# A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models

Dominik Stein
Stefan Hanenberg
Rainer Unland

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

**Abstract.** Specifying queries on models is a prerequisite to model transformations in the MDA because queries select the model elements that are the source of transformations. Current responses to OMG's MOF 2.0 QVT RFP mostly propose to use (and/or extend) OCL 2.0 as specification language for queries. In this paper, we demonstrate that using textual notations (like OCL) quickly leads to complex query statements even for simple queries. In order to overcome this handicap, we present a graphical notation based on the UML that facilitates comprehension of query statements as well as estimation of the (ultimately) selected model elements. We advocate that queries should be specified in terms of user model entities and user model properties (rather than meta-model entities and meta-model properties) for the sake of feasibility and comprehensibility to the user.

## 1 Introduction

Model-Driven Architecture (MDA) [13] aims to assist the development process of software intensive systems by providing a standardized framework for the specification of software artifacts and integration directives. Its key idea is to install traceable relationships between software artifacts of different domains or different development phases. In that way, the MDA aims to improve software quality since software developers can directly relate the final program code to design decisions and/or requirement specifications of the early phases of software development. It allows them to validate and test the final code for compliance to particular requirements, thus making maintenance much simpler. Further, the MDA promotes reuse of existing system solutions in new application domains by means of conceptual mappings and artifact integration.

The principal software artifact of consideration in the MDA are machine-readable models. The underlying technique of the MDA is model transformation. Transformations are accomplished according to the tracing and mapping relationships established between the software artifacts (i.e., between their models).

Striving for a standardized language to define such model transformations, the OMG released the "MOF 2.0 Query / Views / Transformation (QVT)" Request For Proposal (RFP) in April 2002 [14]. It has been one of the mandatory requirements to come up with a query language to select and filter elements from models, which then can be used as sources for transformations. In response to the RFP, several proposals for general-purpose model transformation languages have been submitted (e.g., [1], [4], [7] and [17]). Most of them propose to use (and/or extend) the Object Constraint Language (OCL) 2.0 [15] as query language (e.g., [7] [17] [1]). Having said so, only one proposition [17] provides a graphical representation for its (OCL-based) query language.

We think, though, that a graphical notation to specify and visualize model queries is inevitable for the MDA to drive for success. We think that software developers require a graphical representation of their selection queries, which they can use to communicate their ideas to colleagues, or to document design decisions for maintainers and administrators. A graphical visualization would facilitate their comprehension on where a transformation actually modifies their models. We think that using a textual notation (like OCL), instead, would quickly turn out to lead to very complex expressions even when defining a relatively small number of selection criteria.

In this paper we present a graphical notation to specify selection queries on models specified in the Unified Modeling Language (UML) [16], aiming to overcome the lack of most of the RFP responses when working in a UML model context. We introduce several abstraction means in order to express various selection criteria, and specify how such selection criteria are evaluated by OCL expressions. Query models built from such abstraction means are called "Join Point Designation Diagrams" ("JPDD" in short). JPDDs originate in our work on Aspect-Oriented Software Development (AOSD) [5] in general, and on the visualization of aspect-oriented concepts in particular. JPDDs are concerned with the selection of points in software artifacts that are target to modifications. JPDDs extend the UML with well-defined selection semantics. They make use of, and partially extend, UML's conventional modeling means.

The remainder of this paper is structured as follows: In the first section we emphasize the need of a graphical notation to specify selection queries with the help of an example. After that, we briefly sketch the background that JPDDs originate from, and point to the parallels of query specification in AOSD and MDA. Then, we describe the abstract syntax of our notation. In section 5, we present our graphical means and the OCL expressions by which they are evaluated. We conclude the paper with relation to other work and a summary.

## 2 Motivation

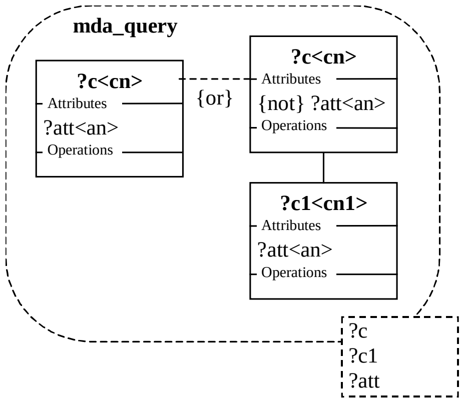
In order to make the motivation of this work more clear, we take a look at a hypothetical, yet easy-to-understand example (adopted from [17]): Imagine, for some arbitrary model transformation, we need a model query that selects all classes with name "cn" that either have an attribute named "an", or – in case not – that have an association to some other class with name "cn1" which in turn has an attribute named "an". Fig. 1, right part, demonstrates how such query would be expressed using the textual and graphical notation as proposed in [17]. Fig. 1, left part, shows the same query, once expressed as an OCL statement, and once expressed as a JPDD.

As you can learn from the example, even a simple model query quickly results in a complex query expression – when using a textual notation. Compared to the OCL code (see Fig. 1, top left part), the textual notation presented in [17] (see Fig. 1, top right part) proves to be more concise. However, comprehension of the query and estimation what model elements finally will be selected is still difficult. The graphical notation shown in Fig. 1, bottom right part, helps to keep track of what is going on in the selection query. However, since the query is specified in terms of meta-model entities and meta-model properties, unnecessary and distracting noise is added to the diagram: A simple association between classes "c" and "c1" is represented by three distinct entities.

Fig. 1, bottom left part, shows what the query looks like using a JPDD. JPDDs represent model queries in terms of user model entities and user model properties. Using user model entities and user model properties for query specification (rather than meta-model entities and meta-model properties) is to the advantage of feasibility and comprehensibility: Software developers work with symbols they are familiar with. They do not need to bother with meta-models. Further, query models turn out to be concise and comprehensible: They specify a minimal pattern to which all ultimately selected model elements must comply.

```
someUmlModel.contents
->select(c: Class | -- c1: Class | -- att: Attribute |
    (c.name="cn" and c.feature->select(f |
        f.oclIsKindOf(Attribute))->includes(att |
            att.name="an") )
or  (c.name="cn" and not c.feature->select(f |
        f.oclIsKindOf(Attribute))->includes(att |
            att.name="an") and
     c1.name="cn" and c1.feature->select(f |
        f.oclIsKindOf(Attribute))->includes(att |
            att.name="an") and
    c.oppositeAssociationEnds->includes(ae |
        ae.participant = c1 )
```

```
(UML.Class, c) [name = "cn", feature =
    { (UML.Attribute) [name="an"] } ]
or
(UML.Class, c) [name = "cn", feature =
    { not (UML.Attribute) [name="an"] } ] and
(UML.Class, c1) [name = "cn1", feature =
    { (UML.Attribute) [name="an"] } ] and
(UML.Association) [connection =
    { (UML.AssociationEnd) [participant = c],
    (UML.AssociationEnd) [participant = c1] } ]
```



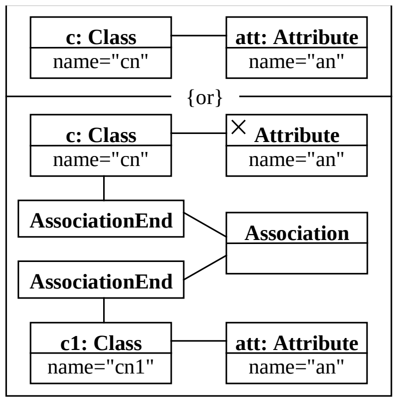


**Fig. 1.** Selection query expressed in OCL (*top left part*), using the textual and graphical notation presented in [17] (*right part*), and with help of a JPDD (*bottom left part*)

## 3 Background

JPDDs originate in our work on AOSD. AOSD deals with the encapsulation of *crosscutting concerns* into separate modular units, called *aspects*. A crosscutting concern is a concern that cannot be cleanly decomposed to the primary decomposition of a program, thus leading to crosscutting code that is scattered throughout every module of the dominant decomposition. This is what became known as the *Tyranny of the Dominant Decomposition* [20]. An aspect encapsulates the crosscutting code of a crosscutting concern. Besides specifying the crosscutting code that should be injected into the primary decomposition, an aspect also specifies the conditions under which the injection shall to take place.

In order to do so, aspect-oriented programming techniques rely on the concepts of *join points* and *weaving*. Join points designate *loci* (in program code) or *instants* (in program execution) at which injection takes place. Weaving defines the exact *manner* in which injection takes place. Since crosscutting usually takes place at more than one join point (in fact, this is the major case that AOSD is focused on), aspect-oriented programming techniques provide various ways to specify selections of join points. For example, join point selection is possible based on lexical similarity of join point properties [11] [12] (e.g., of their name or type declarations), based on the structural arrangement join points reside in [6] (such as presence of particular parameters in an operation's parameter list, or existence of a navigable path to a particular class), or based on the dynamic context join points occur in [12] (e.g., in the scope of a particular object, or in the control flow of a particular method).

We see strong parallels between AOSD and MDA with respect to the selection of locations in software artifacts that are focus of modification. We estimate (e.g., from the examples given in [17]) that selection in MDA also depends on lexical similarities of model element properties – in particular, of their names. Further, structural arrangements, such as the existence of certain features or relationships, are deemed to play a major role in model element selection, as well. Structural constraints may also involve general statements on navigable paths, i.e., indirect associations or indirect generalizations between classifiers.

In the following, we describe the general syntax of JPDDs as it has been depicted in Fig. 1 (bottom left part). After that, we explain the graphical elements that may be used to specify model element selections based on lexical similarities and structural arrangements with JPDDs. We further detail their semantic implications using OCL expressions.

## 4 Abstract Syntax

JPDDs can be fully integrated into the UML, making use of UML's modeling means and its meta-model. Fig. 2, top left part, depicts the abstract syntax of a JPDD: A JPDD 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. 2, bottom right part, depicts the abstract syntax of JPDDs in terms of UML's meta-model: A JPDD represents a namespace which may contain several model elements, each representing a selection criterion. The namespace is provided with a

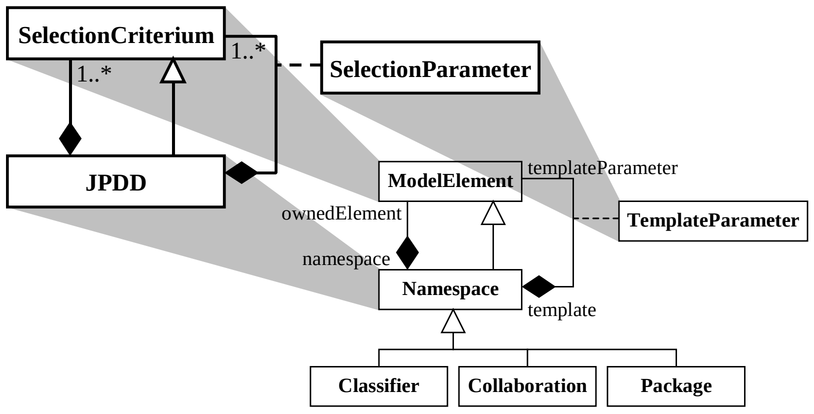


**Fig. 2.** Abstract syntax of JPDDs (*top part*) mapped to UML's meta-model (*bottom part*)

set of template parameters that hold the set of model elements being returned by the query. Being provided with template parameters, a JPDD renders a *template* in the UML. A JPDD may be reified by different types of namespaces – whatever suits the particular needs of the query specification best. A JPDD may be specified as a classifier template, a collaboration template, or a package template, for example. Fig. 1 in section 2 outlines a collaboration template.

Before going into the semantic details of JPDD in the next section, we want to emphasize that the general 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 "generation pattern"), JPDDs are used to identify all model elements that share one common shape (a "selection pattern"). Correspondingly, template parameters of JPDDs are meant to return actual arguments rather than being bound to actual arguments (we describe in the next section, how such arguments are retrieved and affiliated with the individual template parameter). To emphasize this 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).

## 5 Notation and Semantics

In this section we present the core modeling means we developed for specifying selection queries using JPDDs. We explain their graphical notation, and define their semantic implications using OCL meta-operations. Note that not all meta-operations are shown due to space limitations.

### 5.1 Initiating Selection

Selection is accomplished by special meta-operations that are defined on UML's meta-model classes (e.g., on classifiers, attributes, operations, associations, messages, etc.). It sets out with the elements being designated as template parameters, proceeds with their composite elements and their relationships to other elements, starts over with those related elements, and cascades that way throughout the entire JPDD. Fig. 3 sketches how pattern evaluation of is accomplished in case of a template parameter for classifiers ("cPattern"):

The selection is initiated by a special operation defined on the template parameters of the JPDD (see OCL expression no. 1 in Fig. 3): Operation "matchingModelElements" returns all model elements that comply to the selection criteria specified in the JPDD. The operation takes a namespace (e.g., a model, package, collaboration, classifier, etc.) as parameter, whose model elements are to be matched against the selection criteria specified in the JPDD. Matching is accomplished by invoking the appropriate meta-operation on each model element in that namespace.

When matching a classifier pattern ("cPattern"), for example, operations "matchesClassifier" and "matchesRelationships" are invoked on each classifier in the namespace (see no. 2 and 3 in Fig. 3). These operations evaluate if a given classifier in the provided namespace possesses all attributes, operations, associations, and parents, etc., that have been specified in the JPDD. Such evaluations are accomplished by special operations (see no. 4 in Fig. 3), which in turn make use of other operations, and so on (see no. 5 and 6 in Fig. 3; note that not all operation invocations are shown). It is important to note from Fig. 3 how relationship matching involves matching the participating classifiers. That way, evaluation cascades from classifier to classifier, assessing if all selection criteria in the JPDD are fulfilled.

In the subsequent sections, we exemplify selected meta-operations in closer detail.

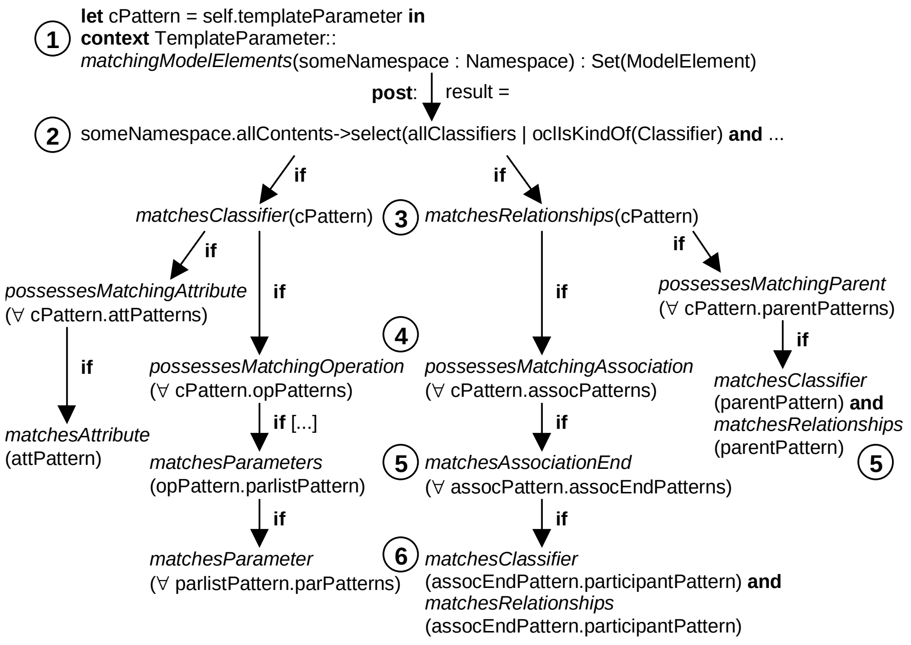

1
**let** cPattern = self.templateParameter **in**
**context** TemplateParameter::
*matchingModelElements*(someNamespace : Namespace) : Set(ModelElement)
**post**: result =
2
someNamespace.allContents->select(allClassifiers | oclIsKindOf(Classifier) **and** ...
if
if
*matchesClassifier*(cPattern)
3
*matchesRelationships*(cPattern)
if
if
if
if
*possessesMatchingAttribute* (∀ cPattern.attPatterns)
*possessesMatchingParent* (∀ cPattern.parentPatterns)
4
if
if
*possessesMatchingOperation* (∀ cPattern.opPatterns)
*possessesMatchingAssociation* (∀ cPattern.assocPatterns)
*matchesClassifier* (parentPattern) **and** *matchesRelationships* (parentPattern)
if [...]
if
*matchesAttribute* (attPattern)
*matchesParameters* (opPattern.parlistPattern)
5
*matchesAssociationEnd* (∀ assocPattern.assocEndPatterns)
5
if
if
*matchesParameter* (∀ parlistPattern.parPatterns)
6
*matchesClassifier* (assocEndPattern.participantPattern) **and** *matchesRelationships* (assocEndPattern.participantPattern)

**Fig. 3.** Cascading evaluation of JPDDs (note that not all evaluation steps are shown)

## 5.2 Classifier Selection

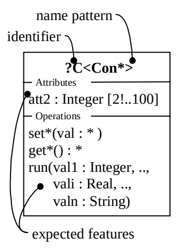
Looking at the selection semantics for classifiers, we may learn about the general selection mechanism for all model elements: Principally, model elements are selected based on the values of their meta-attributes. In case of classifiers, these are the properties "isAbstract", "isLeaf", and "isRoot" (see Table 1, block II).

Besides that, model elements are selected based on their meta-relationships to composite model elements. In case of classifiers, for example, special regards must be given to the features they must posses in order to be selected (see Table 1, block III).

At last, note that name matching of model elements is accomplished with help of name patterns. Name patterns may contain wildcards, such as "*" and "?", in order to select groups of model elements (of same type) based on lexical similarities. All element names in a JPDD represent name patterns by default. In case an element needs to be referenced within the JPDD (e.g., if it needs to be defined as a JPDD template parameter), the element may be given an identifier[1]. In diagrams, identifiers are indicated by a prepending question mark, while the name pattern is enclosed into angle brackets (see "?C<Con*>" in Table 1 for example, or "?c<cn>", "?c1<cn1>", and "?att<an>" in Fig. 1 of section 2).

Having explained these general selection principles, we concentrate on discussing the particularities of other modeling means in the following.

**Table 1.** OCL meta-operation for matching classifiers (*left part*), and its invocation using a sample pattern (*right part*)

| | |
|---|---|
| **context** Classifier:: | A sample class pattern |
| matchesClassifier(C : Classifier) : Boolean | (?C): |
| **post:** result = **-- I. evaluate name pattern** | |
| **if** [...] **-- given an identifier**(see footnote 1) |  name pattern<br>identifier<br>?C<Con*><br>Attributes<br>att2 : Integer [2!..100]<br>Operations<br>set*(val : * )<br>get*() : *<br>run(val1 : Integer, ..,<br>vali : Real, ..,<br>valn : String)<br>expected features |
|     self.*matchesNamePattern*(C.taggedValue->[...]) | |
| **else** **-- default** | |
|     self.*matchesNamePattern*(C.name) | |
| **endif** | |
| **-- II. 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 = '') | |
| **-- III. evaluate attributes and operations** | |
| **and** (C.feature->select(f \| f.oclIsKindOf(Attribute))->forAll(ATT \| | |
|     self.*possessesMatchingAttribute*(ATT)) | ...and its evaluation: |
|     **or** C.feature->select(f \| f.oclIsKindOf(Attribute))->size = 0) | |
| **and** (C.feature->select(f \| f.oclIsKindOf(Operation))->forAll(OP \| | [...]->select(ME \| [...] |
|     self.*possessesMatchingOperation*(OP)) | **and** ME.*matches* |
| **or** C.feature->select(f \| f.oclIsKindOf(Operation))->size = 0) | *Classifier*(?C) [...]) |

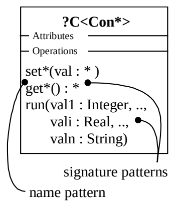
## 5.3 Operation Selection

Special regards in operation selection must be given to the usage of wildcard ".." in the operation's signature pattern. Wildcard ".." allows operation selection based on their structural arrangement – that is, based on existence of particular parameters, while neglecting from others.

[1] In that case, the name pattern is stored (technically) in a special tagged value.

Table 2 gives a detailed description on how such structural arrangements are evaluated by means of an OCL expression: Meta-operation "matchesParameters" compares the overall number (see Table 2, block I) and order (see Table 2, block II) of parameters in the actual operation's parameter list to the one being passed from the JPDD. To do so, the meta-operation makes use of sub-expressions "ownPars", "purePars", and "matchingPars". "ownPars" comprises all parameters of the actual operation ("self"). "purePars" equates the parameter list being passed from the JPDD, neglecting all wildcarded parameters "..". "matchingPars" is a subset of "ownPars", and contains only those parameters that have a matching counterpart in "purePars". Apart from the overall order of parameters, the partial order of parameters at the parameter list's beginning (see Table 2, block III) and its end (see Table 2, block IV) is evaluated. Order evaluation stops (i.e., is always true) when the first wildcarded parameter ".." is reached in the parameter list passed from the JPDD (see collect statement at end of block III and IV).

**Table 2.** OCL meta-operation for matching parameter lists (*left part*), and its invocation using a sample pattern (*right part*)

| | |
|---|---|
| **context** Operation **def:**<br>**let** ownPars = self.parameter->asSequence()<br>**let** purePars(par : Sequence(Parameter)) : Sequence(Parameter)<br>        = par->reject(p \| p.name = '..')<br>**let** matchingPars(par : Sequence(Parameter)) : Sequence(Parameter)<br>        = ownPars->select(p \|<br>        purePars(par)->exists(parp \| p.*matchesParameter*(parp)) )<br>**context** Operation::<br>matchesParameters(par : Sequence(Parameter)) : Boolean<br>**post:** result =     **-- I. compare parameter number**<br>    (matchingPars(par)->size() = purePars(par)->size())<br>        **-- II. compare parameter order**<br>**and** matchingPars(par)->forAll(index : Integer \|<br>    matchingPars(par)->at(index)<br>        .*matchesParameter*( purePars(par)->at(index) ))<br>        **-- III. compare first parameters**<br>**and** ownPars->forAll(i : Integer \| ownPars->at(i)<br>        .*matchesParameter*(par->at(i))<br>   **or** par->collect(j : Integer \| j <= i<br>      **and** par->at(j).name = '..')->size() <> 0)<br>        **-- IV. compare last parameters**<br>**and** ownPars->forAll(i : Integer \| ownPars->at(ownPars->size() - i)<br>        .*matchesParameter*(par->at(par->size() - i))<br>   **or** par->collect(j : Integer \| j <= i<br>      **and** par->at(par->size() - j).name = '..')->size() <> 0) | A sample signature pattern (run) providing a sample parameter list ({val1 : Integer, .., vali : Real, .., valn : String}):<br> ?C<Con*><br>Attributes<br>Operations<br>set*(val : * )<br>get*() : *<br>run(val1 : Integer, ..,<br>vali : Real, ..,<br>valn : String)<br>signature patterns<br>name pattern<br>...and its evaluation:<br>[...]->select(op \| [...] **and** op.*matchesParameters* (run.parameter ->asSequence()) [...]) |

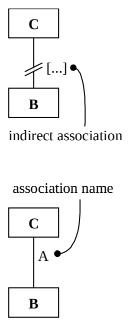
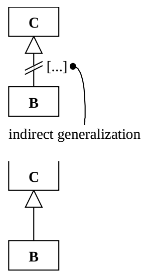
## 5.4 Relationship Selection

When selecting relationships, special regards must be given to indirect relationships. Indirect relationships are a sophisticated means to constrain structural arrangements: Indirect relationships may be used in JPDDs to indicate that a classifier does not need to be directly connected to a particular parent, child, or associated classifier. This means in case of associations, that the particular classifier must be reachable via the designated association, but does not need to be a direct neighbor.

In diagrams, indirect relationships are rendered by a double-crossed line[2]. In Table 3 (top part), for example, there must be a navigable path from class "C" to class "B" for the selection criterion to be fulfilled. The ends of that path must match with the association ends of the indirect association (operation "allIndirectNeighbors" returns the set of all (opposite) association ends being reachable via the passed association).

In case of generalizations, the particular parent or child needs to reside somewhere in the inheritance tree, but does not need to be a direct parent or child. For example, class "C" in Table 3 (bottom part) must be among the ancestors of class "B", and class "B" must be among the descendants of class "C", for the selection criterion to be satisfied (operation "allParents" returns the set of all inherited super-classifiers).

**Table 3.** OCL meta-operation for matching relationships (*left part*), and sample patterns (*right part*). Invocation examples are omitted due to space limitations

| | |
|---|---|
| **context** Classifier::<br>possessesMatchingAssociation(a : Association, c : Classifier) : Boolean<br>**post:** result =<br>**if** [...] **-- indirect association** **-- I. evaluate indirect neighbors**<br>    self.associations->includes(A \| A.*matchesAssociation*(a) **and**<br>    a.allConnections->select(ae \| ae.participant = c)->forAll(ae \|<br>      A.allConnections->select(AE \| AE.participant = self)<br>        ->includes(AE \| AE.*matchesAssociationEnd*(ae) **and**<br>         a.allConnections->select(ae \| ae.participant <> c)<br>        ->forAll(ae2 \| self.*allIndirectNeighbors*(A)<br>        ->includes(AE2 \| AE2.*matchesAssociationEnd*(ae2))))))<br>**else** **-- direct association** **-- II. evaluate direct neighbors**<br>    [...]<br>**endif** |  C<br>[...]<br>B<br>indirect association<br>association name<br>C<br>A<br>B |
| **context** Classifier::<br>possessesMatchingParent(g : Generalization) : Boolean<br>**post:** result =<br>**if** [...] **-- indirect association** **-- I. evaluate indirect parents**<br>    self.generalization->includes(G \| G.*matchesGeneralization*(g) **and**<br>    G.parent->union(G.parent.allParents)->includes(C \|<br>        C.*matchesClassifier*(g.parent) **and**<br>        C.*matchesRelationships*(g.parent)))<br>**else** **-- direct association** **-- II. evaluate direct parents**<br>    [...]<br>**endif** |  |

## 5.5 Multiplicity Restrictions

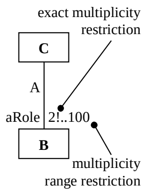
Special attention in association end selection must be paid to the association end's multiplicity specification[3]: Multiplicity of association ends may declare exact upper and/or lower limits; or it may designate the upper and/or lower bounds which the multiplicity of an association end must not exceed or underrun (respectively). Being able to declare exact limits and/or minimal and maximal bounds provides for further flexibility in specification of structural arrangements. Graphically, exact multiplicity

[2] Technically, indirect relationships are defined as a special stereotype of associations or generalizations, respectively.

[3] The same counts for the multiplicity specification of attributes.

bounds are indicated by exclamation marks[4]. The lower multiplicity limit of association end "aRole" in Table 4, for example, denotes a strict limit. Accordingly, association ends are only selected, if their lower multiplicity limit equates "2". The upper multiplicity limit of "aRole", on the contrary, denotes a maximum. Association ends are selected as long as their upper multiplicity limit does not exceed "100".

**Table 4.** OCL meta-operation for matching association ends (*left part*), and its invocation using a sample pattern (*right part*)

| | |
|---|---|
| **context** AssociationEnd::<br>isSuccessfulMatch(ae : AssociationEnd) : Boolean<br>**post:** result = [...]<br>                                **-- evaluate multiplicity**<br>**and** ((**if** [...]  **-- exact limit**<br>        self.multiplicity.range.lower   = ae.multiplicity.range.lower<br>    **else**      **-- minimum bound**<br>        self.multiplicity.range.lower >= ae.multiplicity.range.lower<br>    **endif**<br>**and**   **if** [...]  **-- exact limit**<br>        self.multiplicity.range.upper   = ae.multiplicity.range.upper<br>    **else**      **-- maximum bound**<br>        self.multiplicity.range.upper <= ae.multiplicity.range.upper<br>    **endif**)<br>**or** ae.multiplicity = '') | A sample association end pattern (aRole):<br> exact multiplicity restriction<br>C<br>A<br>aRole<br>2!..100<br>B<br>multiplicity range restriction<br>...and its evaluation:<br>[...]->select(AE \| [...]<br>**and** AE.*isSuccessful Match*(aRole) [...]) |

## 5.6 Message Selection

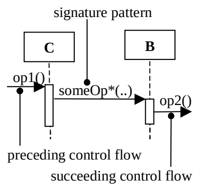
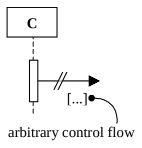
Selection is not restrained to structural aspects of a UML model as they are specified in UML class diagrams, for example. Selection criteria may as well involve behavioral requirements as they are specified in UML interaction and collaboration diagrams. Table 5 shows the notational means to specify selection criteria on messages, and how such criteria are evaluated by an OCL operation.

Messages are selected based on the action they invoke. In case of operation call actions, signature patterns may be used to restrict the operation called. Further, messages are selected based on the base classifiers of their sender and receiver roles. It is important to note that selection is based on the base 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 thereof. The same counts for the associations used for transmitting the messages.

Lastly, messages may be selected based on the control flow they occur in, and/or based on the control flow they invoke. Such control flow is delineated by predecessor and successor messages: Message "someOp" in Table 5, for example, must occur in the control flow of message "op1", and must invoke message "op2". Messages of special stereotype "indirect" can be used to indicate arbitrary control flow that may occur between two successive messages. In diagrams, indirect messages are depicted as double-crossed arrows (see Table 5 for illustration).

[4] Technically, fix upper and lower limits are specified as special stereotypes of multiplicity ranges.

**Table 5.** OCL meta-operation for matching messages (*left part*), and its invocation using a sample pattern (*right top part*). Graphical representation of messages stereotyped as "indirect" (*right bottom part*)

| | |
|---|---|
| **context** Message::<br>isSuccessfulMatch(m : Message) : Boolean<br>**post:** result = **-- I. evaluate name pattern**<br>**if** [...] **-- given an identifier**(see footnote 1)<br>self.*matchesNamePattern*(m.taggedValue->[...])<br>**else -- default**<br>self.*matchesNamePattern*(m.name)<br>**endif**<br>**-- II. evaluate sender/receiver/...**<br>**and** self.sender.base->includes(C \|<br>C.*matchesRelationships*(m.sender) **and**<br>C.*matchesClassifier*(m.sender))<br>**and** self.receiver.base->includes(C \|<br>C.*matchesRelationships*(m.receiver) **and**<br>C.*matchesClassifier*(m.receiver))<br>**and** self.communicationConnection.base<br>.*matchesAssociation*(m.communicationConnection)<br>**-- III. evaluate predecessors/activator**<br>**and** m.allPredecessors->union(m.activator)->reject(m2 \|<br>m2.stereotype->includes(st \| st.name='indirect'))->forAll(m2 \|<br>self.allPredecessors->includes(M \| M.*matchesMessage*(m2)))<br>**-- IV. evaluate successors**<br>**and** m.*allSuccessors*->reject(m2 \|<br>m2.stereotype->includes(st \| st.name='indirect'))->forAll(m2 \|<br>self.*allSuccessors*->includes(M \| M.*matchesMessage*(m2)))<br>**-- V. evaluate action**<br>**and** self.action.*matchesAction*(m.action) | A sample message pattern (someOp*):<br> signature pattern<br>C<br>B<br>op1()<br>someOp*(..)<br>op2()<br>preceding control flow<br>succeeding control flow<br>...and its evaluation:<br>[...]->select(M \| [...] **and** M.*isSuccessful Match*(someOp*) [...])<br> |

## 5.7 Boolean Restrictions

By default, all selection criteria specified in a JPDD are implicitly combined using a boolean "and". That is, all such selection criteria must be fulfilled by a given model element in order to be selected by the query. In some cases, though, we may need to specify alternative, exclusive, or mutual exclusive selection criteria. Table 6 exemplifies how we may use boolean constraints, i.e., "{or}", "{xor}", and "{not}", to render such combinations of selection criteria. Operation "matchesAttribute", for example, specifies that either at least, or exactly, one (respectively) of all model elements interrelated by a boolean constraint must comply to the selection criteria (operation "allConstrainedElements" returns the set of all model elements being interconnected by the passed boolean constraint). Operation "matchesAttributeNot" inverts the result of matching in case the model element is constrained with "{not}". See Table 6 on next page for illustrations.

# 6 Example

With help of the notational means presented in the previous section, we now can define even complex selection queries without getting lost in its specification:

**Table 6.** OCL meta-operations for evaluating boolean combinations on attributes (*left part*), and its invocation using a sample pattern (*right part*)

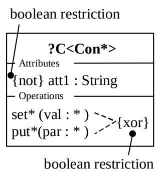
| | |
|---|---|
| **context** Attribute::<br>matchesAttribute(att : Attribute) : Boolean<br>post: result =<br>**if** att.constraint->includes(c \| c.body = 'or') **then**<br>att.*allConstrainedElements*('or')->exists(att1 \|<br>self.*matchesAttributeNot*(att1)<br>)<br>**else**<br>**if** att.constraint->includes(c \| c.body = 'xor') **then**<br>att.*allConstrainedElements*('xor')->collect(att1 \|<br>self.*matchesAttributeNot*(att1)<br>).size = 1<br>**else**<br>self.*matchesAttributeNot*(att)<br><br>**endif endif** | A sample attribute pattern (set*), interrelated to another attribute pattern by means of boolean {xor}:<br> boolean restriction<br>?C<Con*><br>Attributes<br>{not} att1 : String<br>Operations<br>set* (val : * )<br>put*(par : * )<br>{xor}<br>boolean restriction<br>...and its evaluation:<br><br>[...]->select(ATT\| [...]<br>**and** ATT.*matches Attribute*(set*) [...]) |
| **context** Attribute::<br>matchesAttributeNot(att : Attribute) : Boolean<br>**post:** result =<br>**if** att.constraint->includes(c \| c.body = 'not') **then**<br>not self.*isSuccessfulMatch*(att) **-- evaluate non-existence of att's**<br>**else**<br>self.*isSuccessfulMatch*(att) **-- evaluate existence of attributes**<br>**endif** | |

Fig. 4 depicts a sample JPDD that selects all classifiers (identified with "?C") (1) matching the name pattern "Con*"; (2) that do *not* have an attribute matching "att1" of type "String"; (3) that do have an array attribute matching "att2" of type "Integer" whose lower bound equates "2", and whose upper bound does not exceed "100"; (4) that either have an operation matching "set*", or an operation matching "put*" (but not both) that both take one parameter of arbitrary type; (5) that have an operation matching "get*" that returns an value of arbitrary type; (6) that have an operation matching "run" that takes (at least) three parameters: (6a) the first parameter in the operation's parameter list must be of type "Integer", (6b) the last parameter must be of type "String"; (6c) besides that, the operation must take a third parameter of type

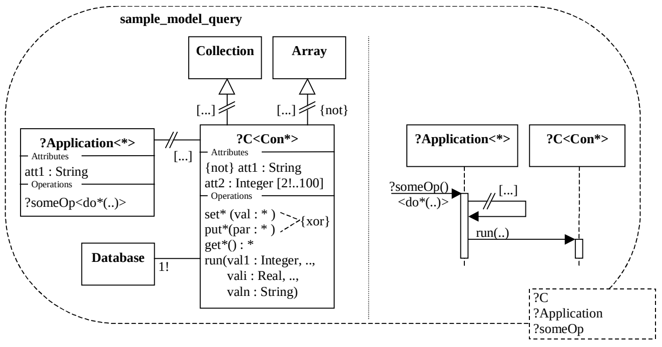


**Fig. 4.** A sample JPDD

"Real" (no matter at which position in the operation's parameter list). Selected classifiers must be (7) subtypes of "Collection"; (8) but *not* subtypes of "Array"; and (9) they have to have an association to exactly one classifier matching "Database".

Besides that, selected classifiers must possess an indirect association (i.e., a navigable path) to a classifier (identified with "?Application") (1) matching "*"; (2) that has an attribute matching "att1" of type "String"; (3) and that has an operation matching "do*" (and identified with "?someOp"), which takes any number of parameters, (3a) and which invokes method "run" on the former classifier (identified with "C") – (3b) no matter when (see double-crossed back loop in right part of Fig. 4) – (3c) using arbitrary values as parameters.

Having found actual model elements that comply to these selection criteria, the JPDD returns the resulting model elements via its template parameters "?C", "?Application", and "?someOp".

## 7 Related Work

Most submissions to OMG's QVT RFP propose to use OCL as a query language. In this section, we reflect on existing approaches for the visualization of OCL expressions with respect to their ability to represent model element selections.

Constraint Diagrams [8], for example, represent a graphical notation to specify invariants on objects and their associations (i.e., links) depending on the state they are in. In consequence to its strict focus on runtime constraints, the notation does not provide for the specification of model element queries, though. In particular, no means are provided to designate model elements that serve as sources for transformations. Further, the notation is not concerned with the specification of structural selection constraints, such as existence of particular features.

Visual OCL [3] [9] is a graphical notation to express OCL constraints. It provides graphical symbols for all OCL keywords, in particular for the "select" statement as we need it for model element selection in MDA. For example, Fig. 5 portrays how the OCL selection specified in section 2 (Fig. 1) would be represented using Visual OCL. Note that similar to [17], Visual OCL does not provide for the specification of model element queries in terms of user model entities. In consequence, users are confronted

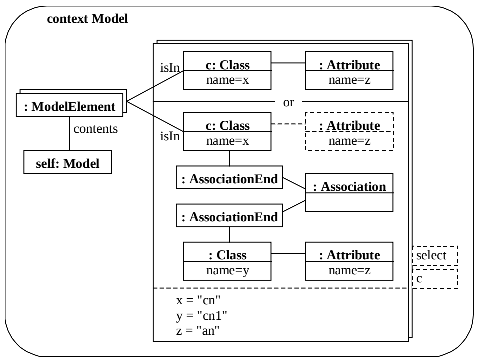


**Fig. 5.** OCL select statement from Fig. 1 (section 2) represented in Visual OCL [3] [9]

with the full load of OCL complexity – in particular when specification of structural abstractions such as indirect associations or generalization relationships is necessary (see section 5.4).

The idea of specifying queries in terms of user model entities we borrowed from the approach of Query-By-Example (QBE) [21], which is a common query technique in the database domain: We specify sample model entities, having sample properties, and determine how selected model elements must relate to such samples. We make use of "operator" symbols (such as wildcards, exclamation marks, and double-crossed lines and arrows) to differentiate whether selected model elements must match the samples exactly, or with a permissible degree of deviation (e.g., multiplicity boundaries may be specified to denote minimum and maximum values rather than perfect matches).

As already mentioned above, AOSD is another application area for JPDDs. Here, JPDD are used to visualize selections of join points, i.e., they render those points in program code, or program execution, that are to be enhanced by an aspect. In [18], we demonstrate by example how JPDDs may be used to model join point selections in most popular aspect-oriented programming languages. To be more precise, we describe how JPDDs may be used to represent *pointcuts* in AspectJ [2], *traversal strategies* in Adaptive Programming [10], or *concern mappings* in Hyper/J [19], etc.

## 8 Conclusion

In this paper, we presented a graphical notation to specify model queries on UML models. We identified model queries to be prerequisites to model transformations as they are specified in the Model-Driven Architecture (MDA). We demonstrated that even simple query specifications tend to become excessive and complex when using a textual notation. Aiming to overcome this quandary, we introduced Join Point Designation Diagrams (JPDD) to specify and represent model queries graphically. We explained their abstract syntax, and the graphical means they may contain to specify the queries' selection criteria. We exemplified OCL meta-operations for the evaluation of such selection criteria on actual user model elements. We demonstrated the use of JPDDs using a quite complex model query, proving that even then the query specification remains comprehensible.

The particular focus of this work has been on providing graphical means for the specification of model element queries based on lexical similarity (e.g., based on name and signature patterns) and structural arrangements (e.g., based on indirect relationships). We extrapolated the need of such selection means from the area of Aspect-Oriented Software Development (AOSD), where JPDDs were originally developed for. We think that providing our graphical means with OCL semantics can assist developers in both AOSD and MDA when specifying and modeling selections. It is important to note that JPDDs are not capable – and not intended – to represent OCL expressions in the general case. Their clear focus is on selection techniques in AOSD. Currently, we are working on further improvements for the specification of structural constraints on operation parameter lists.

With respect to the MDA domain, we think that it is advantageous to specify model queries in terms of user models, rather than meta-models, in order to facilitate their specification and comprehension to the user. In this paper, we have presented a query language for the UML. We advocate for the development of further query languages in other modeling and domain-specific languages. That way, transformations may be specified in terms of user model entities, simply by relating one user-model-based query to another user-model-based query, for the sake of feasibility and comprehensibility to the user.

## References

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

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

[3] Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G., *A Visualization of OCL Using Collaborations*, in: Proc. of UML 2001 (Toronto, Canada, Oct. 2001), Springer, pp. 257-271

[4] CBOP, DSTC, IBM, *Revised Submission for MOF 2.0 Query / Views / Transformations RFP*, 18. Aug. 2003 (http://www.dstc.edu.au/pegamento/publications/ad-03-08-03.pdf)

[5] Filman, R., Elrad, T., Clarke, S., Aksit, M., (eds.), *Aspect-Oriented Software Development*, Addison-Wesley, 2004 (to appear)

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

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

[8] Kent, S., *Constraint Diagrams: Visualizing Assertions in Object-Oriented Models*, in: Proc. of OOPSLA 1997 (Atlanta, Georgia, Oct. 1997), ACM pp. 327-341

[9] Kiesner, Chr., Taentzer, G., Winkelmann, J., *Visual OCL: A Visual Notation of the Object Constraint Language*, TR 2002/23, Technical University Berlin, 2002

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

[11] Lieberherr, K., Lorenz, D., Mezini, M., *Programming with Aspectual Components*, TR NU-CCS-99-01, Northeastern University, 1999

[12] Masuhara, H., Kiczales, G., Dutchyn, Chr., *A Compilation and Optimization Model for Aspect-Oriented Programs*, in: Proc. of CC 2003 (Warsaw, Poland, Apr. 2003), Springer, pp. 46-60

[13] OMG, *MDA Guide Version 1.0*, OMG, 1. May 2003 (omg/2003-05-01)

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

[15] OMG, *UML 2.0 OCL Specification*, Final Adopted Specification, 2003 (OMG Document pct/03-10-14)

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

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

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

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

[20] Tarr, P., Ossher, H., Harrison, W., Sutton Jr., St., *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, in: Proc. of ICSE 1999 (Los Angeles, CA, May 1999), ACM, pp. 107-119

[21] Zloof, M., *Query-by-Example: A Data Base Language*, IBM Systems Journal, Vol. 16(4), 1977, pp. 324-343