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) [16] 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 [17]. 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], [5], [9] and [20]). Most of them propose to use (and/or extend) the Object Constraint Language (OCL) 2.0 [18] as query language (e.g., [9] [20] [1]). Having said so, only one proposition [20] provides a graphical representation for its 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) [19], 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" [26] (or "JPDD" in short). JPDDs originate in our work on Aspect-Oriented Software Development (AOSD) [7] in general, and on the visualization of aspect-oriented concepts in particular. They are concerned with the selection of points in software artifacts that are target to modifications (so-called "join points" in AOSD). They extend the UML with selection semantics. And they make use of, and partially extend, UML's conventional modeling means.

This paper is an immediate follow-up paper of our submission [24] to the "MDA Foundation and Application" workshop [3]. We carefully revised that submission taking into account the comments and remarks that we received at the workshop. Meanwhile, JPDDs have also been presented in [26]. While there we have identified the general need to specify queries on software artifacts as a new evolving design issue, here we concentrate on the integration of JPDDs into the MDA context. In particular, we describe a generic mechanism to map JPDDs onto OCL statements, thus giving way to the integration of our approach with the current QVT submissions.

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. In section 4, we briefly describe the abstract syntax of our notation. We then present the graphical means as well as 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 [20]): 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 [20]. 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 (cf. Fig. 1, top part). As a result, comprehension of the query and estimation what model elements finally will be selected is rather 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 abstraction means they are familiar with. They do not need to bother about 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.

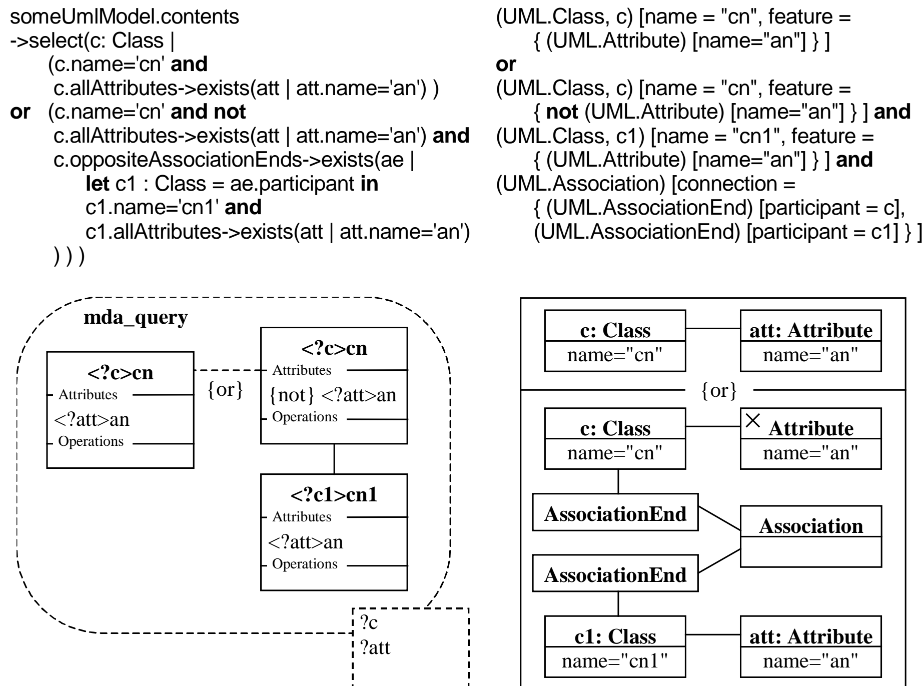


Fig. 1. Selection query expressed in OCL (*top left part*), using the textual and graphical notation presented in [20] (*right part*), and with help of a JPDD (*bottom left part*)

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* [28]. 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 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 similarities of join point properties [14] [15] (e.g., of their name or type declarations), based on the structural arrangement the join points reside in [8] (such as the presence of particular parameters in an operation's parameter list, or the existence of a navigable path to a particular class), or based on the dynamic context join points occur in [15] (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 [20]) 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 explain the graphical elements that we provide to specify model element selections based on lexical similarities and structural arrangements with JPDDs. We briefly sketch their general syntax, and detail their semantic implications using OCL expressions.

4 Notation and Semantics

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). JPDDs can be fully integrated into the UML, making use of UML's modeling means and its meta model: Structurally, JPDDs compare to UML templates of UML namespaces (cf. Fig. 2). Note, though, that semantically JPDDs differ from conventional UML templates since they render a "selection pattern" rather than a "generation pattern". This means in particular that the parameters of JPDDs represent logical variables (which *return* values), while the parameters of a conventional UML template are *fed* with values. To

Fig. 2. Abstract syntax of JPDDs (*top part*) mapped to UML's meta model (*bottom part*) (cf.[26])

emphasize this difference in meaning visually, parameters of JPDDs are summarized at the lower right corner of JPDDs – rather than at their upper right corner as with conventional UML templates (see Fig. 4 in section 5 for an example).

In the following we present the core modeling means that may be used to specify selection queries with help of JPDDs. We explain their graphical notation, and describe how they can be evaluated using OCL meta operations[1]. Such meta operations are appended to UML's meta model classes (e.g., to classifiers, attributes, operations, associations, messages, etc.). Note that not all meta operations are shown due to space limitations. At last, we sketch how the meta operations are deployed in order to retrieve an actual set of matching model elements.

4.1 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

Table 1. OCL meta operation for matching classifiers (*left part*), and a sample class pattern that could be passed as an argument (*right part*)

context Classifier::
matchesClassifier(C : Classifier) : Boolean
post: result = **-- I. evaluate name pattern**
if [...] **-- given an identifier**(see footnote 2)
    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.allAttributes->forAll(ATT | self.*possessesMatchingAttribute*(ATT))
    or C.allAttributes->size() = 0)
and (C.allOperations->forAll(OP | self.*possessesMatchingOperation*(OP))
    or C.allOperations->size() = 0)
	A sample class pattern, which is given an identifier (?C):
	name pattern
	identifier
	<?C>Con*
	Attributes
	att2 : Integer [2!..100]
	Operations
	set*(val : *)
	get*() : *
	run(val1 : Integer, ..,
	vali : Real, ..,
	valn : String)
	expected features

[1] We used *OCL Checker*, version 0.3 (http://www.klasse.nl/ocl/ocl-checker.html), to write the OCL statements, and *OCLE*, version 2.02 (http://lci.cs.ubbcluj.ro/ocle), to typecheck them.

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 possess 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 (see Table 1, block I). Name patterns may contain wildcards, e.g. "*", in order to select groups of model elements 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 parameter), the element may be given an identifier[2]. In diagrams, identifiers are enclosed into angle brackets and are prepended by a question mark (see "<?C>Con*" in Table 1 for example, or "<?c>cn", "<?c1>cn1", and "<?att>an" in Fig. 1 of section 2). They are placed in front of the element they refer to.

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

4.2 Operation Selection

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

Table 2 gives a detailed description on how such structural arrangements are

Table 2. OCL meta operation for matching parameter lists (*left part*), and a sample signature pattern whose parameter list could be passed as an argument (*right part*)

<table>
<tr>
<td>
context Operation def:
let ownPars : Sequence(Parameter) = self.parameter->asSequence()
let patternPars(par : Sequence(Parameter)) : Sequence(Parameter)
= par->reject(p | p.name = '..')
let matchingPars(par : Sequence(Parameter)) : Sequence(Parameter)
= ownPars->select(p |
patternPars(par)->exists(parp | p.matchesParameter(parp)))

context Operation def:
let matchesParameterList(par : Sequence(Parameter)) : Boolean
post: result = -- I. compare parameter order
matchesParameterOrder(matchingPars(par), patternPars(par))
-- II. compare first parameters
and Sequence{1..ownPars->size()}->forAll(i : Integer | ownPars->at(i)
.matchesParameter(par->at(i))
or Sequence{1..par->size()}->collect(j : Integer | j <= i
and par->at(j).name = '..')->size() <> 0)
-- III. compare last parameters
and Sequence{1..ownPars->size()}->forAll(i : Integer | ownPars->at(
ownPars->size() - i)
.matchesParameter(par->at(par->size() - i))
or Sequence{1..par->size()}->collect(j : Integer | j <= i
and par->at(par->size() - j).name = '..')->size() <> 0)
</td>
<td>
A sample signature pattern (run), providing a sample parameter list ({val1 : Integer, .., vali : Real, .., valn : String}):

<?C>Con*
Attributes
Operations
set*(val : *)
get*() : *
run(val1 : Integer, ..,
vali : Real, ..,
valn : String)
signature patterns
name pattern
</td>
</tr>
</table>

---

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

evaluated by means of an OCL expression: Meta operation "matchesParameterList" compares (a) the *overall order* of parameters in the actual operation("self")'s parameter list to the one being passed from the JPDD (Table 2, block I), as well as (b) the *partial order* of parameters at the parameter lists' beginnings (Table 2, block II) and their ends (Table 2, block III). For that purpose, the meta operation defines a couple of sub-expressions: "ownPars" comprises all parameters of the actual operation ("self"); "patternPars" holds the parameters being passed from the JPDD, neglecting all wildcarded parameters ".."; and "matchingPars" is a subset of "ownPars", containing only those parameters that have a matching counterpart in "patternPars".

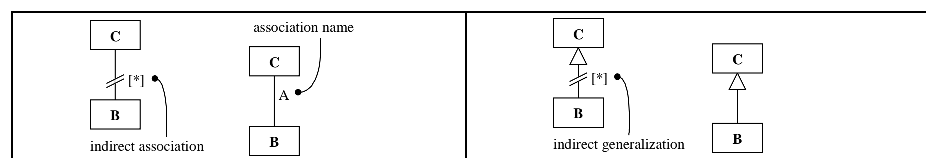
The sub-expressions are used to compare the overall order of parameter lists with help of meta operation "matchesParameterOrder" (not shown here). That operation recursively iterates over "matchingPars" and "patternPars", verifying if (subsequences of) the former contains all the elements belonging to (subsequences of) the latter. The partial order is evaluated based on "ownPars" and the parameter list being passed from the JPDD. 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 II and III).

4.3 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[3]. Table 3 (left part) states, for example, that 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. In case of indirect 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 (right 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. The respective OCL meta operations are omitted here due to space limitations. Please refer to [26] for a detailed description.

Table 3. Sample relationship patterns for (indirect) relationships, which could be passed to a meta operation as an argument (meta operations are omitted here; see [26] for further details)



[3] Technically, indirect relationships are rendered by a special stereotype for associations or generalizations, respectively. Query evaluation is based on the (non-)presence of that stereotype (cf. [26]).

4.4 Multiplicity Restrictions

Special attention in association end selection must be paid to the association end's multiplicity specification[4]: Multiplicity of an association end 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 query specification based on structural arrangements.

Graphically, exact multiplicity bounds are indicated by exclamation marks[5]. 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 a sample association end pattern that could be passed as an argument (*right part*)

context AssociationEnd:: matchesAssociationEnd(ae : AssociationEnd) : Boolean **post:** result = [...] **and** ((**if** [...] **-- exact limit** **-- evaluate multiplicity** self.multiplicity.range.lower = ae.multiplicity.range.lower **else** **-- minimum bound** self.multiplicity.range.lower >= ae.multiplicity.range.lower **endif** **and** **if** [...] **-- exact limit** self.multiplicity.range.upper = ae.multiplicity.range.upper **else** **-- maximum bound** self.multiplicity.range.upper <= ae.multiplicity.range.upper **endif**) **or** ae.multiplicity = '')	A sample association pattern (A), comprising a sample association end pattern (aRole): exact multiplicity restriction C A aRole 2!..100 B multiplicity range restriction

4.5 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 (Table 5, block I). In case of operation call actions, signature patterns may be used to restrict the operation called. Further, messages are selected based on their senders and receivers (Table 5, block II). It is important to note that the OCL operation evaluates the senders' and receivers' base classifiers rather than their role specifications. This is accomplished deeming that selections should consider the full specification of classifiers rather than restricted

[4] The same counts for the multiplicity specification of attributes (see sample classifier pattern in Table 1 for an example).

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

Table 5. OCL meta operation for matching (indirect) messages (*left part*), and a sample message pattern that could be used as an argument (*right part*)

```
context Message::
matchesMessage(m : Message) : Boolean
post: result =                                               -- I. evaluate action
self.action.matchesAction(m.action)
                                              -- II. evaluate sender/receiver/...
and self.sender.base->exists(C |
    C.matchesRelationships(m.sender) and
    C.matchesClassifier(m.sender))
and self.receiver.base->exists(C |
    C.matchesRelationships(m.receiver) and
    C.matchesClassifier(m.receiver))
and self.communicationConnection.base
    .matchesAssociation(m.communicationConnection)
                                                       -- III. evaluate activator
and ((if m.activator.stereotype->exists(st | st.name='indirect') then
    self.activator.matchesReceptionContext(m.activator) and
    self.allActivators->exists(M | M.matchesSendingContext(m.activator))
    else
        self.activator.matches(m.activator)
    endif) or m.activator='')
                                     -- IV. evaluate predecessors/successors/...
and (m.predecessor->forAll(p |
    if p.stereotype->exists(st | st.name='indirect') then
        self.predecessor->exists(P |
            P.matchesSendingContext(p) and
            P.allActivatedMessages->including(P)->exists(M |
                                         M.matchesReceptionContext(p)) )
    else
        self.predecessor->exists(P | P.matches(p))
    endif ) or m.predecessor->size()=0)
and (m.successor->forAll([...]) or m.successor->size()=0) -- analogously
and (m.activated->forAll([...]) or m.activated->size()=0)    -- analogously
```

A sample message pattern (someOp*), and an "indirect" message symbol:

signature pattern
C
B
op1()
someOp*(..)
op2()
activating control flow
activated control flow

C
[...]
arbitrary control flow

projections thereof. The same counts for the associations used for transmitting the messages.

Lastly, messages may be selected based on their activator message, their predecessor and successor messages, as well as based on the messages they are activating themselves (Table 5, block III and IV). This is particularly useful to constrain the (preceding) control flow in which selected messages may occur, as well as the (succeeding) control flow that selected messages may invoke. Message "someOp" in Table 5, for example, must be activated in the control flow of message "op1", and must in turn 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. Technically, indirect relationships are rendered as special message stereotypes. The presence of that stereotype is checked during query evaluation (see Table 5, block III and IV, for illustration). Evaluation of indirect messages is accomplished in two steps: One step is concerned with finding messages that comply to the sending context of the indirect message (i.e. sender role, predecessors, successors, and activator messages); the other step deals with the identification of messages matching to the reception context of the indirect message (i.e. receiver role and subsequently activated messages).

4.6 Combination of Selection Criteria

By default, all selection criteria specified in a JPDD are implicitly combined with "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. In order to render such combinations of selection criteria, we may use constraint strings ("{or}", "{xor}", and "{not}"). The corresponding OCL operations specify that either at least, or exactly, one (respectively) of all model elements interrelated by such a constraint must comply to the selection criteria; or it inverts the result of matching in case the model element is constrained with "{not}". The OCL operations are omitted here due to space limitations. Please refer to [24] for further illustrations.

4.7 Retrieving Matching Model Elements

Retrieval of actual model elements from user models is accomplished using the UML meta model operations as they have been exemplified in the previous sections. A corresponding meta operation is specified for each UML meta model element (whose instances may appear in class/object diagrams or in interaction diagrams, e.g. classifiers, attributes, operations, associations, messages, etc.). In order to retrieve a set of (matching) model elements, the meta operation successively invoke one another so

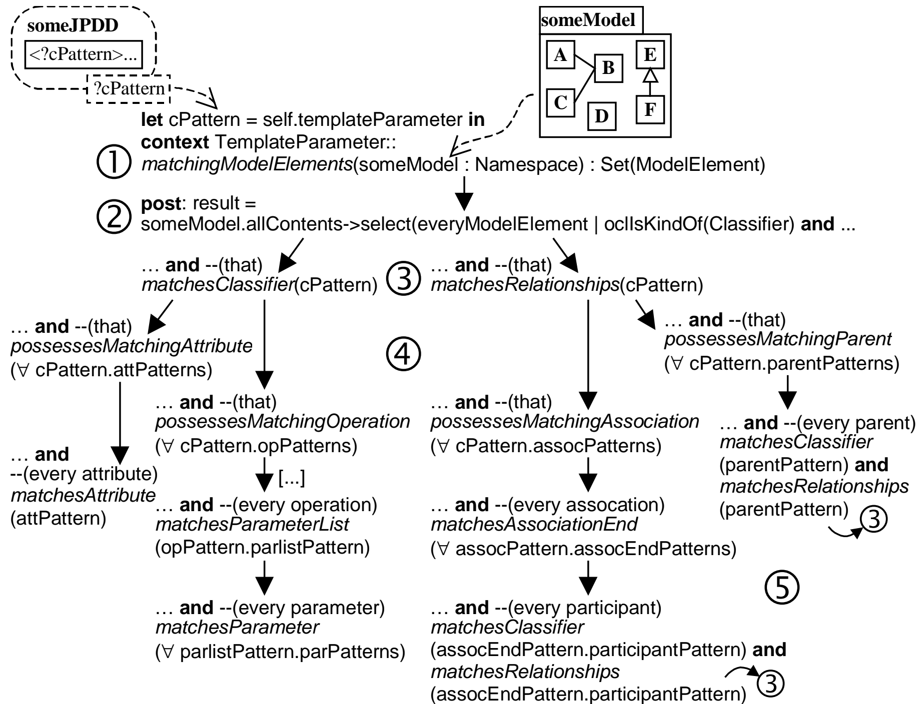


Fig. 3. Cascading evaluation of JPDDs (note that not all evaluation steps are shown)

that all selection criteria specified in the JPDD are evaluated (see Fig. 3). The meta operations take a model element pattern from the JPDD as argument, and compare its characteristics with an actual model element instance of a user model. Starting point of evaluation is a return parameter of the JPDD. For each return parameter of a JPDD, a set of matching elements in the given user model is retrieved.

Fig. 3 exemplifies how the OCL meta operations work together in order to retrieve a set of matching model elements for a classifier pattern ("?cPattern"). The selection is initiated by a special meta operation "matchingModelElements", which is defined in the context of the JPDD parameter and that returns the set of all model elements matching to that parameter (i.e. to classifier pattern "?cPattern"; see ① in Fig. 3). The meta operation takes a UML model (or any other namespace, such as packages, collaborations, etc.) as an argument. The contents of that model (or namespace) are then matched against the selection criteria outlined by the JPDD parameter (i.e. by classifier pattern "?cPattern"), one by one (see ② in Fig. 3). The model elements contained in the model are selected if their meta attributes (in this case, "isAbstract", "isLeaf", "isRoot", etc.) as well as their meta relationships (to other model elements, such as attributes, operations, associations, and generalizations, etc.) comply to the ones defined by classifier pattern "?cPattern" (cf. also section 4.1). This is checked with help of operations "matchesClassifier" and "matchesRelationships" (see ③ in Fig. 3), which in turn make use of operations "possessesMatchingAttribute", "possessesMatchingOperation", "possessesMatchingAssociation", and "possessesMatchingParent" (see ④ in Fig. 3) – and so forth. It is important to note that relationship matching also involves matching the participating classifiers (see ⑤ in Fig. 3)[6]. That way, evaluation cascades from selection criterion to selection criterion, assessing if all selection criteria in the JPDD are fulfilled.

5 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.

Fig. 4, for example, 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 "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".

Furthermore, selected classifiers must possess an indirect association (i.e., a navigable path) to a classifier (identified with "?Application") (1) matching "*"; (2) that

[6] Likewise, attribute and operation matching involves matching of their type and parameter types, respectively (not shown in Fig. 3).

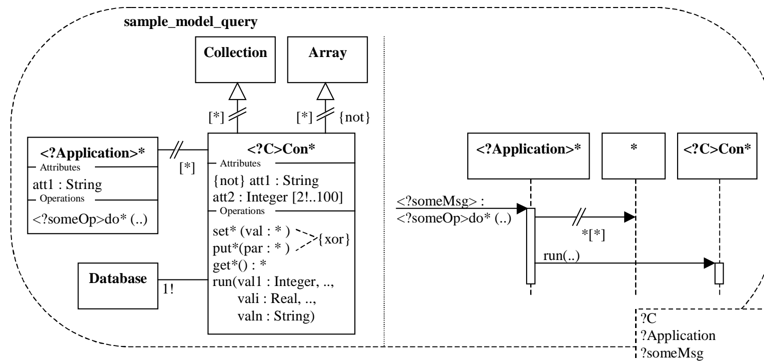


Fig. 4. A sample JPDD

has an attribute matching "att1" of type "String"; (3) and that has an operation matching "do*" (identified with "?someOp"), which takes any number of parameters. That operation must be invoked by some message (identified with "?someMsg"[7]) (3a) which in turn invokes method "run" on the former classifier (identified with "?C") – (3b) no matter when (see "iterating" double-crossed message in right part of Fig. 4) – and (3c) using arbitrary values as parameters. While the left part of Fig. 4 is matched against classifiers in class diagrams, the right part is compared to message specifications in interaction diagrams in which matching classifiers are involved.

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 "?someMsg".

6 Related Work

MDA is closely related to the research field of graph transformations [21]. In both domains, we are concerned with the specification of model (or graph) transformations and – consequently – with the specification of model (or graph) queries. From that perspective, JPDDs compare to the left-hand side (LHS) of production rules as we find them in graph rewrite systems such as PROGRES [23] or AGG [27]. JPDDs differ from LHS specified in PROGRES in their way to specify constraints on (class/object) node attributes. In PROGRES, such constraints are either specified using textual descriptions, or they are attached to the (class/object) node which they apply to by means of a hollow fat arrow. Both representations differ considerably from the class/object notation as it is known from the UML. AGG does a better job in that respect, since attributes are listed within a special attribute compartment inside the node. On the other hand, though, AGG does not provide for the specification of paths (e.g., indirect associations) between (class/object) nodes – such as PROGRES and JPDDs do. The specification means of path expressions in PROGRES go beyond those of JPDDs: PROGRES gives developers fine-grained control over the evaluation process of path expressions (by providing conditional and iterative path expressions).

[7] Note how the identifier of the message is separated from the identifier of the operation (which is being called) by means of a colon.

Furthermore, it permits the specification of optional nodes. Selection criteria specified in JPDDs, on the contrary, must be satisfied as a whole; and their evaluation process is invariable as determined by the OCL statements presented in this paper[8].

Apart from the transformation approaches originating in the field of graph transformations, there are a couple of notations around that are explicitly dedicated to the field of MDA, e.g., the QVT approach presented in [20], or MOLA [10]. The major problem with these transformation languages is that they specify model queries in terms of meta model entities. While this may be more convenient when referring to meta properties that have a standard representation in UML diagrams, it severely hinders the overall comprehension of the queries. Apart form that, JPDDs facilitate the reuse of model queries since they consider model queries as first-class entities[9] which may be involved in multiple transformations.

Considering that most submissions to OMG's QVT RFP propose to use OCL as a query language, JPDDs also relate to existing approaches for the visualization of OCL expressions in general, such as Constraint Diagrams [11] or Visual OCL [4] [12]. Constraint Diagrams 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 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. However, similar to the MDA transformation approaches mentioned above, Visual OCL does not provide for the specification of model element queries in terms of user model entities. In consequence, users are confronted with the full load of OCL complexity – in particular when specification of indirect relationships (see section 4.3) is necessary.

The idea of specifying queries in terms of user model entities we borrowed from the approach of Query-By-Example (QBE) [30], 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., names may be rendered with help of patterns, and/or multiplicity boundaries may be specified to denote minimum and maximum values rather than perfect matches).

As already mentioned above and discussed in [26], 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 [25], we demonstrate by example how JPDDs may be used to model join point selections in popular aspect-oriented programming languages. In particular, we describe how JPDDs may be used to represent *pointcuts* in AspectJ [2], *traversal strategies* in Adaptive Programming [13], or *concern mappings* in Hyper/J [29].

---

[8] Note that we abstract from evaluation problems of OCL expressions, such as the calculation of transitive closures (cf. [22]), for example. We consider these problems to be OCL-specific rather than JPDD-specific.

[9] i.e., as an autonomous entity that can exist without further reference to any other entities

7 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, as well as the graphical means to specify the queries' selection criteria. We specified OCL operations for the evaluation of such selection criteria on actual user model elements. We exemplified the use of JPDDs using a complex model query, demonstrating 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 mapping our graphical means to OCL expressions can assist developers in both AOSD and MDA when specifying and modeling selections. In particular, this allows seamless integration of our JPDDs with various submissions to the MOF QVT RFP, which are proposing to use OCL as a model query language. It is important to note, though, that JPDDs are not capable – and not intended – to represent OCL expressions in the general case. Further, it must be stated that JPDDs may specify only selections on model elements of a kind. It is not possible, for example, to collectively select UML model elements of different types into the same parameter (e.g., classes and associations, or all model elements contained in a model). Instead, a parameter must be defined for each model element type to be selected.

We think, however, that this limitation is more than outranged by the benefits of specifying 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 concentrated on a query language for the UML. We advocate for the development of further user model-based query languages in other modeling and domain-specific languages as well. That way, transformations may be specified as simple as 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/ index-tech.cgi/~checkout~/aspectj-home/doc/progguide/index.html, Jan. 2004

[3] Assmann, U. (ed.), Proc. of MDAFA 2004 (Linköping, Sweden, Jun. 2004), http://www.ida.liu.se/~henla/mdafa2004

[4] Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G., *A Visualization of OCL Using Collaborations*, in: Proc. of UML 2001 (Toronto, Canada, Oct. 2001), LNCS 2185, pp. 257-271

[5] 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)
[6] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): *Handbook on Graph Grammars*, Vol. 2: Applications, Languages, and Tools, World Scientific, River Edge, NJ, 1999
[7] Filman, R., Elrad, T., Clarke, S., Aksit, M., (eds.), *Aspect-Oriented Software Development*, Addison-Wesley, 2005
[8] 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
[9] Interactive Objects Software, Project Technology, *Revised Submission for MOF 2.0 Query / Views / Transformations RFP*, 18. Aug. 2003
[10] Kalnins, A., Barzdins, J., Celms, E., *Model Transformation Language MOLA*, in: [3], pp. 14-28
[11] Kent, S., *Constraint Diagrams: Visualizing Assertions in Object-Oriented Models*, in: Proc. of OOPSLA 1997 (Atlanta, Georgia, Oct. 1997), ACM pp. 327-341
[12] Kiesner, Chr., Taentzer, G., Winkelmann, J., *Visual OCL: A Visual Notation of the Object Constraint Language*, TR 2002/23, Technical University Berlin, 2002
[13] Lieberherr, K., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996
[14] Lieberherr, K., Lorenz, D., Mezini, M., *Programming with Aspectual Components*, TR NU-CCS-99-01, Northeastern University, 1999
[15] Masuhara, H., Kiczales, G., Dutchyn, Chr., *A Compilation and Optimization Model for Aspect-Oriented Programs*, in: Proc. of CC 2003 (Warsaw, Poland, Apr. 2003), LNCS 2622, pp. 46-60
[16] OMG, *MDA Guide Version 1.0*, OMG, 1. May 2003 (omg/2003-05-01)
[17] OMG, *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, 2002 (OMG Document ad/2002-04-10)
[18] OMG, *UML 2.0 OCL Specification*, Final Adopted Specification, 2003 (OMG Document pct/03-10-14)
[19] OMG, *Unified Modeling Language Specification*, Version 1.5, March 2003 (OMG Document: formal/03-03-01)
[20] QVT-Partners, *Revised Submission for MOF 2.0 Query / Views / Transformations RFP*, 18. August 2003 (http://qvtp.org/downloads/1.1/qvtpartners1.1.pdf)
[21] Rozenberg, G. (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 1: Foundations, World Scientific Publishing, River Edge, NJ, 1997
[22] Schürr, A., *Adding Graph Transformation Concepts to UML's Constraint Language OCL*, Electronic Notes in Theoretical Computer Science Vol. 44(4), Elsevier, 2001
[23] Schürr, A., Winter, A., Zündorf, A., *PROGRES: Language and Environment*, in: [6], pp. 487-550
[24] Stein, D., Hanenberg, St., Unland, R., *A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models*, in: [3], pp. 60-74
[25] Stein, D., Hanenberg, St., Unland, R., *Modeling Pointcuts*, Early Aspect Workshop, AOSD '04 (Lancaster, UK, Mar. 2004)
[26] Stein, D., Hanenberg, St., Unland, R., *Query Models*, in: Proc. of UML 2004 (Lisbon, Portugal, Oct. 2004), LNCS 3273, pp. 98-112
[27] Taentzer, G., Ermel, C., Rudolf, M., *The AGG Approach: Language and Environment*, in: [6], pp. 551-603
[28] 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
[29] Tarr, P., Ossher, H., *Hyper/J User and Installation Manual*, IBM Corp., 2000
[30] Zloof, M., *Query-by-Example: A Data Base Language*, IBM Systems Journal, Vol. 16(4), 1977, pp. 324-343