# On Relationships between Query Models

Dominik Stein
Stefan Hanenberg
Rainer Unland

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

**Abstract.** Queries on software artifacts play an important role in novel software development approaches, such as Aspect-Oriented Software Development and OMG's Model Driven Architecture. Keeping them separate from the modifications operating on them has proven to be beneficial with respect to their comprehensibility and their reusability. In this paper we describe what relationships can exist between such stand-alone queries. These relationships allow the combination of existing queries to form new ones, enabling developers to come up with abstractions for common selection patterns.

## 1 Introduction

Queries are an essential software artifact in modern software development techniques, such as Aspect-Oriented Software Development (AOSD) [12] and OMG's Model-Driven Architecture (MDA) [18]. Experiences gained in AOSD have shown that dealing with queries as first-class entities (i.e., as autonomous entities that can exist without further reference to any other entities) helps to understand and reason about the purpose and the effects of aspect-oriented adaptations, and improves the reusability of aspect-oriented code [9] [8]. From these experiences we anticipate that the same benefits will be brought forth to MDA if model queries are handled as autonomous artifacts. However, to actually achieve these benefits, developers require appropriate means to relate two (or more) queries to each other, so that to combine two queries to form a new one (for example).

In this paper, we present a set of binary relationships that can be established between two query models. With help of these relationships, developers can compose new queries from existing ones. Furthermore, the relationships help developers to reason on the semantic dependencies between queries; thus, providing developers with the means to identify recurring "selection patterns" that they may abstract from for future use.

The remainder of this paper is structured as follows: First of all (section 2), we give a brief overview of «Join Point Designation Diagrams» [23], the model query notation used throughout this paper, and introduce the running example for the motivation section. After that, we discuss why and what relationships between query models are needed and beneficial (section 3). Then we give a general description of the relationships that may exist between query models, and detail their semantics,

giving a supplementary example (section 4 and 5). After elucidating related work in section 6, we conclude the paper with a summary and a short discussion in section 7.

## 2 Overview to JPDDs and to the Motivating Example

«Join Point Designation Diagrams» (JPDDs) have been introduced in [23] and [22] as a notation to specify query models. The notation is based on Unified Modeling Language (UML) [20] user-model symbols, and provides means to specify lexical constraints on element names (in terms of name and signature patterns) as well as on the structural context and/or the behavioral context those elements reside in (e.g. the (non-)containment of features in classes and of classes in packages; or the (non-)existence of paths between classes (or objects) in the class (or object) hierarchy and between messages in the call graph, etc.). Fig. 1 gives a graphical overview to the most important symbols, which may be arranged in analogy to standard UML symbols as stated in the UML specification [20].

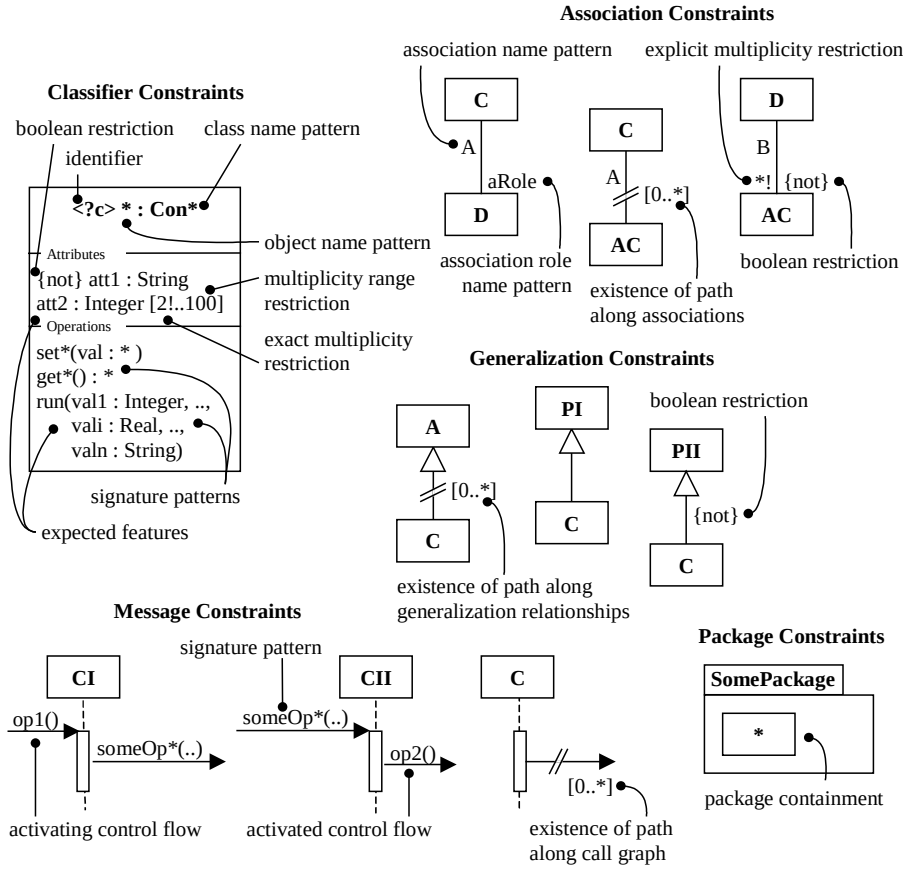


**Fig. 1.** Specifying selection constraints with «Join Point Designation Diagrams» (cf. [23])

Apart from selection constraints, JPDDs may specify those elements that are to be exposed for further adaptations (e.g. for model transformations). These elements are given identifiers, which are prepended by a question mark (`?`) and entangled in angle brackets (`< >`), and which are listed in a parameter box in the lower right corner of the JPDD.

In the following section, multiple sample JPDDs will be given and explained in further detail (see Fig. 2, Fig. 3, Fig. 5, and Fig. 6). These JPDDs are used to represent a pointcut as it is used to implement the decorator design pattern [7] with AspectJ [14]. The example is adopted from [15]; its goal is to monitor the progress of reading some input stream. To do so, it hooks onto method `read` of any `InputStream` object, and ties the monitoring dialog to a GUI component – in this case, a `JComponent` object from the `javax.swing` package.

## 3 Motivation

Relationships between query models are needed for the following reasons: First of all, they are necessary to concatenate selection constraints that are rendered by different diagrams and/or different notations. In Fig. 2, for example, the object diagram on the left outlines the structural selection constraints of a query (select all objects `<?is>` of (sub)type `InputStream`), while the interaction diagram on the right renders the behavioral selection constraints (select all method invocations `<?jp>` to operations named "read" (of objects being (sub)type of `InputStream`), taking an arbitrary number of parameters (`..`), and returning one parameter of type `int`). Usage of relationships in this way permits the representation of each selection constraint in its most appropriate manner. For example, control flow-based constraints can be represented in terms of interaction diagrams; state-based constraints can be represented using state chart notation; and data flow-based constraints can be represented with help of activity diagrams; etc. In the context of MDA, these relationships allow each part of a query to be specified in terms of the diagram and/or notation which it is eventually going to be applied to.

Apart from that, relationships enable developers to come up with abstractions over recurring selection patterns, thus facilitating their comprehension and allowing their reuse in different settings. For example consider Fig. 3, which visualizes an application of the prominent "wormhole selection pattern" [14] as it is regularly used in AOSD. The wormhole selection pattern in AOSD is used to perform adaptations on

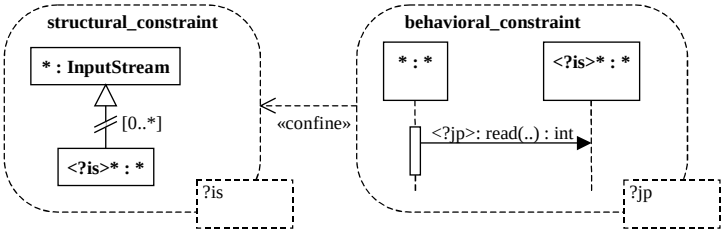


**Fig. 2.** Concatenating selection constraints of different nature (e.g. structural and behavioral selection constraints, in this case)

the behavior of programs which make use of context information from an earlier step in the execution process. The wormhole shown in Fig. 3, for example, selects all method invocations (`<?jp>*(..) : *`) that occur in the control flow (⇢) of any (other) method (`*(..) : *`) invoked on an object of (sub)type `JComponent` (as defined in package `javax.swing`); the receiver object of the latter is exposed by the query model for further adaptations, or for use by another query model. Factoring out the wormhole selection semantics into a separate query (and giving it an appropriate name) helps developers to recognize the intention as well as the effects of the query more easily[1]. Furthermore, the same wormhole can now be reused by multiple (other) queries.

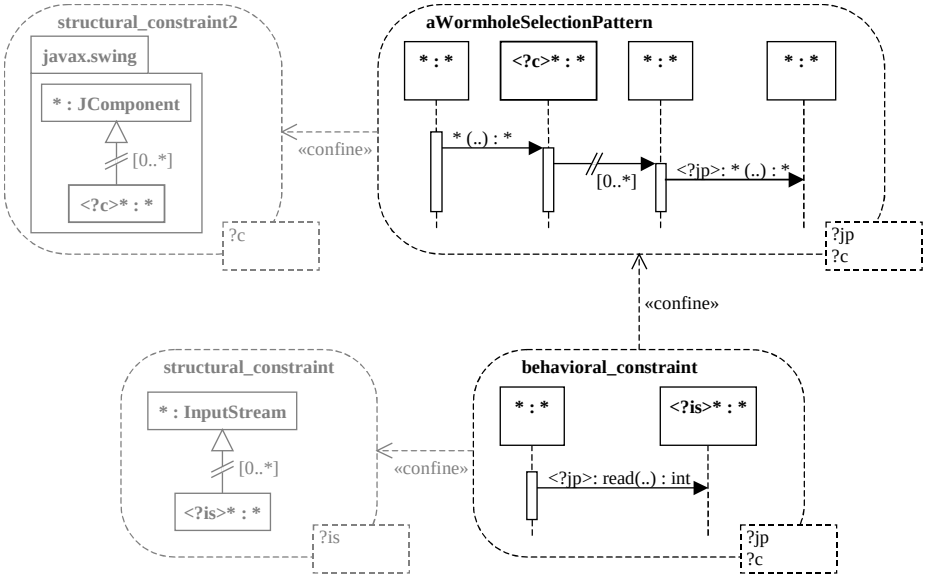


**Fig. 3.** Grouping selection constraints according to intention and effects, and finding abstractions for common selection patterns to facilitate comprehension and enhance reusability

We expect to see many selection patterns like that to arise in the MDA context. Over time, developers are going to recognize that they are using similar selection patterns over and over again. They are going to find suitable idioms for those selection patterns, and we anticipate them to ask for suitable means to abstract over those patterns and to reuse them in most different contexts. The wormhole pattern described above is one example of such recurring selection patterns. It is used to reason about the entities that have participated in (e.g. initiate) a certain task, and to select them for future adaptation (i.e. transformation). Another interesting example is concerned with the reasoning on and selection of recurring method calls; this will be considered in the subsequent chapter 4 in subsection 4.5.

[1] It may be advisable to abbreviate the (explicit) relationships to the structural selection constraints using the (implicit) short hand described in section 4.3 (see Fig. 5), in this case. That way, we have to deal with just two rather than four JPDDs.

# 4 Relationships

Having elucidated necessity and benefits of query model relationships, we now take a closer look at the nature of these relationships and define them more rigorously. We introduce three kinds of relationship: «union», «confinement», and «exclusion». These relationships combine the selection criteria of one query model with the selection criteria of another query model; however, every time in a different manner.

In the following, we refer to the including query model (e.g. JPDD_D in Fig. 4) as the "client" query model, and to the included query model (e.g. JPDD_A, JPDD_B, or JPDD_C in Fig. 4) as the "supplier" query model (in compliance to common UML parlance).

## 4.1 Mapping Rules

All combination relationships can be annotated with mapping rules indicating what identifiers from the including (client) query model are matched to what identifiers from the included (supplier) query model for unification purposes. Mapping rules are enclosed in curly braces (`{ }`) and are prepended by a lower-case Rho "ρ" in the following manner "`ρ{clientid=supplierid}`" (see Fig. 4 for an example). In case no explicit mapping rules are given, correspondence is assumed for all identifiers of same name. If explicit mapping rules are given, though, any identifier from included (supplier) query model and/or the including (client) query model *not* being mentioned in the mapping rules is considered irrelevant for the union, confinement, or exclusion, respectively.

Mappings may only be established between identifiers referring to the same type of element (i.e. classes, objects, attributes, operations, parameters, stimuli, etc.). Furthermore, mapping rules may only be specified for identifiers of the included (supplier) query model if they are exposed in the query model's parameter box. Any other identifier specified inside the included query model is not accessible from outside the query model. In contrast to this, identifiers specified inside the including (client) query model may be involved in mapping rules. This is because the inclusion is assumed to happen in the namespace of the including query model (the including query model is "calling for" the inclusion). Consequently, the identifiers of the

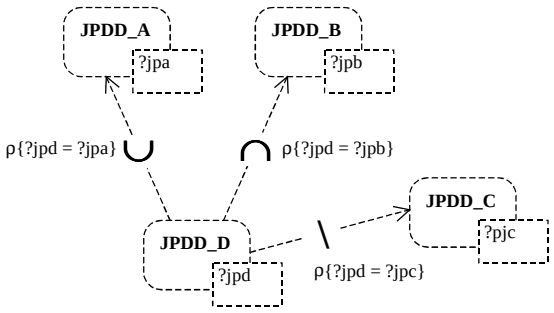


**Fig. 4.** Graphical representation of combination relationships between query models

including query model are deemed to be visible.

It remains to note that if a JPDD calls for multiple unions, confinements, and exclusions at the same time, combining is accomplished in the following order "1. all unions – 2. all confinements – 3. all exclusions". Any other combination order needs to be explicitly modeled using an execution tree.

### 4.2 Union

The first kind of relationship considers the selection criteria of the included (supplier) query model to be *alternative selection criteria* that extend those given in the including (client) query model. Consequently, the elements being designated by the including (client) query model are complemented with those being designated by the included (supplier) query model. Basically, this lead to a union of the designation results of both query models. Therefore, we use a union symbol "∪" to symbolize this kind of relationship (see Fig. 4 for an example). In union relationships, mapping rules must be specified at least for all identifiers contained in the parameter box of the including (client) query model so that no element may be left unspecified in any tuple of the final designation result.

### 4.3 Confinement

The second kind of relationship looks at the selection criteria of the included (supplier) query model as *additional restrictions* that confine the selection criteria of the including (client) query model. Confinement is accomplished using mapping rules which are attached to the relationship and that indicate what identifier from the

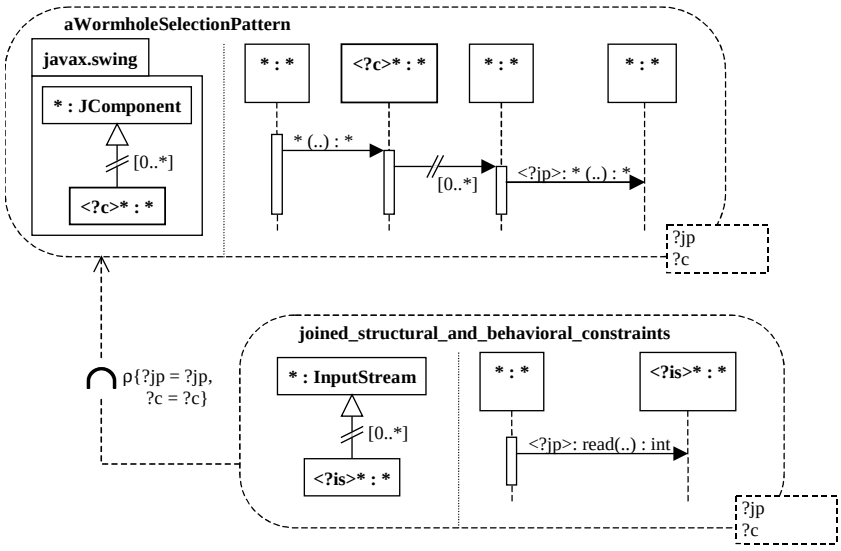


**Fig. 5.** Shorthand for concatenating selection constraints of different nature using a dotted line

including (client) query model confines what identifier from the included (supplier) query model. In consequence, the designation result of the including (client) query model is diminished to those tuples only that have a corresponding tuple in the designation result of the included (supplier) query model. We use an intersection symbol "∩" to indicate this kind of relationship (see Fig. 4 for an example).

As a short hand, confinement relationships can be abbreviated in a single query model by means of a dotted line. This is particularly useful when selection constraints of different nature (e.g. structural and behavioral selection constraints) are to be confined, and no distinct representation of either (or both) selection constraint(s) seems befitting. For example, Fig. 5 demonstrates how the structural and behavioral selection constraints from Fig. 3 (see section 3) can be merged into two rather than four query models. Further merging (e.g. in order to get one single all-inclusive query model) would be possible. Doing so, however, would obstruct the easy recognition of the application of the wormhole selection pattern in the query model.

### 4.4 Exclusion

The third kind of relationship regards the selection criteria of the included (supplier) query model to be *exclusion constraints*, which means that all tuples designated by the including (client) query model are excluded from its designation result if there is a corresponding tuple in the designation result of the included (supplier) query model. Again, correspondence between tuples is established by means of mapping rules that specify what identifier from the including (client) query model matches with what identifier from the included (supplier) query model. We use the difference symbol "\" to designate this kind of relationship (see Fig. 4 for an example).

### 4.5 Example

Having introduced the different combination relationships in detail, we now want to demonstrate their usage and their different effects with help of a small example.

Fig. 6 shows two query models, one of which is already well known from the previous examples (the bottom one). The other (top) one represents a general abstraction over recurring method calls: It selects two messages `<?jp1>` and `<?jp2>`, which both are sent to the same object `<?obj>`, and which both invoke methods with the (same) signature `<?sig>`. Message `<?jp1>` is "chained" to message `<?jp2>` with help of an indirect message symbol, which designates that message `<?jp2>` occurs in the control flow of message `<?jp1>`. Assuming that an object does not have two methods with the same signature, this query model selects all (equivalent) recurrences of a given message in its own control flow. This selection pattern is often needed to ensure that a given adaptation (or transformation) is applied only once (e.g. at the first occurrence of a method call) rather than every time the method is invoked.

The upper (supplier) query model is now included into the lower (client) query model in three different ways: ① by means of a confinement relationship, ② by means of a exclusion relationship, and ③ by means of a combination of both. The confinement relationships are annotated with mapping rules stating that message `<?jp>` from the lower (client) query model is mapped to message `<?jp1>` in the

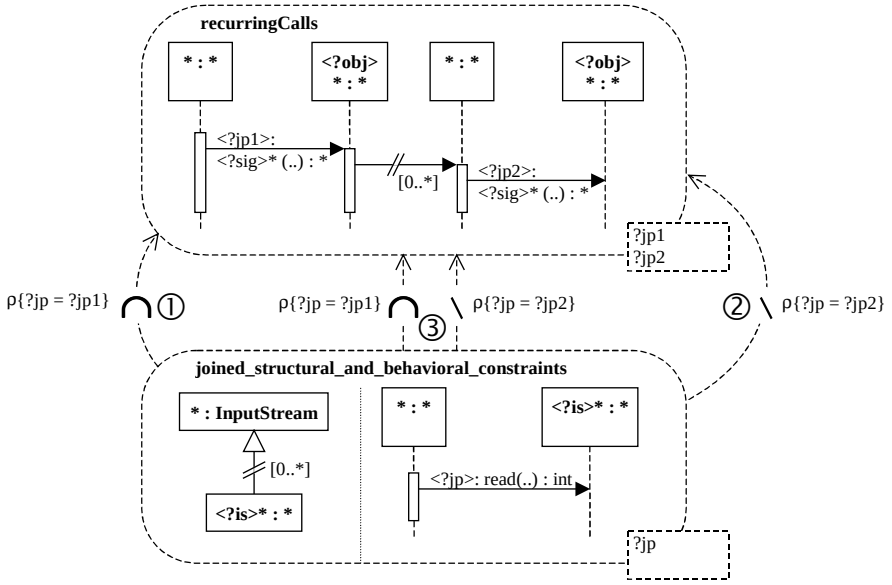


**Fig. 6.** Different usages of combination relationships

upper (supplier) query model. The exclusion relationships are annotated with mapping rules stating that message `<?jp>` from the lower (client) query model is mapped to message `<?jp2>` in the upper (supplier) query model. The effects of these combination relationships are as follows:

In the first case ①, method calls matching the lower (client) query model are only selected if there exists an recurring, i.e. an equivalent, method call in their control flow. As a result, all recursive method calls are selected, but the last. In the second case ②, method calls matching the lower (client) query model are selected only if they do not represent a recurring method call in the control flow of an equivalent method call. In consequence, only the first method call of a recurring set of equivalent method calls is selected, as well as any non-recurring method call. In the third case ③, the selection semantics of ① and ② are combined. As a result, all method calls representing the first ones in a recurring set of equivalent method calls are selected (due to the exclusion relationship); unlike to the previous case, though, selection of non-recurring method calls is not considered (due to the confinement relationship).

## 5 Semantics

In this section, the semantics of the relationships (that have been introduced in the previous section) are specified using the Object Constraint Language (OCL) 2.0 [19]. The OCL is chosen as it has been commonly proposed as the standard query language for QVT model transformations (cf. [10], [21], [6]).

**Table 1.** OCL semantics of combination relationships

```
-- identifier result sets
let QueryA_identifierResultSet(someModel : Namespace) :
      Set(TupleType(idA1 : idTypeA1, ..., idAi : idTypeAi, ..., idAk : idTypeAk, ..., idAn : idTypeAn)) = ...
let QueryB_identifierResultSet(someModel : Namespace) :
      Set(TupleType(idB1 : idTypeB1, ..., idBj : idTypeBj, ..., idBl : idTypeBl, ..., idBm : idTypeBm)) = ...
-- exposed result set
let QueryB_parameterResultSet(someModel : Namespace) :
      Set(TupleType(parB1 : parTypeB1, ..., parBr : parTypeBr, ..., parBs : parTypeBs)) =
   QueryB_identifierResultSet(someModel)->collect(tup |
         Tuple{parB1 = tup.idB1, ..., parBr = tup.idBi, ..., parBs = tup.idBm})
-- result sets to be combined
let T1(..) = QueryA_identifierResultSet(..)
let T2(..) = QueryB_parameterResultSet(..)

-- set of tuple type labels in T1 and T2 having identical names (cf. [3])
let D = p.first.allAttributes->intersection(p.second.allAttributes)
-- set of joinable tuples from T1 ⨯ T2
let X(..) = T1(..)->product(T2(..))->select(p | D->forAll(d | p.first.d = p.second.d))

-- union
T1(..)->union(T2(..))
-- confinement
T1(..)->select(q | X(..)->exists(p | p.first = q))
-- exclusion
T1(..)->reject(q | X(..)->exists(p | p.first = q))
```

### 5.1 General OCL Semantics

Using the Object Constraint Language (OCL), the combination relationship can be defined more rigorously. To do so, we assume that each query model selects a set of model elements – i.e. those model elements that are given an identifier in the query model – from a given user-model. The model elements are returned as a set of tuples, while each tuple represents a distinct combination of model elements (from the given user-model) that satisfies the selection criteria outlined in the query model. In Table 1, these sets are rendered by `QueryA_identifierResultSet` and `QueryB_identifierResultSet`. The operations take one parameter (`someModel`) that is the user-model which is to be queried.

From all model elements in the query model that are given an identifier, only a projection is exposed for further processing. These model elements (i.e. their identifiers) are listed in the query model's parameter box. Again, the model elements are returned as a set of tuples. In Table 1, these set operations are exemplified by `QueryB_parameterResultSet`. And again, the operations take one parameter (`someModel`) identifying the user-model that is to be queried.

Available to a «union», «exclusion», or «confinement» combination relationship are (a) *all* identified model elements of the including (client) query model (i.e. `QueryA_identifierResultSet` in Table 1, subsequently referred to as `T1`)[2], as well as (b) *only the exposed* model elements of the included (supplier) query model

(i.e. `QueryB_parameterResultSet` in Table 1, subsequently referred to as `T2`)[2] (cf. Mapping Rules section).

In case of a union, the set of tuples `T1` and `T2` are unified using the OCL core operation `union` (see Table 1).

In case of a confinement, all pairs of tuples from `T1` and `T2` that comply to each other in all of their attributes having the same name (referred to as `D` in Table 1) are collected from the Cartesian product of `T1` and `T2` (following the approach of [3] to compute a relational join). These pairs of tuples (referred to as `X` in Table 1)[2] are then used to diminish the set of tuples `T1` to those that are *also* part of `X`.

In case of an exclusion, initial proceeding is the same as in case of a confinement. Afterwards, though, the set of tuples `T1` is diminished to those tuples that are *not* part of `X` (see Table 1).

Note that the semantics of both confinement and exclusion relationships are defined such that they allow the combination of query models whose result sets are different in structure (i.e. that comprise different – that is, only partially overlapping – sets of attributes). Note further that in the OCL code (see Table 1) we abstract from the mapping rules. Mapping, i.e. renaming of and projection to relevant identifiers in `T1` and `T2`, cannot expressed trivially – and in a general way (!) – in the OCL (cf. [3], [4]). Hence, this can only be done manually for a particular combination relationship (as it will be demonstrated in the following).

### 5.2 A Concrete Example

After describing the general OCL semantics of the combination relationships in Table 1, we now take a look at their actual implications to a concrete example. We do so by revisiting the example from section 4.5.

First of all, we need to specify the relevant result sets of the query models, i.e. the identifier result set of the including query model as well as the parameter result set of the included query model (see above). Table 2 exemplifies how this is accomplished for query model `recurringCalls`[3] – referred to as `QueryB` in Table 2 (the specification of the result set of query model `joined_structural_and_behavioral_constraints` – referred to as `QueryA` in Table 2 – is omitted here for space reasons): The OCL code[4] starts out with collecting all possible combinations of model elements from the user-model being passed (`someModel`) that are of same type as the identifier model elements in the query model. In this example, the result set therefore contains all combinations of two stimuli model elements (`stim1` and `stim2`), one instance model element (`inst`), and one operation model element (`sig`) that can be found in the user-model. Then, in a second step, this result set is reduced to only those combinations that consist of elements complying to the selection criteria specified in the query model. This evaluation is accomplished by special *matchModelElement* operations, such as they have been specified in [23] [22]. The operations take as parameter an identifier model element from the query model

[2] Note that we abstract from parameter `someModel` of the result sets `T1`, `T2`, and `X` in the subsequent text and in Table 1.

[3] The approach has been inspired by the OCL code in [13].

[4] We used OCLE 2.02 (http://lci.cs.ubbcluj.ro/ocle) to syntax and type check the OCL code. Note that we abstract from any type cast (`oclAsType`) in the code shown.

**Table 2.** Applying the OCL semantics to the example

```
-- identifier result sets
let QueryA_identifierResultSet(someModel : Namespace) :
        Set(TupleType(stimulus : Stimulus, instance: Instance)) = [...]

let QueryB_identifierResultSet(someModel : Namespace) : Set(TupleType(stimulus1 : Stimulus,
stimulus2 : Stimulus, instance : Instance, signature : Operation)) =
-- create Cartesian product of all stimuli, instances, and signatures in someModel
someModel.allContents->select(me | me.oclIsKindOf(Stimulus))->collect(stim1 |
  someModel.allContents->select(me | me.oclIsKindOf(Stimulus))->collect(stim2 |
    someModel.allContents->select(me | me.oclIsKindOf(Instance))->collect(inst |
      someModel.allContents->select(me | me.oclIsKindOf(Operation))->collect(sig |
    Tuple {
      stimulus1 : Stimulus = stim1,
      stimulus2 : Stimulus = stim2,
      instance : Instance = inst,
      signature : Operation = sig
    } ))))
-- select those tuples that match the selection criteria specified in the query model...
->select(tup | tup.stimulus1.matchesStimulus(self.allContents->any(me | me.name="jp1"))
    and tup.stimulus2.matchesStimulus(self.allContents->any(me | me.name="jp2"))
    and tup.instance.matchesInstance(self.allContents->any(me | me.name="obj"))
    and tup.signature.matchesOperation(self.allContents->any(me | me.name="sig"))
-- ...and whose elements are related to each other as specified in the query model
    and tup.stimulus1.receiver = tup.instance
    and tup.stimulus2.receiver = tup.instance
    and tup.stimulus1.dispatchAction.method.specification = tup.signature
    and tup.stimulus2.dispatchAction.method.specification = tup.signature
    and tup.stimulus2.allActivators()->includes(tup.stimulus1))

-- exposed result set
let QueryB_parameterResultSet(someModel : Namespace) :
        Set(TupleType(stimulus1 : Stimulus, stimulus2 : Stimulus)) =
-- project identifier result set to those elements being exposed by the query model
   QueryB_identifierResultSet(someModel)->collect(tup |
        Tuple{ stimulus1 = tup.stimulus1, stimulus2 = tup.stimulus2 })

-- result sets to be combined
let T1(someModel : Namespace) = QueryA_identifierResultSet(someModel)
let T2(someModel : Namespace) = QueryB_parameterResultSet(someModel)
-- set of joinable tuples from T1 × T2 (according to mapping rules outlined in Fig. 6)
let X1(someModel : Namespace) = T1(someModel)->product(T2(someModel))
  ->select(p | p.first.stimulus = p.second.stimulus1)
let X2(someModel : Namespace) = T1(someModel)->product(T2(someModel))
  ->select(p | p.first.stimulus = p.second.stimulus2)

-- confinement (①)
let ResultSet_1(someModel : Namespace) = T1(someModel)
  ->select(q | X1(someModel)->exists(p | p.first = q))
-- exclusion (②)
let ResultSet_2(someModel : Namespace) = T1(someModel)
  ->reject(q | X2(someModel)->exists(p | p.first = q))
-- simultaneous confinement and exclusion (③)
let ResultSet_3(someModel : Namespace) = T1(someModel)
  ->select(q | X1(someModel)->exists(p | p.first = q))
  ->reject(q | X2(someModel)->exists(p | p.first = q))
```

(i.e. `jp1`, `jp2`, `obj`, and `sig`, respectively[5]); the parameter is taken as a selection pattern to which the user-model elements are compared. At last, the OCL code assures that the elements in the remaining combinations actually relate to each other as specified in the query model (the evaluation is based on the (meta-)associations between the model elements as specified in the UML meta-model [20]).

Once we have retrieved the identifier result set of the (included) query model (`QueryB_identifierResultSet`), we need to project that result set such that its tuples only consists of model elements that are actually exposed by the query model. Therefore, a parameter result set (`QueryB_parameterResultSet`) is created from the identifier result set, whose tuples only consist of model elements that are (designated by the identifier model elements) listed in the parameter box of the query model (i.e. the stimuli `stimulus1` and `stimulus2`, in this case).

Now we are ready to perform the confinement and the exclusion, respectively. To do so, two sets `X1` and `X2` of "joinable tuples" are created – one for each mapping specification given in the example (see Fig. 6). The sets comprise all pairs of tuples from `QueryA_identifierResultSet` (referred to as `T1`) and `QueryB_parameterResultSet` (referred to as `T2`) that are referring to the same model elements in their attributes being mapped (i.e. in `stimulus` and `stimulus1`, as well as in `stimulus` and `stimulus2`, respectively). The sets are then used to filter the identifier result set of the including query model (`QueryA_identifierResultSet`, referred to as `T1`) such that it consists only of those tuples ① being part of `X1` (resulting into a confinement), ② being *not* part of `X2` (resulting into an exclusion), and finally, ③ being part of `X1`, but not part of `X2` (resulting into a combined confinement and exclusion).

## 6 Related Work

JPDDs as we have used them in this paper relate to other approaches that provide visualization means for model queries in the MDA domain. Examples of such approaches are basically all proposals to specify model transformations, such as MOLA [11], BOTL [17], or the QVT-Merge submission [21], etc. However, none of these approaches provide means to reason about model queries in isolation. In consequence, no abstraction means are provided to segregate recurring selection patterns, and no relationships are specified to re-use such recurring selection patterns in different discrete application domain-specific selection queries.

Instead, OCL 2.0 is commonly suggested to serve as a (purely textual) query language – e.g. for OMG's transformation language QVT (cf. [10], [21], [6]). Therefore, it is interesting to relate our combination relationships to OCL's proper capabilities to express and calculate combination of sets of tuples. Several studies [16] [1] [3] have been conducted in this regard to investigate the expressiveness of OCL with respect to relational algebra [5], a well-known and well-founded approach to specify operations on (e.g. combinations of) sets of tuples, originating from the

[5] It is assumed that the OCL code is specified in the context of the query model; thus, `self.allContents` refers to all model elements contained in `QueryB`. See [22] for further information on how identifiers and name patterns are stored in the meta-representation of model elements.

database domain. In the latest study [3], it has been stated that since the introduction of tuple types and product types as primitive operators in OCL 2.0, OCL became much more expressive. But still, projection and renaming of tuple attributes needs to be done on a per-case basis and cannot be expressed in a general way. When compared to the mentioned work, it is important to note that our work is not focused on providing a (relational) complete set of graphical operators that could serve as a basis for arbitrary computations over sets of tuples (of model elements). Instead, the goal was to find appropriate abstractions for common combinations of query models and daily needs in query design.

Nonetheless, the identified combination relationships can be compared to relational operations, and would correspond to the following expressions: Assuming that sets of tuples `T1` and `T2` (see Table 1) designate relations, a union relationship equates to the following relational operation: $T1 \cup T2$. A confinement relationship could be expressed as a left semi-join between `T1` and `T2`: $T1 \ltimes T2$. And finally, an exclusion relationship equates to $T1 - (T1 \ltimes T2)$ . A mapping rule serves two purposes: First of all, it projects the second relation `T2` to those attributes that are relevant for the respective relational operation: $\pi_{attT2.1,\ attT2.2,\ ...,\ attT2.n}\ T2$. Furthermore, it renames the projected attributes so that their labels comply to the attribute labels of the first relation `T1` (and the relational operation can actually be performed): $\rho_{attT1.1 \leftarrow attT2.1,\ attT1.2 \leftarrow attT2.2,\ ...,\ attT1.n \leftarrow attT2.n}$

Apart from that, the presented relationships also compare to (and have been influenced by) the combination operators for join point queries (so-called "pointcuts") `&&`, `||`, and `!` in AspectJ [14], the most popular aspect-oriented programming language. If two join point queries are combined by means of an AND-operator `&&`, the result set contains all join points that are picked out by *both* queries – which corresponds to the selection semantics of the confinement relationship. If two join point queries are combined by means of an OR-operator `||`, the result set contains all join points that are picked out by *either* query – which equates to the union relationship. If a join point query is prepended by a NOT-operator `!`, the join points designated by that query are dismissed from the selection result. This complies to the exclusion relationship (that is, to be more exact, the exclusion relationship equates to a combination of an AND-operator and a NOT-operator in AspectJ, i.e. "`&& !`").

## 7 Conclusion

In this paper we elucidated the need of having suitable means to combine query models. We presented three kinds of combination relationships – «union», «confinement», and «exclusion» – and detailed their semantics informally and with help of OCL 2.0 expressions. We demonstrated the usage and the benefits of these relationships with help of two common selection patterns.

By means of the presented combination relationships, developers are able to abstract from recurring selection patterns and to reuse them in different application contexts. Furthermore, they are able to relate queries pertaining to different modeling notations or diagrams (e.g. to both structural and behavioral model specifications); thus, they are able to specify context-sensitive model queries. Of course, in order to efficiently do so in model transformations, the meta models of the involved modeling notations must share some common abstractions.

Provided with the combination relationships presented in this paper, abstractions of recurring selection patterns can be assembled in public libraries (just like abstractions of common transformations can be assembled in public transformation libraries), ready for reuse by the model-driven developer. That way, developers are freed from elaborating on recurring selection semantics again and again (e.g. how to handle recursive method calls; see section 4.5), each time they are dealing with a new problem.

It remains to mention that our work is not concerned with the actual evaluation of query combinations. That is, we do not consider how the resulting sets of tuples (of model elements) are actually extracted from a given user model. In particular, we currently abstract from problems that may occur due to recursive or circular query combinations. Such combinations may lead to infinite loops during query evaluation, or may introduce irresolvable dependencies between the involved query specifications. It is an interesting field for future research to investigate if such problems may be detected prior to query execution time and how they could be possibly resolved automatically based on static query analysis.

## References


[1] Akehurst, D., Bordbar, B., *On Querying UML Data Models with OCL*, in: Proc. of UML'01, Toronto, Canada, LNCS 2185, Springer, October 2001, pp. 91-103

[2] Aßmann, U., Aksit, M., Rensink, A., *Model Driven Architecture* (European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004; Revised Selected Papers), LNCS 3599, Springer, June 2004

[3] Balsters, H., *Modelling Database Views with Derived Classes in the UML/OCL-Framework*, in: Proc. of UML'03, San Francisco, CA, LNCS 2863, Springer, October 2003, pp. 295-309

[4] Blaha, M., Premerlani, W., *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, Englewood Cliffs, NJ, 1998

[5] Codd, E.F., *Relational Completeness of Data Base Sublanguages*, in: Rustin, R. (ed.), Courant Computer Science Symposia, Vol. 6: Database Systems, Prentice Hall, Englewood Cliffs, NJ, 1972, pp. 65-98

[6] Compuware Corporation, SUN Microsystems, *2nd revised submission for MOF 2.0 Query/Views/Transformations RFP*, 11. October 2004 (OMG Document ad/2004-10-03)

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, 1995

[8] Hanenberg, S., Schmidmeier, A., *Idioms for Building Software Frameworks in AspectJ*, 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Boston, MA, March 17, 2003

[9] Hannemann, J., Kiczales, G., *Design pattern implementation in Java and AspectJ*, in: Proc. of OOPSLA'02, Seattle, Washington, SIGPLAN Notices 37(11), ACM, November 2002, pp. 161-173

[10] Interactive Objects Software, Project Technology, *2nd revised Submission for MOF 2.0 Query/Views/Transformations RFP*, 12. January 2004 (OMG Document ad/2004-01-14.pdf)

[11] Kalnins, A., Barzdins, J., Celms, E., *Model Transformation Language MOLA*, in: [2], pp. 62-76

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Chr., Lopes, C.V., Loingtier, J-M., Irwin, J.: *Aspect-Oriented Programming*, in: Proc. of ECOOP '97, Jyväskylä, Finland,

LNCS 1241, Springer, June 1997, pp. 220-242
[13] Kožusznik, J., *Dotazování, pohledy a transformace v MDA*, in: Proc. of Objekty 2003, Ostrava, Czech Republic, ISBN 80-248-0274-0, VŠB-Technical University of Ostrava, November 2003, pp. 120-128
[14] Laddad, R., *AspecJ in Action: Practical Aspect-Oriented Programming*, Manning Publications, Greenwich, 2003
[15] Lesiecki, N., *Enhance design patterns with AspectJ*, Part 1, IBM DeveloperWorks > Java Technology > AOP@Work (http://www-128.ibm.com/developerworks/java/library/j-aopwork5)
[16] Mandel, L., Cengarle, M., *On the Expressive Power of OCL*, in: Proc. of FM'99, Toulouse, France, LNCS 1708, Springer, September 1999, pp. 854-874
[17] Marschall, F., Braun, P., *Model Transformations for the MDA with BOTL*, in: Workshop Proc. of MDAFA 2003, Enschede, The Netherlands, CTIT Technical Report TR-CTIT-03-27, University of Twente, June 2003, pp. 25-36
[18] OMG, *MDA Guide Version 1.0*, 2003 (OMG Document omg/2003-05-01)
[19] OMG, *OCL 2.0 Final Adopted Specification*, 2003 (OMG Document ptc/03-10-14)
[20] OMG, *Unified Modeling Language Specification*, Version 1.5, 2003 (OMG Document formal/03-03-01)
[21] QVT-Merge Group, *Revised submission for MOF 2.0 Query / Views / Transformations RFP*, 2. March 2005 (OMG Document ad/2005-03-02)
[22] Stein, D., Hanenberg, S., Unland, R., *A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models*, in: [2], pp. 77-92
[23] Stein, D., Hanenberg, S., Unland, R., *Query Models*, in: Proc. of UML'04, Lisbon, Portugal, LNCS 3273, Springer, October 2004, pp. 98-112