A logical model query interface*

Harald Störrle
Institute of Informatics
University of Munich
Munich, Germany

Abstract

This paper presents the Logical Query Facility (LQF), a high level programming interface to query UML models. LQF is a Prolog library built on top of the Model Manipulation Toolkit (MoMaT, cf. [8]). It provides a set of versatile predicates that reflects the notions modelers use when reasoning about their models which makes it easy to formulate queries in a natural way. In order to demonstrate the capabilities of LQF in comparison to OCL, we have implemented it as a plug in to the popular MagicDraw UML CASE tool [3], and evaluated LQF with a benchmark suite of frequent model queries.

1 Introduction

1.1 Motivation

Over the last decade, model based and model driven development have turned into mainstream approaches in large scale industrial software engineering projects.[1] Visual languages like UML, EPCs, BPMN, DSLs, etc. play a more and more prominent role in such settings, and as a consequence, models have grown much larger (see cf. [9] and Fig. 1).

Another consequence is that more and more people are involved directly in modeling activities. Today, most modelers in large scale projects are not software engineers, but domain experts. In fact, the integration of domain experts is a crucial success factor in medium to large scale software development efforts. Thus, providing an interactive query facility for modelers is dearly needed in many if not all modeling projects.

From experience we know, however, that many modelers are challenged by the complexity of modeling languages already. Often, they can't (or won't) cope with yet another, complicated language for queries (such as OCL or QVT), let alone query APIs. But the query facilities provided by many tools (full-text search and predefined queries)

*Thanks to Alexander Knapp for generously sharing his OCL expertise.

[1] Since 2004, the author has participated in two such projects as lead methodologist and modeling coach.

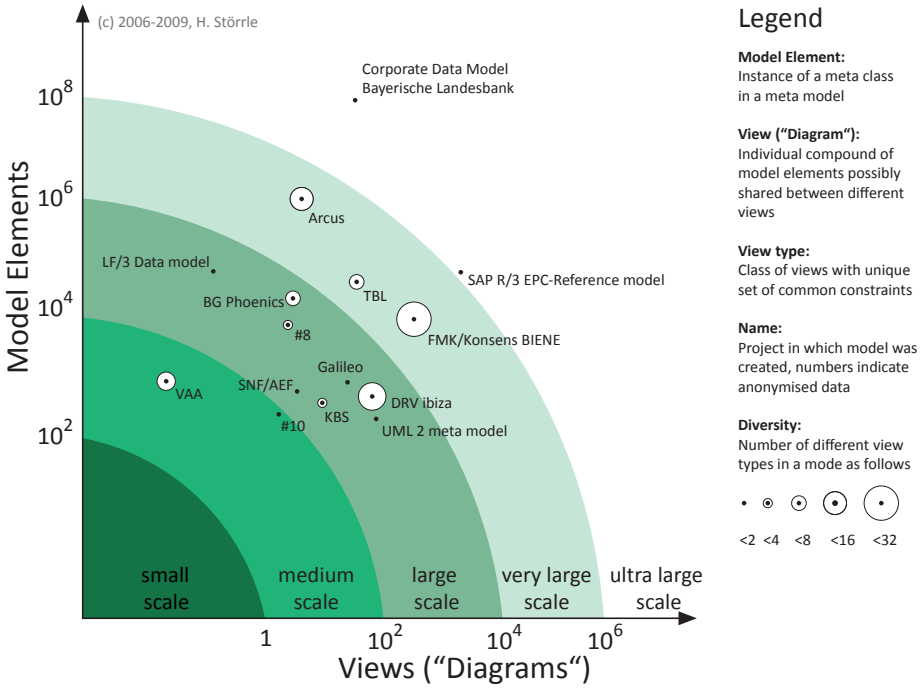


Figure 1: Real life models may become very large (cf. [9]).

are not expressive and flexible enough. This paper reports on our attempt to provide a better query facility which is expressive enough for all queries yet much easier to use.

1.2 Related work

Currently, there are four distinct types of UML model query facility: (1) tool specific queries, (2) application programming interfaces, (3) visual query facilities, and (4) abstract query facilities like OCL.

tool specific facilities Full text search and predefined queries are easy to use, but very limited in terms of expressiveness. For instance, a text search cannot find model structures or patterns, and sets of predefined queries cannot be easily extended. In our experience from industrial modeling projects, this type of query facility is too limited for many tasks.

APIs In contrast, an Application Programming Interface (API) offers complete control over a model and unrestricted expressiveness for querying. However, most CASE tools' APIs are very complex and are built on mainstream programming languages like Java (MagicDraw), C# (Enterprise Architect), or Visual Basic (Rational Rose). So, substantial commitment and effort is required before an end user can use such an API.

visual queries There are also visual query facilities like Query Models [6, 7] and VMQL [10]. Unfortunately, the Query Models approach has never been implemented; VMQL *has* been implemented, but there are no evaluations of its practical value yet.

logic based queries Today, the Object Constraint Language (OCL, [4]) is the de-facto standard language for complex annotations of UML models (such as consistency conditions, pre- and post-conditions). So, one could say that OCL is the "gold standard" of logic based UML query languages. However, OCL lacks several features essential for querying.

We will analyze OCL's deficiencies for querying in detail in the next section as the starting point for our own work.

1.3 Approach

As we have said before, OCL is the de-facto standard for expressing complex properties of UML models but it suffers from several shortcomings as a language for end user model querying. Analyzing these deficiencies will help us define a better query facility.

Pattern Matching OCL provides no pattern matching facilities, e. g., name matching using wild cards. For most users concerned with ad hoc queries, the full power of regular expressions are probably not required. Most of the time, it will be sufficient to allow `*` and `?` in names to match any number of characters and a single character, respectively. Defining such a function is very hard with OCL.

Conceptual Abstraction When expressing queries in OCL, the modeler needs to navigate through concepts defined in the UML meta model which requires substantial expertise. Also, since the concepts used in the UML meta model have little to do with the notions a modeler uses when reasoning about models, a conceptual mismatch arises that interferes with using OCL.

Type System OCL is strongly typed, which many people perceive as disruptive in interactive tasks (such as ad-hoc querying). Moreover, the OCL type system lacks type variables, providing only a limited form of subtyping polymorphism, but no parametric polymorphism (cf. [1]).

Notation Size & Complexity OCL has a rich and complex syntax with more than 50 keywords and standard library functions, plus all the usual operators and constants for arithmetics, boolean logic etc., which implies a considerable learning effort for any user.

Summing up, OCL lacks essential query facilities like pattern matching, it fails to provide a useful abstraction layer on top of the meta model, and its syntax and type system are not very helpful either. All in all, its complexity renders it effectively unusable for the average modeler. As an attempt to overcome these limitations, we have designed the Logical Query Facility (LQF) advancing our own prior work (see [8]).

We pursue three goals with LQF. Firstly, LQF should be *universal*, that is, it should allow all types of queries, including full text search. Secondly, LQF should be *expressive*, i. e., as many queries as possible should be expressible in LQF, including those predefined in typical CASE tools. Thirdly, LQF should be *simple*, that is, we aim to make LQF much simpler to use than OCL or an API. To this end, LQF provides a set of predicates that state important model properties in terms modelers are accustomed to rather than in terms of the underlying meta model (as OCL does).

2 The Logical Query Facility

In this section we will describe the LQF, the MoMaT framework on which it builds, and the $\mathsf{MQ}_{\mathsf{Logic}}$ tool implementing LQF.

2.1 The MoMaT framework

The Model Manipulation Toolkit (MoMaT) is a framework for processing models such as UML models using Prolog. It has been described eg. in [8], and we summarise it here only so that this paper is more self contained.

MoMaT represents model elements as individual facts and models as sets of facts, i. e., a Prolog Database. Consider the example shown in Fig. 3. It shows a simple UML class diagram (top), and its representation as a Prolog module with a set of facts, one for each model element. The blue italic numbers serve as identifiers of model elements. These identifiers are completely arbitrary; any string could be used, or, in fact, the original object identifiers provided for model elements by most contemporary

modeling tools. Every fact describing one model element is described using the `me/2` predicate. Fig. 2 shows how the arguments of theis predicate are to be interpreted.

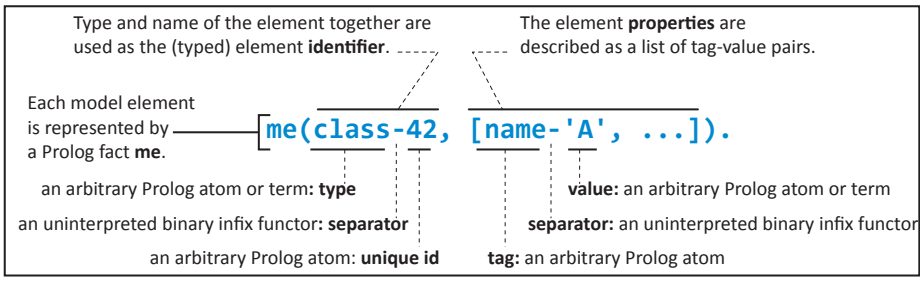


Figure 2: Schematic Prolog representation of a single model element.

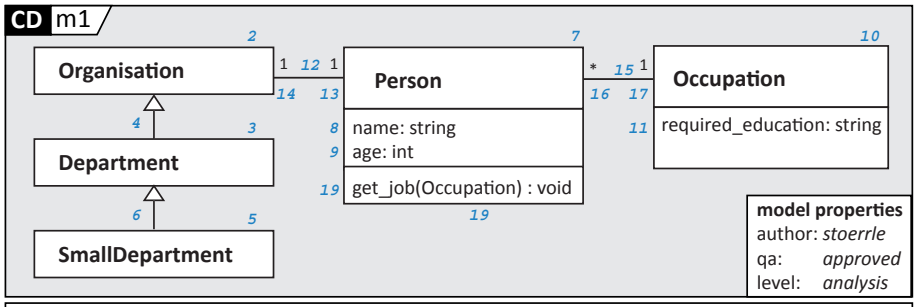
The Prolog representation of models is created automatically by the MX tool (cf. [2]). MX is a standalone tool that processes the files used to store models. Since MX is highly configurable, it can process a very wide range of file formats, that is different versions of UML, XMI, MOF/EMF/ECORE, and different tool manufacturers interpretations of them, but also BPMN/BPEL, and ADL. So far, MX has been used with MagicDraw, EnterpriseArchitect, VisualParadigm, and Adonis. Extending this range is usually a matter of hours. Thus, MX (and MoMat, and LQF) may process a wide variety of models today, and, with a little extra effort, potentially any modeling language.

The Prolog representation shown in Fig. 3 is identical for every source language or file format. The first argument contains the model element type (its meta class, in UML terminology), and and identifier. Both are arbitrary Prolog atoms. The second argument of `me/2` is an unordered list of tag-value pairs, both of which may be arbitrary Prolog expressions, including complex terms, lists, and so on. Note that this representation is purely syntactic: a new modeling language with a different set of concepts (meta classes) is treated just the same and does not require any changes to MoMaT.

This representation alone allows to manipulate models using arbitrary Prolog predicates. For instance, querying for all attributes with type `string` in model `m1` from Fig. 3, we would have to load the output of MX into a Prolog system ("consult the file" in Prolog terminology), and issue a small query at the command line prompt:

```
?- consult('m1').
?- m1:me(property-ID, Attributes),
   memberchk(type-string, Attributes).
```

The query returns all identifiers of model elements of type property (the UML jargon for attribute) in the scope of module `m1`, that have the pair `type-string` among their attributes. In this case, the answer is the set of identifiers 1 and 7. To understand this type of expression, a user needs to know a number of Prolog conventions and syntactic elements.




```
:-module(m1, [me/2,model/2]).
  model(cd-1,           [type-class_diagram, name-m1,language-'UML 2',
                         version-'2.2']).
  me(model-1,           [name-'Data',ownedMember-ids([2,3,5,7,10,12,15]),
                         level-analysis, author-stoerrle, qa-approved]).
  me(class-2,           [name-'Organisation',isAbstract-true,
                         ownedMember-ids([14])]).
  me(class-3,           [name-'Department',ownedMember-ids([4])]).
  me(generalization-4,  [to-id(2),from-id(3),isSubstitutable-true]).
  me(class-5,           [name-'SmallDepartment',ownedMember-ids([6])]).
  me(generalization-6,  [to-id(3),from-id(5),isSubstitutable-true]).
  me(class-7,           [name-'Person',ownedMember-ids([8,9,13,16,19])]).
  me(property-8,        [name-name,visibility-private,
                         multiplicity-interval(0,*),type-string]).
  me(property-9,        [name-age,visibility-private,
                         multiplicity-interval(0,*),type-int]).
  me(class-10,          [name-'Occupation',ownedMember-ids([11,17])]).
  me(property-11,       [name-required_education,visibility-private,
                         multiplicity-interval(0,*),type-string]).
  me(association-12,    [memberEnd-ids([13,14])]).
  me(property-13,       [type-id(2),visibility-private,association-id(12),
                         multiplicity-interval(1,1),type-id(2)]).
  me(property-14,       [type-id(7),visibility-private,association-id(12),
                         multiplicity-interval(1,1),type-id(7)]).
  me(association-15,    [memberEnd-ids([17,16])]).
  me(property-16,       [type-id(10),visibility-private,association-id(15),
                         multiplicity-interval(0,*)]).
  me(property-17,       [type-id(7),visibility-private,association-id(15),
                         multiplicity-interval(1,1)]).
  me(operation-18,      [name-'get_job',ownedMember-ids([19]),
                         visibility-'public']).
  me(parameter-19,      [visibility-'public',type-id(10)]).
```

Figure 3: A simple UML model (top), and its representation in Prolog (bottom). The blue italic numbers serve as identifiers of model elements; for easier reference we have added them in the UML model, close to the respective element. Many of the model element's properties are default values (e. g., visibilities, multiplicities and `isSubstitutable`). The layout of the Prolog representation has been improved for readability. The notation `:-module` is the syntax SWI Prolog uses to define a module.

Modules A module in the Prolog system we use is a flat name space. Elements in this name space may be accessed by prefixing a predicate by the module name and a colon.

Facts A Prolog fact is an identifier followed by a bracketed sequence of arguments which are separated by commas. A fact is terminated by a full stop. The predicate- is defined as an infix operator, so `type-string` really is identical to `-(type, string).`

Variables In Prolog, all identifiers starting with a capital letter are logical variables. The underscore character denotes the anonymous variable.

Queries Stating a fact prompts Prolog to try and find a variable binging that makes this fact true relative to the currently known facts.

Lists Lists are enclosed in square braces, the list elements are separated by commas.

While this type of access brings the full power of Prolog to UML models, it requires considerable knowledge both of Prolog and the respective modeling language. MoMaT provides an abstraction layer that makes it easier to deal with complex operations on models of different kinds. However, since MoMaT provides the full spectrum of operations, it has proved to be too complex for just querying, and definitely too difficult to learn for the casual user. LQF, on the other hand, provides a restricted and specialised set of operators that makes this possible.

2.2 The LQF predicates

The Logical Query Facility (LQF) provides a small set of powerful and generic predicates on top of MoMaT. The LQF predicates capture the properties and relationships of model elements in the terms modelers are accustomed to rather than in terms of the underlying meta model (as OCL does). See Table 1 for a complete reference of the LQF-predicates currently defined. Note that most arguments may be either unbound, bound to items, or bound to sets of items. Predicates from `associated` on also have an additional optional last parameter indicating the number of steps (default is 1).

As a first example, consider again the query we defined in the previous section to determine the string-typed attributes in model m1. Using LQF, this query may be rewritten as

```
exists(property, ID, [type-string])
```

Now consider a more complex example. Assume, we want to check that two model elements E_1 and E_2 are associated. Using OCL this requires us to navigate from E_1 and E_2 to their respective `ownedMembers`, and find an association containing them both. In order to *find* the opposite end of an association partner, a different OCL statement is needed, and in order to get pairs of associated model elements, *yet another* OCL statement is needed.

In contrast, the LQF predicate `associated/2` may be instantiated in all three ways, i. e., with both E_1 and E_2 bound ("check association between them"), with just

one of them bound ("get the other end of an association"), or none of them bound ("find associated pairs of elements"). Additionally, the LQF predicate provides an option to check whether the association is indirect, that is, via a given number of steps (including "any"). Also, it is defined on pairs of elements as well as on sets of elements (for n-ary associations). Finally, it may be used for all kinds of model elements, whereas OCL would require one definition for every pair of element types.

Similar options and usage modes are provided by all other LQF predicates. The predicates concerned with relationships also have an additional optional last parameter indicating the length of the path of the relationship type ("steps"), ranging from 1 (default) to * (any number of steps). For instance, `is_a(A,B)` asserts that there is a generalization relationship between model elements `A` and `B`, while `is_a(A, B, 3)` asserts that there is a chain of at most three generalizations between `A` and `B`. Similarly, `is_a(A, B, *)` asserts that there is a chain of generalizations between `A` and `B`, and it may be of arbitrary length.

2.3 The $\mathsf{MQ}_{\mathsf{Logic}}$ Tool

In order to explore our approach further, we have implemented $\mathsf{MQ}_{\mathsf{Logic}}$, a prototype plug in to the popular MagicDraw UML CASE tool (cf. [3]). It uses the MX model converter [2], some of the infrastructure of the MQ model query tool [11], SWI-Prolog and the JPL Java-Prolog-Bridge library (see `www.swi-prolog.org`). The LQF predicates are implemented as a set of SWI-Prolog modules. Fig. 4 shows an overview of the structure of $\mathsf{MQ}_{\mathsf{Logic}}$. See Fig. 5 for a screenshot of $\mathsf{MQ}_{\mathsf{Logic}}$.

This chart is annotated with the steps involved in creating and executing a query. We will start with the steps marked by white circles.

① First, the user creates or obtains a model and starts the $\mathsf{MQ}_{\mathsf{Logic}}$ system from within MagicDraw.

② The model is exported by MagicDraw and stored as a XMI-file in the local file system.

③ Using the MX tool (cf. [2]), the XMI file is converted into a set of prolog facts.

Steps ② and ③ are performed completely automatically. Note that ③ modifies only the format, but leaves the semantic contents of the model completely unchanged. After changes to the model the user must refresh its Prolog representation which repeats steps ② and ③.

In order to execute a LQF query, the following steps must be executed (marked with numbered black circles in Fig. 4).

❶ The user inputs an ordinary Prolog query as plain text to the $\mathsf{MQ}_{\mathsf{Logic}}$ input window, using the predicates defined by LQF (see Table 1).

❷ The query is sent as-is to the Prolog engine via the JPL Java-to-Prolog bridge.

❸ The query is executed as-is, dynamically using LQF predicates.

Table 1: The predicates defined by LQF.

`exists(TYPE, ID, PROPS)`

There is an element of type `TYPE` identified by `ID` with the properties listed in `PROPS` as `Key-Value` pairs. Note that at least one of the `Key` or the `Value` must be instantiated.

`sub_type_of(SUPERTYPE, SUBTYPE)`

In the underlying modeling language, `SUBTYPE` is more special than `SUPERTYPE`.

`attribute_of(TYPE, ID, Value)`

In the underlying modeling language, `SUBTYPE` is more special than `SUPERTYPE`.

`name(ID, NAME)`

The element identified by `ID` has the qualified name `NAME`.

`match(VAL, PATTERN)`

Value `VAL` matches the pattern `PATTERN` (both parameters must be instantiated).

`distinct(IDS)` All elements in `IDS` are distinct.

`occurs_in(ID, D)`

The element identified by `ID` occurs in the diagram identified by `D`.

`associated(ID-SET)`

All elements in `ID-SET` are part of an nary association, where $n \geq |$`ID-SET`$|$.

`rel(ID, ID', RTYPE)`

There is a relationship of type `RTYPE` between the element(s) identified by `ID1` and the element(s) identified by `ID2`. If both `ID1` and `ID2` are sets, all pairs of identifiers must be in the relationship.

`is_a(ID, ID')`

There is a generalization relationship between the element(s) identified by `ID` and the element(s) identified by `ID'`. If both `ID` and `ID'` are sets, all pairs of identifiers must be in the relationship.

`depends(ID, ID')`

There is a dependency relationship between the element(s) identified by `ID` and the element(s) identified by `ID'`. If both `ID` and `ID'` are sets, all pairs of identifiers must be in the relationship.

`connected(ID, ID')`

There is any kind of connection between the element(s) identified by `ID` and the element(s) identified by `ID'`. If both `ID` and `ID'` are sets, all pairs of identifiers must be connected.

`precedes(ID, ID')`

There is a sequential ordering relationship between the element(s) identified by `ID` and the element(s) identified by `ID'` (e. g., before/after, incoming/outgoing etc.). If both `ID` and `ID'` are sets, all pairs of identifiers must be in the relationship.

`calls(ID, ID')`

There is a calling relationship between the element(s) identified by `ID` and the element(s) identified by `ID'`. If both `ID` and `ID'` are sets, all pairs of identifiers must be in the relationship.

`contains(ID, ID')`

There is a whole-part relationship between the element(s) identified by `ID` and the element(s) identified by `ID'` (e. g., class/attributes, package/members, state/substate etc.). If both `ID` and `ID'` are sets, all pairs of identifiers must be in the relationship.

❹ The results are presented back to the user. Currently, this feedback is restricted to simple values such as (qualified) names of model elements.

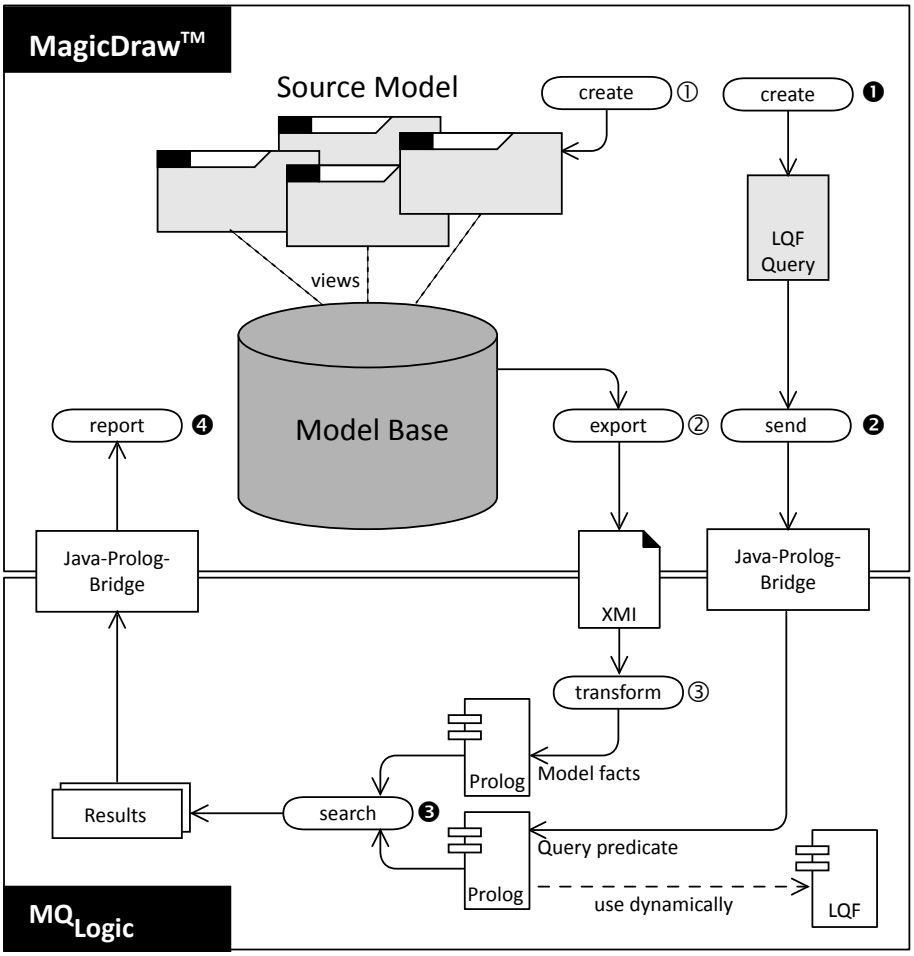


Figure 4: Overview of our prototype implementation of $\mathsf{MQ}_{\mathsf{Logic}}$.

3 Evaluation

Since $\mathsf{MQ}_{\mathsf{Logic}}$ allows us to run arbitrary Prolog queries against the model, we may issue every computable query. So, in terms of expressiveness, LQF is equally powerful as any API offered by any UML tool (assuming unrestricted read access to the complete model by the respective API). Similarly, any computable function may theoretically be expressed in OCL, so there is not difference in terms of expressiveness between these

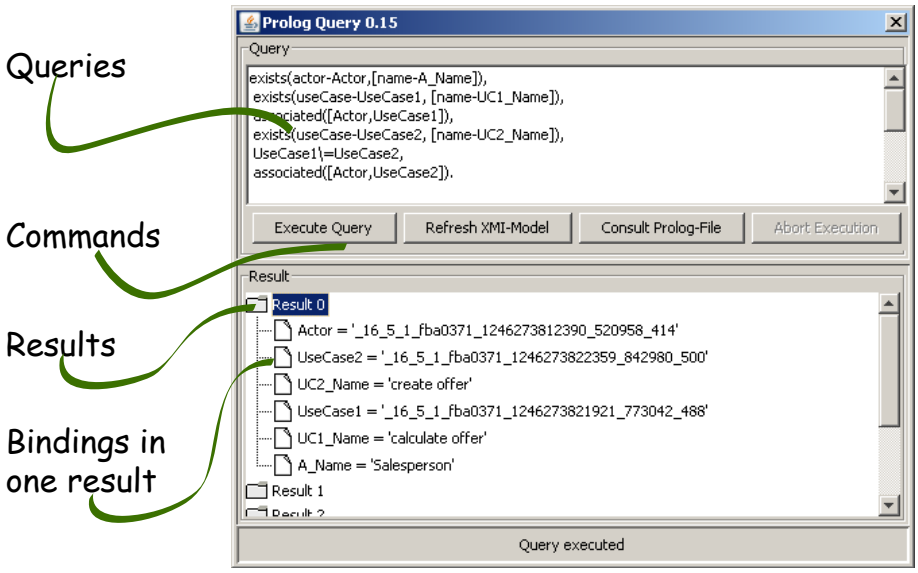


Figure 5: Screenshot of the MQ_{Logic} prototype running in MagicDraw.

three alternatives. Now, the crucial question is, whether creating and/or understanding queries in LQF is easier than in OCL. In order to find out, we tested the MQ_{Logic} tool. Over the last years we have collected a suite of the ten most popular queries (beyond full text search) people have wanted to run against their models (see Fig. 6). We will use them as a benchmark to evaluate predefined APIs, LQF and OCL, contrasting how they represent these queries. Due lack of space, we will discuss only the first six queries in this paper.

- text search with pattern matching
- search for particular attribute values
- unconnected nodes/subgraphs
- all transitive super classes
- counting elements of given types
- undefined attributes
- elements of a given type
- structural patterns
- invisible model elements
- references to an element

Figure 6: Some of the most frequent types of queries in industrial modeling projects.

3.1 Text search with pattern matching

Probably *the* most frequent query is to do a full text search for a given string over a complete model. Most (though not all) CASE tools offer this functionality. A sample application might be "Find all occurrences of 'foobar' in any attribute of any model element." In LQF, this is a rather simple expression.

```
exists(_, Element, [Attr-Val]),
match(Val, '*foobar*').
```

Recall that all variables are written with leading capitals except the underscore which is the anonymous variable. Analogously, we could ask for an element whose name is restricted by a wild card pattern. For instance, when looking for all occurrences of the factory-pattern, we might ask for "All classes whose name ends with 'Factory'". In LQF, this may be expressed as follows.

```
exists(class, C, [name-N]),
match(N, '*Factory').
```

To our knowledge, such queries can't be expressed in OCL.

3.2 Search for undefined attributes or particular values

One of the most common queries is to ask for "unfinished work", for instance, any attributes that should be filled but are not. For instance, operations of classes may or may not have a visibility. So, when looking for operations that lack a value for "visibility", in LQF we would have to say

```
exists(operation, Element, Attributes),
not(member(visibility, Attributes)).
```

while we could not express this in OCL.

Since most tools do not distinguish between attributes that are left empty on purpose and attributes that have not yet been filled, it is common to set attributes of the latter kind with a dummy default value like '??' or 'ToDo', indicating unfinished business (if an automatic default is not available, it may be replaced by manual work). Then, a full text search could find such markers. However, it must also be possible to restrict the search scope and the the text search must be guaranteed to access all fields. Unfortunately, these preconditions are rarely met (we know of no such example). Thus, querying for such values across all types of attributes is a convenient way of checking for unfinished business. To our knowledge, this can't be expressed in OCL. In LQF, the query would read

```
exists(_, Element, [Attribute-'??']);
exists(_, Element, [Attribute-'ToDo']).
```

3.3 All elements of a given type

The first query from our benchmark that may be expressed in OCL is a query for all elements of a given type, say, classes, in a given model. Using LQF, this query could be expressed as `exists(class, C, [])`. In OCL, we would have to use the `allInstances` construct, as in `Class.allInstances()`.

```
context Package
  self.packagedElement->select ( t | t.oclIsTypeOf(Class)).
```

Both queries are of approximately similar complexity, but it is already clear that the second query requires knowledge of the UML meta model (i. e., the meta association `packagedElement`), but also that the OCL syntax is rather complex (i. e., the difference between the `.` and the `->` operator, and the keywords `self`, `select` and `oclIsTypeOf`). This type of query is also easily expressed in many tools' query facilities using predefined queries.

3.4 All transitive super classes

Collecting all (transitive) super classes of a class named "Contract" amounts to computing a fixed point, which is a rather challenging task for the ordinary modeler (and for quite a computer science graduate, too). Expressing this in OCL adds an additional level of syntactic complexity, as the following code demonstrates.

```
def:
superClasses_1_1(baseClass: Class) : Set(Class) =
   if self.hasGeneralization()
   then self.generalization.general.
          oclAsType(Class)->asSet())
   else Set{}
   endif

def:
superClasses_n_1(baseClasses: Set(Class)) : Set(Class) =
   baseClasses->forAll( bc | superClasses_1_1(bc) )
     ->flatten()->asSet()

def:
superClasses_n_n(baseClasses: Set(Class)) : Set(Class) =
   let next = superClasses_n_1(baseClasses)
   in if next.equals(baseClasses)
      then return baseClasses
      else return superClasses_n_n(next)
      endif
```

We first define `superClasses_1_1` to compute the set of direct super classes of a single class, the simplest case. In the next step, we lift this function to sets of base

classes, defining `superClasses_n_1`. The `flatten` operator transforms sets of sets of items into sets of items. Finally, chains of inheritance relationships are computed by `superClasses_n_n`, which also includes an implicit occurs check. Our query for the super classes of `Contract` can thus be expressed as follows.

```
let baseClass = self.packagedElement
   ->select( x | x.oclIsTypeOf(Class)
     ->select( x | x.name = 'Contract')
       ->asOrderedSet->at(1)
in superClasses_1_1(baseClass)
```

With LQF, all this complexity is encapsulated in the `is_a` predicate so the respective query is rather simple.

```
  exists(class, Sub,   [name='Contract']),
  exists(class, Super, []),
  is_a(Sub, Super, steps=*).
```

There are three reasons for this succinctness. First, the notion of "is a superclass of" used to characterize the query in natural language is present in LQF, but not in OCL. Creating such an abstraction in OCL requires considerable work and expertise. Second, the OCL syntax is rather complex, thus difficult to master. Third, OCL's type system intervenes, forcing us to include type casting operations like `asOrderedSet()`.

Note also, that in the case of OCL, we would have to define similar functions for *every single* type of relationship that may occur transitively. In LQF, on the other hand, the `rel` predicate covers all type of relationships. Additionally, the most frequent cases (generalization, calling, precedence etc.) are also provided with convenience predicates.

So, while we could hide the complexity of the fixed point computations in OCL behind suitable library functions created by experts, there would have to be a large set of similar functions for different types and usage modes. Six years after the last OCL version was finalized, no such library seems to exist. And even if it did exist, the user still would have to learn a large set of functions with complex syntax.

3.5 Structural patterns

Consider next the query for a particular structure, e. g.: "Collect all actors associated to at least two different Use Cases." This query represents a large class of queries for local model structures and are useful for design pattern mining. In OCL, this query may be expressed as follows.

```
context Package
def:
actorUseCaseAssoc(a: Actor, u: UseCase) : Bool =
  let types : set(Element) =
    self.packagedElement->asSet()->
      select(assoc | assoc.isKindOf(Association)).
```

```
        ownedElement.type->asSet().
  in let participants : set(Element) = {a, u}.
    in types.intersection(participants) = participants

def:
actorWithTwoUCs(a: Actor) : Bool =
  self.packagedElement->asSet()-> select(ucs |
     ucs.isKindOf(UseCase))
      ->collect( uc | actorUseCaseAssoc(a, uc))
        ->count() > 1

def:
allActorsWithTwoUCs() : Set(Actor)=
  self.packagedElement->asSet()->
    select(a | a.isKindOf(Actor))
      ->collect( a | actorWithTwoUCs(a))-> asSet()
endpackage
```

In LQF, this query would read as follows (this is also the query we show in Fig. 5).

```
  exists(actor, Actor,[]),
  exists(useCase, UC_1,[]),
  exists(useCase, UC_2,[]),
  distinct([U1, U2]),
  associated([Actor,UC_1]),
  associated([Actor,UC_2]).
```

3.6 OCL-APIs

While the OCL as such does not offer much to support querying. In that respect, it is fairly well comparable to MoMaT without LQF as an additional abstraction layer on top of it. It seems that no such query API exists for OCL. In fact, it seems that there are few OCL APIs for whatever purpose publicly available.

One notable exception is the UML, however, which defines 77 auxiliary functions and helpful abbreviations for defining OCL queries. These include a number that may improve writing queries in OCL, for instance

- `allParents()` returning the transitive closure of the Generalization relationship;
- `general` abbreviates `generalization.general`;
- `<EXPR>[<TYPE>]` abbreviates `<EXPR>.oclAsType(<TYPE>)` where `<EXPR>` is any OCL expression and `<TYPE>` is any meta class (type cast in QVT);
- `opposite` abbreviates access to the opposite end of a (binary) association.

This collection of OCL predicates and shorthands is not really an API, it has not been designed to facilitate end user queries. It is just the collection that happened to be helpful when defining the constraints of the UML standard document. So, it is not complete or orthogonal. For instance, there is no predicate for the transitive closure of the *aggregation* relationship, `allParents` lacks an occurs check, there is no predicate to collect all inherited features, and so on. Also, many of the features of LQF like pattern matching, and predicate overloading are not defined. Still, using these auxiliary predicates makes OCL much better usable than pure OCL, as our experiments have shown (see next Section).

4 Experimental evaluation of LQF

While we believe our approach is obviously better than OCL, we are biased of course, compromising our judgment. Our claim of superiority is mostly concerned with the usability, most notably the understandability of LQF as a model query language. Obviously, such a claim can only be examined empirically. We have therefore devised a questionnaire with a set of tasks to help answer these questions. A complete account of these experiments, unfortunately, would be beyond the scope of this paper and will be submitted elsewhere. Without going into the details, we only summarize our findings here.

The experiment consisted in a questionnaire where subjects were asked to match queries described in natural language and queries described in OCL and LQF, the latter being our two experimental conditions. In a second task, subjects were asked to judge as correct or not pairs of given matches of a natural language query and a query expressed in OCL or LQF. Next, subjects were asked to compare the time and effort it took them to complete OCL and LQF tasks, and their personal opinion of the understandability of the respective languages. Finally, some of the subjects participated in structured interviews to further elaborate on their experiences and feelings concerning the tasks.

Unsurprisingly, we could demonstrate that subjects made many more mistakes using OCL than they did using LQF, for all tasks, and for all categories of errors. Subjects also consistently judged their effort with OCL tasks much higher than LQF tasks and generally found LQF much better understandable than OCL (which was generally judged as very difficult to understand). These findings were also confirmed by post-experiment interviews. Interestingly, the occupation of the subjects (students, IT professionals, scientists) and their prior knowledge of OCL did not influence these results substantially.

As we have said, none of these findings were surprising, quite the opposite. An interesting phenomenon occurred, however, when adding another experimental condition besides OCL and LQF, namely, OCL plus the convenience functions defined en passant in the UML standard (see [5]). We called this query language "OCL+UML".

The error rates of OCL+UML were slightly lower than those of LQF, and similarly, the subjective judgments were slightly better. However, when controlling for prior OCL knowledge, the relation between LQF and OCL+UML flipped, both in error rates and judgments. That is: subjects with no prior OCL exposure performed better on LQF

than on OCL+UML, and subjects with OCL exposure performed better on OCL+UML than on LQF. In most cases, the exposure was a rather substantial MDA course the students acting as subjects had just finished.

5 Discussion

5.1 Summary

This paper presents the Logical Query Facility (LQF), a very high-level Prolog API suitable for querying UML models ad-hoc by end-users. We have implemented the MQ_L tool, a plug in to the popular MagicDraw CASE tool implementing LQF. It allows to access all languages supported by MagicDraw, i.e., all of UML, a variety of UML profiles, and BPMN. Executing a query in MQ_L amounts to translating a UML model into a Prolog rule base, and executing the LQF-based query predicate on it. LQF builds on the MoMaT system (see [8]). It shares some of the infrastructure of VMQL [10], but follows a distinct approach defining its own language, and providing its own tool.

5.2 Contribution

Our approach attempts to achieve *universality*, *expressiveness*, and *simplicity* (cf. Section 1.2). We have evaluated the universality and expressiveness of our approach against these goals by collecting a test suite of common queries and checking that all of these queries can be expressed in LQF. We have evaluated the simplicity of our approach by contrasting the OCL and LQF representations of these queries. It is obvious that LQF expressions are much simpler and shorter than corresponding OCL expressions. We have tried to confirm this finding by a controlled experiment. Although our results seem to confirm our hypothesis, we do not have sufficiently many data points yet to truly support our claim. Further experimentation is clearly called for.

LQF offers two advantages over OCL, today's de-facto standard for querying UML models. First, it shields the modeler from the complexity of the UML meta model so that a modeler may express queries using familiar concepts. Second, it provides a very small, yet powerful interface as all predicates may be used in different usage modes (i. e., different patterns of instantiating parameters). As our experiments have demonstrated, this interface is truly easy to understand.

While we cannot be sure that our sample of queries is truly representative for all application contexts, it surely is sufficient to contrast the different approaches. Obviously, all text based query facilities for visual query languages suffer from the media gap between query and model. To which degree this impedes querying is currently an open question.

5.3 Future work

There are a number of promising routes for future work. First of all, LQF lacks means to access the diagrammatic aspect of models, i. e., visual features of diagrams such as

relative position, size, and so on. Also, accessing the meta model in the same way as the model would allow parameterization over concepts.

Then, $\mathsf{MQ}_{\mathsf{Logic}}$ is just a prototype. It currently lacks features for visualization of query results, debugging support, and productivity features like syntax highlighting, auto completion and so on.

Finally, the syntax seems to be suboptimal. Whether the improvements come from visual notations like VMQL (cf. [10]) or controlled natural language constructs can only be determined empirically.

References

[1] Luca Cardelli and Peter Wegner. On Understanding Types Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4), December 1985.

[2] Josef Edenhauser. MX – Model Exchange Tool. Master's thesis, Innsbruck University, 2008.

[3] No Magic, Inc. *USERS MANUAL (version 16.5)*, 2009. available online at `http://www.magicdraw.com`.

[4] OMG. UML 2.0 OCL Specification (ptc/03-10-14). Technical report, Object Management Group, October 2003. available at `www.omg.org/docs/ptc/03-10-14.pdf`.

[5] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, V2.2 beta (ptc/08-05-04). Technical report, Object Management Group, May 2008. Available at `www.omg.org`, downloaded on March 6^{th}, 2009.

[6] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Query Models. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proc.* 7^{th} *Intl. Conf. Unified Modeling Language (≪UML≫'04)*, number 3273 in LNCS, pages 98–112. Springer Verlag, 2004.

[7] Dominik Stein, Stefan Hanenberg, and Rainer Unland. On Relationships between Query Models. In A. Hartman and D. Kreische, editors, *Proc. Eur. Conf. Model Driven Architecture – Foundations and Applications (ECMDA-FA 2005)*, number 3748 in LNCS, pages 77–92. Springer Verlag, 2005.

[8] Harald Störrle. A PROLOG-based Approach to Representing and Querying UML Models. In Philip Cox, Andrew Fish, and John Howse, editors, *Intl. Ws. Visual Languages and Logic (VLL'07)*, volume 274 of *CEUR-WS*, pages 71–84. CEUR, 2007. Available at `ftp.informatik.rwthaachen.de/Publications/CEUR-WS`.

[9] Harald Störrle. Large Scale Modeling Efforts. A Survey on Challenges and Best Practices. In *IASTED Intl. Conf. Software Engineering (SE'2007)*, pages 382–389. IASTED, 2007.

[10] Harald Störrle. VMQL: A Generic Visual Model Query Language. In Martin Erwig, Robert DeLine, and Mark Minas, editors, *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'09)*. IEEE Computer Society, 2009. to be published.

[11] Mathias Winder. MQ – Eine visuelle Query-Schnittstelle für Modelle, 2009. Bachelor thesis, Innsbruck University.