Abstract
In large scale model based development, analysis level models are more like knowledge bases than engineering artifacts. Their effectiveness depends, to a large degree, on the ability of domain experts to retrieve information from them ad hoc. For large scale models, however, existing query facilities are inadequate.

The Visual Model Query Language (VMQL) is a novel approach that uses the respective modeling language of the source model as the query language, too. The semantics of VMQL is defined formally based on graphs, so that query execution can be defined as graph matching. VMQL has been applied to several visual modeling languages, implemented, and validated in small case studies, and several controlled experiments.

Keywords: Model Querying, Unified Modeling Language (UML), Object Constraint Language (OCL), Domain Specific Languages (DSL), End User Modelers

1. Motivation

Many software development approaches today use models instead of code, e.g., in model based and model driven software development (MB/MDSD), Domain-Specific Languages (DSLs), Business Process Management (BPM). As a consequence of this shift of focus, models have become a prime asset and tasks such as version control and configuration management, consistency checking, transformations, and querying of models are much more common and important today than they used to be. In fact, in all of the large scale industrial modeling projects we have studied (or participated in ourselves), the analysis level models were effectively repositories whose main purpose was to store and distribute domain knowledge. Obviously, the benefit of such models

Email address: hsto@imm.dtu.dk (Harald Störrle)
largely depends on them being readily available, and indeed in the modeling projects 
we have studied, modelers urgently asked for a powerful and easy to use model query 
facility (MQF). Unfortunately, though, the facilities offered in current CASE tools 
are not sufficient for real-life projects.

Since 2005, we have been collecting data on model sizes by interviewing expert 
modelers from industry on their experiences. The initial results have been published 
in [45]; Fig. 1 presents the models of this study together with new data points we 
have acquired since.

In this chart, the x-axis ("Views") shows the number of views (or "Diagrams"), 
the y-axis ("Model Elements") shows the number of model elements, and the size 
of the circles ("Weight") shows the number of concepts in the respective modeling 
languages, and thus the information content of each of the model elements. For a 
UML model, say, model elements are instances of meta classes, while the weight is 
the number of meta classes and stereotypes.

These dimensions represent different aspects of the size of a model that influence 
the effectiveness of approaches to query these models. For instance, a large weight 
means that there are many different searchable elements, which limits the usefulness 
of text search since many occurrences of keywords will be found out of context. Also, 
a large weight allows a significant reduction of the search space in many practical 
settings, because a typed element search needs to consider only the subset of model 
elements instantiating the modeling concept sought after.

In practice, there are several ways to mitigate the need for model query facilities. 
For small and medium scale models, there is no need for a query facility at all: 
manual search or plain memorizing will be more effective. If the number of views 
is relatively small (i.e. below 10), large overview diagrams ("wallpapers") might 
be the best way to maintain overview, and thus get questions answered. Clever 
organization of the model will allow expert modelers to find what they are looking 
for manually even in large scale models. However, non-experts find it difficult to 
query large scale models this way, in particular if they also have large weight and 
many views. Finally, if large scale models are machine generated they typically lack 
a helpful structure, and elements of secondary notation that will assist manual search 
(in particular naming conventions and layout). So, for very large models, obviously, 
manual search is hopeless, even by modeling experts.

In the next Section, we will give an introductory example of VMQL, along with 
a sample model base, followed by an explanation of the research method used in 
this project (Section 3). We continue with the comprehensive presentation of the 
VMQL notation (Section 4) and semantics (Section 5). In Section 6 we will present 
our prototype implementation of VMQL. Section 7 discusses existing model query
Figure 1: A survey of large scale models, see [45]. The x-axis ("Views") shows the number of views (or "Diagrams"), the y-axis ("Model Elements") shows the number of model elements. The size of the circles ("Weight") shows the number of concepts in the respective modeling languages.
facilities and Section 8 deals with evaluating VMQL against the major requirements for ad hoc query facilities. This evaluation is carried out by controlled experiments and case studies, and also compares VMQL with related work. Section 9 concludes this article with a brief summary of VMQL’s contributions.

2. The VMQL Approach

In this section we will introduce VMQL by an example. Throughout this paper we will use the model base presented in Fig. 2 and Fig. 3. This model originated from an industrial domain analysis project for an insurance administration software system; the original model contained approximately 180 classes and 300 business processes. Although the part presented here is simplified and adapted for presentation, this is not an unrealistic example (except for its minuscule size). This and similar modeling projects provided both the motivation for this research and the guidance which queries are the most relevant from a practical point of view. The model is expressed in UML 2.2 (cf. [32]) which we have chosen as our main point of reference because of its widespread adoption. For the same reason, most of our examples use class diagrams, the most well known notation of UML. When referring to UML meta classes we will write them using the UML convention, i.e. in camel-caps (as in AssociationClass or MergeNode). Individual elements of a model will be presented in a fixed width typeface (as in Person).

2.1. An introductory example

Assume that you are a modeler working with the model base shown in Figures 2 and 3 and that you want to find out what insurance products are modeled. This is easy, if this query exists as a predefined query in the CASE tool used. However, it is impossible to define in advance all the queries that modelers will find useful. Writing a query using the CASE tool’s API (provided it has one) is feasible only for those modelers that are also skilled programmers and have spare time for writing and debugging such a program: a set of prerequisites not met by most modelers in real life. Issuing a full text search is not an option either, as the keyword "Product" will yield a great number of false positives, and many products will go undetected. Using a built-in query language is also not an attractive option, for the modeler would have to learn yet another language for this special purpose. Also, many tools do not provide such a facility, and if they do, it might be very hard to use it. Consider as an example the Object Constraint Language (OCL) which might be used for expressing queries on UML models. It is extremely difficult to use, even for expert users (see Section 8.4 below for evidence).
Figure 2: A model fragment of an insurance administration system. This model originated from an industrial domain analysis project and has been simplified and adapted for this paper (continued in Fig. 3).
Figure 3: A model fragment of an insurance administration system. This model originated from an industrial domain analysis project and has been simplified and adapted for this paper (continued from Fig. 2)
Figure 4: Using VMQL to query for all products in the model base shown in Fig. 3: query (left, top); list of bindings for candidate solutions (left, bottom); visualization of first solution (right).
2.2. Terminology

So, how can model querying be supported in a powerful, flexible, and yet user-friendly way? We believe that VMQL is at least a partial answer. Consider Fig. 4 for a first glimpse of how it works. Read the query on the top left of Fig. 4 as follows: Find all classes that have a super class named "Product", and then bind their name to the variable "Result". A VMQL implementation will present all possible bindings to a user, and lets him or her browse and inspect the candidates in their respective definition or usage environment (see Fig. 4 right). For better result interpretation the binding is highlighted by fat green outlines by the tool.

A VMQL query can be seen as a set of minimum constraints each individual result must satisfy. So, it is irrelevant whether Product or LifePlan have attributes or associations other than those specified by the query. More technically speaking, executing a query amounts to computing an injective mapping from the elements in the query model to the elements in the base model, such that all constraints associated with the query model are satisfied. We call such a mapping a binding.

The process of querying a large model or set of models can be broken down into four basic steps (see Fig. 5): first, the user must formulate and input the query. Second, the query must be executed and a result set collected. Third, the results must somehow be visualized to be accessible to the fourth and last step, the interpretation of the results by the user. Obviously, the end-to-end effectiveness of model querying depends on all of these steps, and errors or shortcomings in one step will limit the effectiveness of the subsequent steps. Therefore, it is essential to view the whole process and not just individual steps.

To this end, we propose to express queries as annotated model fragments, using more or less the same notation that is already being used for expressing the models to be queried—plus a range of optional annotations (constraints) to provide more expressiveness and flexibility for queries. We will call the modeling notation host language and the model to be queried source model in the remainder. The complete set of source models is collected in the model base. Executing a query (that is, a
query model) amounts to finding matching fragments in the source model and thus establishing an injective function from model elements of the query model to model elements of the source model (binding) and values for all free variables in the query (valuation).

Finally, the results must be displayed back to the user, and again, we use the notations of the host language for this. Obviously, then, anybody who can model can also read and write queries with virtually no additional learning effort. Since the results are also presented in the host language, there are no semantic gaps between model base, query, and query result. We expect that this makes querying models much easier, and first practical experience has confirmed this.

3. Methodology

VMQL is the result of a four year research project following the design science paradigm (cf. [19]). In this section we will briefly explain the iterations of this research project.

3.1. Starting point

The starting point of this research comes from first hand experience with very large scale industrial modeling projects. We observed that model query facilities (MQFs) are needed, when models grow to a certain size. In large scale modeling projects, however, many modelers come from the business side rather than the technology side of development, that is, they may not have a mathematical or engineering background. Addressing this audience with a model query facility, therefore, entails stringent usability requirements which are not met by the MQFs of today’s CASE tools. As a consequence, employing logical formalisms, programming languages, or concepts like recursion in a model query facility may be a considerable obstacle to its adoption by the prospective users.

On the other hand, any audience will demand as much expressive power and flexibility as they can get, and complexity is no prerogative to technology. In fact, some of the most complex concepts may be found in application domains like tax legislation (e.g. at the data point "FMK/Konsens BIENE" in Fig. 1). If these complex and subtle issues are to be accessible by model querying, the respective query facility must support rather intricate queries, too.

Existing query facilities do not meet these demands. Either, they offer too limited expressive power (like full text search), or they are too difficult to apply by end user modelers for ad hoc querying (like using a CASE tool API). We will discuss the respective advantages and disadvantages in detail in Section 8.
3.2. Step one: technological feasibility

In the next step, a technological basis suitable for querying models had to be selected. We have evaluated and compared SQL, OCL, and Prolog as platforms. SQL would offer the highest performance in terms of model size handling and query execution and the best choice of tools, but Prolog offers a much higher level of abstraction, higher computational power with an altogether acceptable tool environment. OCL offers perfect integration with UML (and a few other modeling languages), but very bad usability, and, at the time of this decision, also extremely bad tool support. Also, even highly optimizing OCL compilers made for rather slow query execution. Surprisingly, Prolog proved to be much faster.

We started to build up a framework of tools and libraries to support a multitude of operations on models, including queries, called the Model Manipulation Toolkit (MoMaT, see [44] for details). This work yielded two insights: First, the approach as such was feasible and practical, also from a performance point of view (see [11] for a comparison). Second, the level of abstraction was too generic and difficult to use by end user modelers. Thus, further work was necessary.

3.3. Step two: appropriate abstraction level

In the second step, we looked for the right concepts to query models with, that is, the abstraction level appropriate for end user modelers. Our search was informed by the experience the author gathered as a method and modeling consultant in industry at that time. This led to the development of the Logical Query Facility (LQF, see [46]). In controlled experiments reported below in Section 8.4 we could prove that modelers performed much better on certain model querying tasks when using LQF than when using OCL. This was not surprising result, since the abstractions implemented in LQF were created precisely such that they matched the needs of end user modelers, and OCL lacks such an abstraction layer. However, in-depth qualitative interviews with individual study participants revealed that the relative usability of a MQF might not just depend on the right abstraction level, but also on a second factor: concrete syntax. So, in a follow-up experiment, we equipped OCL with a library simulating the same level of abstraction as offered by LQF. We called this approach OCL+. It turned out, that subjects performed much better on OCL+ than on pure OCL, though still not quite as good as on LQF. This led to the question whether the concrete syntax could be further improved to yield yet better performance in query tasks.

1In the meantime, more powerful OCL tools have emerged, in particular as Eclipse plugins, e.g. [10].
3.4. Step three: supportive concrete syntax

In the third step, we tried to equip LQF with a visual syntax. This "front end" evolved and eventually became VMQL. An early version of VMQL has been published in [47]. It has been extended and improved since, and the results have been validated experimentally, as reported below (see Section 8.4).

VMQL aims at supporting end user modelers in formulating and issuing ad-hoc queries. It is safe to assume that people working with a large scale model are always familiar with the language used for creating the model in the first place. If queries may be expressed as (annotated) models, formulating queries requires less prior knowledge, and there is a smaller media mismatch between the models queried, the queries, and their results than for other query facilities. Thus, for our target group model querying ought to be easier using VMQL than using LQF or OCL+. Conceptually, though, VMQL offers more or less the same expressive means as LQF.

On the other hand, by using essentially the same notations for specifying models and queries on them, VMQL is generic: it may be used for almost any visual modeling language, including individual visual languages like ER-diagrams, use case maps, BPMN, MSCs, SADT, Petri nets or state machines, and also the members of visual language families like UML [32, 43], ARIS [13], and IDEF [29], each of which contains several different visual languages. Moreover, VMQL can even be applied to Domain-Specific Languages (DSLs) not yet defined.

4. Syntax

The syntax of VMQL includes the syntax of the host language, with all of its notational elements. Additionally, VMQL offers a range of constraints to be attached to model elements as comments equipped with the stereotype <<vmql>>. Table 1 presents a complete list of VMQL constraints. We will explain them in turn below. If nothing else is specified in this table, the respective constraints are applicable to any types of model elements. Several constraints may be connected using the usual logical connectives and brackets. For instance,

\[
mclass <: \text{Action and not mclass} = \text{ChangeEventAction}
\]

is valid for all model elements that are instances of Action or its (transitive) subclasses, except ChangeEventAction. Note that <: denotes the subtype relation of concepts of the host language (i.e., generalization in the meta model); the asterisk *
stands for any value, i.e., paths of arbitrary length, arbitrary values for \texttt{mclass} or \texttt{mattr}, or arbitrary multiplicity.

Generally, the Prolog syntax and conventions apply: variables names start with capital letters, and single quotes are used to escape atom names starting with capital letters. Also, variables may be used several times in one query, but they always refer to the same value in one result, just like mathematical variables and unlike container-like variables as known from most imperative programming languages. Finally, conjunction is usually abbreviated by separating expressions by commas, while the notation for disjunction is the semicolon.

Elements of secondary notation, such as layout conventions, are not part of VMQL if they are not represented in the abstract syntax of the host language. Similarly, semantical properties not expressed in the meta model are not (readily) accessible for VMQL queries (more on this issue in Section \ref{sec:conclusion}).

4.1. Ground queries

Ground queries are VMQL queries that use only the means of the host language, but no VMQL constraints. Two examples of queries together with their result are shown in Fig. \ref{fig:ground}. Again, the results are marked by green highlighting and fat outlines.

Note that there are actually three results since the last constraint—an attribute of type Date—has three possible bindings: the attributes ”birth date”, ”last change”, and ”entry”, respectively.

When running Query 2 (see Fig. \ref{fig:ground}), no binding can be established, and so, no results will be returned. There are three points of failure: the model base does not have a class \texttt{Address} with (1) an attribute \texttt{zip} of type \texttt{String}, (2) an association to a class \texttt{Product}, and (3) with value \texttt{true} for the meta attribute \texttt{isAbstract}.

Thus far, only elements of the host language have been used, in other words: these queries can be used in any visual language, and any tool. Now we turn to constraints that require more interference with the host language.

4.2. Name and identity constraints: \texttt{match}, \texttt{name}, \texttt{distinct}, and \texttt{once}

Query 1 and Query 2 in Fig. \ref{fig:ground} are ground queries: they do not contain variables; now we introduce constraints on names and element identity. Suppose a modeler is

---

3Note that modelers may find it hard to distinguish between primary and secondary notation. Consider e.g. containment of Parts in Classes in UML composite structure diagrams, cf. e.g. \cite[Fig. 9.26, p. 193]{uml} which belongs to the primary notation while the customary vertical arrangement of use cases in a system is not.

4In UML, using an italic font for the class name shows that it is abstract. This property is reflected by the boolean meta attribute \texttt{isAbstract}, cf. \cite[p. 53]{uml}.
<table>
<thead>
<tr>
<th>Constraint</th>
<th>Informal Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>match</td>
<td>Restricts the name of the constrained model element by a wild card expression or regular expression.</td>
<td>match pa?tern*</td>
</tr>
<tr>
<td>distinct</td>
<td>Enforces that a set of constrained model elements of the same type are pairwise distinct.</td>
<td>distinct</td>
</tr>
<tr>
<td>once</td>
<td>Enforces that a solution occurs only once in the set of all solutions.</td>
<td>once</td>
</tr>
<tr>
<td>steps</td>
<td>Defines the length of a path between two connected model elements. Only one type of relationship may occur on the path. Applicable values are integers &gt; 0 or * for arbitrary length &gt; 0. Applicable to elements that are subclasses of Relationship.</td>
<td>steps = 3 steps &lt; 3 steps = * steps = 2; 3 steps &gt; 1, steps &lt; 4</td>
</tr>
<tr>
<td>indirect</td>
<td>alias for steps = *</td>
<td>indirect</td>
</tr>
<tr>
<td>mclass</td>
<td>Allows the constrained element to be of a different meta class than actually specified in the query.</td>
<td>mclass = Class mclass = Class; Component mclass = * mclass &lt;: Feature</td>
</tr>
<tr>
<td>mattr</td>
<td>Constrains the value of a meta attribute that has or has not a representation in the concrete syntax. If given a value expression, the meta attribute’s value must conform to it. If given a variable (i.e. any expression starting with $), the value of the meta attribute is bound to that variable if possible. Variable may be bound several times, but only with one value.</td>
<td>mattr isRoot = true mattr aggregationKind = composition; none mattr isAbstract = * mattr name = $N</td>
</tr>
<tr>
<td>name</td>
<td>alias for mattr name = $N.</td>
<td>name = $N</td>
</tr>
<tr>
<td>precision</td>
<td>Reduces the precision level used in the model matching to values below 1.</td>
<td>precision = 0.8</td>
</tr>
<tr>
<td>strict</td>
<td>Enforces that a query element must match exactly with a result element, i.e., the binding is bijective rather than injective.</td>
<td>strict</td>
</tr>
<tr>
<td>optional</td>
<td>Allows that a constrained model element of a query may or may not appear in the result.</td>
<td>optional</td>
</tr>
<tr>
<td>either</td>
<td>Allows a set of alternatives for a constrained model element of a query to appear in the result. Applicable only to non-empty sets of model elements.</td>
<td>either</td>
</tr>
<tr>
<td>not</td>
<td>Prevents a result from containing a match for the constrained model element of the query.</td>
<td>not</td>
</tr>
</tbody>
</table>

Table 1: Overview of VQML constraints, grouped by the subsection in which they are covered. If nothing else is said in the middle column, these constraints may be applied to model elements of any type (i.e., instances of any meta class). The precise meaning is defined in Table 2.
Figure 6: Sample VMQL ground queries and their results in the model base defined in Fig. 2. Query 1 has exactly three results, the first one being shown on the right. Query 2 (shown below Query 1 on the left) has no results.
Figure 7: Sample VMQL name queries and their results in the model base defined in Fig. 2. Query 3 (top) has exactly one result. Query 4a (bottom left) has no results, Query 4b (bottom middle) has no results, Query 4c (bottom right) has no results.

looking for a special health-related product. He remembers that the name contained something like "Disease" or "Medical" and probably the name ends with "Plan", as most such products may do. Simply running a full text search will get thousands of false positive hits, since these words occur all over the model, e.g., in comment fields, use cases, activities and so on. In VMQL, he may issue a query like Query 3 shown in Fig. 7 constraining the name of the result elements by an expression with wild cards (the asterisk for any sequence) and choice (the vertical bar). Brackets may be used to group subexpressions. All the usual features of regular expressions are allowed (i.e., option, choice, repetition).

Now assume that a modeler is looking for all classes associated directly with Person. Since Person is a very important class in the model, he can’t simply find the diagram where it is being defined and then look for the neighbors. Likewise, finding all references to Person is useless as it produces too many false positive matches. Query 4a, on the other hand, asks for the classes directly associated to Person straight away.

However, Query 4a will also find Person as a neighbor of itself due to the reflexive Association covers. In order to avoid this, the two Classes in the query must be constrained as being distinct, see Query 4b. Both Query 4a and Query 4b will find Contract twice, since there are two distinct Associations between Person and Contract. In order to avoid this, the unnamed Class in the query may be constrained also to occur only once in the solution set, see Query 4c. The following table sum-
marizes the number of matches for the various neighbors of Person by Query 4a to Query 4c.

<table>
<thead>
<tr>
<th>Bindings</th>
<th>Address</th>
<th>Person</th>
<th>Contract</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 4a</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Query 4b</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Query 4c</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

4.3. Path-related constraints: steps and indirect

Path constraints refer to the frequently found edge-node-structures in models. Using steps, an individual link in such a path may be generalized to a proper path of a specified length. Paths of finite but arbitrary length are captured by the length *. A special case is the indirect constraint which is just an alias for steps = *.

Returning to our running example, assume we are looking for all classes associated with Person. At first sight, Query 5 in Fig. 8 appears to do this. However, it yields four different bindings for the variable Result, namely Address, Person, Contract, and Product, because these are the only direct neighbor of Person; Supplier is not a solution. In order to find this class also, the steps constraint is introduced. So, Query 6 will yield Supplier as a new different binding for the variable Result.

However, several of these different bindings are found more than once. For instance, Association-paths of length one starting from Person will find Person by virtue of the reflexive Association covers as specified in the class diagram ”Insurance Entities” of Fig. 2. Association-paths of greater length will find the results over and over again, since associations are not directed. This may be suppressed by the once constraint. So, Query 7 yields every solution only once. Note that steps = * is defined as if there were also a once constraint. In order to also suppress solutions yielded from reflexive relationships, the distinct constraint may be used (see Query 8).

The paths constrained using steps may contain different types of intermediate nodes, but only one type of link. Consider Query 9, which looks for all Actions being surrounded by a pair of ForkNode and JoinNode, which indicate that they could be processed in parallel with other Actions of the same Activity. The Action ”verify customer account” in the activity diagram ”Coverage Quote Processing” (see Fig. 3) is a solution to Query 9 irrespective of the DecisionNode immediately afterwards on the path.

4.4. Meta model constraints: mclass and mattr

Meta model constraints allow direct access to the abstract syntax (i.e., the meta model) of the host language. So, they are specific to one particular version of a fixed
Figure 8: Sample VMQL path queries and their results in the model base defined in Fig. 2.
The `mclass` constraint allows one to enlarge the set of model element types matching a model element in the query. It is useful whenever the concrete syntax is ambiguous (several concepts sharing the same grapheme, e.g., different kinds of Action, or Class vs. AssociationClass), or when concepts have no visual representation (e.g., Classifier, ModelElement).

In our running example, let’s go back to the very first query, All Products of Fig. 4. It did not quite live up to its name by leaving out some elements in the specialization tree below Product, namely GroupPlan and AllRoundHealthPlan.

These two model elements were not found by the query All Products for two different reasons. Observe that the title of GroupPlan is set italics. In UML, that means that this is an abstract class, i.e., technically, the value of the meta attribute `isAbstract` is set to true. Any newly created Class (like the one in our query) has its meta attribute `isAbstract` set to true, which is the default value, and so the abstract class GroupPlan does not match the concrete class in the query. Explicitly allowing any value for this meta attribute fixes this.

For AllRoundHealthPlan, the case is quite different. In the class diagram ”Standard Product Catalog” in Fig. 2 AllRoundHealthPlan is defined as being an AssociationClass rather than a Class. In the UML meta model, AssociationClass is a subclass of Class. Again, since the type of AllRoundHealthPlan and the query element are not identical, they will not match. Explicitly allowing this additional
type alternatively to Class, or allowing any of its direct or indirect subtypes by using \texttt{mclass <: Class} will obtain the desired result. However, then other subclasses of Class, Component, say, would be included as well. In order to be more specific about the admissible types, they might be listed explicitly as in \texttt{mclass = Class}, \texttt{AssociationClass} or constrained by a combination of several \texttt{mclass} expressions. This feature is particularly useful when the meta model contains large generalization trees, e.g., for the \texttt{Action}, \texttt{Classifier} or \texttt{Behavior} hierarchies.

Obviously, using the \texttt{mclass} and \texttt{mattr} constraints ties queries to the particular meta model underlying the modeling language used, so some of the generality of VMQL is lost here: should the meta model change, the query may become faulty. However, we get a great deal of expressiveness in return, useful for all meta model-based modeling languages. Also, the fragility introduced by this constraint type is localized and thus easy to adapt.

The \texttt{mattr} constraint may also be used with variables instead of values or value expressions in order to look up the value, bind it to the respective value and refer to it in some other place of the query. Consider \texttt{Query 11} in Fig. 9. There, we look for refactoring candidates by querying two distinct subclasses of an unnamed class that both have an attribute with the same (arbitrary) name and type. This is ensured by placing constraints on the attributes that bind the values of the meta attributes \texttt{name} and \texttt{type} to the variables \texttt{N} and \texttt{T}, respectively. Note that constraints always override actual model values so that the bogus name “foo” plays no role. It is placed here only to make the attributes visible in the diagram.

Clearly, creating meta model constraints is a rather complex task and requires considerable understanding of the meta model. It is unlikely that many end user modelers, that is, VMQL’s intended audience, are able to use this feature without help. We have included this type of constraint anyway, because it significantly increases the expressiveness of VMQL. Also, providing default meta model constraints (see Section 4.6) allows to take advantage of this feature in a way suitable for end user modelers, too.

4.5. Matching constraints: \texttt{strict, precision, optional, either, and not}

The modeler may exert a certain amount of control over the matching process. The \texttt{precision} constraint defines the degree of similarity required for two model elements to be considered as matches (the default is 1). Reducing \texttt{precision} will admit more solutions and usually changes the order in which results are being presented to the user. Modifying the matching precision can be helpful when developing queries, because low-precision queries will return the expected results even for overspecified queries, if only at a lower rank. Similarly, reducing the precision may help avoid
small mistakes like, e. g., spelling of types (cf. `int` vs. `Int`), or names (e. g., color vs. colour). The drawbacks include increased processing times and vastly increasing numbers of false positives.

The exact opposite behavior is achieved by the `strict` constraint. It enforces that matches for query elements or models must not have additional parts and properties. So, for instance, Query 12 in Fig. 10 yields `Address` and `Supplier`, but not `Person`, `Product`, or `Contract`.

If two or more model elements are connected by an `either` constraint, any one of them may occur in a match. Declaring a model element as `optional` will allow results to not contain a match for the respective query element. Query 13 in Fig. 10 finds `Person`, irrespective of whether the address is modeled as an attribute or as an associated class. Finally, the `not` constraint forbids the designated element from occurring in a result. Query 14 in Fig. 10 will yield all classes without subclasses.

### 4.6. Syntactic fine tuning

To increase the usability, we allow several forms of syntactic sugaring beyond mere aliasing, as done before. First, instead of attaching several constraints in their own boxes to one element, logical connectors may be used to combine several constraints in one comment box. We have used this improvement repeatedly already, e. g. in queries 7 through 11. Following the Prolog conventions, commas are used for conjunction, and semicolons for disjunction. Alternatively, the keywords `and` and `or` may be used. The unary Prolog negation operator `\+` may also be used (identical to the `not` constraint), brackets must be used to disambiguate terms.

Second, constraints may be applied to several model elements at one time so as to express that individual copies of the constraint are applied to each of the model elements. This may considerably reduce typing effort, and may even help convey the meaning of the constraint, as is the case for Query 11 which may be shortened to Query 11 (2) (see Fig. 11). For the `distinct` and `either` constraints, attaching them
to a set of more than two model elements is supposed to mean that the constraint applies to all pairs of model elements of the same type in the set. Observe that the model elements in a group constraint may have different types without any impact on the language. A constraint not connected to any model element in a query is taken to hold for all of them. Typically, this only ever occurs for the \textbf{precision} constraint.

As experience shows, the intuition of UML modelers is often inconsistent with the definitions in the UML standard. Put another way: the UML meta model is rather intricate and difficult to understand. Therefore, a modeler’s intuition of any given query might not be accurate. For instance, when looking for all references to a type, in UML it is not sufficient to look for instances of the meta class \texttt{Class} that represent the type, but also to similar meta classes like \texttt{AssociationClass} (see Fig. 9). We have seen some instance of this phenomenon earlier when motivating meta model constraints.

Additionally, other diagram types may also contain model elements whose concrete syntax and intuitive meaning is that of a class, but that are actually instances of other meta classes, for instance \texttt{LifeLine}, \texttt{ObjectName}, and \texttt{ActivityParameterNode}. The query \textbf{References to 'Contract'} shown in Fig. 11 (right) fixes this, and also finds \texttt{Contract} in the running example (Figures 2 and 3) not just in the class diagrams, but also in the activity diagram and in the sequence diagram. Observe that \textbf{References to 'Contract'} also ranges over arbitrary types of diagrams by using a variable "$\textbf{DiagramType}" instead of a concrete diagram type. This feature is currently

Figure 11: VMQL queries using syntactic sugaring (left). Fine-tuning VMQL to UML can be done by introducing default constraints: finding all references to a type in UML is more than asking for a \texttt{Class} (right).
Class -> ( mclass = Class
    ; mclass = DataFlowNode
    ; mclass = ActivityParameterNode
    ; mclass = LifeLine
    ; mclass <: StructuralFeature)
    , matth isAbstract = *

AssociationClass ->
    ( mclass <: Association
        ; mclass = Class
        ; mclass = DataFlowNode
        ; mclass = ActivityParameterNode
        ; mclass = LifeLine
        ; mclass <: StructuralFeature)
        , matth isAbstract = *

Action -> mclass <: Action
    , not(mclass = SendEventAction)
    , not(mclass = ReceiveEventAction)

Figure 12: The complete default constraints for fine-tuning VMQL to UML. These defaults replace recurring complex constraints as shown in Fig. 11 (right).
missing from our implementation due to restrictions imposed by the host CASE tool MagicDraw.

Obviously, having to type a long constraint as in Fig. 11 (right) is quite annoying. More importantly, however, end user modelers will likely not be able to create (or understand) such a constraint. VMQL can take care of this by adding default constraints that change query models in such a way that they mean what users (typically) want to say. This is particularly useful for ambiguous syntactic elements as the square box standing for, among others, Class. The three defaults shown in Fig. 12 have been found helpful for adapting VMQL to UML. They address the issues most frequently occurring, in our experience. For languages other than UML, different defaults might be useful. Currently, such abbreviations are implemented as defaults that are overwritten by user specified meta model constraints on these elements, but other possibilities are being explored.

5. Semantics

The semantics of VMQL is described in three steps: the translation from plain models without constraints (see Section 5.1), the meaning of constraints (Section 5.2), and the matching process between queries and base models (Section 5.3). After that, we will discuss issues related to the semantics of the host language in Section 5.4 using UML as the example again.

5.1. Translating models

Models and model queries may be represented as graphs. This will allow us to formally define matching between graphs, that is, the most important step in query execution. As a first step, we will define a kind of labeled graph suitable for representing models.

**Definition 1** A **Model Graph** is a tuple \( (N, \mathcal{T}, \text{type}, E, \text{source}, \text{link}, A, \text{slot}, V, \text{val}) \) such that

- \( N \) and \( E \) are finite sets of nodes and edges, respectively, with \( E \cap N = \emptyset \);
- \( \mathcal{T} \) is a domain of types representing the modeling concepts (i.e., the meta classes of the modeling language);
- \( \text{type} : N \rightarrow \mathcal{T} \) is a function equipping every node with a type;
- two functions \( \text{source} : E \rightarrow N \) and \( \text{link} : E \rightarrow N \) defining the origin and target of each edge;
• $\mathcal{A}$ is a domain of attribute names representing the modeling concepts’ properties (i.e., the meta attributes of the modeling language);

• $\text{slot}: N \rightarrow 2^\mathcal{A}$ is a function associating every node with a set of attribute names; the same name is used to associate attribute names to edges, i.e. $\text{slot}: E \rightarrow \mathcal{A}$;

• $\mathcal{V}$ is a domain of values representing the properties’ values (i.e., the values stored in the slots realizing meta attributes); and

• $\text{val}: N \times \mathcal{A} \rightarrow \mathcal{V}$ is a partial function associating a value to every combination of nodes $n$ and attributes $a$ defined on that node, i.e., $a \in \text{slot}(n)$.

The notation $g_x$ is used to access the component $x$ of a graph $g$, i.e., $g_E$ denotes the edges of $g$. The element $\bot$ ("undefined") is not element of any domain. It is also required that $\forall e, e' \in E: (\text{slot}(e) = \text{slot}(e') \land \text{source}(e) = \text{source}(e')) \implies e = e'$ for all model graphs.

Model graphs may represent both base models and queries on them. The names $\text{slot}$ and $\text{link}$ in this definition are motivated by the corresponding UML terminology. The slightly unusual codomain of $\text{link}$ is necessary for expressing relationships whose target is an ordered set of elements $\mathcal{T}$, $\mathcal{A}$, and $\mathcal{V}$ are determined by the underlying meta model and are included to make the definition self-contained so as to be applicable to instances of arbitrary meta models, e.g., different versions of UML and, indeed, languages other than UML.

Fig. 13 shows visually, how a model query is translated into its Prolog representation step by step. The first step is not a translation as such, but simply a different view on the same entity: inside any UML-compliant tool, the model query in Fig. 13 (top left) is already represented in a data structure very similar to the structure outlined by the object diagram shown in Fig. 13 (top right).

Such an object structure directly corresponds to a graph in the mathematical sense, as shown in Fig. 13 (middle). Here, we depict nodes as black filled circles with white numbers denoting their identity. Nodes are always tagged by a type, shown in hexagons. Nodes may have any number of slots represented as rectangles with rounded corners attached to them which show the current value and the name of the corresponding attribute or operation (a "Feature" in UML terminology). Nodes may also be connected by labeled edges which we represent as directed arcs.

\footnote{For instance, this is the case for all variants of containment in UML models are of this type, e.g., $\text{Association.ownedEnd}$, $\text{Interface.ownedAttribute}$, or $\text{Operation.ownedParameter}$. Such links typically account for over 80% of all links in a domain model.}
The transformation of such a graph into Prolog is straightforward, see Fig. 13 (bottom): model graph nodes become facts of the `me` predicate, and model graph edges are represented by references to other nodes. If there are several edges in a model graph that start from the same node and that carry the same slot name, they are joined into one attribute in the implementation (see the list-valued `ownedMember` slot of `class-1` in see Fig. 13, bottom). Observe that the layout and the identifiers used in Fig. 13 are consistent to support tracking the elements.

5.2. Semantics of constraints

A constraint can be seen as a predicate to characterize bindings as admissible or not. Table 2 below describes the meaning of each constraint. We use the notation \( \perp \) for "undefined", a value contained in no domain. The notations \([\perp = x]\) shall be false for all \( x \) other than \( \perp \) in any domain.

**Definition 2** Given two model graphs \( q \) and \( m \) representing a model query \( q \) and a base model. A Binding \( \beta \) is an injective function \( \beta : q_N \rightarrow m_N \). This binding is said to satisfy a constraint \( c \), if the condition for this type of constraint as defined in Table 2 holds.

This definition covers constraints on individual nodes as well as constraints on sets of nodes (i.e. constraints of type `distinct` and `either`).

Clearly, the definitions given in Table 2 are tied to UML because they refer to specific elements of the UML meta model (e.g., in the definition of the `steps` constraint). However, it is also clear that it is fairly easy to replace these for another language with another meta model, so the genericity of VMQL is not impeded.

5.3. Matching models and queries

As shown in Fig. 13, models and model queries may be represented as graphs. This allows us to formally define a notion of matching between model graphs, in particular those representing models and queries on them. Since matching is the most important step in query execution, this definition also specifies a large portion of the implementation of query execution.

**Definition 3** Given two model graphs \( q \) and \( m \), and a Binding \( \beta : q_N \rightarrow m_N \). This binding is a Match, if those nodes from the query and the base model that are bound together have (1) the same type and (2) the model nodes have at least the same slots, values, and links as the query nodes they are bound to. More formally,
Figure 13: The transformation of a model query (top left), its tool-internal data structure representation as defined by the UML standard (top right), a conceptual representation as a labeled graph (middle), and finally the resulting Prolog code (bottom). Note that layout and identifiers are consistent to support tracing. The Prolog code has been simplified by removing tool-specific information like version, loaded libraries and stereotypes, and information about the model (UML version, view type, and so on).
<table>
<thead>
<tr>
<th>Constraint</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>match</strong></td>
<td>If a query node $n$ is constrained by <code>match expr</code>, then for any binding $\beta$ in the result set, it must be the case that $val(\beta(n), \text{name}) \in L$, where $L$ is the language defined by the regular expression <code>expr</code>.</td>
</tr>
<tr>
<td><strong>distinct</strong></td>
<td>If a set $N$ of query nodes is constrained by <code>distinct</code>, then for any binding $\beta$ in the result set, it must be the case that $\forall_{n,n' \in N} : n \neq n' \implies \beta(n) \neq \beta(n')$.</td>
</tr>
<tr>
<td><strong>once</strong></td>
<td>The <code>once</code> constraint only removes duplicate results, so the set of results is not affected.</td>
</tr>
<tr>
<td><strong>steps</strong></td>
<td>If a query node $n$ with $\text{type}(n) = t$ is constrained by <code>steps op expr</code>, then for any binding $\beta$ in the result set, there is a sequence $s = n_1 n_2 \ldots n_k$ of nodes in the model graph such that $\forall_{i:1 \leq i \leq k} : \text{type}(n_i) = t$, and $\beta(n) = n_1$. Also,</td>
</tr>
<tr>
<td>&amp;</td>
<td>- if $t$ is <code>Generalization</code> we demand that $\forall_{i:1 \leq i \leq k} : \text{to}(n_i, \text{general}) \subseteq \text{to}(n_{i+1}, \text{ownedMember})$ and $\forall_{i:1 \leq i \leq k} : n_{i+1} \in \text{to}(n_i, \text{ownedMember})$.</td>
</tr>
<tr>
<td>&amp;</td>
<td>- If $t$ is <code>Association</code>, we demand that $\forall_{i:1 \leq i \leq k} : \text{to}(n_i, \text{ownedMember}) \cap \text{to}(n_{i+1}, \text{ownedMember}) \neq \emptyset$.</td>
</tr>
<tr>
<td>&amp;</td>
<td>We abbreviate ${n' \in N \mid \exists e \in E : \text{source}(e) = n \land \text{slot}(e) = s}$ as $\text{to}(n, s)$. The length of the sequence $k$ must be consistent with the constraint’s expression, e.g. $\text{steps} &lt; 5$ demands that $k &lt; 5$ (similar for other types of nodes).</td>
</tr>
<tr>
<td><strong>indirect</strong></td>
<td>The constraint <code>indirect</code> is a syntactic alias of the constraint <code>steps = *</code> with the semantics as defined above.</td>
</tr>
<tr>
<td><strong>mclass</strong></td>
<td>If a query node $n$ is constrained by <code>mclass op expr</code>, then for any binding $\beta$ in the result set, it must be the case that</td>
</tr>
<tr>
<td>&amp;</td>
<td>- if <code>op</code> is <code>=' then $\text{type}(n) \in T$, where $T \subseteq T$ is the set of types that </code>expr` evaluates to;</td>
</tr>
<tr>
<td>&amp;</td>
<td>- if <code>op</code> is <code>:&lt;:</code> then $\text{type}(n) \in T$, where $T \subseteq T$ is the set of types that <code>expr</code> evaluates to plus all their subtypes.</td>
</tr>
<tr>
<td>&amp;</td>
<td>The expression <code>*</code> evaluates to $T$.</td>
</tr>
<tr>
<td><strong>mattr</strong></td>
<td>If a query node $n$ is constrained by <code>mattr a = expr</code>, then for any binding $\beta$ in the result set, it must be the case that $\text{val}(n, a) \in V$, where $V \subseteq V$ is the set of values <code>expr</code> evaluates to. Note that the expression <code>*</code> evaluates to the set of all values possible for the respective $a$.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>The constraint <code>name = expr</code> is a syntactic alias of the constraint <code>mattr name = expr</code> with the semantics as defined above.</td>
</tr>
<tr>
<td><strong>precision</strong></td>
<td>Use approximate matching (Algorithm 2) instead of the exact matching (Algorithm 1).</td>
</tr>
<tr>
<td><strong>strict</strong></td>
<td>If a query node $n$ is constrained by <code>strict</code>, then for any binding $\beta$ in the result set, it must be the case that $\text{slot}(n) = \text{slot}(\beta(n))$ and $\forall_{n \in \mathcal{E}} : \forall_{a \in A} : \text{link}(n, a) = \text{link}(\beta(n), a)$.</td>
</tr>
<tr>
<td><strong>optional</strong></td>
<td>The <code>optional</code> constraint only provides convenience to the user and optimization information for the tool. Omitting it does not change the result set.</td>
</tr>
<tr>
<td><strong>either</strong></td>
<td>If a set $N$ of query nodes is constrained by <code>either</code>, then for any binding $\beta$ in the result set, it must be the case that $\forall_{n,n' \in N} : n \neq n' \implies \neg(n \in \text{dom}(\beta) \land n' \in \text{dom}(\beta))$.</td>
</tr>
<tr>
<td><strong>not</strong></td>
<td>If a query node $n$ is constrained by <code>not</code>, then for any binding $\beta$ in the result set, it must be the case that $n \notin \text{dom}(\beta)$.</td>
</tr>
</tbody>
</table>

Table 2: Semantics of VQML constraints.
\( \beta \) is a match for query \( q \) against model \( m \) (written \( \text{match}(q, m, \beta) \)) if and only if node-match\((q, m, \beta)\) and edge-match\((q, m, \beta)\) with the following abbreviations hold.

\[
\text{node-match}(q, m, \beta) = \forall n \in q_N : \\
(\text{type}(n) = \text{type}(\beta(n)) \\
\land \forall a \in \text{slot}(n) : \text{val}(n, a) = \text{val}(\beta(n), a))
\]

\[
\text{edge-match}(q, m, \beta) = \forall e \in q_E : \exists e' \in m_E : \text{slot}(e) = \text{slot}(e') \\
\land \beta(\text{source}(e)) = \text{source}(e') \\
\land \beta(\text{link}(e)) = \text{link}(e')
\]

Recall that \( \text{val} \) is a partial function, \( \bot = x \) is \textit{false} for all \( x \) other than \( \bot \).

Using the definitions from this section as boolean expressions in pseudo-code, we can now define algorithms for matching model graphs, and thus for executing queries with VMQL. We first consider the exact match algorithm (see Algorithm 1).

\begin{center}
\textbf{Algorithm 1:} Exact matching algorithm (without optimizations)
\end{center}

\begin{algorithm}
\textbf{Input:} two model graphs \textit{query} and \textit{model}
\textbf{Output:} a set of bindings
\begin{algorithmic}
\State \( c \leftarrow \) extract constraints from \textit{query};
\State Result \( \leftarrow \emptyset ;
\State B \leftarrow \) enumerate all bindings between \textit{query} and \textit{model};
\While {\( B \neq \emptyset \)}
\State select some \( b \in B \);
\State \( B \leftarrow B \setminus \{b\} ;
\If {\text{node-match}(\textit{query}, \textit{model}, b)}
\If {\text{edge-match}(\textit{query}, \textit{model}, b)}
\If {\text{satisfies}(b, c)}
\State Result \( \leftarrow \) Result \( \cup \) \{b\};
\EndIf
\EndIf
\EndIf
\EndWhile
\State return Result;
\end{algorithmic}
\end{algorithm}

Trivially, this algorithm terminates, as model graphs are finite by definition. The actual implementation deviates a little from this abstract specification for optimization purposes, e.g., the set of all bindings is computed lazily, that is, exactly when
they are needed. Also, Prologs extremely fast pattern matching capabilities are used to avoid enumerating pairs of elements with different and/or unsuitable types. Further optimizations like ordering the constraints reduce the search space further.

The exact matching algorithm reduces false positives and runtime, but requires more effort on behalf of the user, as queries must be specified exactly right. A more advanced solution returns all bindings, ordered by decreasing similarity (see Algorithm 2 below). First, we have to define a suitable notion for approximate matching.

**Definition 4** Let $\beta$ be a binding between two model graphs $q$ and $m$ representing a query and a base model and $t \in [0..1]$ a threshold value. The similarity between $m$ and $q$ is $s = \frac{1}{2} (\text{node-similarity}(q, m, \beta) + \text{edge-similarity}(q, m, \beta))$ and $\beta$ is an Approximate Match for query $q$ against model $m$ (written $\text{approximate-match}(q, m, \beta, t)$) if and only if $s \geq t$ using the following abbreviations.

\[
\text{node-similarity}(q, m, \beta) = \frac{1}{2} \cdot |q_N| \sum_{n \in q_N} \left( [\text{type}(n) = \text{type}(\beta(n))] + \frac{1}{|\text{slot}(n)|} \sum_{a \in \text{slot}(n)} \text{sim}(\text{val}(n, a), \text{val}(\beta(n), a)) \right)
\]

\[
\text{edge-similarity}(q, m, \beta) = \frac{1}{|q_E|} \sum_{e \in q_E} \left( [\exists e' \in m_E : \text{slot}(e) = \text{slot}(e')] + [\beta(\text{source}(e)) = \text{source}(e')] + [\beta(\text{link}(e)) = \text{link}(e')] \right)
\]

where $\text{sim}$ denotes a suitable similarity function, i.e. $\text{sim}(x, y) = [x = y]$ in the simplest case, or the edit distance (also known as Levenshtein distance) for names. The notation $[\phi]$ for boolean predicates $\phi$ means that $[\phi] = 1$ if $\phi$ holds, and $[\phi] = 0$ otherwise.

The factors 2 and 3 normalize edge similarity and node similarity to $[0..1]$. Using the notion of similarity match, we can now define an algorithm to compute approximate matches.

Choosing $t > \frac{1}{2}$ usually ensures that only elements of the same meta class match, which is often desired. Choosing $t = 1$ yields the same results as exact matching. Of course, this algorithm is slower than Algorithm 1 defined above, because the similarities of many pairs of nodes have to be computed. In fact, it is quite easy to
Algorithm 2: Approximate matching algorithm (without optimizations)

Input: two model graphs query and model, and a similarity threshold value \( t \)
Output: a set of bindings

\[
c \leftarrow \text{extract constraints from } \text{query};
\]
\[
\text{Result} \leftarrow \emptyset;
\]
\[
B \leftarrow \text{enumerate promising bindings between query and model (best first)};
\]

\[\text{while } B \neq \emptyset \text{ do} \]
\[
\text{select some } b \in B;
\]
\[
B \leftarrow B - \{b\};
\]
\[
n \leftarrow \text{node-similarity}(\text{query}, \text{model}, b);
\]
\[
e \leftarrow \text{edge-similarity}(\text{query}, \text{model}, b);
\]
\[
\text{if } \frac{n+e}{2} \geq t \text{ then}
\]
\[\text{if } \text{satisfies}(b, c) \text{ then}
\]
\[\text{ Result } \leftarrow \text{Result } \cup \{b\};
\]
\[\text{end}
\]
\[\text{end}
\]
\[\text{end}\]
\[\text{return Result;}
\]

construct queries with run times beyond practical boundaries. However, it is possible

to reduce practical processing times for realistic cases to an acceptable level by using

some heuristics that exploit the structure of models, see \[8.6\]

5.4. Limits of VMQL

It is important to understand that VMQL obtains its genericity from restricting

itself to the syntactical level. Only those properties of the model captured in the con-

crete syntax may be exploited in VMQL queries. For instance, elements of secondary

notation are not covered, including diagram layout, annotations, and naming con-

ventions. Any model information expressed this way can not be accessed by VMQL.
The same is true for restrictions or refinements of the host language meta model that

are expressed outside the meta model, e.g., in OCL constraints, or informal anno-

tations, but also semantic, and pragmatic conditions imposed by the development

methodology.

As an example, consider the Generalization relationship in UML. The (informal)

UML semantics states that if a Generalization is specified from a Classifier A to a
Figure 14: VMQL works on the syntax, so queries referring to the semantics cannot be expressed directly: Valid call cannot be derived from UML Inheritance; Query 15(1) yields only Product, Query 15(2) yields also all subclasses of Product.

Classifier B, A will be able to access those Features of B that have public visibility.

Fig. 14 (left) illustrates this case: When defining the class diagram UML inheritance, the Interaction defined in the sequence diagram Valid call is indeed admissible, although the abstract syntax of UML does not warrant this interpretation: it is the semantic definition of Generalization which declares that, behind the scenes, the method name m is included in the name space of B, although it is not defined by B but some other class which generalizes B. Dynamically, the call to m() is understood to be delegated to the super class A by the UML equivalent of a runtime environment.

So, in order to query the model defined in Fig. 2 and Fig. 3 for all classes with the attribute effective in their name space, Query 15 (1) is not sufficient, but we need Query 15 (2).

Also, there is currently no convenient way to formulate queries in VMQL that can refer to model elements presented in more than one diagram at the same time. For instance, queries on, say, a business process overview defined in a use case diagram and the process details defined in an activity diagram can not be expressed in VMQL. Similarly, queries that relate to more than one activity (e.g., an Activity A containing a CallActivityAction refined to some other Activity B) are currently out of the scope of VMQL.

6For the sake of the argument, we are ignoring the visibility level "protected" here.
6. Implementation

We have implemented the ModelQuery system (MQ) to realize VMQL (see [49]). It is a plugin to the MagicDraw UML™ CASE tool (see www.magicdraw.com) and uses SWI Prolog and the JPL Java-Prolog-Bridge library (see www.swi-prolog.org). Fig. [15] shows an architectural overview of MQ and the main data flows.

MQ provides not just a query facility for VMQL, but also provides additional syntactic sugaring by way of specialized query diagrams for class and activity queries. MQ does not yet implement all constraint types and lacks several features necessary for industrial usage, such as robustness, helpful error messages, online help, query optimization, and so on. While the user interaction is tool specific, the query engine and the model-to-Prolog conversion are not (i. e., the lower part of Fig. [15]). It should thus be fairly easy to port MQ to other UML tools, to DSL tools, or, in fact, to any CASE tool as long as it provides an open plugin API. Fig. [16] presents a screen shot of the MQ system.

6.1. Query execution

The process of executing a query is shown by the numbers in black/white circles in Fig. [15]. We will start with the white circles for transforming the model base first. These steps occur only once for all queries until the model is being modified, which currently requires to repeat them.

1. A source model is created using some modeling language (UML in this implementation) and stored in the tool’s model base.

2. The model base is exported to a file. Here, we use MagicDraw’s built-in XMI export facility.

3. The file representing the model base is transformed into Prolog-predicates by the MX-tool (see [15]), as described in Fig. [13].

Observe that the XMI-to-Prolog transformation is bidirectional. It neither adds nor removes anything, it changes merely the model’s representation. This process is highly generic, covering a wide range of XMI versions, different tools’ dialects of them, and completely different formats, as found in, e. g., DSLs. To some degree, it is exactly this step that makes VMQL a generic approach. Using files allows for loose integration so that it is easy to replace MagicDraw by another UML tool, and UML by another modeling language.

Now we turn to the queries and their evaluation, presented as numbered black circles in Fig. [15]. Since VMQL queries are just annotated fragments of regular
Figure 15: The architecture of the ModelQuery system and the main data flow of executing queries. Numbers in black circles indicate the sequence of steps in creating and executing a query. Numbers in white circles indicate the steps for creating or changing models in the model base. Rectangles are used to represent data, rectangles with rounded corners are used to represent actions. Arrows indicate data flow.
Figure 16: A prototype implementation of VMQL. In the foreground, MQ presents a list of all queries defined at that point (top left). The highlighted query is shown explicitly at the top right, the results of this query are shown below. One of the results is highlighted, and a diagram in which this binding appears is shown in the background, the binding elements highlighted by bold green outlines.
models, executing a query on a given model base boils down to finding matches between the query and the model base, and checking the constraints provided by the annotations.

① A query is entered as a regular UML model with constraints packaged in stereotyped comments.

②, ③ Then, the model query is exported to XMI and transformed into a Prolog predicate, just like the source model. Constraints are directly mapped to predefined Prolog predicates and added to the query.

④ Now the predicate resulting from translating the model query is run on the Prolog-database resulting from transforming the source model. The two models are matched and the constraints are evaluated.

⑤, ⑥ Finally, the user selects one of the matches found in the previous step. To support this, a list of all diagrams containing elements of the match is computed. These may be either exact or approximate matches as controlled by the precision constraint. If appropriate, the list is sorted by decreasing similarity. The diagram selected in the previous step is presented, and all elements of the binding are highlighted with fat green outlines (like in Fig. 4). The user may return to the previous step and select another match, and eventually terminate this query.

Translating a query (step ③) is a two-step process. Firstly, the model query is transformed into Prolog code (via XMI), just like a regular source model. Secondly, the constraints are translated into predefined Prolog predicates. The constraints are translated one-to-one into calls to predefined Prolog predicates that are used in the process of computing matches. Adding another constraint to VMQL, thus, effectively means adding one more predicate. That is to say, it is extremely easy to extend VMQL, or to adapt it to some other modeling language that may have different modeling concepts or different meta models. This also makes the implementation extremely compact and elegant, as most of the process is handled by Prologs built in facilities.

7. Existing model query facilities

Broadly speaking, there are eight classes of facilities that may be used to retrieve information from a (large) model (see Fig. 17). Mostly, they have not been created specifically for this purpose.
**CASE tool APIs** Most CASE tools like *Rational Rose, Sparx Enterprise Architect* or *MagicDraw UML* provide an application programming interface (API) that may be used to query models, among other things. Obviously, such facilities potentially offer unlimited expressiveness but require a user to express a query as a program, using some particular programming language (Visual Basic, .Net, and Java, respectively, for the tools just mentioned), the API as such, and a programming environment. Typically, these incur considerable technical complexity, which make it fairly difficult to run ad hoc user queries against a model. So, only very proficient users will be able to query models using an API; modelers from the business domain will typical not be able to take advantage of it.

**Low Level Query Facilities** Instead of programming a query in some CASE tool, one may also consider to work on the models as given in some underlying representation, such as an XML file or a relational databases. Such low level query facilities include XPath, XQuery, SQL, QBE, and so on. Like APIs, these approaches also provide extensive control over how a query is executed and impose little or no restrictions in terms of expressiveness. On the other hand, they are just as difficult to use as APIs for ordinary modelers. End users
will not be able to use them, and certainly not for ad hoc querying.

**Full Text Search** On the other hand, many tools also offer full-text-search that might be used for querying models. Obviously, such facilities are extremely easy to use, for all types of users, but they provide only very simple types of queries. On the one hand, the result depends on how the full text search is implemented, so there is no way to assess the recall (cf. [34]). On the other hand, domain-relevant keywords typically occur in many places, so that precision is diminished. In other words, full text search on models produces both many false negatives and many false positives. Extending full text search by wild cards and regular expressions does not help, either.

**Model Visualization Tools** In this category, we collect all approaches to collect information from a model and present it in a way that shall support a human user in inferring the information needed. This includes proper visualizations like inheritance or aggregation tree views as offered by many case tools, automatically generated overview diagrams presenting call hierarchies, wallpaper-like views of very large models, and traditional reports, effectively serving the same purpose. Model visualization tools can be seen as a restricted class of predefined queries.

**Predefined Queries** Many tools offer the most common queries as predefined commands. Obviously, such queries are easy to use, and they may be arbitrarily complex. However, it is not possible to create new queries ad hoc. Adding parameters and logical connectors increases the flexibility somewhat (cf. the ADONIS tool [20]), but are of little help without abstraction mechanisms like quantifiers, and type variables.

**Domain Specific Query Facilities** Powerful query facilities have been introduced that achieve both expressiveness and usability by sacrificing genericity, that is, by restricting the approach to specific application domains (cf. [14][2]). Transferring such an approach to a different application domain tends to be very difficult, or even impossible, so that, for instance, DSLs or generic languages like UML, IDEF, or ARIS/EPCs will not be supported very well by such an approach. VMQL, on the other hand, tries to be as generic as possible (see Section 8.3).

The only types of model query facility that (potentially) offer a large or even unlimited expressiveness for queries, combined with large genericity are textual and
visual query languages. We will treat these last two classes of query facilities in greater detail now.

Textual languages include the Object Constraint Language (OCL, [31]), query extensions of OCL (cf. [1]), the Model Manipulation Toolkit (MoMaT, [44]), the Logical Query Facility (LQF, [46]), the UML Model Transformation Tool (UMT, [30]), the Model Transformation Language (MOLA, [22, 21]), the Atlas Transformation Language (ATL), or the OMG’s Query-View-Transformations standard (QVT, [35]). As outlined before, a textual query language does not blend well with a visual modeling language, and formalisms like first order logic are not acceptable to domain modelers. Also, experience shows that the languages mentioned are quite difficult to handle, even for experts. In the case of OCL, we will substantiate this observation by experimental results later on. Since the OCL is probably the best known of these approaches, we take it as the gold standard for comparing usability. While visual languages are not necessarily any better usable than textual queries, they often have more appeal to potential users. Visual OCL [6] only visualizes an OCL expression used for querying, not the query as such: from a visual OCL diagram, only the structure of the OCL expression may be inferred, not the structure of the query result. Additionally, all approaches mentioned in this paragraph have a certain version of the UML meta model hardwired into them. Any changes to the meta model will break some queries.

Both of these disadvantages are avoided by Constraint Diagrams (CD, [24]). Here, the concrete syntax (which is much less likely to change than the meta model) is used to formulate a query. Unfortunately, Constraint Diagrams are rather limited with respect to the modeling language they may be applied to (basic class diagrams). Also, it seems that Constraint Diagrams have never been implemented as a tool.

The BP-QL language (see [5]) allows visual queries in a similar fashion than VMQL, but is restricted to BPMN. However, it also allows queries along semantic relationships like refinement and calling, a feature which is currently rather cumbersome to match in VMQL. Unfortunately, this feature relies heavily on a semantics of BPML that has been defined for this purpose in [5]. Therefore, it can not be used for other languages. In fact, it may not even be compatible with the implementation of other BPMN/BPEL tools since the semantics of BPMN/BPEL is currently not standardized. So, query results yielded by BP-QL may be inconsistent with the underlying BPEL engine.

Probably the approach closest to VMQL are Query Models (QM, [41, 40, 42]). Like Constraint Diagrams and VMQL, QM uses a variant of the host language to express additional constraints. Unlike VMQL, however, query models are supposed to be translated into OCL. The QM approach does not provide a generic translation
of queries into OCL, it only gives a handful of examples how such a translation might look like. So, QM is more like a visualization of certain predefined OCL queries. Also, QM has only been elaborated for a few examples of class and sequence diagrams in the context of aspect-oriented modeling. It is not clear whether it may be generalized to other notations and other types of queries. QM has never been elaborated to the point of an implementation that actually executes a query.

8. Evaluation

So far, we have described the elements of VMQL and how it was implemented. In this section, we will assess the contribution of VMQL relative to the requirements for ad-hoc model query facilities. To this end, we will first define these requirements (Section 8.1), and then analyze VMQL with respect to them in turn (Sections 8.2 to 8.6). Using these criteria, we also compare existing model query facilities as discussed in Chapter 7 with the VMQL evaluation results from this chapter (see Table 3).

8.1. Requirements for ad-hoc model querying facilities

Practical experience suggests that the following are the most important requirements for facilities to support ad hoc querying in visual modeling language environments.

expressiveness Any query facility should impose as few restrictions as possible on the scope of queries that may be expressed and evaluated by it.

genericity Any query facility should be applicable to as wide a range of languages as possible, including at least common dialects and variants of the most important languages (such as different versions of UML or UML profiles), possibly a broad range of different less common languages, and maybe even as-yet undefined languages like future visual DSLs.

usability Any query facility must support users with little or no technical background, and no time and motivation to undergo additional training to be able to use the query facility.

practicality Any query facility must be actually implemented to be of any use, and ideally integrated into one or more common CASE tools, or at least easily adaptable.
Table 3: A comparison of the different model query facilities discussed in Chapter 7 according to the criteria defined in Section 8.1, grouped by what is used for queries.

<table>
<thead>
<tr>
<th>APPROACH</th>
<th>TYPE</th>
<th>SEM-Antics</th>
<th>EXPRESSIVENESS</th>
<th>GENERICITY</th>
<th>PRACTICABILITY</th>
<th>PERFORMANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>predefined</td>
<td>selection</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>code</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>code</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>code</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>code</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>code</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>formula</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>formula</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>visual OCL</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>formula</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCL</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCL+</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>visual OCL</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>formula</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>diagram</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>diagram</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>diagram</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>diagram</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>diagram</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>diagram</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>diagram</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>diagram</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. depends on tool
2. tool does not execute queries
3. implementations do not support visual notation of standard
4. not standardized
5. not implemented
6. on top of OCL, thus restricted by performance of OCL
7. depends on variant/tool
8. according to [11]
9. according to several MDA experts
10. only examples of a translation from QM to OCL specified
**performance** Any query facility focusing on ad hoc querying must work at acceptable levels of performance. So, firstly, response times have to allow for interactive work. Secondly, recall is more important than precision: there is always a human being interpreting the results.

Obviously, the ideal query facility would combine the highest possible levels of expressiveness and genericity with the best usability for the broadest possible audience plus a convincing integration into whatever environment is at hand, executing queries within milliseconds.

In the next sections, we will discuss how VMQL satisfies these requirements, and compare it to other approaches. We will mainly focus on comparing VMQL with OCL and QM as the closest competitors for VMQL. While OCL has not been designed with ad hoc querying in mind, it may be used for this purpose, and it is the only language in this comparison that is of any practical relevance at all. QM is similar to VMQL from the visual point of view, but has never been implemented.

### 8.2. Expressiveness

The expressiveness of a language can be analyzed from a semantic (or logic) and from a pragmatic viewpoint. We discuss the logic viewpoint first. Intuitively, a VMQL query without constraints can be seen as a formula of the first order predicate calculus (FOPC) as follows.

![Diagram](image)

\[
\begin{align*}
A(x, y) & := (type(x) = Class) \land val(x, name) = Foo \\
& \land (type(y) = Class) \land val(y, name) = Bar \\
& \land \exists g : (type(g) = Generalization) \\
& \land x \in link(g, general) \\
& \land g \in link(y, ownedMember)
\end{align*}
\]

Here we have reused the names from Definition 1 in a loose sense as logical predicates. This mapping can be extended to also cover the VMQL constraints `mattre`, `mclass`, `match`, `not`, `strict`, and `distinct`. In fact, this is also possible for the constraint `steps` (and `indirect`), as the following example shows.

---

7The optimization constraints (`optional`, `either`, `once` and `precision`) do not affect the expressiveness and may be ignored for this consideration.
Whether or not VMQL is less powerful than FOPC, and if so, which fragment of FOPC best characterizes VMQL is currently unknown. In particular, there may be a Description Logic (see [28]) that is close to VMQL. However, most extensions of the \texttt{steps} constraint will exceed FOPC. For instance, querying for shortest or longest paths rather than any path requires a fixed point operator. Also, using variable lengths determined by some other expression will require quantification over sets of variables. So, as it is now, VMQL has at most the expressive power of FOPC.

From a pragmatic viewpoint, expressiveness can be assessed by analyzing and comparing the language constructs. Looking at the other diagram based model query facilities surveyed in Section 7 above (i.e., CD, BPQL, and QM), there are two ways, in which VMQL is more expressive. First, VMQL allows to express queries on all types of models in a language while the others only allow one (CD, BPQL) or two (QM). Second, VMQL provides constraints that are not expressible in the other languages, most notably \texttt{match, mclass/mattr}, and \texttt{steps/indirect} (QM offers a limited variant of \texttt{indirect}).

VMQL and OCL are not directly comparable, since both allow queries the other cannot express. For instance, there is no way to encode arithmetic expressions or arbitrary recursive functions in VMQL, which is easy to do in OCL. In fact, OCL can express any computable function (cf. [26, 8]), which VMQL cannot. Beckert et al. have translated OCL into FOPC so we know that OCL is at most as expressive as FOPC (see [4]). On the other hand, since OCL quantifiers are always bounded, OCL must be strictly weaker than FOPC. Schmitt has defined iteration logic $\mathcal{L}_{it}$ as a formalization of OCL and could show that

$$\mathcal{L}_{FO} \leq \mathcal{L}_{it} < \mathcal{L}_{tc}$$

where $\mathcal{L}_{FO}$ and $\mathcal{L}_{tc}$ denote first-order logic and transitive closure logic, respectively (see [36]).

On the other hand, VMQL may express meta level queries like \texttt{Query 16} (see Fig. 18, "Find all elements named 'Foo' of any type!"). Due to its type system, OCL only permits strictly typed expressions, which is somewhat limiting for interactive use. Finally, VMQL allows approximate queries using the \texttt{either} constraints (see
queries 13 and 15), and it even allows direct control over the precision by using the `precision` constraint (see Query 17). As far as expressiveness is concerned, all comparisons between VMQL and OCL also apply to the other OCL-based query languages, in particular, OCL\(^+\) and Visual OCL.

As far as OCL is concerned, this comparison is not entirely convincing, though: OCL is a compiled textual language with a static type system originally intended to express constraints, whereas VMQL is an interpreted generic visual language with a dynamic type system created for ad hoc querying.

8.3. Genericity

Almost all of the examples we have seen so far are on UML 2.2. class diagrams, because this is probably the most widely known modeling notation used today. However, VMQL is generic in the sense that it works for all of UML, and also many languages besides UML. In fact, since VMQL uses essentially the same notations for specifying models and for specifying queries on them, it may be used for almost any visual modeling language, including ones that have not been defined yet (i. e., visual DSLs). VMQL only requires that there is something like textual comments to annotate elements with constraints. In Fig. 19, we show some queries that use other UML notations besides class diagrams, including UML profiles like SysML or custom-defined profiles, and all their elements, e. g., new stereotypes and custom notations.
Other model query facilities do not share this property. For example, a query expressed in OCL 1.1 for a UML 1.4 model is restricted to just this version of OCL and UML. Moving it to another setting might break it due to the changes to the meta model. Since the concrete syntax of UML changes at a much lower rate, approaches that start there are less susceptible to that type of change.

We have applied VMQL to about 70% of the concepts and notations of UML 2.2, SysML and other UML profiles. We have also used VMQL manually for notations as diverse as Event Process Chains (EPCs, cf. [9, 23]), the Business Process Modeling Notation (BPMN, cf. [33]), all of IDEF, MSCs, SDL, BON, a range of DSLs, and several others. Fig. 20 shows an example of the same query expressed as annotated class diagrams in UML notation, Coad/Yourdon notation (see [12]), and Martin/Odell notation (see [27]).
Note that it is not only the concrete syntax that differs between these notation, but also the concepts: the Coad/Yourdon meta model for class models consists of ten meta classes with 23 meta attributes and ten meta associations (see [12, p. 205]), while the corresponding fragment of the UML 2.1.2 meta model consists of no less than 55 meta classes with about 40 meta attributes and well over 100 meta associations (excluding inherited concepts and features, see [32]).

In Table 3, we rank the genericity of query APIs and sets of predefined queries lowest, as they only apply for a single tool. Constraint Diagrams and BPQL are defined only for a single type of models (static structure and process models, respectively), while Query Models incorporate both UML class and sequence models. OCL-based approaches work for a whole family of languages like UML, but are restricted to a fixed and MOF-compliant language. Also, not all of UML is covered equally well. VMQL covers all of UML and similar visual notations, but some of the constraints and defaults defined in VMQL are specific to UML so that they must be adapted before they can be applied to other languages, thus limiting the genericity of VMQL. In contrast, LQF and MoMaT provide a somewhat higher degree of genericity at the price of less usability.

8.4. Usability

We wanted to establish the degree of usability of VMQL as compared to OCL and the Logical Query Facility (LQF), the Prolog-based query language we have developed in previous work (see [46]). To this end, we conducted a series of controlled experiments. We first drafted a questionnaire on reading tasks. As described in Section 3.3, the queries used to construct these tasks were derived from a collection of queries originating from industrial large scale modeling projects. That is, all queries are essentially real queries, although some modifications and simplifications had to be made for this experiment. For instance, real names have been replaced by ‘XYZ’ to avoid shortcutting. An example of a result set description is ”The set of all classes associated to class ‘XYZ’”.

We ran the initial questionnaire with five student participants and conducted in-depth follow-up interviews. Analyzing the misunderstandings and errors participants had made led to considerably enhanced instructions and modified tasks. We soon realized that OCL was not very popular with the study participants. In fact, some of them were so dissatisfied with OCL that they simply refused to complete the OCL

---

8 More information on one of these projects is available in [48]. The author was participating in this project between 01/2004 and 05/2009 as a consultant, coaching end user modelers in modeling and methodology.
So, in order not to de-motivate future participants, we dropped OCL from the comparison.

However, several study participants pointed out that the comparison between OCL and VMQL is inadequate anyway because of the very different levels of abstraction they operate on. In order to provide a fair comparison, we enhanced OCL by a special purpose library mimicking the concepts provided in VMQL. We call this language OCL+. So, all in all, the experiments covered four languages: VMQL, LQF, OCL+, and, initially, plain OCL. To our knowledge, there is no other publicly available OCL library suitable for querying.\textsuperscript{10} Later on, a writing task was added to the questionnaire.

The experiments were run between May and December 2009 with 38 participants altogether, 31 of which were graduate students at the time, the rest were IT professionals with 3 to 15 years of job experience. Six data points were excluded (the five subjects participating in the prototyping phase, and one participant who seemed to have filled out the questionnaire without reading the instructions). 19 of the study participants were tested on writing queries, all of them were students. All subjects participated voluntarily for no reward with complete anonymity. No oral instruction were given: the questionnaire contained an instructions sheet as its first page. The experiments were run in Munich, Augsburg, and Lyngby, partially by the author, partially by a colleague, and partially by the participants themselves. The tasks were given to the participants in three different sequences by random assignment. The number of tasks was the same for all experimental conditions. Taken together, these measures ensure a high degree of validity of our results.

The experiments consisted of two query reading-comprehension tasks; the latest experiments also contained a query creation task. All tasks compared the query languages listed above. We asked subjects to work on three types of tasks: a difficult reading task (Task A), an easy reading task (Task B), and a writing task (Task C). The reading tasks referred to a list of eight queries for each of the languages compared in the experiment. These lists were partially overlapping, that is, some of the queries were expressed in one or two different languages. In task A, subjects were presented with ten expressions describing a set of model elements and were asked to

\footnotesize

\textsuperscript{9}Some original quotes from follow up interviews include "Language C really sucks" and "it’s a pain in the arse". During the experiments, the languages were anonymized, C was the name for OCL.

\textsuperscript{10}The OCL predicates defined by the UML standard are not sufficient to lift OCL to the same level of abstraction as VMQL. Akehurst and Bordbar show only how to emulate SQL-like behavior in a previous OCL version, see \cite{1}.  

46
note which queries for any language produced such a result. For each task, between
zero and 4 correct answers were possible. In task B, subjects were presented with
twelve expressions as in task A together with a query from the query list and were
asked to assess whether or not the query and the result matched.

Apart from the issue mentioned already with plain OCL, task completion was
generally high (91% and 96% for reading tasks, and 58% for writing tasks). For
Tasks A and B, tasks were either not completed, or the answer was right or wrong.
In the writing task, the answers were awarded between 0 and 4 points for how close
they came to the right answer. Fig. 21 provides an overview of our experimental
results. Only the results for VMQL and OCL+ are shown, results for plain OCL are
much worse than for OCL+.

The experimental results consistently show better results for VMQL tasks than
for OCL tasks of all types: the subjects made less errors, gave more correct answers,
and tried to answer more tasks for VMQL than for OCL+ (see Fig. 21 a through
c). Unsurprisingly, we found that subjects made more mistakes and gave less correct
answers when asked to write queries rather in comparison to reading them. But there,
too, VMQL tested much better than OCL+ (Fig. 21 c). The most striking result is
the high number of uncompleted OCL+-writing tasks. We could not conduct follow
up interviews with many of the subjects on this finding. Two possible explanations
seem to be likely: subjects either had not enough motivation or they saw not even
a remote chance of succeeding. Either way, subjects performed much better with
VMQL. Additionally, on closer inspection, most of the mistakes made under the
VMQL condition in this task could have been easily avoided with a small introduction
to VMQL. So, the margin is likely to increase significantly when investigating this
issue further.

These factual results coincide nicely with the personal assessment asked by the
subjects (see Fig. 21 e & f). Here, we asked for absolute judgments on a five-point
Likert scale. This often skews data towards the middle value. On average, subjects
agreed that both reading and writing VMQL was significantly easier than for OCL+,
and again, the margin was higher for writing than for reading. This finding was
confirmed by the personal confidence subjects reported for the respective languages

The grading scheme is 4 points for exactly right answers; 3 points for answers with minor errors;
2 points for answers with major errors; 1 point for some elements of an answer; 0 points for nothing
recognizable at all.
Figure 21: Summary of experimental results studying the usability of VMQL.
(and again, lower confidence for writing than for reading). Finally, VMQL is also the winner in terms of popularity, as Fig. 21 (f) shows.

In order to obtain qualitative data, too, we asked participants to conduct a brief follow-up interview. Fourteen subjects agreed to this, seven of which were practitioners. We particularly asked for an introspective account of how they tackled the different languages which nine were able to explain in reasonable detail. Unanimously, these nine subjects reported very similar experiences: for OCL queries, subjects described their method as a kind of stepwise mental execution of the expressions. For VMQL queries, on the other hand, subjects reported that they first scanned the set of all possible expressions provided with the questionnaire to narrow the field down to a few likely candidates per task, and then selected their answers by checking some details. Five out of seven practitioners reported that instead of just checking a few details for VMQL queries, they systematically evaluated them mentally, too. For the writing tasks, most subjects reported that they reproduced parts of the queries they had been provided with examples, adapting them to the task at hand.

For the reading tasks, there are no apparent error patterns. Without a better understanding of how queries are executed mentally, it is hard to classify the errors made. Future work will have to resolve this issue. For the writing tasks, subjects reported that they were lost in “a forest of symbols”, as one of them put it. Creating a query by selecting from the exemplars considered before and adapting it in a correct or nearly correct way was apparently very difficult. Many subjects simply gave up (completion rate 32%), some students also with an expression of frustration. For VMQL, the completion rate was more than twice as high (68%). Subjects still made many mistakes but most of these were only slight mistakes like forgetting to specify some detail, omitting a variable for the result, or omitting a distinct constraint which would yield the desired output, plus additional false positives. Even so, however, there are still more correct answers for writing VMQL queries (41%) than correct and incorrect attempts at writing OCL queries combined (32%).

One explanation for the results on writing queries might be that it was easier for subjects to memorize VMQL queries due to their visual nature, which would be consistent with the general intuition on visual languages. Interestingly, however, several subjects reproduced appropriate VMQL queries in writing tasks but changed their layout as compared to the original, indicating that there might be other processes involved as well. Future experimentation will have to clarify this. Also, since subjects assessed OCL tasks as much harder and time consuming than VMQL tasks, it appears that some subjects have been seduced to apply less diligence to the VMQL tasks than to the OCL tasks. Future experiments may be able to solve this question.
Table 4: The case studies on which VMQL was evaluated.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Model Size</th>
<th>Queries</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ME</td>
<td>D</td>
<td>CD</td>
</tr>
<tr>
<td>Library</td>
<td>177</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>Insurance</td>
<td>282</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Vehicles</td>
<td>505</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>UML 2.2</td>
<td>1587</td>
<td>109</td>
<td>15</td>
</tr>
<tr>
<td>LF4</td>
<td>10293</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Analyzing the different batches of results by different experiment leaders showed no significant differences, very likely an effect of using only (identical) written instructions. Analyzing the results of practitioners vs. those of students showed generally better results for practitioners: they usually made no mistakes at all, for all languages, but gave similar feedback concerning effort and confidence as the students. Practitioners achieved higher completion rates on the tasks they were tested on than students, but this may be due to the different experimental situations (classroom/mail vs. one-on-one).

In Table 3, most code approaches are ranked with the lowest usability because they are the least suitable for end user modelers. LQF and OCL+ rank better because they provide adequate abstractions allowing at least to understand existing queries. VMQL provides the same abstractions, but better syntax as demonstrated by the experimental results. Predefined sets of queries are ranked best, though, as these are the easiest to use.

8.5. Practicality

In Section 8.3, we have listed many notations to which the VMQL approach has been applied. We have also tested VMQL in practice; Table 4 provides an overview of the case studies we ran with VMQL. The model sizes are given by the number of model elements (ME) and diagrams (D); the number of queries are split up into class diagram queries (CD), activity diagram queries (AD), and all other types of queries.

With the MQ system, our implementation of VMQL (see Section 6), we have also studied VMQL in practice which led to a great number of improvements, additions, and changes, so that currently, MQ must be considered obsolete. Therefore, we have not yet run real industrial case studies in industry, which is part of our future work.

However, using MQ, we have not encountered any specific problems in terms of
extending it towards other languages, and we can’t think of any reason why it should not be possible to use VMQL for other visual modeling languages and to integrate it into other tools. Given the architecture of MQ and the nature of our approach, this should actually be fairly straightforward.

In Table 3, we have ranked the practicality of competing model querying approaches based on the availability of an implementation, its integration in a CASE tool, the validation of the approach in realistic case studies, and whether or not it is usable for languages with practical relevance.

8.6. Performance

For an interactive system, performance is a critical requirement. In the following, we refer to the steps shown in Fig. 15. All measurements were taken on the author’s subnotebook computer (Centrino Duo, 1.2GHz, 2GB Ram, Windows XP). The manual steps (1, 3, 5) are not taken into account. The steps executed by the host CASE tool (MagicDraw in the current setting) dominated the query process: visualization (step 2) was in the order of a few seconds, but exporting a model and the queries (step 2/2) to XMI takes between 10 and 20 seconds for small scale models, and between tens of second or even a few minutes for medium scale models. A drastic improvement should be possible here based on a better understanding of the MagicDraw API. Step 3/3, on the other hand, takes much less than a second for small and medium scale models, and may reach a few seconds for large and very large scale models.

The query processing as such (step 3) varies mostly with the query size, and to a lesser degree with model size. If there are query results at all, the first results usually start arriving within one or two seconds. If this is not the case, or if the complete set of query results is relevant to the user, an exhaustive search is conducted. There, small queries like Query 3 (Fig. 7) are also terminated in less than a second, while complex queries like Query 9 (Fig. 8) may take up to a minute for large scale models. In [11], the LQF-library underlying VMQL has been tested approximately an order of magnitude faster than highly optimized, state-of-the-art OCL queries, irrespective of the model size.

Nevertheless, enumerating all bindings between a query model \( q \) and a base model \( m \) has a worst case complexity in \( O(|m||q|) \) where \( |m| \) and \( |q| \) are the numbers of model elements in \( m \) and \( q \), respectively. So, increasing the query or forcing the system to exhaustively search for all matches can easily result in run times beyond practical boundaries. Also, constraints like indirect effectively compute a transitive closure. Obviously, thus, it is easy to create enormous search spaces.

However, as the measurements reported above show, it is possible to reduce
processing times for realistic cases to an acceptable level. This is achieved by using some heuristics exploiting the structure of the models, including enumerating only bindings with pairs of identical types, restricting the bindings to particular local patterns of model elements (i.e., only some particular types, sizes, properties, and branching structures), and presenting initial results while looking for further matches in the background. The user can also help pruning the search space by several measures.

• Reduce $|q|$ by providing a smaller query whose elements are of different types. For instance, Query 3 contains only the one model element absolutely needed.

• Reduce $|m|$ by providing a query scope smaller than the overall model. For instance, Query 18 (Fig. 18) will further reduce the run time of Query 3.

• Add detail to query model elements, in particular unique detail and concrete names. For instance, Query 2 contains more detail than Query 1 (e.g., names and types for all features) and may thus be executed more quickly.

• Execute tight constraints first and loose constraints last. For instance, the ordering of constraints when executing Query 11 (Fig. 9) is relevant. The optimal sequence would be to first execute distinct, then the two mclass constraints, then the two mattr constraints on the subclasses. Finally the mattr constraint on the super class (which is not really a constraint at all in this query) should be executed last.

• Avoid or defer "expensive" constraints. The constraints precision, indirect, steps, either, optional tend to inflate the search space, slowing down query execution.

• Add or fast-track "cheap" constraints. The constraints distinct, strict, not, and name tend to reduce the search space, speeding up query execution.

• Avoid <: and * in the mclass and mattr constraints.

Finally, in an interactive query system, the worst case complexity for the last answer is much less important than the latency between issuing the query and receiving the first answer. If the answer to a query is obtained as one of the first results, then the user will not even notice that getting all answers may be very time consuming. A future implementation of VMQL may analyze queries for their expected runtime before executing them and optimize them, or warn the user, if a long computation my be expected.
In Table 3, we can provide only partial information concerning performance, because some of the approaches surveyed here are not implemented. Also, there are no benchmarks for these tasks. The programming based approaches are assumed to be (potentially) the fastest while [11] show that OCL is inferior to MoMaT, so we assume the same holds for OCL-based approaches (i.e., QVT, OCL+, QM) vs. MoMaT-based approaches (i.e., LQF). QM is likely to require substantial additional time to translate query models into OCL before executing queries.

9. Summary

VMQL is a visual language for ad-hoc querying of large scale analysis level models as used and maintained by domain experts. While model bases to be queried may become very big, queries usually are rather small. We focus on this scenario because it is by far the most relevant use case for model querying in industry.

VMQL is much more expressive than simple text search using wild cards and regular expressions, but does not allow the degree of control a query API would. However, domain experts are usually not able to take advantage of APIs, and certainly not for ad hoc queries. In contrast, it is much easier to formulate queries in VMQL, than it is in OCL, even when equipping OCL with a library defining query predicates similar to those used in VMQL. Additionally, VMQL can be applied to any visual language, and small changes to the VMQL implementation presented above will suffice to adapt it to other languages and tools.

Finally, VMQL as a language is created in such a way that it facilitates all four stages of querying: query formulation, query execution, result visualization, and result interpretation (cf. Fig. 5). Many other approaches do not have this property. For instance, the Query Models approach only provides examples of query diagrams and their OCL counterparts (see [40]). The resulting OCL expressions are enormously complex, and it is not clear how the translation process can be automated, and in fact, this has not been done since the approach has been published only an editor for Query Models, but not a query engine. So the Query Models approach only addresses the first out of four steps to query models.

In contrast to similar approaches like Constraint Diagrams and Query Models, VMQL is actually implemented, even if our implementation is a prototype, and does not yet support all of the language features described in this paper. In contrast to (visual) OCL, and QVT, VMQL is not tied to a meta model, and the query structure is determined by the source model structure, not some formalism.

12 “There is no translation into a QVT query language or a similar thing.”, [39]
VMQL is as visual a language as its host language is. By using the concrete syntax of the host language for query specification and result presentation, our approach avoids a meta model and modeling language lock-in and guarantees minimal effort to read and write queries and results by the user. This is in stark contrast to most approaches based on attributed graph grammars (cf. [38, 25]) and their implementation (cf. Fujaba [7, PROGRESS [37]), that focus on working at the meta model level. It is precisely this difference that makes VMQL superior in terms of usability and popularity.

As it is now, VMQL has been applied to most of the notations in UML 2.2, SysML and some other UML profiles, BPMN 1.2 [33], ARIS/EPCs [13] and IDEF [29], or for individual visual languages like ER-diagrams, use case maps, BPMN, MSCs, SDL, SADT, and BON. VMQL has been evaluated in three ways. Firstly, the approach has been implemented as a tool which considerably smoothed the language and improved the whole approach by uncovering several blind spots such as how do best deal with great numbers of solutions, multiple bindings, and so on. Secondly, we applied VMQL to a real life case study. To this end, we gathered query-like questions modelers would have in the industrial projects that we have worked on, and have tried to formulate them using VMQL. This step again improved the language by adding some syntactic sugar, and some extra features of high practical relevance (e. g., the multiplicity-constraint). Thirdly, we ran a series of controlled experiments to establish in a quantitative way the degree

Our current implementation of VMQL is tightly integrated with the MagicDraw 16.5 CASE tool, but can be used with loose integration (data exchange by files) for VisualParadigm, Enterprise Architect, Fujaba, and BOC ADONIS. If MQ were to be adapted to another, non-MOF-based language, changes would have to be made only to the transform step in the lower half of Fig. 15. If MQ were to be adapted to another UML CASE tool than MagicDraw, changes would have to be made only those parts shown in the upper half of Fig. 15.

VMQL is not just a research vehicle but targets real world application. While the MQ prototype is certainly nowhere near actual usage in the field, it leads the way to such a future tool.

Acknowledgements

I would like to thank the anonymous reviewers for their detailed and insightful reviews, and Patrick Könemann, Andrew Fish, and Alexander Knapp for comments and suggestions along the line of this research project in the last years. Also, I am indebted to J. Edenhauser and M. Winder for creating part of the implementation
and all of those students and colleagues of mine that participated in the usability experiments.

References


