VMQL: A Generic Visual Model Query Language

Harald Störrle
Institute of Informatics, University of Munich, Munich, Germany
Harald.Stoerrle@pst.ifi.lmu.de

Abstract

Shifting the focus from code to models in software development brings into view model-related tasks such as querying which are not very well supported by current CASE tools. Existing textual query languages like OCL are often not acceptable for domain modelers. Also, most query languages suffer from a mismatch between models, queries, and results. The Visual Model Query Language (VMQL) tries to overcome this by using a modeling language also as the query language and result presentation language.

1 Introduction

Many software development approaches today use models instead of code for many purposes (e. g., model based and model driven development, Domain-Specific Languages (DSLs), Business process management). As a consequence of this shift of focus, tasks such as version and configuration management, consistency checking, transformations, and querying of models are much more common today than they used to be. Unfortunately, these and other tasks are not well covered in current CASE tools. But from practical experience we have learned that modelers dearly need, among others, an ad-hoc query facility covering more than just full text search and a set of predefined queries.

In prior work [11], we have explored several textual languages to query models, in particular OCL, SQL, and Prolog. All of these serve this purpose, with varying degrees of expressiveness, efficiency, and usability. However, all of them are textual in nature, so that there is a media mismatch between the models queried, the queries, and the results found. Also, all of these languages involve a substantial amount of formalism which makes these languages difficult to use for a fair number of (potential) modelers, because at least domain and requirements modelers are usually not software engineers. However, *any* modeler must be familiar with the modeling language at hand—after all, this is a necessary prerequisite for becoming a modeler in the first place. So, using the modeling language as the query and result presentation language would make querying models much easier for them.

In order to support ad hoc querying, the VMQL aims at expressing model queries as (annotated) models. Our approach is generic in that it is applicable to a wide range of visual languages, including all notations in UML, IDEF, ARIS, as well as traditional notations such as ER-diagrams, Petri nets or state machines, and even domain-specific languages not yet defined. In order to provide an optimal trade off between expressiveness and complexity, the syntax of VMQL is structured in two layers. The lower layer providing full expressiveness and little implementation effort at the cost of reduced usability. The upper layer provides syntactic sugar for the most frequent types of queries, providing very succinct and easy to use notations but incurring more implementation effort.

Most popular CASE tools like *Rational Rose*, *Sparx EnterpriseArchitect* or *MagicDraw UML* provide an application programming interface (API) that may be used to query models. Obviously, such a facility potentially offers unlimited expressiveness but requires a user to express a query as a program, using some particular programming language (Visual Basic, .Net, and Java, respectively, for the tools jut mentioned), and a complex API and programming environment. Thus, it is fairly difficult to run ad hoc user queries against a model, and so, only very proficient users will be able to query models using an API.

On the other hand, many tools also offer full-text-search, and/or a selection of predefined (parameterizable) queries. For instance, *MagicDraw UML* and *Adonis* provide full-text search and predefined queries. Such query facilities are rather easy to use, but offer only limited expressiveness. For instance, in Adonis it is not possible to query for all model elements of any type satisfying some condition (e. g., "has been changed yesterday"). Either way, both types of facilities—APIs and predefined queries—are specific to one given tool.

More generic approaches that define a proper query language independent of a specific tool include the Object Constraint Language (OCL, [6]), the Model Manipulation Toolkit (MoMaT, [11]), or the OMG's Query-View-Transformations standard (QVT, [8]). 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. Visual OCL [1] only visualizes an OCL expression used for querying, not the query as such: from a visual OCL diagram, the structure of the result can not be inferred. Additionally, all approaches mentioned in this paragraph have a certain version of the UML

978-1-4244-4876-0/09/$25.00 ©2009 IEEE

meta-model hardwired into them. Any changes to the meta-model will break some queries.

Both of these disadvantages are avoided by Constraint Diagrams [4]. 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.

Probably the approach closest to VMQL are Query Models (QM, [9, 10]). 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 diagrams into UML, it only gives a handfull of examples how such a translation might look like. So, QM is more like a visualization of certain predefined OCL queries, and inherits the respective problems of (visual) OCL. 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, and indeed, it is not clear how the translation from Query Models to OCL could be automated in general. Table 1 provides an overview comparison of the approaches discussed above. The usability scores shown refer to the ease of usage nonexpert users experience with ad-hoc-queries. Expressiveness scores are increasing with higher numbers of (dialects of) modeling languages and types of query.

The starting point of our work is the observation that providing interactive query facilities for modelers is essential in many modeling projects. However, many modelers are overwhelmed by the complexity of the modeling language they (have to) use, and can't cope with yet another, complicated language for queries (such as OCL or QVT), let alone query APIs. On the other hand, the full-text search and predefined queries provided by many tools are not nearly expressive and flexible enough.

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

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 experiences have confirmed this.

Using essentially the same notations for specifying models and for specifying queries on them means that this approach may be used for almost *any* visual modeling language, that is, not just all diagram types of the UML, but also all other visual language families (like ARIS [2], IDEF [5]) or individual visual languages like ER-diagrams, BPMN, MSCs, SADT and so on. And, more importantly, it is also applicable for visual languages not yet defined, i. e. Domain-Specific Languages (DSLs). In this paper, however, we will restrict ourselves to UML to mitigate the limited space and yet reach the broadest audience possible.

The rest of this paper is organised as follows. In the next section, we will go through the VMQL language and explain all notations, followed by a section with sample queries for a variety of modeling languages such as UML Class, Activity, and State Machine diagrams. Then we will explain how VMQL queries are executed and briefly portray our experimental implementation of VMQL.

2 Language elements of VMQL

In the remainder, we will use the source model visualised by the diagrams presented in Fig. 1. This model is extracted from the analysis level models of an industrial project; ll queries discussed subsequently will be run against it.

First of all, the VMQL language consists of the host language itself, with **all** of its notational elements that have a representation in the abstract syntax. Elements of secondary notation, such as implicit layout conventions, are excluded.[1] Some sample queries are presented in Fig. 2.

Sample Query 1 (see Fig. 2) contains a class `Product` and a `Person` with three attributes (`name` of type `String`, some attribute of type `Date`, and an attribute called `gender`). Running this query against the model base of Fig. 1 ought to produce exactly one result mapping as shown in the following sketch.

[1] Note 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. [7, Fig. 9.26, p. 190] which belongs to the primary notation while the vertical arrangement of use cases in a system is not.

Table 1. Approaches to querying models, ordered by decreasing score.

APPROACH	USABILITY				EXPRESSIVENESS			TOOL	
NAME [REF.]	READ QUERY	WRITE QUERY	READ RESULTS	QUERY AD HOC	MODEL TYPES	QUERY TYPES	QUERY LANG.	AVAIL-ABILITY	INTE-GRATION
VMQL (this paper)	●●●	●●●	●●●	✓	●●●	●●	diagram	✓	✓
predefined queries	●●●	●●●	●[1]	✓	●	●[1]	selection	✓	✓
Query Models [9]	●●●	●●●	●●●	×	●	●	diagram	×[2]	×
Visual OCL [1]	●	●	●	×	●●	●●●	diagram	✓	×
OCL [6]	●	●	●	✓	●●	●●●	formula	✓	✓
query APIs	●	●	●	×	●	●●●	code	✓	✓
MoMaT [11]	●	●	●	✓	●●●	●●●	code	✓	×
Constraint Diagr. [4]	●●	●	●	×	●	●●	diagram	×	×
QVT [8]	●	●	●	×	●●	●●	code[3]	✓	×

[1] depends on tool [2] tool doesn't execute queries [3] implementations don't support visual notation of standard

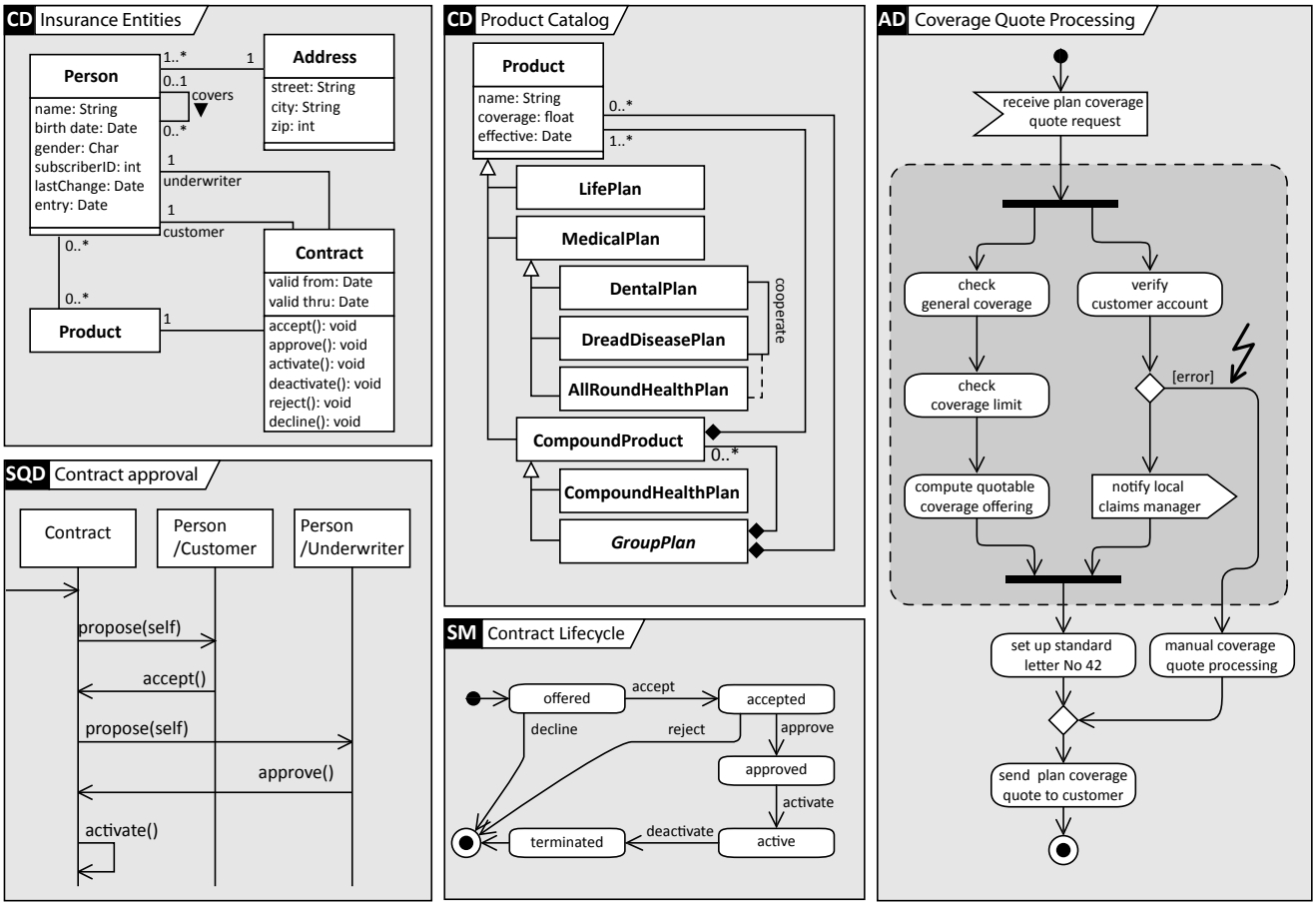


Figure 1. The model base against which all sample queries described in this paper are supposed to run.

Table 2. Overview of VQML constraints.

CONSTRAINT	applicable to		parameters
`name`	1	any type of model element	regular exp., var. name[1]
`ref`	1	any type of model element	variable name
`steps`	1	relationship	relational expression
`multiplicity`	1	relationship end	cardinality
`mclass`	1	any type of model element	set of meta-classes
`mattr`	1	any type of model element	meta-attribute, value
`precision`	$0, 1^2$	any type of model element	float

CONSTRAINT	applicable to	
`acyclic`	1	relationship
`irreflexive`	1	relationship
`distinct`	2	model elements of same type
`or`	n	model elements of same type
`not`	1	any type of model element
`optional`	1	any type of model element
`strict`	1	any type of model element

[1] : optional [2] : constraint without parameter refers to whole query

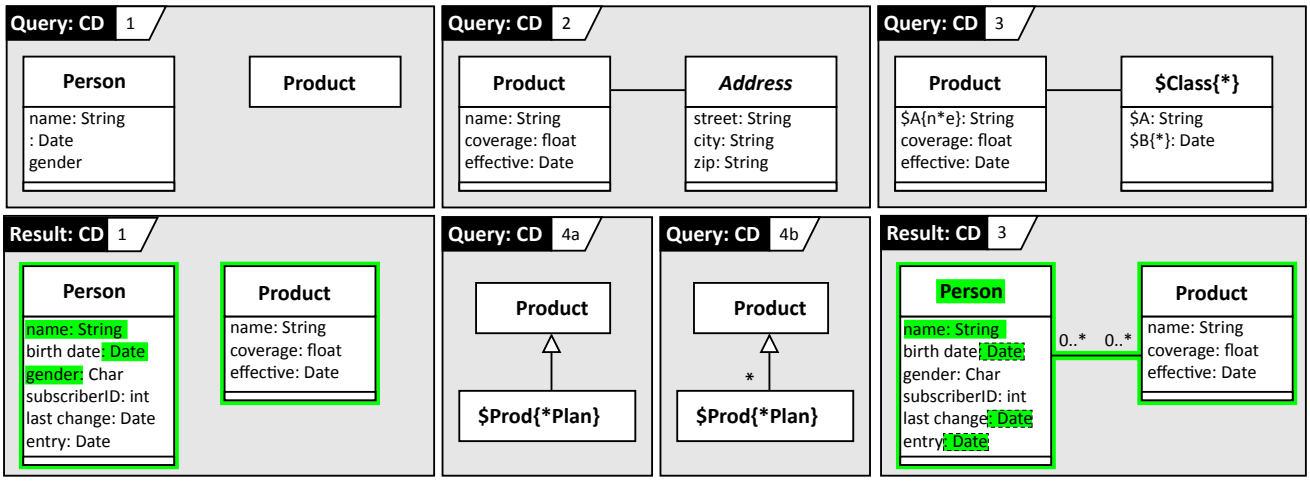


Figure 2. Some sample VMQL class model queries on the model base of Fig. 1, together with some of their results. Alternative bindings for Result 3 **are displayed by highlighting with dashed outline.**

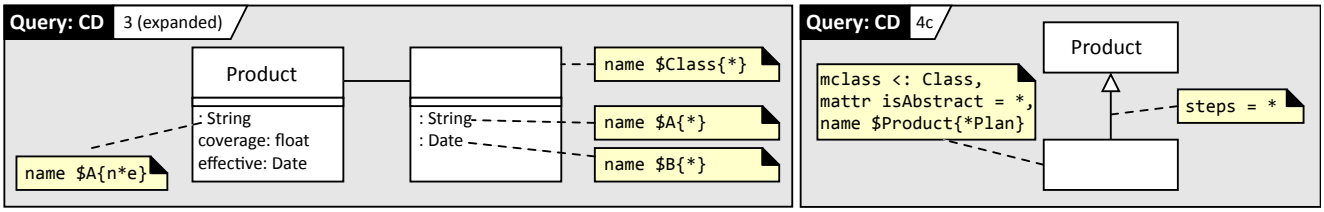


Figure 3. Two queries with expanded ("non sugared") syntax.

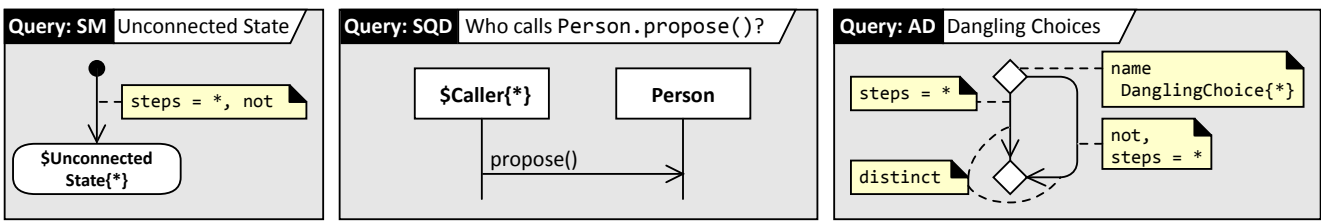
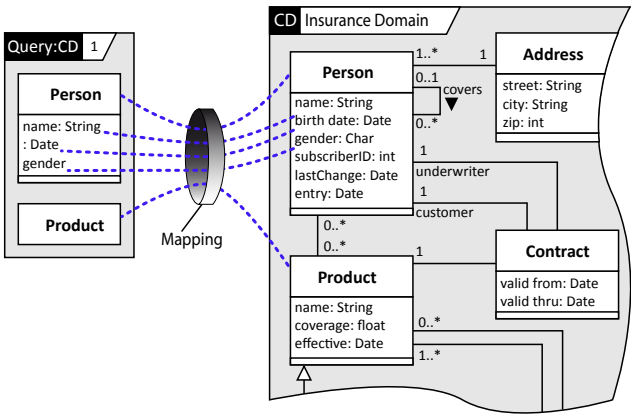


Figure 4. Queries for UML Activity and State Machine diagrams.



This query result could be visualised by highlighting, as shown in Result 1 in Fig. 2. Observe that the mapping covers just five elements from the model base: the classes named `Person` and `Product`, the attributes name, birth date, and gender of `Person`, and just those properties shown in Query 1. Additional elements or properties of the model base that do not occur in Query 1 are ignored.

Let's now turn to the next example. When running Query 2 (see Fig. 2), no mapping can be established, and so, no results will be returned. There are three points of failure: the model base does not have a class `Address` with (1) an attribute `zip` of type `String`, (2) an association to a class `Product`, and (3) with the value `true` for the meta-attribute `isAbstract`.[2]

The queries encountered so far are ground queries; that is, they do not contain variables. In Query 3 we introduce variables and element name match expressions. An expression of the form `$Var{regexp}` may replace the name of a model element. This expression declares the variable `Var`, and constrains the element's name to be in the set of strings described by the regular expression `regexp`. In Query 3, for instance, the expression `$A{n*e}$` used as the name of the first attribute will restrict the set of matching attributes in the model base to those whose names start with a "n", followed by an arbitrary sequence of symbols, and end with "e". So, Query 3 looks for pairs of associated classes with the following properties:

- both classes have `String`-typed attributes whose names match `n*e`, and the result is bound to the same variable `A` (i. e., the names coincide);
- one of the classes is named `Product` and contains the attributes `coverage` and `effective` of types `float` and `Date`, respectively;
- the other classes' name matches `*` and is bound to the variable `Class`. It has an attribute of type `Date`, whose name is bound to the variable `B`.

Thus, mappings quite similar to those shown in the sketch above are established. In fact, there are three possible mappings, that is, three solutions to Query 3. In order to save space in this presentation, the three alternative solutions have been combined into one diagram in Result 3 of Fig. 2. As there are free variables in this query, also a binding between variables and values is established. Each mapping corresponds to one of the following bindings.

Binding	`Class`	`A`	`B`
1:	"Person"	"name"	"birth date"
2:	"Person"	"name"	"last change"
3:	"Person"	"name"	"entry"

Variables may be used several times in one query, but they always refer to the same value in one result, though this value may vary for different results. All the usual features of regular expressions are allowed (i. e., option, selection, repetition, and wild cards).

Instead of replacing the name of a model element by an expression as described above, the constraints may also be expressed as a comment attached to the model element. We call this the *"expanded"* notation; Fig. 3 presents the expanded version of Query 3 as an example. In fact, the expanded notation is the primary way to define constraints, as it allows a greater variety of constraints, and it is also generally applicable: almost any modeling language will provide something similar to a comment attached to an element.

So, the notation presented above is just a form of "syntactic sugaring" to increase usability while restricting the generality of the approach and increasing the implementation effort. However, there are also drawbacks to the expanded syntax. For instance, non-unary constraints may not be expressed with the expanded notation if the host modeling language or tool does not support comments to be attached to two or more model elements simultaneously. In that case, syntactic sugaring may be used to overcome tool or host language restrictions.

Observe that constraints always override features in a model. For instance, if a model element has a name *and* a name constraint, the latter takes precedence. This feature is necessary when tools impose restrictions that interact with the query as such.

Now assume we are looking for all `Products` of our insurance domain that are `Plans`. At first sight, Query 4a in Fig. 2 appears like doing this, however, it yields only two bindings for the variable `Prod`, namely `LifePlan` and `MedicalPlan` because all other classes matching the name constraint are not direct subclasses of `Product`. In order to also access all indirect subclasses the steps constraint is introduced, see Query 4b in Fig. 2. This query yields five bindings as expected, because `GroupPlan` is

[2] In UML, using an italic font for the class name shows that it is abstract. This property is reflected by the boolean meta-attribute `isAbstract`, cf. [7, p. 52].

abstract and AllRoundHealthPlan is an AssociationClass rather than a Class in the UML meta-model. In order to also include these two in the result set, Query 4c (Fig. 3) is needed.

In order to include GroupPlan and AllRoundHealthPlan in the result set, two additional constraints must be put in place to form Query 4c (see Fig. 3). First, we must allow other UML meta-classes than Class to match our query. We could just enumerate the relevant meta-classes here (i.e., write mclass [Class, AssociationClass]). However, in order to provide more generality, we might prefer mclass <: Class to allow any *subclass* of the meta-class Class, which also includes AssociationClass. This feature is particularly useful for large generalization trees, e. g., for the Action or Behavior hierarchies. Second, we must allow any value of the meta-attribute isAbstract in a matching model element. This is achieved by the constraint mattr isAbstract *.

Obviously, using the mclass and 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.

We will now briefly explain the complete set of all constraint types currently defined in VMQL. Table 2 provides a synopsis. Due to the restricted space in this paper, we can't go into details here.

Identifier constraints refer to the proper names of model elements: name declares a regular expression to match the name of a model element, and ref declares a reference to a variable defined before.

Path constraints refer to edge-node-structures: steps gives a limit to the length of a path (as in steps < 42 or steps $= *$), acyclic and irreflexive are exactly as is to be expected from graph theory (irreflexive is really a frequent special case of distinct for relationships without cycle of 1 or more steps).

Metamodel constraints refer to the meta-model of the host language: mclass allows to define the set of matching model element types (i. e., meta-classes), mattr allows to specify the values of meta-attributes.

Match constraints refer to the process of matching: precision defines the degree of similarity required to consider two model elements as matches (default is 1), strict enforces that matches for query elements must not have additional parts and properties.

Identity constraints refer to the role of model elements: distinct enforces that two matches be non-identical, or joins alternative match candidates, optional declares an element as not essential, and not forbids designated elements.

To increase the usability, we allow several forms of syntactic sugaring. First, the unary constraint of Table 2 may be applied to several model elements at one time. This is supposed to mean that an identical copy of the constraint is applied to each of the model elements individually. Also, instead of attaching several constraints in their own boxes to one element, constraints may use one comment-box together to form a complex constraint. In that case, the individual constraints may be separated by commas (conjunction), or semicolons (disjunction). Brackets must be used to disambiguate terms.

VMQL is not restricted to querying class diagrams, or, indeed, any particular modeling language: every host diagram that allows constraint annotations similar to those presented above may be used as a query. See Fig. 4 for an example of a State Machine query (left), and an Interaction query (middle). These queries yield all states not connected to the initial state (Fig. 4, left); and all interaction partners calling method propose of Person (Fig. 4, middle).

Also, queries may be used for a variety of simple consistency checks, see Fig. 4 (right), where we ask for all choice paths that are not properly joined togehter again. Observe, that this query uses a name constraint for the query diagram as such – this is one feature difficult to implement for most CASE tools.

3 Query execution

Query models are annotated fragments of regular models. Executing a query with respect to a given source model essentially requires finding portions of the source model that resemble the structure and properties of the query model and also satisfy its annotations. Fig. 5 shows a synopsis of the query execution process.

First, the user creates the source models in the model base, exports them to an XMI file, and transforms it into a Prolog database (represented in Fig. 5 by numbers in white circles). These two steps occur once for all subsequent queries. For instance, the Prolog representation of class Address of Fig. 1 is shown below.

```
me(class-25,   [ownedMember-ids([26,29,31,33]),
                name-'Address']).
me(property-26,[association-id(89),type-id(35),
                visibility-'private',
                multiplicity-interval(1,*)]).
me(property-29,[visibility-'private',name-'street',
                multiplicity-1,type-'String']).
me(property-31,[visibility-'private',name-'city',
                multiplicity-1,type-'String']).
me(property-33,[visibility-'private',name-'zip',
                multiplicity-1,type-int]).
```

Each model element is transformed to one Prolog clause of the predicate me, whose first argument are a pair of type

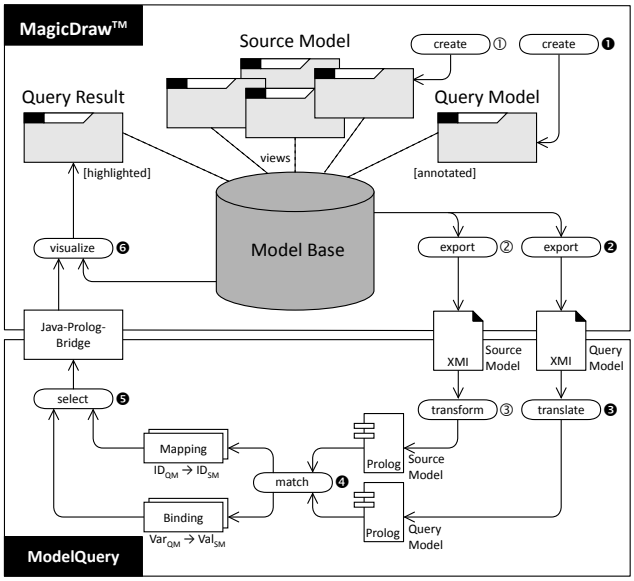


Figure 5. The architecture of the ModelQuery system. Numbers indicate the sequence of steps in creating and executing a query.

and internal identifier (usually an integer), and whose second argument is a property list of tags for meta-attributes and their values. References to identifiers are marked with an `id` or `ids`-term. The code has been slightly beautified for improved readability. The transformation is implemented in the MX-tool [3]. Observe that this transformation is bijective and content preserving. It neither adds nor removes anything, it changes merely the model representation. This process is highly generic, covering a wide range of XMI versions, different tools' dialects of them, and completely different formats e. g. from DSLs. To a large degree, it is precisely this step that makes VMQL a generic approach.

We now describe the process of creating and executing a query and visualizing the results. These steps are represented in Fig. 5 by numbered black circles.

❶ A query is entered either as a regular UML model with extended syntax annotations or as a sugared syntax query diagram, as described above.

❷, ❸ Then, the query model is exported to XMI and translated into a Prolog-predicate. The model elements are transformed by the same procedure as the source model. Constraints are directly mapped to predefined Prolog predicates and added to the query (details below).

❹ Now the predicate resulting from translating the query model is run on the Prolog-database resulting from transforming the source model.

❺, ❻ Finally, the user selects one of the matches found in the previous step. To support this, a list of all diagrams is computed, that contain elements of the match ("hits"). This list is sorted by number of hits. The diagram selected in the previous step is presented, and all hits are highlighted (like in Fig. 2). The user may return to the previous step and select another match.

Translating a query is a two-step process. Firstly, the query model is transformed into Prolog code (via XMI), just like a regular source model. Secondly, the constraints are translated into predefined Prolog predicates. For instance, consider again Query 4c presented in Fig. 3. It is transformed into the following Prolog code.

```
me(class-0,  [name-'Product']).
me(class-1,  [ownedMember-ids([2])]).
me(comment-3,[annotatedElement-id(1),
   body-'mclass <: Class,
         mattr isAbstract = *,
         name $Product {*Plan}']).
me(generalization-2, [to-id(0),from-id(1)]).
me(comment-4,[annotatedElement-id(2),
         body-'steps = *']).
```

The complex constraint with id 3 is translated to the following prolog clause

```
Constraint_1 = [
  mattr( q4c, 1, isAbstract, =, any),
  mclass(q4c, 1, '<:', Class),
  name(  q4c, 1, Product, '*Plan')]
```

where `q4c` is the model identifier, and `1` is the identifier of the model element to which the constraint refers. The predicates `mattr`, `mclass`, and `name` are predefined and check the respective constraint on the specified model element. Such predefined predicates may, of course, be of arbitrary complexity. Keep in mind that in Prolog, variables start with a capital letter.

Matching the constrained elements amounts to calling the following clauses

```
match(q4c, me(me(class-0,  [name-'Product'])),
      [],          Binding_0),
match(q4c, me(class-1,[ownedMember-ids([2])]),
      Constraints_1, Binding_1).
match(q4c, me(generalization-2, [to-id(0),from-id(1)]),
      Constraints_2, Binding_2).
```

And that is it – the Prolog engine handles all the rest.

4 Implementation

We have implemented the ModelQuery system (MQ) to realize VMQL (see [12]). It is a plugin to the popular MagicDraw UML™ CASE tool (see `www.magicdraw.`

com) and uses SWI Prolog and the JPL Java-Prolog-Bridge library (see www.swi-prolog.org).

MQ provides not just a generic query facility using extended syntax, but also 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. 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. 5). 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. For want of space, we cannot include a screenshot here. A more detailed version of this article is available at www.pst.ifi.lmu.de/~stoerrle.

5 Discussion

VMQL is a language to query model bases. It is as visual 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.

VMQL has been used manually for notations as diverse as the complete UML 2.1.2, BPMN, ARIS/EPCs, all of IDEF, MSCs, SDL, BON, a range of DSLs, and several others. We have implemented VMQL as a plugin for the MagicDraw UML tool. We also tested our approach in practice for about 70% of the concepts and notations of UML 2.2, SysML and other UML profiles, and BPMN. The following table 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.

case study	model size		queries		
	ME	D	CD	AD	other
Library	177	4	18	2	3
Insurance	282	5	6	10	8
Vehicles	505	1	20	12	0
UML 2.2	1587	109	15	0	0
LF4	10293	1	8	0	0

We have not encountered any specific problems, and we can't think of any reason why it shouldn't 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 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 constraints 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.

The query execution performance degrades rather quickly with increasing model sizes. Thus, we are currently working on automatic query optimization by reordering the sequence in which the matches and constraints are evaluated, as this significantly shapes the search space. It would also be interesting to see how well this approach is doing for other use cases than ad-hoc-querying, i. e., for model transformation specifications, model completion, pattern retrieval, and, most of all, enforcement of modeling guidelines and validity rules. Finally, it appears to be promising to study combinations of queries, such as logical connectors or pipelining.

References

[1] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A Visualisation of OCL using Collaborations. In M. Gogolla and C. Kobryn, editors, *Proc. 4^{th} Intl. Conf. Unified Modeling Language (≪UML≫'01)*, number 2185 in LNCS. Springer Verlag, 2001.

[2] R. Davis. *Business Process Modelling with ARIS: A Practical Guide*. Springer Verlag, 2001.

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

[4] S. Kent. Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. In *Proc. Intl. Conf. Object-Oriented Programming, Systems, and Languages (OOPSLA97)*, pages 327–341. ACM Press, 1997.

[5] Integration Definition for Function Modeling. Technical report, Computer Systems Laboratory, National Institute of Standards and Technologies (NIST), 1993.

[6] OMG. UML 2.0 OCL Specification (OMG Final Adopted Specification, ptc/2003-10-14). Technical report, Object Management Group, Oct. 2003.

[7] OMG. OMG Unified Modeling Language, Superstructure, V2.1.2 (formal/2007-11-02). Technical report, Object Management Group, Nov. 2007.

[8] QVT-Partners. Revised submission for MOF 2.0 Query/-Views/Transformations RFP (version 1.1, 2003-08-18). Technical report, OMG, Aug. 2003.

[9] D. Stein, S. Hanenberg, and R. Unland. Query Models. In T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, editors, *Proc. 7^{th} Intl. Conf. Unified Modeling Language (≪UML≫'04)*, number 3273 in LNCS, pages 98–112. Springer Verlag, 2004.

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

[11] H. Störrle. A PROLOG-based Approach to Representing and Querying UML Models. In P. Cox, A. Fish, and J. Howse, editors, *Intl. Ws. Visual Languages and Logic (VLL'07)*, volume 274, pages 71–84. CEUR, 2007.

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