MOCQL: A Declarative Language for Ad-Hoc Model Querying

Harald Störrle

Institute of Applied Mathematics and Computer Science
Technical University of Denmark
hsto@dtu.dk

Abstract. This paper starts from the observation that existing model query facilities are not easy to use, and are thus not suitable for users without substantial IT/Computer Science background. In an attempt to highlight this issue and explore alternatives, we have created the Model Constraint and Query Language (MOCQL), an experimental declarative textual language to express queries (and constraints) on models. We introduce MOCQL by examples and its grammar, evaluate its usability by means of controlled experiments, and find that modelers perform better and experience less cognitive load when working with MOCQL than when working with OCL. While MOCQL is currently only implemented and validated for the different notations defined by UML, its concepts should be universally applicable.
Keywords: OCL, UML, model querying, empirical software engineering, Prolog

1 Introduction

1.1 Motivation

Many software development approaches today use models instead of or alongside with code for different purposes, e. g., model based and model driven development, Domain-Specific Languages (DSLs), and Business process management. As a consequence, 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 fixed set of predefined queries. The natural choice of language when it comes to selecting a powerful general-purpose model querying language is the Object Constraint Language (OCL [10]), at least for UML and similar languages. However, OCL is often perceived as too complex for many modelers, let alone domain experts without formal training in Computer Science. So, the goal of this paper is to try and come up with better solutions to this problem. In order to achieve high levels of accessibility, we are prepared to even sacrifice a certain degree of theoretical expressiveness, as long as the practically relevant cases are covered. Aspects other than usability are beyond the scope of this paper.

1.2 Approach

Our first approach to improving the usability of model query languages was based on the commonly held assumption that visual languages are generally easier to understand than textual languages. Since OCL is a purely textual language, a visual language might perform better. Following this assumption, we defined a predominantly visual alternative to OCL, the Visual Model Query Language (VMQL [15]). We could indeed demonstrate that VMQL is easier to use than OCL [17], while being applicable to the same kinds of tasks OCL is targeted at (i.e., both for querying and expressing constraints, see [7] for the latter).

During these research projects, however, we also found evidence that the visual nature of VMQL is only one of several factors contributing to its usability, and quite possibly not the largest one. In order to pursue this lead, and building on the lessons learned in the design of VMQL, we invented a new textual language on the fly. To our surprise, this improvised language was even more effective than VMQL, and was preferred by users when given a choice. Thus, we refined and elaborated this language which has now evolved into the Model Constraint and Query Language (abbreviated to MOCQL, pronounce as "mockle").

The design of MOCQL is informed by the lessons learned during the design of VMQL, and they share conceptual and implementation element, yet MOCQL is a genuinely new language. Our working hypothesis is that MOCQL offers better usability than both OCL and VMQL. MOCQL is not conceptually restricted to UML, but the current implementation and validation have so far only covered this notation. Thus, while we believe that MOCQL is suitable as a *universal* model query language, no such claim will be raised here. Also, we envision using MOCQL on model repositories such as CDO, EMFStore, Morsa, and ModelBus.[1] However, this has not yet been attempted.

In the remainder of this paper, we will first introduce MOCQL by example, showing how it may be used to query models in a concise and modeler-friendly way. We provide a (simplified) grammar for MOCQL, informally describe its semantics, and briefly report on its implementation. Then, we report on two controlled experiments to assess the relative effectiveness of OCL, VMQL, and MOCQL from a user's point of view. We conclude by comparing MOCQL with existing approaches, highlighting the contributions, and outlining ongoing research.

2 Introducing MOCQL

We will now show some examples of MOCQL queries. In order to explain their meaning, we also present OCL queries with the same effect. Note that no formal relationship between MOCQL and OCL exists, in particular, there is no automatic translation between these languages. All queries are assumed to be executed on the models shown in Fig. 1. For simplicity, we shall assume that

[1] See `www.eclipse.org/cdo`, `www.emfstore.org`, `www.modelum.es/morsa`, and `www.modelbus.org`, respectively.

Fig. 1 shows two models M1 and M2, the former of which is completely covered by diagram M1, and the latter is completely covered by the diagrams M2a and M2b.

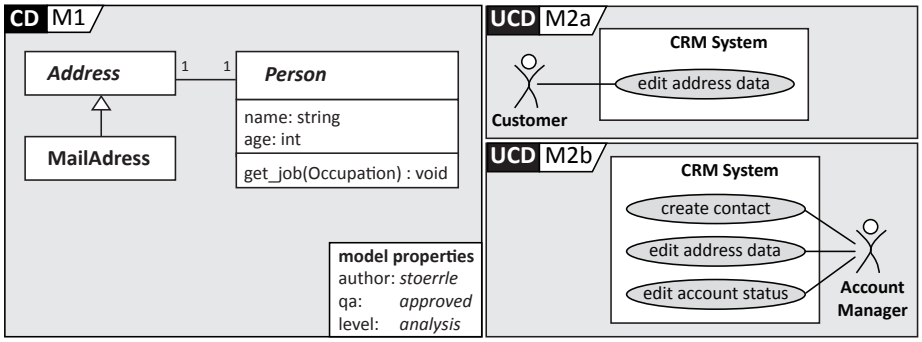


Fig. 1. The sample models used as the base model for all queries described in this paper.

Suppose, a modeler is looking for all classes whose name ends with "Address" in model M1. On the given sample model, the result would be the set of the two classes named "Address" and "MailAddress". Such a simple query would probably be best answered by using whatever search facility any given modeling tool provides; most tools allow specifying the meta-class of the search target and pattern matching for names. Using OCL, the query might look something like the following.

```
(1a)   Class.allInstances()
          -> select(c | c.name.contains_substring("Address")).
```

Here, we assume an OCL-function `contains_substring`. Such a function is not part of the OCL or UML standards [10, 9], but it could probably be programmed and thus be available in some hypothetical OCL Query Library. In MOCQL, this query can be expressed in a very natural way:

```
(1b)   find all classes $X named like "*Address" in M1.
```

Understanding this query requires much less knowledge about the UML meta model than the corresponding OCL expression, which makes it more readily understood by modelers (as we shall show below). Also, this expression is easy to modify and extend, e.g., we want to add the constraint that the classes we look for are abstract. In OCL, this could be expressed by query (2a).

```
(2a)   Class.allInstances()
          -> select(c | c.name.contains_substring("Address")
                        && c.isAbstract = true).
```

Compare this with the corresponding MOCQL query (2b), which we believe is significantly clearer, and easier to understand.

```
(2b)   find all abstract classes $X named like "*Address" in M1.
```

Suppose we were to look for abstract classes that do not have subclasses. The typical built-in search facilities of most modeling tools would not support such a query. In OCL, we would have to write something like the following.

```
(3a)   Class.allInstances()
           ->select(c | c.isAbstract=true).
                   intersection(c | c.general->isEmpty())
```

Even more understanding of fine details of the UML meta model are required here (e.g., the property "general"), and knowledge of the different navigational operators in OCL (i.e., the dot vs. the arrow), which at least many students struggle with. In MOCQL, we can instead write

```
(3b)    find all abstract classes $X in M1 where
              there are no classes $Y such that $X generalizes $Y
```

which was created from the first MOCQL query. This expression just requires to know the name "Generalization" for the relevant UML relationship. Thus, less knowledge about the UML meta model is required when using MOCQL as compared to when using OCL. Together with its user friendly syntax, MOCQL also allows domain experts to query models in a straightforward way.

Let us now turn to model M2. Assume, a modeler wants to find out all the actors involved in a given use case named "edit address data". In MOCQL, the following query would achieve this goal, returning the actors "Customer" and "Account Manager".

```
(4)    find all actors $X where $X is associated to $Y and
              there is a useCase $Y named "edit address data".
```

The way associations are represented in the UML meta-model would make this a rather complex query were we to express it in OCL. Observe, that elements of the UML meta model (i.e., meta classes and meta attributes) are not part of the MOCQL syntax. They are treated as strings and passed on "as is" to the query execution procedures.

MOCQL offers capabilities for all kinds of models occurring in UML, including use case models, state machine models, and activities. Suppose, for instance, we are looking for activities that contain Actions unconnected to the initial node. In MOCQL, this could be expressed by query (5).

```
(5)   find all actions $Unconnected in M3 such that
             there is no initialNode $Initial such that
                   $Initial precedes $Unconnected transitively.
```

Finally, suppose we want to query two models simultaneously, to find elements that have the same name. In MOCQL, this can easily be achieved by query (6).

```
(6)   find all $X1 named $NAME in M1 such that
                there is $X2 named $NAME in M2.
```

Relaxing the query to check for similar names rather than exact matches only requires to add `like` before the second occurrence of `$NAME`. An EBNF grammar of MOCQL is shown in Fig. 2. This grammar has been simplified for purposes of presentation.

3 Semantics

The semantics of MOCQL consists of two parts. On the one hand, there is a particular model representation that facilitates the operations involved in the query execution, storing models as Prolog knowledge bases. On the other hand, there is a mechanical translation of MOCQL queries into Prolog programs that are then executed on the knowledge base, utilizing a set of predefined Prolog predicates.

3.1 Model representation

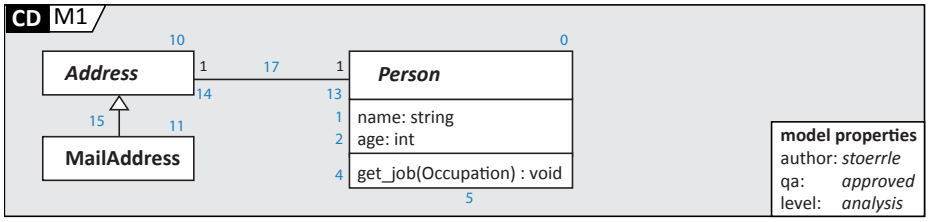
The model representation we use for MOCQL has been used for implementing VMLQ [17], and a set of other advanced operations on models such as clone detection [16], or difference computation [18]. In this representation, models are looked at as knowledge bases, and individual model elements are considered facts that are stored in a Prolog data base. Queries and other operations on models are implemented as Prolog predicates over this knowledge base, i.e., queries are translated into Prolog predicates by a definite clause grammar (DCG, a kind of Prolog program). Those query predicates are then simply executed, calling a small library of predefined search functions. This greatly simplifies the implementation, while making it easy to extend and experiment with, which is the main design objective at this stage of the development of MOCQL.

First, the user creates source models, exports them to an XMI file, and transforms it into a Prolog database. Each model element is transformed to one Prolog clause of the predicate `me`, see Fig. 3 for an example (edited for improved readability). The first argument of each `me`-fact is a pair of type and internal identifier (usually an integer). The 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.

This conversion has several advantages over XMI. On the one hand, it is much more compact than XMI, which also speeds up processing of models. In particular, with the given representation, we are able to keep even very large models in-memory all the time, which is not always the case for the typical data

```
COMMAND      ::= find SET_SPEC
               | count SET
               | save  SET as NAME
               | load NAME
               | assign SET (u | n) SET to VARIABLE
               | clear VARIABLE

SET          ::= SET_SPEC | VAR
VAR          ::= $A | $B | $C | ...
SET_SPEC     ::= (A_QUANTIFIER | E_QUANTIFIER) ELEMENT_SPEC

A_QUANTIFIER ::= forall | each | every | all
E_QUANTIFIER ::= [there (is | are)] [ARTICLE]
ARTICLE      ::= a | an | the | those | some | no

ELEMENT_SPEC ::= [abstract | concrete] TYPE [VAR] [NPROP] [MPROP]
                 (where | such that) PROPS

TYPE         ::= activity | activities | class | classes | useCase | ...
               | element
               | element VAR
               | element VAR with id STRING
ATTR         ::= name
               | isAbstract
               | ownedMember
               | ...

PROPS        ::= PROP
               | not PROP
               | PROP LOG_OP PROPS
PROP         ::= ATTR CMP_OP VAL
               | VAR REL_OP VAR [directly | transitively]
               | SET_SPEC

VAL          ::= defined | bool | int | float | string | ...
MPROP        ::= in MODELNAMES
MODELNAMES   ::= MODELNAME
               | MODELNAME & MODELNAMES

NPROP        ::= named STRING
               | named like PATTERN

LOG_OP       ::= and | or
CMP_OP       ::= is | are | = | has | have | is like | < | > | is not | ...

REL_OP       ::= REL_OP_AKT ARTICLE | (is | are) (ASSOC_REL | REL_OP_PAS by)
REL_OP_AKT   ::= generalizes | specializes | includes   | extends
               | follows     | precedes    | succeeds   | owns      | ...

ASSOC_REL    ::= associated to | part of

REL_OP_PAS   ::= generalized | specialized | included   | extended
               | followed    | preceeded   | succeedes  |   ...
```

Fig. 2. Simplified EBNF grammar of MOCQL.




```
:-module(m1, [me/2, view/2, model/2]).
   model(m1,             [level-analysis, author-stoerrle, qa-approved]).
   view(cd-1,            [type-class_diagram, name-'Sample model',language-'UML 2',
                          version-'2.1.1',elements-ids([0,1,2,4,5])]).
   me(class-0,           [name-'Person',attributes-ids([1,2,3,14]),
                          operations-id(4)]).
   me(feature-1,         [name-name, type-string]).
   me(feature-2,         [name-age, type-int]).
   me(operation-4,       [name-'get_job', parameters-ids([5]), result-void]).
   me(parameter-5,       [type-id(6)]).
   me(class-10,          [name-'Address', attributes-ids([13])]).
   me(class-11,          [name-'MailAddress']).
   me(feature-13,        [multiplicity-1, type-id(0)]).
   me(feature-14,        [multiplicity-1, type-id(10)]).
   me(generalization-15, [from-id(11), to-id(10)]).
   me(association-17,    [ends-ids([13, 14])]).
```

Fig. 3. Example for the Prolog representation of sample model M1.

structures for processing XML-files. On the other hand, it is very easy to access models represented in this way by using the Prolog command line interface, or to exchange code operating on models while keeping the model loaded, which is a tremendous help during development. Finally, the representation is generic enough to allow for all kinds of models, including those that do not have a MOF-like meta meta model.

Moreover, observe that the conversion takes only a few milliseconds, and is fully reversible: It neither adds nor removes anything, it merely changes the model representation. The conversion is triggered automatically when trying to access an XMI file in a MOCQL query, so it is completely transparent to the user.

3.2 Query translation

The second part of the semantics is the translation of queries into executable Prolog code. Executing the queries amounts to executing this code. Let us again consider the introductory examples from Section 2 in order to see how these are interpreted. In the first step, the query is parsed, creating an abstract syntax tree. This step reduces the syntactic sugar, i.e., it reduces plural expressions to singular, and converts syntactic alternatives into a single expression. For instance, the expression

```
(1b)   find all classes $X named like "*Address" in M1.
```

results in the parse tree shown in Fig. 4. This expression is then translated into the following sequence of Prolog predicates.

```
all( 'M1', [X], [type(class), is_like(name, '*Address')]),
show('M1', [X]).
```

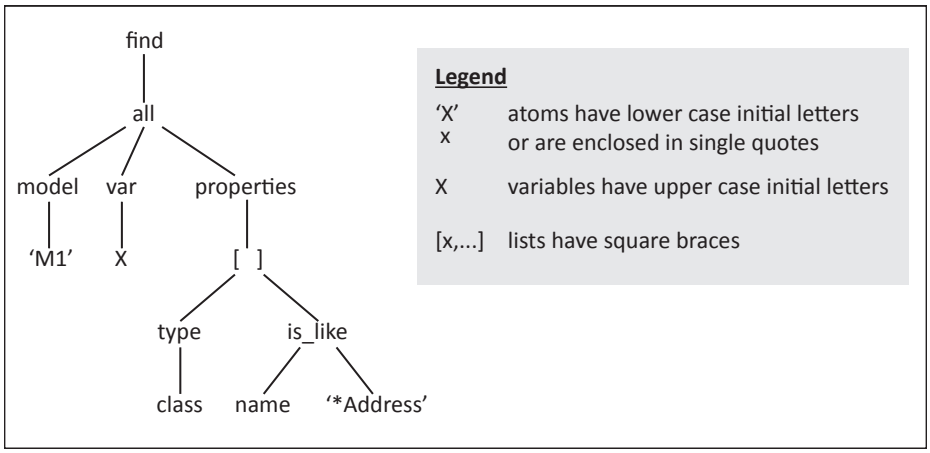


Fig. 4. The abstract syntax tree resulting from parsing and simplifying query (1b).

The predicates `all` and `show` are defined accordingly, so that executing this Prolog program actually executes the query. The atomic value `M1` refers to the name of the model to be queried, the list `[X]` is the list of all variables occurring in the expression.

Similarly, the third query example from Section 2 is translated, so that

```
(3b)   find all abstract classes $X in M1 such that
             there are no classes $Y where $X generalizes $Y
```

becomes

```
all( 'M1', [X],    [type(class), is(isAbstract,true) ]),
none('M1', [X, Y], [type(class), generalizes(X,Y)    ]),
show('M1', [X]).
```

As before, `none` is a predicate defined as part of MOCQL, so that executing this program simply executes the query. Observe that Prolog variables (i.e., `X` and `Y`) are *logical* variables, that is, they are unified rather than containers with assigned values. Thus, the sequence of the first two clauses of this program does not change the result. It does change the execution behavior, though, in particular the required computational resources.

4 Implementation

Fig. 5 shows the overall architecture of MOCQL. Processing queries is done in three steps. First, the base model is transformed from an XMI file to the Prolog transformation described in in Section 3.1 above. Observe that this transformation is purely syntactical and typically too fast for the user to notice. Then, query expressions are transformed into Prolog predicates as shown in Section 3.2, which refer to the predicates defined in the Model Querying Library. Finally, this predicate is then executed.

MOCQL shares the Model Querying Library with previous research including VMQL, but is otherwise independent. Future extensions of MOCQL to allow querying of other modeling languages such as BPMN would require some changes to this implementation. In particular, a new translator from the source model format into the Prolog format would be required, and some amendments the MOCQL grammar to cover new language concepts, i.e., the non-terminals `TYPE`, `ATTR`, and `REL_OP_AKT`. Whether amendments to the Model Querying Library would be necessary, is unclear. Thus, extending the scope of MOCQL to other modeling languages beyond UML is closer to porting a programming language to a new processor architecture than creating a new programming language.

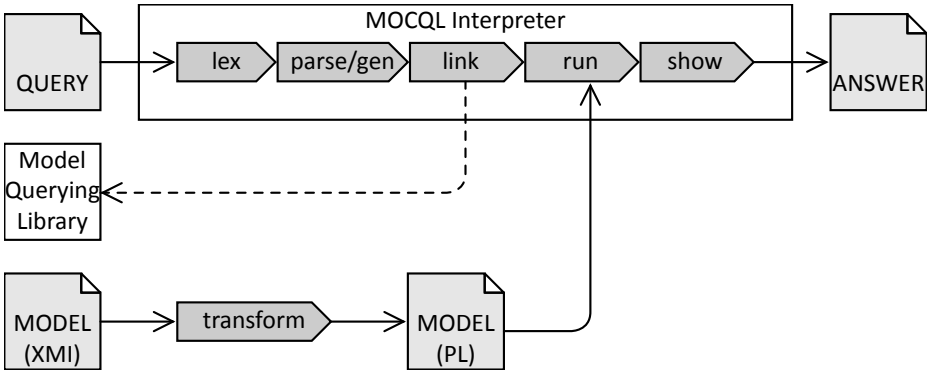


Fig. 5. Architecture of MOCQL: the Model Querying Library is shared with previous research, including VMQL.

5 Usability evaluation

In this section we evaluate MOCQL. We focus on usability, reporting two controlled experiments that study and compare the usability of VMQL, OCL, and MOCQL, respectively. At the end of this section, we briefly discuss other qualities.

5.1 Experiment design

Both experiments used the same randomized block design, but tested different sets of languages: OCL vs. MOCQL, and VMQL vs. MOCQL, respectively. The experimental setup consisted of four parts asking for (A) demographic data, (B) finding query expressions matching an English description, (C) checking the match between a given query expression and its English description, and (D) asking for subjective judgments regarding the languages tested. Task B contained 28 subtasks while Task C contained 12 subtasks.

In each of the experiments, 17 subjects participated, most of them being graduate students, but also including six IT professionals and a researcher. Different sequences of tasks were randomly assigned to them in order to control learning effects and bias. Our study was blinded by naming the languages A, B, and C, respectively. Going by the self-assessment in the demographic part of the questionnaire (Task A), the participants had little knowledge of either of the tested languages. The participants of the second experiment were recruited from the "Elite SE study line", an educational program that admits only students of very high aptitude.

We controlled the variables language (OCL, VMQL, MOCQL), query expression, and task, and recorded the correctness of the answers, the time taken, and the subjective assessment. The latter was divided into three different measures asking for preference, effort, and confidence in the result. The experiment was run as a pen-and-paper exercise. Participants were offered to talk about the experience or comment on the questionnaire, an opportunity some of them took.

5.2 Observations

We first discuss the objective measure of the number of correct answers given by the subjects. We have normalized the absolute numbers to percentages. A perfect score would reveal the same frequency for each language. Table 1 shows the observations, Fig. 6 visualizes them.

Clearly, subjects perform better on both tasks under the treatment MOCQL than under the treatment OCL. In fact, several subjects complained about OCL in follow-up interviews or comments on the questionnaire margins. In the second experiment, we see that subjects perform better using MOCQL than VMQL with what appears to be a smaller margin. Observe that there is a variation in the scores for MOCQL between the two experiments, which we explain by variations in the subject populations, i.e., participants of Experiment 2 can be expected to have a far beyond average general intelligence. The relative difficulty between the two tasks is consistent across both experiments, further confirming the validity of our findings.

Let us now turn to the subjective assessments. Participants were asked to record their subjective assessment on a 5-point Likert scale which we normed to the interval 0..10 for easier presentation. Since these are subjective measures anyway, we combined the results from both experiments in this presentation. Table 2 shows the observations, Fig. 7 visualizes them.

Table 1. Performance of subjects in tasks B and C.

Language	Experiment 1		Experiment 2	
	Task B	Task C	Task B	Task C
MOCQL	82.1%	58.7%	83.9%	62.7%
VMQL	-	-	74.2%	49.0%
OCL	54.8%	38.1%	-	-

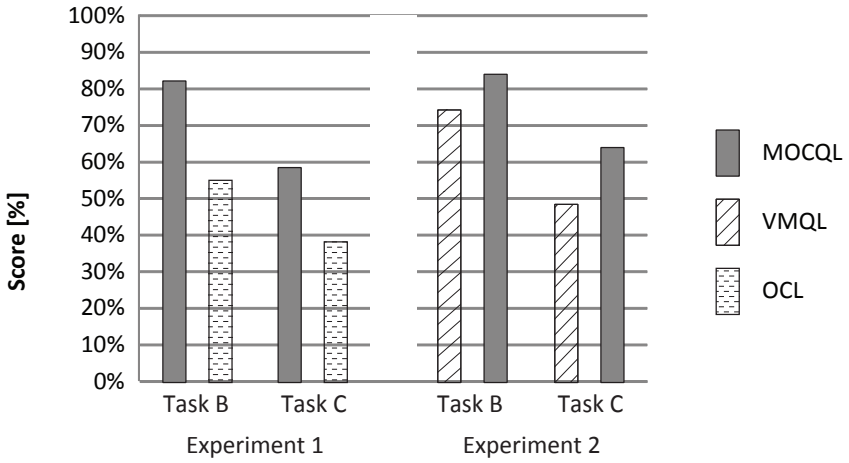


Fig. 6. Performance of subjects in tasks B and C: visualization of the data in Table 1.

All three measures consistently show the same trend of OCL scoring lower than VMQL, which in turn scores lower than MOCQL. As before, there is a larger difference between OCL and VMQL, than there is between MOCQL and VMQL. We see that the ratings for "Understandability" and "Confidence" are particularly low for OCL, which is consistent with the post-experimental remarks by participants.

5.3 Validity

With 34 participants, three different tasks, and 28/12/3 different measurements within each task, we have a fairly large sample size. Due to the study design, we can safely exclude bias through learning effects or variations in the subject population. All results are consistent with each other, with only the minor fluctuations between experiments that are to be expected in any kind of human factor study.

Obviously, the task presentation would influence the outcome: since MOCQL tries to imitate a natural language in its concrete syntax, there is high degree of proximity to the task description, that is provided in written English, biasing the result in favor of MOCQL. However, we stipulate that describing a query

Table 2. Subjective assessment of cognitive load.

Language	Understandability		Effort		Confidence	
	μ	σ	μ	σ	μ	σ
MOCQL	8.0	1.88	6.2	3.38	8.5	2.20
VMQL	7.0	2.28	7.5	3.30	7.7	2.90
OCL	3.8	1.80	8.8	1.78	3.3	1.55

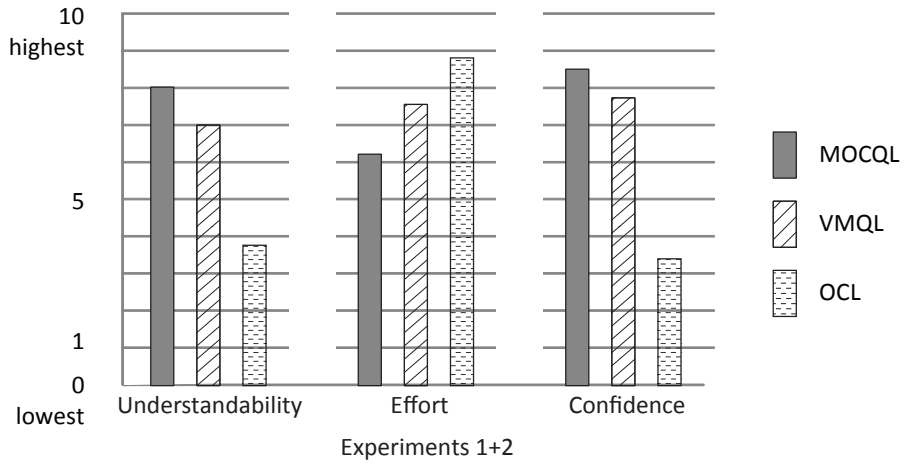


Fig. 7. Subjective assessment of cognitive load (averages across all subjects, normalized to the interval 0 (lowest) to 10 (highest).

in plain English is exactly what a modeler does when faced with a search task. Allowing him or her to express queries that way is, thus, not an undue influence on the experiment. Quite contrary, that is exactly the point of MOCQL.

A potential threat to validity is the fact that we did not test the respective whole languages, that is, there are parts of OCL, VMQL, and MOCQL that have not been subjected to experimental validation of their usability. In that sense, the validity of inferences regarding the languages as such is limited. Due to the conceptual differences between the languages, however, it would be difficult to completely compare them.

One might also object that the subjects—students—are not representative for the audience MOCQL is targeted at, i.e., people with little or no UML knowledge. In fact, the participants of Experiment 2 have been tested after they had just completed a one-term intensive course on UML, MDA, and OCL. However, even that degree of UML/OCL knowledge and additional cues and auxiliary high-level functions did not lead to an improved performance on the OCL tasks.

Finally, we have used different measurements to capture aspects of cognitive load (cf. [11]), yielding consistent results. Subjective assessments of cognitive

Table 3. Testing Hypothesis with the Binomial test.

Hypothesis	Task	Significance
Experiment 1:	B	$p < 10^{-7}$ ***
Subjects perform better using OCL than MOCQL	C	$p < 10^{-5}$ ***
Experiment 2	B	$p = 0.0033$ *
Subjects perform better using VMQL than MOCQL	C	$p = 0.059$ ·

load have been found to be very reliable indicators of the objective difficulty of a task (cf. [8]).

5.4 Inferences

We tested the hypothesis that subjects performed better using OCL than MOCQL in Experiment 1. Using a binomial test in the R environment [12], we can reject this hypothesis with high certainty ($p < 10^{-7}$ for task B, and $p < 10^{-5}$ for task C). Similarly, we can reject the hypothesis that subjects performed better using VMQL than MOCQL in Experiment 2, though there is not necessarily a significant result for task C ($p = 0.0033$ for task B, and $p = 0.059$ for task C).

5.5 Interpretation and conclusions

It is obvious that OCL offers little to modelers when it comes to querying models, and our investigation establishes the consequences as a fact. In previous research [17], we have shown how users performs better using VMQL than OCL, over a range of tasks. The current results show that user perform better using MOCQL than both OCL and VMQL. Both in our previous research and the current results, users also show a much higher acceptance (and thus, motivation), for MOCQL and VMQL than they exhibit for OCL, consistently across different task types, many different queries, and different measurements, which all consistently point in the same direction. Our results are mostly significant, some of them to the extreme. Our study exhibits a high degree of validity.

We have thus provided substantial evidence in support of our initial working hypothesis that MOCQL offers better usability than both OCL and VMQL, as outlined in Section 1.2. We believe it is safe to assume, that these results are generalizable to other contexts, such as different subject populations or different queries. Also, we expect these findings to carry over to extensions of MOCQL that have not yet been tested.

6 Related Work

There are essentially three kinds of query facilities. First, there are basic tools like full text search and sets of predefined queries. These sacrifice expressiveness for usability, leaving modelers with little leverage. On the other end of the

spectrum, there are application programming interfaces of modeling tools, which offer maximum expressiveness to the modeler, but require substantial expertise which only few modelers possess. Certainly, domain experts, which are in the focal point of our work, lack this capability.

Between these two extremes, there are model query languages varying along different dimensions. One the one hand, there is of course OCL [10] as the most widely used model query language. OCL also seems to be the only generically applicable textual model query language (disregarding non-semantic facilities such SQL, XPath, and similar). As our studies clearly show, OCL is not suited for ad-hoc querying by domain experts. In fact, even highly trained professionals and top-notch students with substantial training in OCL have serious trouble using it.

On the other hand, there are the visual model query languages like QM [14, 13], BP-QL [3], BPMQ [2, 1], Visual OCL [4, 5], and VMQL [15, 17, 7] (see [17] for a detailed comparison). These come with the explicit or implicit promise of higher usability, exploiting the fact that most modeling notations are also visual, and it is intuitively appealing to express queries the same way as base models. However, little evidence has been published to support this intuition; only VMQL seems to have been evaluated from this angle. From the results presented above and in previous studies, respectively, it is clear that OCL performs poorly, and that both MOCQL and VMQL perform better than OCL. Surprisingly, though, MOCQL even surpasses VMQL with respect to usability. This contradicts the common intuition about textual vs. visual notations and demands further inquiry.

We expect Visual OCL to perform similar to OCL since it is just a visualization of OCL; the other visual model query languages should yield results similar to those of VMQL since they are based on a similar paradigm and in some cases offer similar solutions (e.g., the treatment of transitive edges in BP-QL and QM, or negation in BP-QL).

Most model query languages are restricted to express queries on a single notation or a small set of related notations. For instance BPMN-Q addresses BPMN and (to some degree) EPCs, QM address a subset of UML class and sequence diagrams, and CM address only elementary class diagrams. On the other hand, OCL and Visual OCL apply to all MOF-based notations; VMQL and MOCQL even go beyond that requirement.

There are large differences with respect to the tool support a modeler might obtain for the model query languages mentioned. Only for OCL is there a choice of quality tools from different sources. Most of the other tools have been implemented as academic prototypes only, or not even that (e.g., CD and QM).

OCL (and, potentially, Visual OCL) offer maximum expressiveness through defining recursive functions. Most other model query languages mentioned above seem to have been analyzed from this perspective. VMQL does do not offer user-defined recursive functions, and is thus less expressive than OCL, though the exact degree of expressiveness is currently unknown. Similarly, MOCQL does not allow the definition of recursive functions, but it should be not too difficult to add such a feature. Observe also, that MOCQL provides features that are relevant for

practical model querying, but currently missing in OCL, such as using wild-card expressions, executing queries across several models, type variables, or access to model element identifiers.

7 Discussion

7.1 Summary

In this paper we have introduced the Model Query and Constraint Language (MOCQL), by means of example and a (simplified) grammar. We report on user studies comparing OCL, VMQL, and MOCQL, and find strong evidence that MOCQL offers higher usability than both OCL and VMQL in a number of ways. This is particularly true when comparing MOCQL and OCL. At the same time, MOCQL offers a high degree of expressiveness. MOCQL can be applied to the whole range of modeling notations present in the UML, not just structural models or meta-models. In fact, MOCQL is not conceptually restricted to UML: we believe it is applicable to any modeling language that has a meta-model or where a meta-model can be constructed, including BPMN, EPCs, and DSLs.

7.2 Contributions

The contribution of this paper is to provide evidence for two observations. Firstly, we maintain that usability is an important concern when it comes to model query languages, but has been largely ignored in existing languages, most notably OCL. Thus, it is relatively easy at this point to achieve substantial improvements over the state of the art. Secondly, it is not so much the concrete syntax that contributes to usability, but the abstract syntax, that is, the conceptual constructs of the query language. In this paper, we show that a textual concrete syntax can actually perform better than a visual concrete syntax, which is somewhat in contradiction with the commonly held belief of visual notations generally being "better" than textual ones.

7.3 Limitations

In its current state, MOCQL has several shortcomings. Firstly, it lacks a formal semantics. Given the time and difficulty it took to arrive at a formal semantics for OCL, we consider this more of a challenge and future work than a lasting deficit.

Secondly, MOCQL currently lacks the capability to define recursive functions, and thus complete expressiveness. MOCQL was designed with the practical modeler in mind, thus, many of the functions that modelers have to define themselves in OCL are built into MOCQL, thus reducing the need for such a feature.

Thirdly, MOCQL allows many expressions that are either hard to process, or may be confusing. For instance, MOCQL allows to express queries with double negation. Clearly, this is computationally inefficient, and since we use the regular

negation-as-failure semantics of Prolog, the result might not be what the user expects. Moreover, since double negation is inherently cognitively difficult, using it will be a challenge. We currently lack empirical evidence on the actual usage of MOCQL in the field.

7.4 Future Work

Clearly, the current limitations of MOCQL are some threads of our ongoing and future work. In particular, it would be interesting to apply MOCQL to other modeling languages, such as BPMN and EPCs and see whether the current MOCQL is up to this task, or requires extensions and amendments. Also, parts of the implementation would have to be adapted to accommodate for different model representations.

Then, performance is obviously an issue for practical model querying, in particular for using MOCQL for interactive operations on large models. We generally have very good experience with the performance of the technology underlying our approach in comparison with current OCL implementations (see also [6]), and experience so far indicate that MOCQL might in fact be dramatically faster than existing OCL tools. Still, we will have study and document the run-time performance of MOCQL.

Moreover, our initial research hypothesis is based on the intuition, that the major improvement in usability would derive from using a visual rather than a textual concrete syntax for querying. Thus, one would expect a similar effect for, say, the Visual OCL [4, 5]. Doing pairwise comparisons of OCL, Visual OCL, VMQL, and MOCQL, respectively, and studying the factors impacting modeler understanding with qualitative methods such as think aloud protocols might allow us to develop a theory about how queries are being processed by modelers. This, in turn, could be valuable in informing future language design practice.

References

1. Ahmed Awad. *A Compliance Management Framework for Business Process Models*. PhD thesis, Hasso Plattner Institute, Univ. of Potsdam, 2010.
2. Ahmed Awad, Gero Decker, and Mathias Weske. Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In *Proc. Intl. Conf. Business Process Modeling (BPM)*, volume 5240 of *LNCS*, pages 326–341. Springer Verlag, 2008.
3. Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. Querying Business Processes. In *Proc. 32nd Intl. Conf. Very Large Data Bases (VLDB)*, pages 343–354. VLDB Endowment, 2006.
4. Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency Checking and Visualization of OCL Constraints. In Bran Selic, Stuart Kent, and Andy Evans, editors, *Proc. 3^{rd} Intl. Conf. Unified Modeling Language (≪UML≫'00)*, number 1939 in LNCS, pages 294–308. Springer Verlag, 2000.
5. Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A Visualisation of OCL using Collaborations. In Martin Gogolla and Chris Kobryn, editors, *Proc. 4^{th} Intl. Conf. Unified Modeling Language (≪UML≫'01)*, number 2185 in LNCS, pages 257–271. Springer Verlag, 2001.

6. Joanna Chimiak-Opoka, Michael Felderer, Chris Lenz, and Christian Lange. Querying UML Models using OCL and Prolog: A Performance Study. In Alain Faivre, Sudipto Ghosh, and Alexander Pretschner, editors, *Ws. Model Driven Engineering, Verification, and Validation (MoDeVVa'08)*, pages 81–89, 2008.
7. G. Costagliola et al., editor. *Expressing Model Constraints Visually with VMQL*. IEEE Computer Society, 2011.
8. Daniel Gopher and Rolf Braune. On the psychophysics of workload: Why bother with subjective measures? *Human Factors*, 26(5):519–532, 1984.
9. OMG. OMG Unified Modeling Language (OMG UML), Superstructure, V2.4 (ptc/2010-12-06). Technical report, Object Management Group, January 2011.
10. OMG. OCL Specification v2.3.1 (formal/2012-01-01). Technical report, Object Management Group, January 2012.
11. Fred Paas, Juhani E. Tuovinen, Huib Tabbers, and Pascal W.M. Van Gerven. Cognitive Load Measurement as a Means to Advance Cognitive Load Theory. *Educational Psychologist*, 38(1):63–71, 2003.
12. R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011.
13. Dominik Stein, Stefan Hanenberg, and Rainer Unland. A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models. In Uwe Aßmann, editor, *Proc.* 2nd *Working Conf. Model Driven Architecture: Foundations and Applications (MDAFA 2004)*, number 3599 in LNCS, pages 77–92. Springer Verlag, 2004. available at `www.ida.liu.se/~henla/mdafa2004`, updated papers appeared later as LNCS 3599, Springer.
14. Dominik Stein, Stefan Hanenberg, and Rainer Unland. Query Models. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proc.* 7th *Intl. Conf. Unified Modeling Language (≪UML≫'04)*, number 3273 in LNCS, pages 98–112. Springer Verlag, 2004.
15. Harald Störrle. VMQL: A Generic Visual Model Query Language. In Martin Erwig, Robert DeLine, and Mark Minas, editors, *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'09)*, pages 199–206. IEEE Computer Society, 2009.
16. Harald Störrle. Towards Clone Detection in UML Domain Models. *J. Software and Systems Modeling*, 2011. (in print).
17. Harald Störrle. VMQL: A Visual Language for Ad-Hoc Model Querying. *J. Visual Languages and Computing*, 22(1), February 2011.
18. Harald Störrle. Making Sense to Modelers - Presenting UML Class Model Differences in Prose. In Joaquim Filipe, Rui Csar das Neves, Slimane Hammoudi, and Lus Ferreira Pires, editors, *Proc. 1st Intl. Conf. Model-Driven Engineering and Software Development*, pages 39–48. SCITEPRESS, 2013.