Improving the Usability of OCL as an Ad-hoc Model Querying Language

Harald Störrle

Department of Applied Mathematics and Computer Science
Technical University of Denmark, Matematiktorvet, 2800 Lyngby, Denmark
hsto@dtu.dk

Abstract. The OCL is often perceived as difficult to learn and use. In previous research, we have defined experimental query languages exhibiting higher levels of usability than OCL. However, none of these alternatives can rival OCL in terms of adoption and support. In an attempt to leverage the lessons learned from our research and make it accessible to the OCL community, we propose the OCL Query API (OQAPI), a library of query-predicates to improve the user-friendliness of OCL for ad-hoc querying. The usability of OQAPI is studied using controlled experiments. We find considerable evidence to support our claim, that OQAPI facilitates user querying using OCL.

1 Introduction

Interactive querying by modelers is an important task in many model-based practices (cf. [7]). Where the full-text search and predefined queries provided by many tools are not expressive and flexible enough, the Object Constraint Language (OCL, [14]) is an obvious choice as a model query language. However, OCL has a reputation as being "hard", and we have observed many a modeler struggle with OCL, both in industry and academia. This has even been quoted as the main obstacle to adopting OCL in industry: "*OCL has not yet been broadly adopted by practitioners because they find it difficult to define OCL expressions*" (cf. [5, p. 665]).

In previous research, we had hypothesized that there might be at least two different factors contributing to the usability of model query languages (see Fig. 1). First, we hypothesized that there is a "language gap" between a textual model query language and a visual modeling language; providing a visual query language instead of a textual one (such as OCL) would avoid this gap, and contribute to usability. Thus we have defined the Visual Model Query Language (VMQL, see [19]), and could indeed prove that it is easier to use than OCL.

Evaluating the participant feedback from this initial study revealed, however, that there is a second factor, namely, a "semantic" or "concept gap" between the concepts modelers use and the concepts provided by a low-level query language such as OCL. Since, inadvertently, VMQL closed both of these gaps at the same time, we could not identify the respective contributions of these two factors.

Thus, we have defined the Model Query and Constraint Language (MOCQL, [20]), a language that shares the high level concepts of VMQL, but not the visual syntax. Again, we could demonstrate higher usability than OCL—but also higher usability than VMQL. This leads to two predictions. First, it suggests that closing the concept gap has (much) more impact than closing the language gap. If this is correct, the Visual OCL [4] would exhibit essentially the same level of usability than OCL. Second, equipping OCL with similar query concepts as defined in VMQL and MOQCL would improve the usability of OCL. In this paper, we are testing the second prediction by introducing the OCL Query API (OQPAI), a library of OCL querying predicates emulating the query concepts offered by VMQL and MOQCL, and test its usability.

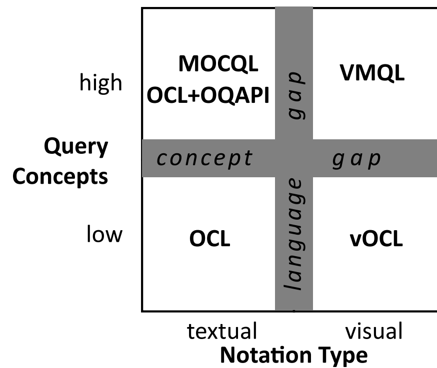

Fig. 1. Two independent factors contribute to the usability of a model querying language, the concrete syntax and conceptual abstraction level.

2 The OCL Query API

Based on previous research on the model query languages [17–20], we have identified the most important model query elements for ad-hoc domain model querying. We have translated them into OCL predicates, and organized them into concepts appealing to end user modelers, resulting in the OCL Query API (OQAPI).

In the following, we discuss a few design issues of OQAPI, explaining its major predicates and general style. Due to lack of space, however, this explanation is not complete. Note also, that there is currently no complete implementation of OQAPI. An overview over its predicates is found in Table 1.

2.1 Meta-class selection

Doubtlessly the most frequent query is looking for a certain element by its name and/or type. For instance, modelers would look for "the class called 'x"' or "all methods whose name starts with 'set"' (i.e., all setters). Since these operations

are so frequent, OQAPI provides specific support for them. As a concrete example, consider finding all classes in a model that have the name “Address”. Using OQAPI this query might be expressed as query (1a), while bare OCL would require something like (1b).

```
(1a)   classes() -> named("Address")
(1b)   Class.allInstances() -> select(c | c.name == "Address")
```

Unfortunately, solution (1b) works only in some contexts, but not, say, for selecting `Actions` of a given name, since `allInstances` is only defined for `Classifier`, and does not cover sub-meta-classes. So, `action.allInstances()` is not proper OCL code, and even if it were, no instances of subclasses of Action would be found. Since there are no type variables in OCL, we cannot implement a generic lookup function:

```
def lookup(<T> <: Element) : Set(<T>)=  -- not possible in OCL!
  model->collect(x|x.isOclTypeOf(<T>)).
```

Thus, OQAPI provides explicit lookup-functions for all (!) relevant meta classes, i.e., `classes()`, `actions()`, `useCases`, and so on. It also provides matching filters, i.e., `is_class()`, `is_action()`, `is_useCases` etc.

2.2 Attribute value selection

In (1b), we have used `select` for selecting by attribute values, but there is also the OCL function `collect` that allows more compositional selection: OCL allows to use `collect(a=v)` for searching for model elements with a specific value `v` in a given attribute `a` (see [14, p. 32]). Of course, such clauses can be cascaded. For instance, query (2) below looks for abstract classes called “Address”.

```
(2)    classes()->collect(name="Address")->collect(isAbstract=true)
```

For the most frequent meta-attributes, specific selection predicates are provided, in particular `named`. Besides exact matching, however, wild-card matching is frequently called for. Currently, there is no such feature in OCL. Using the `substring` functions of the OCL standard library, it is easy to implement a `named_like` function for the simplest cases with only one '*' wild card.

```
match(s:String, pattern:String): Boolean =
  let star = pattern.indexOf("*")
  let head = pattern.substring(0,star-1)
  let tail = pattern.substring(star+1,l-1)
  in
    match_tl(s, tail) && match_hd(s, head)
match_tl(s, pattern) =
  pattern =
  s.substring(s.length()-pattern.length(),s.length()-1)
match_hd(s, pattern) =
  pattern =
  s.substring(0,pattern.length()-1)
```

Clearly, anything up to and including linguistic and phonetic search would be possible, too.

2.3 Association

One of the properties of OCL is that it reflects immediately the structure of the underlying meta-model. This can be understood as a feature (degree of control), but also as a bug (lack of abstraction). Since the UML meta-model encodes many relationships that are quite clear and intuitive to modelers in unobvious and complicated ways, we think the latter point of view is more accurate. This is probably best demonstrated using association (but applies similarly to generalization, and containment). Consider first a case where we want to find the actors involved in a given use case named "edit address data". Using OQAPI, we can issue the following query.

```
(3)    actors()->associated_to()->(is_useCase()
                                  && named_like("edit address data") )
```

OQAPI offers the simple `is_associated_to` predicate to replace the cumbersome navigation made necessary by the complex encoding in the UML meta-model. In bare OCL, simply checking that a given Classifier `C1` is associated to another given Classifier `C2` makes us navigate to the respective `ownedMembers` of C_1 and C_2, then find an Association `A` that refers `ownedMembers` of C_1 and C_2 by its `memberEnd` meta-attribute. In OCL, this reads as follows.

```
   let End1 = C1.ownedMember
in let End2 = C2.ownedMember
in let Ass = Association.allInstances()
in collect(a | Ass->includes(a)
               and a.memberEnd->intersects(End1)
               and a.memberEnd->intersects(End2) )->notEmpty()
```

The clear and simple concept of an association is lost in this expression. OQAPI, on the other hand, provides the function `associated` hiding all the UML complexity behind an intuitive name, and the query simply becomes `E1.is_associated_to().contains(E2)`.

2.4 Other relationships

Another relationships that is treated in a similar way is containment. While the UML meta-model encodes it by the meta-attribute `ownedMember`, OQAPI offers the functions `owns` and `is_part_of`. Likewise for generalization: the UML meta-model encodes this relationship as a part of the specialization that points to the generalization, OQPAI offers the terms `specializes` and `is_generalized_by` and so on. Observe that OQPAI offers both the active and the passive form, both for generalizes and specializes, such as to not impose stylistic restrictions on the modeler. Likewise, overloading is used to define both set and element-based variants of predicates.

As a third example, consider a process model expressed by activity diagrams. Suppose, we are looking for Activities that contain Actions unconnected to the initial node. Using OQAPI, this could be expressed by query (4).

Table 1. Overview of the OQAPI functions. INTV refers to an interval specification

Concept	**Main Predicate**	**Syntactic Sugar Predicates**
type selection	`classes()`,...	`is_class()`,...
association	`associates_to()`	`associated_to()`, `associated_to(INTV)`, ...
element name	`named()`	`named_like()`
containment	`owns()`	`is_owned_by()`, `contains()`, `is_part_of()`, `owns(INTV)`, ...
generalization	`generalizes()`	`specializes()`, `is_generalized_by()`, `inherits()`, `owns(INTV)`, ...
control flow	`comes_after()`	`comes_before()`, `precedes()`, `succeeds()`, `is_preceded_by()`, `precedes(INTV)`, ...
inclusion	`includes()`	`is_included_by()`, `is_included_by(INTV)`, ...
extension	`extends()`	`is_extended_by()`, `is_extended_by(INTV)`, ...

```
(4)   actions()->preceded_by(transitively)->contains()->is_initialNode()
```

This example also illustrates another aspect: namely these concepts are often intuitively understood as being transitive, while writing recursive queries is a major challenge to many modelers.

Also, notice the (optional) parameter of this predicate: It specifies the length of the path between the nodes in the precedence-relationship. If this parameter is omitted, a distance of one is assumed, i.e., direct precedence. Other values include minimal and maximal distances (see [21] for more details).

3 Usability Evaluation

In a series of controlled experiments, we analyzed the performance of modelers when using different model query languages. In this paper, we report only on the results obtained for two model query languages, the OCL and OCL with OQAPI. Results on other model query languages are reported in [20] and [19].

Study Design We created sets of twelve queries ("stimuli") in OCL with and without OQAPI, and a list of 32 plain English descriptions of queries ("responses"). In the first task, subjects were given ten stimuli and asked to find matching responses, with zero to four matches being applicable to each stimulus. In the second task, subjects were asked given twelve groups of one stimulus and three alternative responses, one of which was correct. In the third task, subjects were given four responses and asked to create the correct stimulus. In the fourth task, subjects were asked to asses the readability and writability of the query languages, and the subjective effort and confidence.

The languages in the experiment were named as A to D.[1] The instructions were given in print, after completing one task, subjects were instructed not to

[1] Recall, that these experiments included not just OCL and OCL+OQAPI.

Table 2. Observations made in the experiments: The significance is computed by a two-tailed t-test for two samples with unequal variance. The effect size level is given in Cohen's terminology. The different n reflect the mortality in the experiment for the reading tasks; for the writing tasks, no analyzable results were obtained under the OCL condition.

	OCL (n=18)	**OQAPI** (n=25)	**Improvement** [%]	**Significance** (p-Value, level)		**Effect size** (Cohen's d, level)	
Understandability							
μ	3.76	4.78	27.13%	0.097	.	-0.559	M
σ	1.80	1.84	2.22%				
Writability							
μ		5.53					
σ		2.53					
Effort							
μ	8.75	6.25	-28.57%	0.015	*	0.918	L
σ	1.78	3.23	81.69%				
Confidence							
μ	3.33	5.23	57.14%	0.013	*	-0.881	L
σ	1.55	2.50	61.29%				

turn back to previous tasks. Subjects were asked to provide additional comments on the questionnaires and in unstructured follow-up interviews.

Observations Probably the most striking observation is that some subjects refused to work on tasks involving the OCL treatment; thus the substantially smaller number of participants for the OCL condition, in particular for the writing task, see Table 2. One of the participants remarked "*after [the other] languages, it's hard to get yourself to work on [OCL]. [It] is rather, well, relatively complicated, I kept thinking, jeez, why does it have to be quite as complicated. [The others] are quite easy in comparison, these are easy to understand.*" Other participant made similar remarks, such as: "*All in all it was ok...I found OCL horrible*" or "*[the other language] was ok, but [OCL] is difficult to understand, you have to follow the algorithm. That's ok, it works, but it's more effort.*" Probably the most poignant comment was "*[OCL] was really pissing me off*".

Coming to the objective measures (see Table 2), we see that subjects perform better when using OQAPI: subjects score almost 30% better when using OQAPI as compared to using bare OCL. At the same time, the reported effort (a measure for cognitive load) goes down almost 30%. Most strikingly, the confidence in the correctness of the result increases by almost 60%. Surprisingly, we also see that in the "OQAPI" condition, variances generally increase as when compared to the "bare OCL" condition. For understandability scores, this is only a slight increase that may not be significant. However, for confidence, we see an increase of over 60%, and for effort, we even see an increase of over 80%.

Interpretation The observations of the different performance scores and load measures are consistent, as are the impressions from the post experiment interviews. The differences between the two treatments are statistically significant, and demonstrate a medium to large effect size of using OQAPI, in the conventional terminology of Cohen. Together, this implies strong support for our initial hypothesis.

We interpret the massive increase in variance as a sign that there is a strong difference between individuals in the capacity to take advantage of the support offered by OQAPI. In other words, on average, everybody benefits from using OQAPI, but it is easier to use for some people than for others.

Threats to validity While it is always desirable to have larger numbers of data points to achieve better p-values, the effect size of the phenomenon studied here is such that even the modest n we provide in our study is sufficient to achieve statistically significant results. A larger threat to validity is the relatively small number of queries sampled in our evaluation. Moreover, we cannot claim that they are representative with any degree of certainty. Clearly, this would require a generally accepted body of sample queries, or a benchmark, both of which are not present in the case of OCL. In fact, as we have remarked earlier, there does not seem to be a single OCL library or API that is published.

Probably the most significant threat to validity is the high degree of mortality in this study, that is, the large number of subjects that completed the procedure only partially as described as the beginning of Section 3 above: for the writing task, mortality was 100%. However, in itself this is testament to the very low degree of usability that bare OCL provides, as compared to using OQAPI.

4 Related Work

Many model query languages have been defined beyond those discussed in this paper, the first being Constraint Diagrams [10], Query Models, and Join Point Designation Diagrams [15, 16]. In the area of business process modeling, there have been many new proposals in the last decade, e.g., the BPQL [11], BP-QL [3], BPMN-VQL [6], BQL [9], and BPMN-Q [2]. To our knowledge, none of these have been evaluated with a view to usability.

In contrast to the general interest in defining new query languages, there seems to be no interest in improving OCL by providing query libraries: Despite our best effort, we could not find *any* published libraries of OCL functions, or definitions of APIs, for any purpose. We have contacted three leading experts from the OCL community who confirmed this observation. Chimiak [5] claims to have defined 50 libraries with 4.5kLOC combined, but these are not published.

The *OCL Standard library* ??defined where?? defines only low level functions and operations on container types. The UML standard [13] does define some predicates that appear in OQAPI, but defines them in an unsystematic way and scattered over the whole UML standard document. The *QVT standard* [12] mentions a `UmlUtilities` library with operations such as `getAllAbstractBaseActors`

(p. 70) and `getAllDerivedClasses` (p. 72) but there is no reference to or listing of the library as such. QVT also defines a *Standard Library* with some useful functions (see [12, pp. 107]) rather similar to those included in the UML. Again, they are not presented as coherent API and suffer from the same shortcomings found in the function definitions of the UML standard.

5 Relevance of OQAPI

Since OCL has existed since almost twenty years now without any libraries published for it, we think it is reasonable to fundamentally question the relevance of predicate libraries such as OQAPI: if nobody needed them so far, why now? First of all, APIs are obviously in wide spread use in programming and enjoy great popularity, so it is intuitive to us to assume that OCL users would benefit from similar facilities. In fact, Chimiak [5] has argued for the extension of OCL to better support the use of libraries like OQAPI, too, highlighting reuse, modularity, and separation of concerns as the main benefits of such libraries.

Secondly, some people argue that usability is not a major issue, since OCL is supposed to be used only by experts. However, this is not the position of the OMG, which declare usability a major concern: "*The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use. OCL has been developed to fill this gap. It is a formal language that remains easy to read and write. It has been developed as a business modeling language [...]*" (cf. [14, p. 5]).

Thirdly, it could be argued that the comparison of OCL with and without OQAPI is unfair in a sense, since OCL is primarily a model constraint language targeted at expert modelers. Indeed, some seem to believe that OCL was never intended as a model query language, e.g., "*OCL was originally designed specifically for expressing constraints about a UML model*" (cf. [1, p. 91]) and "*OCL was not originally designed to be a query language*" (cf. [1, p. 102]). However, the OCL standard itself makes it quite clear that OCL has indeed been created with with both constraining and querying models in mind: "*[OCL] expressions typically specify invariant conditions [...] or queries over objects described in a model.*" (cf. [14, p. 5]).

Finally, and rather ironically, more evidence for the relevance of OQPAI is provided by the UML and QVT standards themselves: when inspecting these documents closely, we found a sizable number of instances where auxiliary functions are defined that perform a part or a special case of an OQAPI function (see Table 3). In some cases, these functions were defined repeatedly, sometimes under different names.

6 Discussion

In previous research, we created several model query languages, each of which provided higher usability than OCL. None of these, however, is a *practical* alter-

Table 3. OCL query functions defined in UML (top, [13]) and QVT (bottom, [12]).

Classifier :: inherit :	Set(NamedElement) ↦ Set(NamedElement)	p. 50
Classifier :: parents :	() ↦ Set(Classifier)	p. 53
Classifier :: allParents :	() ↦ Set(Classifier)	p. 54
Classifier :: allFeatures :	() ↦ Set(Feature)	p. 54
Element :: allOwnedElements :	() ↦ Set(Element)	p. 64
StateMachine :: ancestor :	State × State ↦ Boolean	p. 565
Transition :: containingStateMachine :	() ↦ StateMachine	p. 573
UseCase :: allIncludedUseCases :	() ↦ Set(UseCase)	p. 597
Profile :: allOwningPackages :	() ↦ Set(Package)	p. 664
Element :: subobjects :	() ↦ Set(Element)	p. 107
Element :: allSubobjects :	() ↦ Set(Element)	p. 107
Element :: subobjectsOfType :	OclType ↦ Set(Element)	p. 107
Element :: subobjectsOfKind :	OclType ↦ Set(Element)	p. 107
Model :: objects :	() ↦ Set(Element)	p. 108
Model :: objectsOfType :	OclType ↦ Set(Element)	p. 109

native to OCL, due to the lack of supportive materials, mature tool implementations, existing knowledge base and so on. In this paper, we apply the lessons learned from our previous work to defining the OCL Query Library (OQAPI). A conservative extension to OCL such as this does not share the weaknesses of our previous approaches: it is easy to deploy wherever OCL is already in use. Thus, OQPAI trades reduced usability for extended usage scenarios.

[5] has clearly identified the need for OCL libraries. Still, however, it seems that no such libraries have been published, ever: none of the OCL experts we have consulted was aware of any such library. So, OQAPI is a first step to addressing a practical need, and our evaluation demonstrates the benefit of using OQAPI in an empirical way. En passant, it provides evidence to the common perception that OCL is difficult to use.

References