ABSTRACT

The Visual Model Query Language (VMQL) has been invented with the objectives (1) to make it easier for modelers to query models effectively, and (2) to be universally applicable to all modeling languages. In previous work, we have applied VMQL to UML, and validated the first of these two claims. In this paper, we apply VMQL to the Business Process Modeling Notation (BPMN) to evaluate the second claim. We explore the adaptations required, and re-evaluate the usability of VMQL in this context. We find similar results to earlier work, thus both supporting our claims and establishing the usability of VMQL beyond the realm of UML.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques—Computer-aided software engineering (CASE), Flow charts

Keywords
Model Querying, VMQL, BPMN, OCL, UML, Model Repositories

1. INTRODUCTION

1.1 Motivation

Some of the largest models in the world are process models, e.g., the SAP R/3 Reference Model. Such models store a wealth of information about organizational processes and their implementation, so it is vital that this information be accessible to a great number of people in the respective organization. However, most existing query facilities have not been conceived and implemented with a non-IT savvy user in mind: they offer only full-text search, a set of predefined queries, and/or a full-blown programming interface.

We are interested in model users that on the one hand require more than full-text search and predefined queries, yet are not able or willing to program a query that they want to run. This case is particularly prevalent in ad-hoc onetime-only querying of large models, where a full-text search invariably offers insufficient precision and programming is not economically efficient.

1.2 Approach

In prior work, we have introduced and evaluated the Visual Model Query Language (VMQL, [22, 23, 24]). We could show that it enjoys a number of desirable properties when compared to the de-facto standard for model querying in the context of UML, the Object Constraint Language (OCL, [17]):

- VMQL is much easier to use than OCL in the sense that modelers perform substantially better when tested empirically;
- VMQL imposes less learning effort than OCL;
- VMQL can easily be extended to act as a language for constraint definition, thus covering both main application areas of OCL (querying and constraint formulation).

There are also some drawbacks, however, in that VMQL is less expressive than OCL (which is Turing-complete). Also, VMQL has not yet been evaluated with regards to whether it can be implemented in a computationally efficient way; the existing implementation is relatively limited.

2. OVERVIEW OF VMQL

As its name suggests, VMQL is a predominantly visual language: it pursues the idea of query-by-example, that is, expressing model queries as (annotated) models. A VMQL query on a UML model is, simply put, a UML model fragment that may be annotated by a small number of constraints defined by VMQL. With a particular encoding of base models, such queries can be executed by similarity-based matching of models. By virtue of this architecture, we have argued, it should be easy to apply VMQL to modeling languages other than UML, for instance IDEF, ARIS/EPC, and BPMN, though we have so far not provided evidence to substantiate this claim.

We begin by introducing VMQL in its original setting as a query language for UML models. We will call the modeling notation host language and the model to be queried source model in the remainder. The complete set of source models is collected in the model base. Executing a query (that is, a query model) amounts to finding matching fragments in the source model and thus establishing an injective function
from model elements of the query model to model elements of the source model (binding) and values for all free variables in the query (valuation). Finally, results are displayed back to the user in the notation of the host language. 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. Consider the source model presented in Fig. 1, representing the product catalog of an insurance company. Fig. 2 contains examples of VMQL queries on this model. Query 1 finds all subclasses of the Product class - that is, it answers the question “What insurance products does this company offer?”. The indirect constraint in the query indicates that all transitive subclasses in the hierarchy should be returned. The mclass <: Class constraint indicates that instances of all subclasses of the Class UML meta class should be returned (including, for example, instances of the AssociationClass meta class). Finally, the mattr isAbstract = * constraint specifies that source model classes bound to the query model element may have any value for the isAbstract meta attribute: this way, the GroupPlan abstract class can also be returned.

Query 2 returns all classes which do not have any subclass. The not constraint used here specifies that the query model element to which it is associated (the subclass) must not appear in any bindings. Finally, Query 3 identifies candidate for a pull-up attribute refactoring: subclasses which have an attribute with the same name and type, in which case the attribute may be safely moved to the super class. The distinct constraint ensures that the two subclasses do not coincide, while the match * constraint states that the common attribute may have any name (“too” is just a placeholder for anchoring the VMQL constraints). Note the use of variables (denoted in VMQL by strings beginning with the $ character) to bind the name and type of the common attribute to the same value in both subclasses.

Table 1 lists all constraints supported by VMQL. Most are independent of the host language, as illustrated in Fig. 3. Here, the same query is expressed in three different notations: UML Class Diagrams, Coard/Yourdon notation, and Martin/Odell notation, respectively. Each query may be applied to a source model represented in the corresponding modeling notation. The any, steps = min, and steps = max constraints are introduced specifically for BPMN, and their usage is exemplified in Section 3.

3. ADAPTING VMQL TO BPMN

To exemplify the usage of VMQL on BPMN models, consider the source model in Fig. 4. The model represents the procedure followed by an insurance company for processing insurance quote requests. Several VMQL queries on this model are shown in Fig. 5 and discussed in the following paragraphs. The purpose of these examples is to demonstrate how VMQL may be applied to a BPMN source model. The examples do not cover all the constraints listed in Table 1, but they do address both the main positive and problematic aspects of querying BPMN models using VMQL. For comparison purposes, OCL versions of the queries in Fig. 5 are presented in Appendix A.

Suppose a business analyst is interested in finding all activities that deal with insurance coverage. He can succinctly express this request in VMQL using Query 4 from Fig. 5. Let us consider each element of this query and its role. The single diagram in the query model consists of a task named $A$ and a text annotation acting as a container for VMQL constraints. The name of the task indicates that the names of any matching activities in the source model must be bound to the variable $A$. The text annotation starts with the vxml keyword, indicating that it should be interpreted as a set of VMQL constraints. The mclass <: Activity constraint ensures that all BPMN activity types are considered. In this example, this constraint enables the query to also return the compute coverage collapsed subprocess, which according to the BPMN meta model is actually not a task (rather, Task and SubProcess are both subclasses of the Activity abstract meta class). Finally, the name = *coverage* constraint uses a regular expression to specify that all activities matching the query must contain the string “coverage” in their name.

Query 5 detects if the create account and consult account tasks are executed exclusively, in parallel, or in some other manner, depending on the gateway preceding them. The vxml mclass <: Gateway constraint indicates that the gateway preceding the tasks may be of any type as long as it is a subclass of the Gateway abstract meta class. The vxml steps = min constraint limits the number of possible matches by stating that only the gateway closest to the two tasks should be returned. Considering the source model, Query 5 will then yield that the two tasks are executed exclusively. The steps = min constraint, along with the converse steps = max constraint, were introduced to VMQL specifically for the context of BPMN. It should be noted that VMQL was initially designed with UML in mind, which (with the exception of Activity Diagrams), places less emphasis on flow constructs. As such, the BPMN-specific extensions to VMQL aim to improve its expressiveness for flow.
Figure 2: VMQL queries on the insurance product catalog for: finding all product types (top left), all classes which do not have sub-classes (bottom left), and all subclasses having a common attribute (right).

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Meaning</th>
<th>Introduced for</th>
<th>Applicable to</th>
</tr>
</thead>
<tbody>
<tr>
<td>match</td>
<td>Restricts the name of the constrained model element by a wildcard expression or regular expression, e.g., <code>match pattern*</code></td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>distinct</td>
<td>Enforces that a set of constrained model elements are pairwise distinct.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>once</td>
<td>Enforces that a solution occurs only once in the set of all solutions.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>all, all as set</td>
<td>Aggregates all solutions for the constrained elements into a single solution for the whole query, and builds collection variables indicated by terms initiated by `$$. Adding the option as set removes duplicates.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>any</td>
<td>When applied to a BPMN sequence flow, allows matched elements to belong to any subclass of SequenceFlow.</td>
<td>BPMN</td>
<td>BPMN</td>
</tr>
<tr>
<td>steps</td>
<td>Defines the length of a path between two connected model elements. Only one type of relationship may occur on the path. Applicable values are integers &gt; 0, <code>min</code> and <code>max</code> for shortest and longest path, or * for arbitrary length &gt; 0, e.g., <code>steps = 3</code>, <code>steps &lt; 3</code>, <code>steps = min</code>, <code>steps = *</code>, <code>steps = 2; 3</code>, or <code>steps &gt; 1</code>, <code>steps &lt; 4</code>.</td>
<td>UML (min and max values introduced for BPMN)</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>indirect</td>
<td>Defines a path of arbitrary length between two connected model elements; alias for <code>steps = *</code>.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>mclass</td>
<td>Allows the constrained element to be of a different meta class than actually specified in the query. E.g., the annotation <code>mclass = Class; Component</code> allows the annotated element to match with classes as well as components, <code>mclass &lt;: Classifier</code> will allow matching with all sub meta classes of Classifier, and <code>mclass = *</code> matches with any meta class.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>mattr</td>
<td>Constrains the value of a meta attribute that has or has not a representation in the concrete syntax. If given a value expression, the meta attribute’s value must conform to it. If given a variable (i.e. any expression starting with $), the value of the meta attribute is bound to that variable if possible. Variable may be bound several times, but only once for any meta attribute, e.g., <code>mattr isRoot = true, mattr aggregationKind = composition; none, mattr isAbstract = *, or mattr name = $N</code>.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>name</td>
<td>Alias for <code>mattr name</code>.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>precision</td>
<td>Reduces the model matching precision level to values below 1.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>strict</td>
<td>Enforces that a query element must match exactly with a result element, i.e., the binding is bijective rather than injective.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>not</td>
<td>Prevents results from containing matches for the constrained model elements.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>optional</td>
<td>Specifies that a model element may or may not appear in results.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>either</td>
<td>Allows a set of alternatives for a constrained model element of a query to appear in the result. Applicable only to non-empty sets of model elements.</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
<tr>
<td>in, sum, min, not_in,...</td>
<td>The usual functions to be used on collection variables (similar to OCL).</td>
<td>UML</td>
<td>UML, BPMN</td>
</tr>
</tbody>
</table>

Table 1: Overview of VQML constraints (see [24] for semantics).
Figure 3: Different notations for the same query: the same query as annotated class diagrams in UML (left), Coad/Yourdon (middle), and Martin/Odell notations (right).

Figure 4: Process model for handling coverage quote requests in an insurance administration system.

Figure 5: VMQL queries on the Coverage Quote Request Processing source model shown in Fig. 4.
related queries.

The goal of Query 6 is to determine which part of the process has access to the Contract data input before a quote is sent to the customer. The indirect constraint by itself does not suffice in this case, as it assumes that all relationships along a path must have the same type. Since in Query 6 the constraint is bound to an association, it will not include any sequence flows in the resulting bindings, resulting in an empty set of bindings (there is no sequence of associations linking the Contract data input to the send quote to customer task). The right answer set can be obtained by adding the mclass :<: SequenceFlow ; DataInputAssociation constraint, which states that both instances of mclass <: SequenceFlow ; DataInputAssociation constraint, which states that both instances of DataInputAssociation and SequenceFlow can be included in query solutions.

A similar yet potentially more confusing situation arises in Query 7. Here, the intention is to find all sets of tasks which can only be executed exclusively. Intuitively, this could be achieved by Query 7 even without including the VMQL constraint. In this form, the query would correctly identify the pair of tasks, but would fail to detect that the create account task may not be executed together with the consult account task. This is because the consult account task is preceded by an instance of DefaultFlow, while the query model contains an instance of NormalFlow. By adding the vmql any constraint to the normal flow in the query model, we specify that any kind of flow (that is, any subclass of the SequenceFlow meta class) should be matched. The any constraint is another addition made to VMQL with BPMN in mind. Note that implementing it requires considering the BPMN meta model, thus going beyond VMQL’s principle of only operating at a syntactical level, and making the constraint only applicable to BPMN. Nevertheless, it is illustrative of how VMQL may overcome the expectation miss-match that users often experience when a modeling language’s meta model does not conform to their intuition.

Query 8 illustrates the difference between syntactic and semantic querying. The intention of the query is to verify if the modify coverage limit task may be reached after the check general coverage task is executed. Syntactically, the source model contains a path between the two tasks. Therefore, Query 8 will return this path. However, the question of reachability may not be reliably answered without considering semantics. Indeed, the path connecting the two tasks in the source model contains a false condition on one of its flow arrows, with the intended meaning that control will never traverse this flow (i.e. task modify coverage limit will never execute). Answering this type of inquiry about process execution is a desirable but as yet unavailable feature in VMQL. Support for behavioural queries will require the use of model checking at the implementation level, in a similar manner to the solution proposed by query languages such as BPMN-Q [4] (see Section 5).  

4. USABILITY EVALUATION

We 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 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 them difficult to use for a fair number of (potential) modelers, because at least domain and requirements modelers are usually not software engineers or software developers. 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, the premise of our experiment is that using the source modeling language as the query and result presentation language should make querying models much easier for them.

4.1 Methods and materials

4.1.1 Study design

We created a controlled experiment with a randomized blocking design, treating subjects with two different query languages (VMQL vs. OCL), both expressing queries on BPMN source models. We observed scores and subjective assessment of the task difficulty. In order to control learning effects, we created two questionnaires with the sequence of the two treatments interchanged, and assigned participants randomly to one of them.

Our experiment consisted of three phases. First, participants were given a set of written instructions, including cheat-sheets on the various treatments. These instructions were issued orally, too. In a second phase, subjects were challenged with two tasks, both of which were first asked for one, then for the other query language. In Task A, subjects were presented with eight queries, each of which was given three interpretations in plain English. Exactly one of these was correct in every case. In Task B, subjects were presented with four queries in plain English and were asked to write down the corresponding query in one of the two query languages. In the third phase, subjects were asked to subjectively assess the difficulty of the tasks and treatments, and provide some demographic data. See Appendix B for more details on the questionnaire contents.

4.1.2 Participants

We tested 24 undergraduate students recruited from a class the authors teach. Participation was strictly voluntary: all participants explicitly consented to being part of the experiment. The subjects had little UML experience and little to no experience in either of the query languages or the host language (i.e., BPMN). The questionnaires were anonymous except for those participants that chose to identify themselves. Our study is governed by the principles laid down in the APA guide [2].

4.2 Observations and inferences

We first describe the scores subjects achieved in the reading task. Overall averages are reported in Table 2 and visualized in Fig. 6. We have normalized all numbers to the scale 1…10. We can see that the average score is considerably higher for the VMQL condition than for the OCL condition. Looking at individual scores, we find that 47% of the participants performed better with VMQL than with OCL, and only 32% performed better with OCL. The remaining 21% of the participants achieved a perfect score. We see a matching picture when considering the variance: the standard deviation is larger for the OCL condition and smaller for the VMQL condition. The results are statistically significant (two-tailed t-test, $p = 0.015$); the differences in the variances might be significant ($p = 0.54$, F-test).
Table 2: Individual scores in the reading and writing tasks, respectively.

<table>
<thead>
<tr>
<th>Language</th>
<th>Reading</th>
<th>Writing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$</td>
<td>$\sigma$</td>
</tr>
<tr>
<td>VMQL</td>
<td>8.00</td>
<td>4.00</td>
</tr>
<tr>
<td>OCL</td>
<td>6.82</td>
<td>4.65</td>
</tr>
</tbody>
</table>

Figure 6: Visualization of scores for reading and writing tasks (see Table 2 for means and standard deviations). Figures are normalized to the range 1...10.

With regards to the writing task, the scores are considerably lower than for reading, but again, the scores for the VMQL task are higher than for the OCL task. Despite the low number of data points (completion rate for this subtask was only 27% as compared to 84% for the reading task), these results are highly significant (two-tailed t-test yields $p = 0.009$, F-test yields $p < 10^{-6}$).

These findings are corroborated by the subjective assessments (see Fig. 7 and Table 3). Again, we have normalized all numbers to the scale 1...10. It is obvious that subjects assess the OCL tasks as much more difficult than the VMQL tasks. Testing these data with a two-tailed t-test, we find that the results are statistically significant, both for the assessment as such ($p = 0.047$), and even more so for the variance ($p = 0.026$). We see similar results for the writing-task ($p = 0.073$ and $p = 0.018$). The results for the assessment of effort points in the same direction, but are slightly less clear (not statistically significant).

4.3 Interpretation

It is obvious that participants perform better on average when using VMQL than when using OCL. This holds true for both the overall average and the individual performance, and both with regards to the absolute score and the variance. These findings are corroborated by the subjective assessments, where subjects consistently report lower cognitive load on the VMQL tasks than on the OCL tasks.

Probably the most indicative result is seen in the variances. Overall, the variances are relatively high, which we attribute to the somewhat low qualification level of the subjects. However, the variances are smaller for the VMQL conditions than for the OCL conditions, for all measurements. We explain this as a sign that the difficulty of using VMQL is smaller than the difficulty of using OCL, so that the subjective variation in general cognitive ability is more visible in the more difficult task. Or, put in other words: clever subjects can cope with the difficulty of OCL, less capable subjects cannot. For VMQL, there is less need to cope, and so we see less variance.

4.4 Threats to validity

There are several potential threats to validity. We eliminated bias through the experimenter by assigning the tasks randomly, providing written instructions, and asking the subjects to fill in and return the questionnaires anonymously. We eliminated bias through learning effects by the randomized blocking design of our study, with all treatments occurring with approximately the same frequency. This way, any learning effects are canceled out.

Bias through unrepresentative population sample is controlled by a relatively large sample size ($n = 24$), but it must be said that the population is relatively homogeneous.
Table 3: Subjective assessment of individual difficulty for different tasks and for different treatments (rows). Figures are normalized to the range 1 . . . 10.

<table>
<thead>
<tr>
<th>Assessment Task Language</th>
<th>Difficulty</th>
<th>Effort</th>
<th>Difficulty</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>µ</td>
<td>σ</td>
<td>µ</td>
<td>σ</td>
</tr>
<tr>
<td>VMQL</td>
<td>5.47</td>
<td>1.64</td>
<td>5.80</td>
<td>1.81</td>
</tr>
<tr>
<td>OCL</td>
<td>6.95</td>
<td>2.79</td>
<td>6.99</td>
<td>2.42</td>
</tr>
</tbody>
</table>

and might not represent the target audience adequately. Another potential source of bias is the measurement procedure, in particular wrt. cognitive load measures. We have taken two different measurements that can be understood as aspects of cognitive load (cf. [19]). Both of these measurements show the same effect, though to varying degrees. Using subjective assessments rather than objective measures such as skin conductivity or pupillary dilatation is justified by the high correlation between subjective and objective assessments of cognitive load (cf. [10]). Only some of the observations are statistically significant, but they are highly consistent. This means that they support and corroborate each other, allowing valid conclusions.

5. RELATED WORK

Model querying has received attention from both the Business Process Modeling (BPM) and Model Based Software Development (MBSD) communities. This section presents existing work on this topic from the two areas, justifying our claim that VMQL is the first model query language to purposefully address the requirements of both.

OCL features prominently as a model query language in the context of MBSD, mainly due to its close association with the UML [1]. However, as demonstrated in Section 4, OCL lacks usability. This may explain why it has failed to gain traction with BPMN users, despite its well documented expressive power [13]. Building on the foundation of OCL, the Query/View/Transformation (QVT, [16]) standard proposes a pair of imperative and declarative languages (QVT-Operational and QVT-Relations) for querying MOF-based models. Just as OCL, these languages do not appear to have been adopted by BPMN practitioners.

Apart from the existence of the OMG standards, MBSD does not seem to emphasize model querying as a primary concern, and often subsumes it to the more general problem of model transformation. Among the relatively few dedicated model query languages proposed in this area, Constraint Diagrams (CD, [12]) adopt a query-by-example approach comparable to that of VMQL. Unlike VMQL, CDs introduce additional notation elements to UML. The approach is not implemented, and is restricted to querying UML Class Diagrams.

In the context of Aspect Oriented Modeling (AOM), a crosscut specification is a direct application of model querying. Join Point Designation Diagrams (JPDD, [21]) are a model querying approach designed for crosscut specifications that may also be used for expressing general model queries [20] and model constraints [25]. JPDDs extend the concrete syntax of UML Class Diagrams, Object Diagrams, and Sequence Diagrams to allow the specification of query patterns. JPDD queries can be mapped to OCL and executed on models, and are implemented by the M4JPDD Eclipse plugin [25]. The main disadvantage of JPDDs when compared to VMQL is that they only cover a relatively small subset of UML, leaving out behavioural models such as Activity Diagrams. Another AOM approach that bears some resemblance to VMQL is MATA [27], which proposes an aspect composition language based on graph transformation rules expressed in concrete syntax. The left-hand side (LHS) of a MATA rule is similar to a VMQL query, but lacks the expressiveness afforded by VMQL’s annotations. The benefit of the MATA implementation is that graph transformations provide a well defined formal semantics and allow static analysis of rules (e.g. critical pair analysis). However, MATA is also limited in scope, covering only UML Class Diagrams, Sequence Diagrams, and State Machine Diagrams.

Model querying plays a central role in BPM, where it is a fundamental operation when working with both individual models and collections of models. The role of model querying is more clearly defined in BPM than in MBSD: queries are used to verify standards compliance, identify refactoring candidates, and identify process templates representing candidates for re-use [8]. Approaches proposed for querying business process models are not all targeted at the same modeling language: while some primarily address the BPMN notation [3, 7, 15], others focus on different visual or textual representations of business processes [5, 11, 14, 28, 6].

Some business process query languages have a purely textual syntax. This is the case with BPQL [14], which operates on the XPDL [26] business process interchange format, and BQL [11], which operates on Petri nets as abstractions of business process models. The approach featured in [6], on the other hand, proposes an XML dialect for representing queries on a custom process repository. This approach has the advantage that any XML tool may be used to create and validate queries. It is also well suited for supporting a query specification GUI, but may result in query definitions which are too verbose to be feasibly constructed by hand.

BP-QL, a visual language for querying processes expressed using the Web Services Business Process Execution Language (WS-BPEL, [18]) is proposed in [5]. Queries are expressed in a notation based on WS-BPEL, and executed by processing the XML representation of models. BP-QL has been implemented, but is restricted both conceptually and practically to querying WS-BPEL models. Another query-by-example approach is presented in [28], where a feature-based search is adopted with the goal of improving query processing times. Both models and queries are represented in a custom graph-based notation: Process Graphs, and Query Process Graphs, respectively. Models may be translated to this generic notation at query execution time.

BPMN-Q [3] is one of the query languages explicitly addressing BPMN models. It is conceptually similar to VMQL, though it replaces model annotations with a number of special visual notation elements. BPMN-Q has also been adapted
to verify model constraints [4] using Computation Tree Logic (CTL) [9]. Another approach to querying BPMN models is the Process Pattern Modeling Language (PPML) [15], which addresses only structural queries. PPML provides a notation similar to BPMN for expressing “patterns” (that is, queries) to be identified within a host BPMN model. Queries are executed by a rule-based inference engine, which also supports semantic annotations of process models. BPMN-VQL [7] proposes a similar structural querying solution for BPMN, with an increased focus on semantic querying. While all of these languages adopt a query-by-example philosophy, none of them propose an equivalent to the annotations offered by VMQL, thus limiting the queries they may express. Also, none of the mentioned BPMN query approaches takes advantage of BPMN’s MOF foundation to explore querying other modeling languages, as is the case with VMQL.

6. CONCLUSIONS

6.1 Summary and contributions

In this paper we have successfully adapted the Visual Model Query Language (VMQL) to express BPMN queries, in addition to its original UML querying capabilities. Only very minor additions (namely, introducing the any, steps = min, and steps = max constraints) had to be made to the language to allow formulating queries on BPMN models. This supports our original hypotheses that VMQL is indeed universally applicable. At the same time, we were able to maintain the usability advantage that VMQL exhibited over OCL when tested on UML.

6.2 Limitations and future work

Our work provides contributions to a more general and comprehensive comparison of model querying approaches. There are three main issues we have not addressed so far. First, in the process modeling world, model querying enjoys a much greater attention than in the UML modeling world. Thus, there are many more approaches to querying BPMN and EPC models than there are to querying UML models. We have so far not compared the usability of our approach to that of any of the model query languages that have been proposed for BPMN model querying.

Second, we do not yet have any data on how our approach and others compare in terms of performance, which is clearly an important question for practical applications. It would be interesting, in this context, to study whether VMQL is suitable for querying large model repositories such as CDO, EMFStore, Morsa, and ModelBus\(^1\). An important step towards this goal is to extend our current implementation of VMQL, which is only functional for UML models, allowing it to process languages such as BPMN and EPCs.

Third, one advantage OCL has over all other model querying approaches is the fact that it offers a formal semantics. This is an essential ingredient to argue about the correctness of queries, and determine the expressiveness of a query language. For these purposes, we plan to endow VMQL with a formal semantics.


7. REFERENCES


APPENDIX

A. BPMN QUERIES EXPRESSED IN OCL

OCL versions of the VMQL queries presented in Fig. 5 are listed below. Listings 1-5 contain OCL query definitions, while Listing 6 defines several helper operations.

Listing 1: Query 4 expressed in OCL

```ocl
context Process
def: query4() : Set(Activity) =
    Activity.allInstances()->select(name.indexOf('coverage') <> 0)
```

Listing 2: Query 5 expressed in OCL

```ocl
context Process
def: query5() : Set(Gateway) =
    Task.allInstances()->select(name='check general coverage')
    ->collectNested(closestPrecGateways())
    ->flatten()->asSet()
    ->intersection(
        Task.allInstances()->
        select(name='create account')
        ->collectNested(closestPrecGateways())
        ->flatten()->asSet())
```

Listing 3: Query 6 expressed in OCL

```ocl
context Process
def: query6() : Boolean =
    Task.allInstances()->
    select(oclIsKindOf(Task))
    ->collect(oclAsType(Task)).dataInputs()->
    exists(name = 'contract')
```

Listing 4: Query 7 expressed in OCL

```ocl
context Process
def: query7() : Set(Task) =
    ExclusiveGateway.allInstances()->
    collectNested(outgoing)
    ->flatten()
    ->collect(targetRef)
    ->select(oclIsKindOf(Task))
    ->collect(oclAsType(Task))
    ->asSet()
```

Listing 5: Query 8 expressed in OCL

```ocl
context Process
def: query8() : Boolean =
    Task.allInstances()->
    select(name='modify coverage limit')
    ->collectNested(indirectPrecFlowNodes())
    ->flatten()->
    exists(name='check general coverage')
```
Listing 6: OCL helper operations

context Activity
def: dataInputs() : Set(DataInput) =
    dataInputAssociations
    ->collectNested(sourceRef)
    ->flatten()
    ->select(oclIsTypeOf(DataInput))
    ->collect(oclAsType(DataInput)) ->asSet()

context FlowNode
def: indirectPrecFlowNodes() : Set(FlowNode) =
    if (incoming ->isEmpty()) then Set()
else
    incoming
    ->collect(sourceRef)
    ->union(
        incoming
        ->collect(sourceRef)
        ->flatMap(sourceRef.indirectPrecFlowNodes())
        ->flatten())
    ) ->asSet()
endif

context FlowNode
def: closestPrecGateways() : Set(Gateway) =
    if (incoming ->collect(sourceRef)
        ->isNotEmpty()) then Set()
else
    if (incoming ->collect(sourceRef)
        ->exists(oclIsKindOf(Gateway))) then
        incoming
        ->collect(sourceRef)
        ->select(oclAsType(Gateway))
        ->asSet()
    else
        incoming
        ->collect(sourceRef)
        ->collectNested(closestPrecGateways())
        ->flatten()
        ->collect(oclAsType(Gateway))
        ->asSet()
    endif
endif

B. QUESTIONNAIRE SUMMARY

Two versions of the questionnaire were handed out to study participants in order to control learning effects. One version begins each task with VMQL questions, while the other begins with OCL questions. To mitigate participant biases about the query languages, VMQL and OCL are referred to as Language X and Language Y, respectively. The questionnaire is structured into the following blocks:

- **Introduction to BPMN:** The BPMN meta model elements appearing in the questionnaire are introduced.
- **Introduction to the Query Languages:** The subsets of VMQL and OCL required to complete the Tasks A and B are introduced.
- **Task A - Comprehension:** Participants are shown eight VMQL and OCL queries, and are asked to select the appropriate plain English interpretation of each.
- **Task B - Production:** Participants are shown four queries in plain English, and are asked to write down the queries’ VMQL and OCL representations.
- **Subjective Assessment:** Participants provide ratings for the perceived difficulty and effort involved in completing Tasks A and B using a 5-point Likert scale.

As an example, Fig. 8 presents the VMQL version of Question 2 of Task A. The OCL version of the same question is presented in Listing 7. The queries are equivalent, and their purpose is to identify all Activities executed immediately before the EndEvent of a Process. To answer the question, participants must select one of the following options (of which option C represents the correct answer):

A Which Activities are executed more than once?
B Which Task precedes the first ExclusiveGateway?
C Which Activities are executed immediately before the EndEvent?
D I don’t know.

Listing 7: Question 2 of Task A expressed in OCL

context Process
def: query() : Set(Activity) =
    self.flowElements ->select(
        e | e.oclIsKindOf(Activity) and
        e.outgoing ->collect(
            targetRef.oclIsTypeOf(EndEvent)
            ) ->size > 0)

In Task B, participants are presented with plain English query descriptions, and must produce VMQL and OCL queries adhering to these descriptions. Participants are instructed not to consult Task A, so that the produced queries are a result of their own understanding of VMQL and OCL. The following four plain English query descriptions were used:

A Which Tasks are not connected to the End Event?
B Which Parallel Gateways have branches that start with the same task?
C What precedes Task A?
D What are the Choreography Tasks on which Participant A and Participant B collaborate?