View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Online Research Database In Technology

Technical University of Denmark

Efficient Model Querying with VMQL

Acretoaie, Vlad; Störrle, Harald

Published in:
Proceedings of the First International Workshop on Combining Modelling with Search- and Example-Based Approaches (CMSEBA 2014)

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Acretoaie, V., & Störrle, H. (2015). Efficient Model Querying with VMQL. In R. Paige, M. Kessentini, P. Langer, & M. Wimmer (Eds.), Proceedings of the First International Workshop on Combining Modelling with Search- and Example-Based Approaches (CMSEBA 2014) (pp. 7-16). (CEUR Workshop Proceedings, Vol. 1340).

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Efficient Model Querying with VMQL

Vlad Acretoaie and Harald Störrle

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

Abstract. *Context:* Despite model querying being an important practical problem, existing solutions lack either usability, expressiveness, or generality. The Visual Model Query Language (VMQL) is a query by-example solution created to satisfy these requirements simultaneously.
Objective: In the present paper we study whether VMQL queries can be executed in an efficient way, such that VMQL is suitable for ad-hoc model querying in practical settings involving large models.
Method: We study VMQL query execution performance on sets of models ranging over a broad spectrum of sizes and degrees of complexity. The models are based on large and realistic case studies.
Results: We observe that our approach exhibits competitive performance, while providing superior usability and generality.

1 Introduction

Interactive model querying is an important task in most modeling scenarios. However, manually browsing the model for query purposes is only practical for trivially small models. Consequently, several model querying approaches have been proposed.

Query By Full-Text Search is very easy to use. All major modeling tools support this feature, with varying degrees of refinement (e. g., wildcards, regular expressions, filters, logical connectors). However, it offers only limited expressiveness, and often yields a large number of false positive results. Furthermore, it cannot be used to search for structural or meta-level information in a model.

Query By Navigation provides search results by traversing a model via links between model elements. This includes APIs for programming queries explicitly, and executing queries expressed in the Object Constraint Language (OCL [9]) or a similar formalism. Navigational approaches expose the meta-level structure of models to users, which both accounts for their versatility and the high demands on the expertise of the user formulating queries.

Query By-Example allows expressing queries as small, possibly annotated, model fragments that are matched against a source model. Wherever a sufficient degree of similarity is found, a binding between the two models can be established, which corresponds to one query result. One of the exponents of this approach is the Visual Model Query Language (VMQL [13]).

In previous work we have shown that VMQL queries are easier to understand and formulate than their OCL counterparts, even when OCL is enhanced with

a query API partially hiding meta-model complexity [13]. However, VMQL has so far not been demonstrated to be Turing-complete, so it is less expressive than OCL in a theoretical sense. Originally defined for UML, VMQL has since been extended to also allow querying BPMN [10] models, as detailed in [14]. Beyond queries, VMQL can also be used to specify model constraints [12].

VMQL addresses the ad-hoc querying application scenario, where a user (often a domain expert) explores a model interactively. Ad-hoc model querying depends on (a) an expressive query language of high usability, and (b) query processing that is fast enough to satisfy an interactive operation.

However, query by-example approaches generally do not scale well, since both source and query models are treated as graphs. Executing a query amounts to mapping the query model graph Q to the source model graph S, and returning bindings $b: Q \to S$ between the elements of the query and source models. A valid binding must bind *all* query elements, i. e., every b is total, so there are up to $B = |S|^{|Q|}$ bindings. Enumerating all the bindings is only an option for small values of S and Q. In the case of ad-hoc queries, $|Q| << |S|$ and $|Q|$ is not very large (typically in the range 5..20 model elements). S, on the other hand, can be very large: models with $|S| \approx 10^3 ... 10^5$ elements are common. Thus, a naive approach cannot deliver the performance required for ad-hoc querying.

The remainder of this paper is structured as follows. Section 2 briefly presents VMQL, Section 3 describes the algorithms underlying the execution of VMQL queries and provides implementation details, Section 4 presents an evaluation of VMQL query execution performance, Section 5 highlights related work, and Section 6 concludes the paper and outlines future work.

2 Presenting VMQL By-Example

Our running example represents a process model for handling insurance benefit claims. The source model is the UML Activity Diagram "Coverage Quote Processing" in Figure 1, while the query model is the UML Activity Diagram "Query A" in the same figure. The red dashed lines highlight a valid binding. In this case, this is the only binding. Since "Query A" does not use any annotations or variables, it is a *base query*. Additional queries are shown in Figure 2.

Query B introduces VMQL variables: strings starting with the `$` character which typically store the value of a source model meta-attribute. The query finds all `Actions` which can raise an exception to be handled by the "manual coverage quote processing" `Action`. It returns two bindings, with the `$FailingAction` variable instantiated to the names of the matching `Actions`: "compute quotable coverage offering" and "verify customer account", respectively.

Query C introduces VMQL *constraints* represented as `Comments` stereotyped by `<<vmql>>`. The `steps` constraint indicates that matches may include paths of indefinite length containing only `Flows` of the same type as the `Flow` to which the constraint is anchored. The `name` constraint implies a particular name for a model element (in this case, any string starting with "check"). Thus, the query

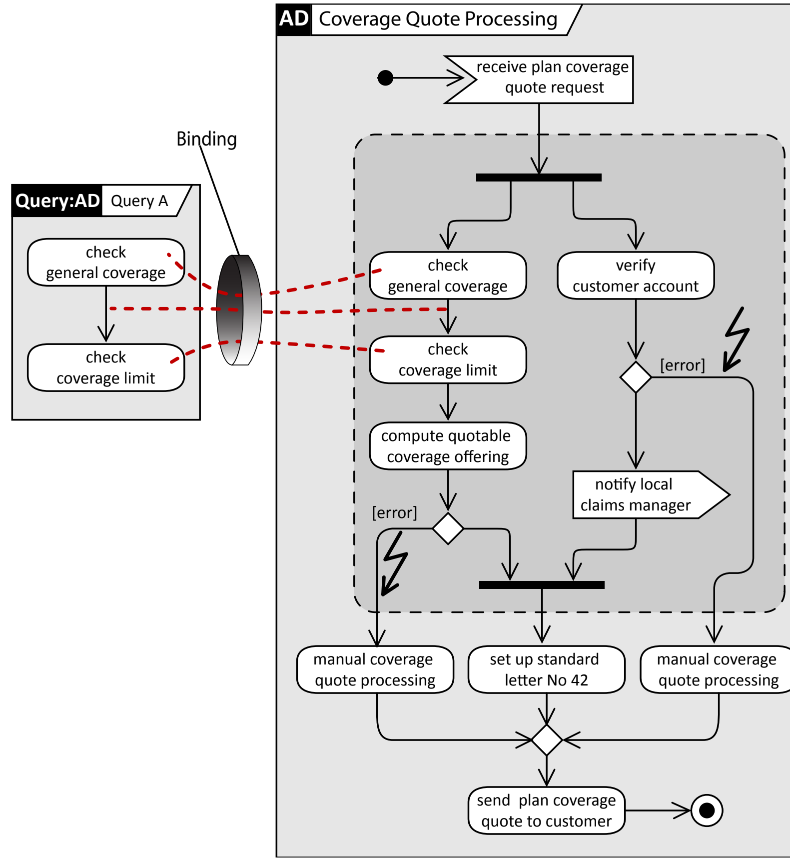


Fig. 1. Answering a query amounts to finding valid mappings from query model elements (left) to source model elements (right).

`Action` is bound to the source model `Actions` "check general coverage" and "check coverage limit", respectively.

Query D is a more general version of Query A, using wildcards. Executing query D will result in the same binding as Query A. Queries A through D only illustrate the most basic constructs of VMQL; a complete list is available in [13].

3 Algorithms and Implementation

VMQL queries are executed by computing a set $B = \{b_1, b_2, ..., b_n\}$ of bindings between the elements of a query model and those of a source model. A binding $b_i = \{(q_id_1, s_id_1), (q_id_2, s_id_2), ..., (q_id_m, s_id_m)\}$ is a set of m pairs, each consisting of the ID of a query model element and that of the matching source model element, where m is the number of elements of the query model.

Query model matching is performed in two stages. First, source model elements that do not match any query model element are quickly discarded. This

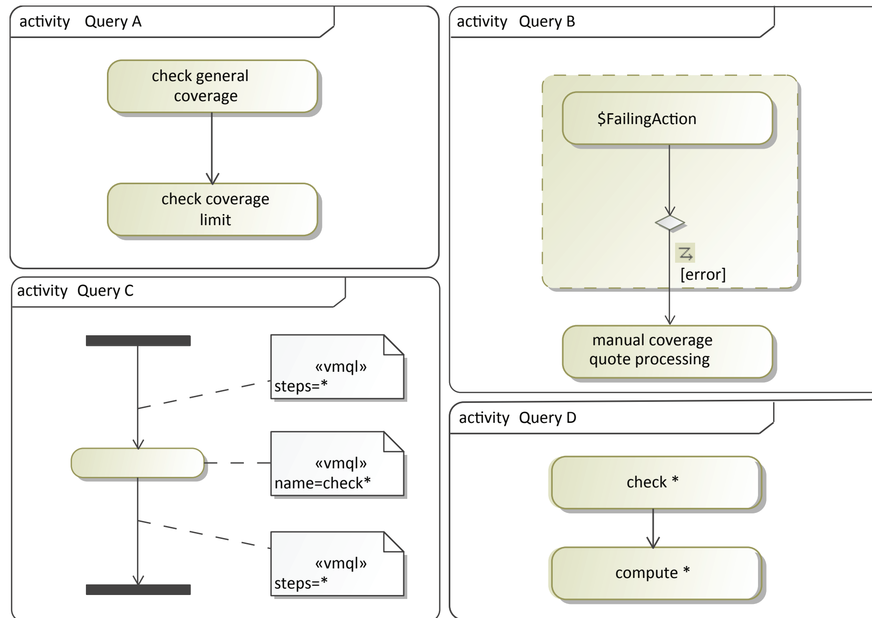


Fig. 2. Some sample VMQL queries: a base query, a query including a variable, a query including a wildcard expression, and a query including paths of undefined length and a wildcard expression (clockwise from top left)

is achieved by only considering element types and meta-attributes with fixed values, while ignoring meta-attributes that represent references to neighbouring model elements.

In the second stage, Algorithm 1 also takes into account references between query model elements. It terminates upon the convergence of two binding lists: *old_bindings* and *new_bindings*. At each iteration, a new version of the bindings is computed by considering each query/source model element pair and verifying its consistency with the rest of the bindings through the VERIFY_LINKS function, which checks if model element references in the query model are reflected by the source model. Additional functions are required for applying VMQL constraints to the results of Algorithm 1. Since constraints are applied locally to binding elements, their processing time is linear in the size of the query model.

VMQL is implemented by the MQ-2 tool [1], a plug-in for the MagicDraw modeling environment [8]. MQ-2 may be downloaded[1] or tested online through SHARE [15]. The query execution engine is implemented in SWI-Prolog [17]. Models are stored as Prolog hash maps of model elements, so loading a model amounts to consulting a Prolog knowledge base. Each model element takes the form `me(type-id, [tag-value, ...])`, where `type` is a model element's meta-class, `id` is an arbitrary unique identifier, `tag` is an atom representing one of the element's properties, and `value` is the respective value for this property.

[1] `http://www2.compute.dtu.dk/~hsto/tools/tools.html`

Algorithm 1 REFINE_BINDINGS

```
1:  Inputs:
2:      bindings - a list of bindings
3:  Outputs:
4:      r_bindings - a refined list of bindings
5:  begin
6:  old_bindings ← bindings
7:  new_bindings ← bindings
8:  repeat
9:      old_bindings ← new_bindings
10:     new_bindings ← ∅
11:     for all (q_id, s_ids) ∈ old_bindings do
12:         new_s_ids ← ∅
13:         for all s_id ∈ s_ids do
14:             if VERIFY_LINKS(old_bindings, q_id, s_id) then
15:                 new_s_ids ← new_s_ids ∪ {s_id}
16:             end if
17:         end for
18:         new_bindings ← new_bindings ∪ {(q_id, new_s_ids)}
19:     end for
20: until old_bindings = new_bindings
21: return new_bindings
22: end
```

4 Evaluation

4.1 Methods and Materials

To evaluate the performance of our approach, we investigate different sizes of source and query models, where model size is measured as the number of elements included in a model. The model samples originate in students' course work in graduate and undergraduate courses at the authors' host institution. They consist of UML Activity Diagrams representing analysis-level process models created by teams of 4–6 people over several months. We split the models into three batches, each consisting of 7 source models, 14 successful query models (i.e. queries producing bindings), and 14 unsuccessful query models (i.e. queries not producing bindings). Source model sizes range between 50 and 1000 model elements, while query sizes range between 1 and 50 elements. Each query execution is replicated 20 times in the interest of accuracy. All queries and their observed execution times are available online[2]. The experiments are executed on a Windows 7 Enterprise machine with 8 GB DDR3 memory and an Intel Core i7-3630QM CPU. The 64-bit version of SWI-Prolog 6.6.1 is used, with the limits of the local, global, and trail stacks set to their default values of 256 MB each.

[2] http://www.compute.dtu.dk/~rvac/experiments/vmql_performance

Table 1. Execution times of successful queries measured in milliseconds; model size is measured as the number of contained model elements.

Src.	**Query size (no. of model elements)**													
size	1	2	3	4	5	6	7	8	9	10	20	30	40	50
50	.3	.9	1.4	1.5	2.2	2.7	3.5	4.3	5.4	5.1	16.9	33.1	59.3	104.2
100	.3	.7	2.0	2.3	3.0	3.8	5.1	6.5	8.0	7.9	29.3	47.8	82.4	159.1
250	.3	1.6	5.3	5.9	10.4	8.9	13.1	17.6	28.9	23.8	86.3	129.0	197.6	320.6
500	.6	3.0	15.4	13.2	16.1	19.0	30.9	48.8	57.9	67.3	231.9	336.3	500.8	736.1
750	.9	5.2	21.9	24.1	28.8	33.8	63.8	82.2	117.0	130.3	464.2	670.0	999.2	1389.4
1000	1.2	8.0	34.1	37.3	44.1	51.4	91.1	135.6	182.6	227.6	756.6	1074.3	1644.3	2246.8

Table 2. The influence of VMQL constraints on query execution time; model size is measured as the number of contained model elements.

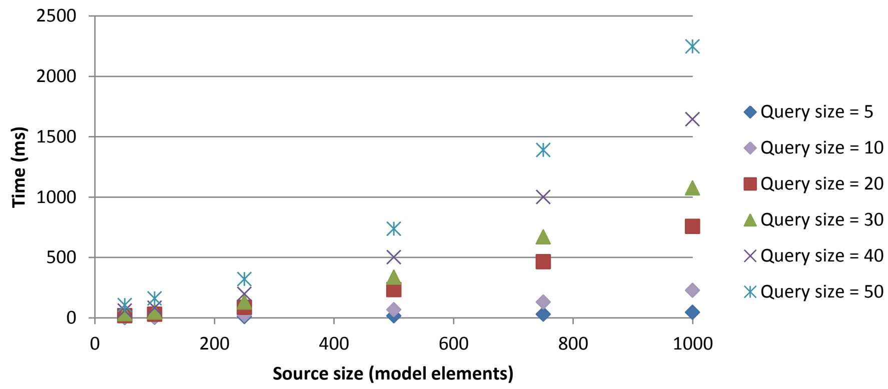
Query	**Query size**	**Query contents**	**Execution time (ms)**
A	4	base query	2.4
B	7	VMQL variable	6.5
C	6	path constraint, wildcard	21.4
D	4	wildcard	2.0

4.2 Observations

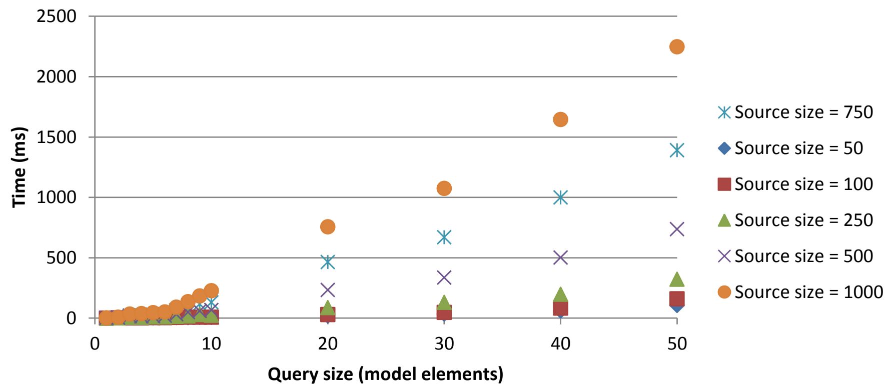
Execution times for all combinations of source and successful query models are listed in Table 1. As Figures 3a and 3b show, increasing either the source or the query model size yields longer execution times. In both cases, it appears that execution time increases at a low polynomial rate. The highest observed times are approximately 2s, while the execution times for most realistic ad-hoc queries (i.e. those with less than 20 elements) are below 1s even for large source models. For source models with less than 250 elements, query execution is instantaneous from a user's perspective.

Unsuccessful queries are executed faster than successful ones (see Figure 4). Also, the execution time of unsuccessful queries is less dependant on source model size. This is due to the "fail early" approach of our algorithms: easily verifiable properties are considered first in order to prune the search space.

Table 2 shows the observations for executing the more complex queries introduced in Section 2. The base query (Query A) and the query containing a wildcard constraint (Query D) have the lowest execution times. Query B has a slightly higher execution time, likely due to the extra variable instantiation step. As expected, the path constraint in Query C makes it the most time consuming in our sample. Paths of indefinite length require computing transitive closures, which is problematic for most model querying approaches (see [2, 18]).



(a) Execution times by source size for various query sizes



(b) Execution times by query size for various source sizes

Fig. 3. The influence of source and query size on query execution time

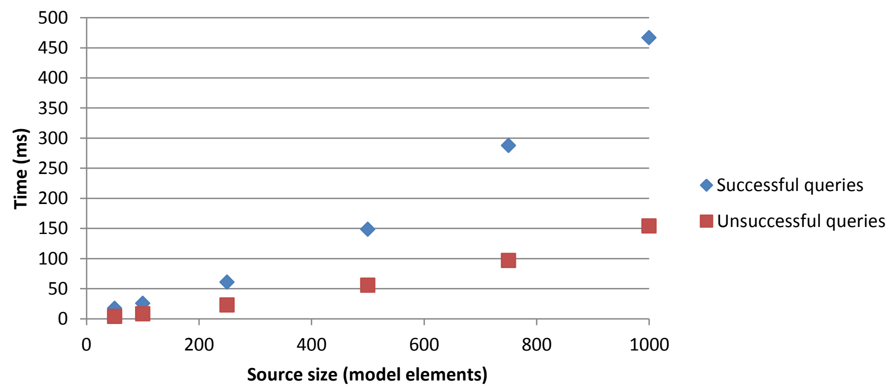


Fig. 4. Execution times for successful and unsuccessful queries

Table 3. Query execution times reported in the literature for various model querying approaches; model size is measured as the number of contained model elements.

Approach	Applicability	Source size	Query size	Time (ms)
VMQL (this)	Model querying	50..1000	1..50	0.3..2,247
EMF-IncQuery [5]	Model querying	6×10^5	~ 15	20..30
BP-QL [4]	Model querying	~ 30	~ 10	2,000..4,000
IncQuery-D [6]	Repository querying	$10^5..5 \times 10^7$	8	10..40
Hawk [3]	Repository querying	$7 \times 10^4..5 \times 10^6$	-	$172..15 \times 10^3$
MorsaQL [11]	Repository querying	$7 \times 10^4..5 \times 10^6$	-	13..180
BPMN-Q [2]	Repository querying	42418	3..13	190..284,000
FNet [18]	Repository querying	12300+	2..55	10..920
YNet [7]	Repository querying	15×10^6	5..202	30..80

4.3 Threats to Validity

The largest threat to validity of our evaluation is the nature of the models it is based on: they originate in academical course work. However, the models are realistic in the sense that their structure, size, and complexity resemble those of models that we have encountered in industry. The same caveat applies to the queries, as we cannot assert that the sample used is representative of practical applications. Furthermore, there is no research on what realistic queries are. We attempt to mitigate this threat by considering a variety of query sizes and types.

4.4 Comparison and Discussion

Models an order of magnitude larger than the ones we have considered are not uncommon in practice. Based on Figure 3b, we estimate that a VMQL query consisting of 20 model elements will be executed on a model containing 10^4 elements in ~ 10 seconds. Similarly, based on Figure 3b, we deduce that queries containing more than 50 elements lead to execution times in the order of seconds on moderately large models. Therefore, scenarios such as large-scale online model differencing for model version control are outside the scope of our solution.

Table 3 summarizes performance experiments reported for model and process querying approaches. The performance of VMQL is comparable to existing ad-hoc model querying solutions, but is surpassed by model repository querying approaches which employ offline element indexing, a process requiring up to several hours to complete [7]. However, Table 3 cannot form the basis of an accurate comparison, since the experiments were performed under different conditions.

As mentioned in Section 1, query by-example solutions are more difficult to scale than navigational approaches (e. g. OCL) due to the large number of source model elements to be considered at the start of the matching process. To address this issue, the VMQL matching algorithm prunes the search space

based on element types and fixed-valued meta-attributes. Local-search based graph pattern matching improves this heuristic by taking into account common structures identified in a set of typical models or in the source model itself [16].

Due to the generality of the matching algorithm, the performance figures reported for VMQL are independent of modeling language and diagram type.

5 Related Work

Model querying is an important topic in Model-Based Software Engineering (MBSE), both in itself and as a part of other operations (e. g. model transformation). In this area, the topic of fast ad-hoc querying has been addressed by EMF-IncQuery [5], an approach based on incremental graph pattern matching.

In Business Process Modeling (BPM), model querying plays several roles, including compliance verification and process template discovery. BP-QL [4] is a matching-based business process model querying language. Execution times for BP-QL queries are in the order of seconds, increasing polynomially with source model size and exponentially with query model size.

The related problem of efficiently querying model repositories has been addressed by both the MBSE and BPM communities. Approaches originating in MBSE (IncQuery-D [6], Hawk [3], and MorsaQL [11]) derive their performance from offline indexing and the adoption of efficient storage layers such as dedicated graph databases. Meanwhile, BPM-based solutions (BPMN-Q [2], FNet [18], and YNet [7]) place a higher emphasis on algorithmic efficiency, while also taking advantage of offline indexing techniques. In all cases, the indexing process is time consuming, reported at 405 seconds for the SAP R/3 Reference Model in the case of BPMN-Q, and at 25 hours for a repository containing 6×10^5 models in the case of YNet. This aspect implies that solutions relying on model pre-processing may not be suitable for the task of ad-hoc querying.

6 Conclusions

In this paper we have shown that the performance of VMQL is adequate for the task of interactive, ad-hoc model querying. On the other hand, VMQL is outperformed by approaches designed for model repository querying. However, such approaches require substantial model pre-processing, which makes them less suitable for the ad-hoc model querying scenario addressed by VMQL.

We have previously shown that, compared to other model query languages, VMQL offers superior usability and generality [13, 12], and supports a broad application scope [14]. The present performance evaluation adds to the body of evidence placing VMQL as a competitive model query language.

As future work, we intend to study the scalability of our implementation to ultra-large sized models, as well as the impact of VMQL annotations on performance. We also plan to extend our approach to address model warehouse querying. This will likely demand a redesign of the matching algorithm to take advantage of caching techniques and other types of pre-processing.

References

1. Vlad Acretoaie and Harald Störrle. MQ-2: A Tool for Prolog-based Model Querying. In *Proc. co-located Events 8th Eur. Conf. on Modelling Foundations and Applications (ECMFA'12)*, pages 328–331.
2. Ahmed Awad and Sharif Sakr. On efficient processing of BPMN-Q queries. *Comp. Ind.*, 63(9):867–881, 2012.
3. Konstantinos Barmpis and Dimitris S. Kolovos. Towards Scalable Querying of Large-Scale Models. In *Proc. 10th Eur. Conf. on Modelling Foundations and Applications (ECMFA'14)*, volume 8569 of *LNCS*, pages 35–50. Springer, 2014.
4. Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. Querying business processes with BP-QL. *Inf. Syst.*, 33(6):477–507, September 2008.
5. Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental Evaluation of Model Queries over EMF Models. In *Proc. 13th Intl. Conf. Model-Driven Engineering Languages and Systems (MODELS'10)*, volume 6394 of *LNCS*, pages 76–90. Springer, 2010.
6. Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. IncQuery-D: Incremental Graph Search in the Cloud. In *Proc. Ws. Scalability in Model Driven Engineering (BigMDE'13)*, pages 4:1–4:4, New York, NY, USA, 2013. ACM.
7. Tao Jin, Jianmin Wang, Marcello La Rosa, Arthur ter Hofstede, and Lijie Wen. Efficient querying of large process model repositories. *Comp. Ind.*, 64(1):41–49, 2013.
8. NoMagic INC. MagicDraw UML 17.0.3, 2014. `http://www.nomagic.com/products/magicdraw`.
9. Object Management Group (OMG). Object Constraint Language (OCL), Version 2.3.1, 2012.
10. Object Management Group (OMG). Business Process Model and Notation (BPMN), Version 2.0.2, 2013.
11. Javier Espinazo Pagán and Jesús García Molina. Querying large models efficiently. *Inf. Softw. Tech.*, 56(6):586–622, 2014.
12. Harald Störrle. Expressing Model Constraints Visually with VMQL. In *Proc. IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC'11)*, pages 195–202. IEEE Computer Society, 2011.
13. Harald Störrle. VMQL: A Visual Language for Ad-hoc Model Querying. *J. Visual Languages and Computing*, 22(1), February 2011.
14. Harald Störrle and Vlad Acretoaie. Querying Business Process Models with VMQL. In *Proc. 5th ACM SIGCHI Ann. Intl. Ws. Behaviour Modelling – Foundations and Applications (BMFA'13)*. ACM, 2013.
15. Pieter Van Gorp and Steffen Mazanek. SHARE: a web portal for creating and sharing executable research papers. *Procedia Comp. Sci.*, 4:589–597, 2011.
16. Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Softw. Syst. Model.*, pages 1–25, 2013.
17. Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.
18. Zhiqiang Yan, Remco Dijkman, and Paul Grefen. FNet: An Index for Advanced Business Process Querying. In *Proc. 10th Intl. Conf. Business Process Management (BPM'12)*, volume 7481 of *LNCS*, pages 246–261. Springer Verlag, 2012.