Querying UML Models using OCL and Prolog: A Performance Study*

Joanna Chimiak–Opoka, Michael Felderer, Chris Lenz
Institute of Computer Science,
University of Innsbruck
Technikerstrasse 21a, A–6020 Innsbruck

Christian Lange
Dep. of Math. and Computer Science,
Eindhoven University of Technology
Den Dolech 2, NL–5600 MB Eindhoven

Abstract

The size of Unified Modeling Language (UML) models used in practice is very large and ranges up to hundreds and thousands of classes. Querying of these models is used to support their quality assessment by information filtering and aggregating. For both, human cognition and automated analysis, there is a need for fast querying. In this context performance of model queries becomes an important issue. We investigated performance characteristics of two different querying engines: one using the Object Constraint Language (OCL) and the other using Prolog. Our comparison is based on equivalent queries in both languages. We applied the queries to 118 models of a size up to 10000 classes to analyze model load and evaluation time. Our preliminary results show that if execution time of queries is linear then Prolog is faster. For one of the presented cases, the execution time in Prolog was nonlinear and thus higher. Further studies should focus on a performance analysis reflecting expressiveness aspects. Our experimental material is accessible to enable future replications of this study.

1 Introduction

The Unified Modelling Language (UML) is commonly accepted as a standard language for object–oriented analysis and design of software systems. This fact has the following consequences. Many tool vendors are interested in providing support for UML, what causes its broader usage also in industrial projects, where *large scale models* are used. The size of industrial models ranges up to hundreds and thousands of classes [9], hence scalability becomes an issue. On the other hand UML provides a *multi–diagram view* enabling description of a software system from different perspectives. The multi–diagram view causes *cross–diagram relationships* [4] which can result in inconsistencies of models. Therefore model querying is important, as it provides information aggregating and filtering enabling easier detection of model flaws. For both, human cognition and automated analysis, there is a need for fast querying. In particular, fast feedback is desired for automated improvement of models using techniques from search–based software engineering (SBSE) [8]. In the case of SBSE, efficient queries are needed to realize the fitness function. In this context performance of model queries becomes an important issue, which impacts the usability of queries in the model validation and verification process.

In general we distinguish two approaches to *UML model querying*. In the first one, UML models can be queried based on their original representation by means of model query languages. In the second one UML models have to be mapped to another representation and queried with an appropriate query language. We investigated model querying with Object Constraint Language (OCL, [11, 14]) and Prolog [12], as representatives of the two approaches.

Model Querying with OCL We selected OCL as representative of model querying languages because of its expressiveness and its growing popularity. Despite the fact that OCL was originally designed for expressing constraints about a UML model, its ability to navigate the model and form collections of objects, caused its usage as a query language [1]. Usage of OCL has begun to exceed its original purpose even further and OCL can be used as a basis for a number of different languages [2]. For example, OCL is a base for the Query/Views/Transformations (QVT[1]) standard. In our recent project we have successfully been using OCL 2.0 in model assessment [6]. However, OCL is under continuous improvement to support new aspects, the ex-

*The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsfrderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

[1] QVT specification at http://www.omg.org/

pressiveness of OCL as a query language is no more a critical question. The OCL becomes widely supported by by commercial and non–commercial modeling tools [3]. Thus a new question arises, the question about performance characteristics of OCL. We are especially interested in performance of OCL in the context of model querying. To the best of our knowledge, there are no studies investigating this aspect. The lack of performance studies is caused by a few reasons. The OCL specifications are changing from one version to another and the changes have to be reflected in OCL interpreters. There are no dominant interpreters for OCL and no benchmarks to compare the performance of tools. There are first signals that the performance of OCL becomes an interesting topic, as the first studies on optimisation of OCL, e.g. [10], appear.

Model Querying with Prolog Prolog is a declarative programming language based on the logic programming paradigm. Due to its rule based approach, it has successfully been applied to implement query engines for query languages based on various data models, e.g. for querying RDF data with SPARQL[2]. Störrle has shown that Prolog can also be used to query large scale industrial software models [13]. Prolog provides meta–model independent facilities to implement query engines for many different practically relevant modeling languages such as UML or Event–driven process chains (EPC). Model elements in these languages can be transformed into facts in Prolog and can then be checked with Prolog rules and queried with Prolog queries. Therefore, Prolog allows for the implementation of a generic and powerful querying approach for software models. Since the 1980's there have been many improvements on the performance of query execution in Prolog ([7],[5]) which lead to the implementation of performant Prolog interpreters, such as SWI–Prolog[3] which has been used in the experiment presented in this paper.

Paper structure The rest of this paper is organised as follows. In the next sections we present our experiment at conceptual (section 2) and technical level (section 3). Next we present results of the experiment (section 4) and we conclude our work (section 5). Additionally, in the appendix we give source code of all queries used in the experiment.

2 Experiment description

In this section we describe the design of the conducted experiment, in particular its goal, the type of the experiment, and the dependent and independent variables.

The goal of the experiment was **performance analysis of model querying** with two different query evaluation frameworks. Beside the performance comparison we would like to observe aspects typical for individual query languages.

We conducted our research as a **laboratory experiment** [16]. In the experiment we observed one *independent variable*, namely the evaluation time and how it reacted on changes of three *dependent variables*, namely model, query and evaluation frameworks.

Evaluation time of queries is measured in terms of run time in milliseconds (without I/O operations).

A set of models of a similar structure and different sizes was generated with a developed model generator (section 3.1). We decided to use generated models rather than models from real projects, as in the second case it would be difficult to control model characteristics (structure, size).

A set of queries was selected and implemented by the authors. In our choice we decided to compare queries of the same semantics instead of comparing queries of similar structure. We ran the experiment for eight queries. These queries are relevant for model analysis and are of different complexity and structure. However, due to space limitations we have to limit the presentation in this paper to three queries. The selected queries are representative in terms of type and results for the entire set of our queries. The simplest queries return a single value by element counting, more complex ones return collections of elements according to given selection criteria. The most complicated queries require recursive calls of methods or predicates.

Evaluation frameworks were selected to represent query languages originating from different paradigms. We selected OCL as "a formal language used to describe expressions on UML model" [11] and Prolog as a general purpose logic programming language, for which UML models have to be converted to an internal representation (section 3.3). Our selection criterion was usage of interpreters in academic applications. Therefore we have decided to use SWI–Prolog (section 3.3). Based on our observation made during MoDELS'2006 and 2007 that the number of research projects using or developing Eclipse Modeling Framework Technologies is increasing, we predict that the OCL interpreter provided by Eclipse (section 3.2) will become popular in academic projects soon.

At this point we have to stress that language performance strictly depends on quality of its compiler or interpreter. Thus the results are typical for these particular interpreters, as a consequence they cannot be generalized for the languages. However, the selection of other interpreters would not change the order of magnitude of time required for evaluation, the results can differ in details.

[2] `http://www.swi-prolog.org/packages/SeRQL/` implements SPARQL support for SWI–Prolog

[3] `http://www.swi-prolog.org/`

3 Experiment environment

In this section we describe all software and hardware settings of our experiments. Simple UML models, containing class and object diagrams, have been generated with a model generator developed by the authors (section 3.1). The model queries were expressed in OCL and Prolog and evaluated by the OCL interpreter (section 3.2) and the Prolog interpreter (section 3.3), respectively. All experiments were performed on the same computer (section 3.4).

3.1 Model Generator

For the representation of the models we used the XMI standard version 2.1[4] and the UML schema version 2.0.0[5] provided by Eclipse. Technically the UML API[6] has been used to create the model in the heap and save them later as XMI representation in UML files.

For the experiment we created models with following elements: PrimitiveTypes, Classes, Generalizations, Attributes, Associations, Instances, Slots and Links. The model generator enables generation of parametrised and randomised models. The parameters for model generation are described in Table 1.

Table 1. Configuration parameters for the model generator

abr.	name	description
NoC	Number of Classes	The main input factor. It determines the size of the model.
AC	Abstract Class Factor	Determines abstractness rate, when expressed as percentage it equals 100%/AC.
Gen	Generalization Factor	Expresses the probability (1/Gen) for each class to have a generalization.
NoAt	Number of Attributes Factor	Determines the range (0..NoAt-1) for a number of generated attributes, it is randomly decided how many attributes will be created.
IpC	Instances per Class Factor	Determines the range (0..IpC-1) for a number of generated instances per class.
NoAs	Number of Association Factor	Determines the range (0..NoAs-1) for a number of generated associations per class.
NoS	Number of Slots Factor	Determines the percentage of slots (NoSF%) which are created for one attribute.
NoL	Number of Links Factor	Determines the percentage (NoL%) of links which are created for one association.

In the generation process, the model and all types of model elements are created by dedicated methods[7]. For the experiment we used the configuration presented in Table 2 to generate 118 models up to 10000 classes.

Table 2. Experiment configuration

abr.	used values	interpreted in a model
NoC	1–10000	1–10000
AC	20	5% of NoC
Gen	20	5% of NoC
NoAt	10	0–9 attributes per class
IpC	10	0–9 instances per class
NoAs	5	0–4 associations per class
NoS	80	80% of attributes per class
NoL	80	80% of assocations per class

3.2 OCL Evaluation Suite

In our study we used an OCL interpreter[8] from a project within the Eclipse Modeling Framework Technology (EMFT[9]). We developed an OCL evaluation suite as a front end for the interpreter. The major requirements for the evaluation suite were reproducibility of results and high level of automation (batch mode, results logging). The second requirement was crucial as multiple query evaluations were a time consuming task. To satisfy these requirements we developed a command line program enabling evaluation of expressions against a UML model. The OCLTestSuiteImpl is configurable with four input parameters and two input files[7].

After execution the OCLTestSuiteImpl prepares the OCL environment (EMFT), registers the UML meta model and registers all defined OCL methods from the defines.ocl. After that the model with the name autogenerated-model-⟨NoC⟩-⟨MSN⟩.uml will be loaded, where a number of classes (⟨NoC⟩) and a model serial number (⟨MSN⟩) are given as parameters. Each expression in the expression.ocl file will be evaluated against the model. For each execution it is possible to define the number of evaluations against the model (size of a cycle). In a case of multiple evaluations, the average execution time is regarded. Each execution is logged into the log file[7], and the result to a result file specifically for that execution. The repetitions parameter determines the number of evaluation executions.

3.3 Prolog Evaluation Suite

We have implemented an evaluation suite in SWI–Prolog[10] which has the same input and output interfaces

[4] http://schema.omg.org/spec/XMI/2.1/
[5] http://www.eclipse.org/uml2/2.0.0/UML/
[6] org.eclipse.uml2.uml
[7] additional description can be found at http://squam.info/doc/papers/Opoka2008UMLModelQuering_appendix.pdf and the generator can be downloaded from http://squam.info/source/performance/
[8] http://www.eclipse.org/emft/projects/ocl/
[9] http://www.eclipse.org/emft/
[10] http://squam.info/source/performance/

as the OCL evaluation suite mentioned above. We just use ISO–compliant functionality of SWI–Prolog so that our queries can be executed on any other ISO–compliant Prolog interpreter. First we import the test model from XMI format into Prolog facts stored in Prologs in–memory database. Therein each fact is represented in the following manner: `model:me(metaclass#id,[tag-value,...])`. We then use `findall` queries which allow for formulating efficient queries on the Prolog database (see section 4 and Appendix A for examples of such queries) to compute desired results. Each test model is loaded, its internal representation is exported, then it gets asserted and all queries are executed on it one by one in a failure–driven loop which enforces backtracking and therefore frees memory for each query. Finally for every test model the query results and the runtime parameters are logged.

3.4 Hardware and Software Settings

To make results comparable the applications have been executed always on the same machine with the same hardware and software configuration (Table 3) and with a minimal load of the operating system.

Table 3. Hardware and Software Parameters

Parameter	Value
CPU(s)	1x3.20GHz
RAM	1015MB DDR–SDRAM
Operating system	Microsoft Windows XP (2002) Professional 5.01.2600 (Service Pack 2)
Java Virtual Machine	1.6.0
OCL interpreter	Version 1.1.0.v200703301439
Prolog interpreter	SWI-Prolog 5.6.34

3.5 Remarks

In Table 4 we compare main characteristics of the evaluation suites described in sections 3.2 and 3.3. Both frameworks use the same input format for models and convert models into internal representations. Note that for Prolog the representations of models and queries are uniform.

Table 4. Some characteristics of the suites

	OCL	Prolog
model external formats	XMI/UML	
model internal format	EMF	Prolog
query language	OCL	Prolog
platform	Eclipse, Java	SWI–Prolog

4 Experiment results

In this section we present results for three queries (sections 4.1–4.3) and additionally for model load (section 4.4). Each query is at first presented in an informal way where we describe when it can be used. Next we present its implementations in OCL and Prolog and the corresponding listings are given in Appendix A. After the presentation of a query we analyse its performance characteristics.

4.1 Q1: Overall number of model elements

Query Q1 is an example query determining model size. Such a query can be used in project management, e.g. for estimation of effort of the implementation. However, the query could be formulated in different ways: in case other elements have to be counted, the general structure of the query would remain the same.

For our models we decided to count all elements, i.e. the following types are taken into consideration: Class, Association, Instance Specification (object and link), Instance Value, Literal, Primitive Type, Property (association and attribute), Slot, and Model.

This query is interesting because of two reasons: the first is related to the characteristics of the model generation algorithm, and the second to our performance study. The relation between number of classes and number of model elements describes a characteristic of created models and the developed generator. The query is interesting from the performance point of view, as it requires an iteration over all models elements. The main reason for the difference in the evaluation time is the way of model representation in both frameworks.

Number of classes to number of elements relation The number of classes (NoC) is a parameter of the model generation algorithm and, for a given configuration (Tables 1, 2), the number of model elements depends on NoC. In the configuration of the experiment (Table 2) the relation between number of classes was linear up to the randomisation factor (Fig. 1). A change in generator parameters would change a proportion factor, but the dependency will remain linear, unless the generation algorithm is changed.

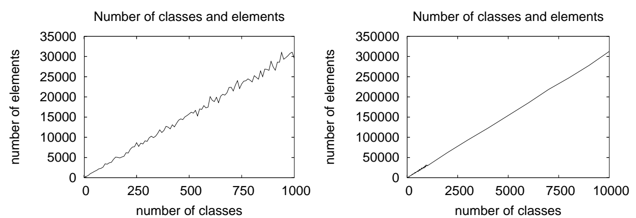

Figure 1. The relationship between number of classes and elements (as evaluated in Q1)

For the following diagrams we decided to use the number of elements as a model size factor instead of the number of classes. However, a character of dependencies in the following diagrams would be the same for both metrics, the first reflects an actual size of models.

Q1 in OCL For Q1 we defined OCL method modelSize() in the Package context (Listing 1 in Appendix A). In this method sets with elements of desired types are created and their magnitudes are added. To collect particular types of elements we used allInstances() method and ownedElement attributes.

We observed that the method allInstances() requires relative long evaluation time. The query could be optimised by definition of collecting methods for each type of elements. Optimisation could lower the evaluation time but the query would be less general and an additional programming effort would be necessary to define the collecting methods (see Listing 3 in Appendix A for example method for classes collection: getAllClasses(), the method is specific for the generated model structure).

Q1 in Prolog The first query can efficiently be executed by one findall predicate to generate a list of all model elements and by then determining the length of that list calling the length predicate (see Listing 2 in Appendix A).

The execution time of Q1 For both query languages the evaluation time of query Q1 was linear (Fig. 2). The evaluation of Prolog[11] was significantly faster. The evaluation in OCL was slower because of separate and repeated iterations over different types of elements. In Prolog it was possible to collect all types of elements with a single predicate. This difference in evaluation is caused by the difference in the internal representation of models. In EMF models are stored as a structured collection and in Prolog as a list (as described in section 3.3). For this type of query the representation as list is more effective.

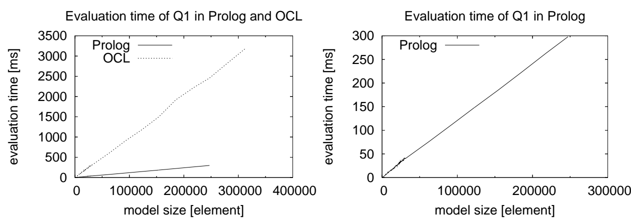

Figure 2. The execution time of Q1

Query Q1 was in general the slowest query in OCL and this is because of multiple usages of the allInstances() method. We made an additional experiment with the method called sequentially and nested. Results of this experiment confirmed that additional sequential calls increase the slope of the function but its linear character is preserved. Thus the sequential calls can be considered as harmless. In opposite, nested calls should be used very carefully, as they results in exponential evaluation time. In our opinion, in practice the nested calls are not necessary to use.

[11] for Prolog not all models could be evaluated, see section 4.4 for explanation

4.2 Q2: List of all classes with a given number of associations

Query Q2 represents a more complex query which, if slightly changed, can be used to check if modeling guidelines are adhered to. With such a query one could find all classes which have too many or too few associations. The query can be extended with additional restrictions, like information about packages or stereotypes, to support more complex guidelines.

Q2 in OCL For query Q2 we defined three methods which were split only for the sake of readability (Listing 3 in Appendix A). The first method defined in context Class, namely getAssociationsFromClass(), is an auxiliary one and enables collection of outgoing associations from a set of properties owned by a given class. The other methods are defined in context Package. The second one for collecting all classes, named getAllClasses(), is an example of dedicated collecting methods for elements of particular type (as mentioned in section 4.1). This method is specific for the structure of generated models and does not reflect full range of classes occurrences, such as e.g. nesting of classes. And the third method namely getClassesWithNumberOfAccosciations(...), collects at first all classes and selects a subset of classes with a given number of outgoing associations.

Q2 in Prolog Q2 is the most time consuming query in Prolog. This is mainly caused by calculating the list of all classes which occur exactly two times in the multi–set of all classes that have associations. In more detail, the query works as follows. First, the identifiers of all classes that have associations are queried via the association model elements holding those identifiers. Then the predicate get_dup determines all classes in that list which occur exactly two times. Finally, the names of theses classes are determined with a findall predicate (see Listing 4 in Appendix A).

The execution time of Q2 For query Q2 evaluation time in OCL was significantly shorter than in Prolog (Fig. 3). Moreover, the evaluation time in OCL is linear and in Prolog it is quadratic.

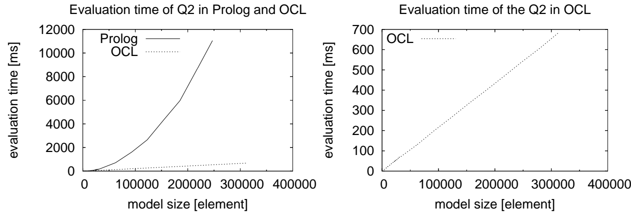

Figure 3. The execution time of Q2

In general query Q2 was the slowest one in Prolog. The evaluation time grows quadratic because the complexity of its bottleneck, the computation of those classes which occur exactly two times in a class list is quadratic.

4.3 Q3: Maximal depth of the inheritance tree

Query Q3 is a non–trivial query as for its evaluation the whole inheritance tree must be examined. Similar to query Q2 it can be used to control if modeling guidelines are fulfilled. The depth of the inheritance tree is used as a typical metric in object oriented modeling or programming.

Q3 in OCL Query Q3 was split into three methods (Listing 5 in Appendix A). The first one had to be separated because of its recursive calls, and the second and the third were split only for the sake of readability.

The first method defined in context Class, namely DITantiCycle, enables navigation from a given class up to the top of the inheritance tree and counting of levels. The information on visited classes is saved, thus when the same class is met the method stops. It is necessary to avoid looping in case of cyclic inheritance relations.

The second method defined in context Package, namely getDepthsOfTheInheritanceTree(), collects results from the first methods for all classes.

The last method also defined in context Package, namely getMaximalDepthOfTheInheritanceTree(), selects the maximum among collected inheritance tree depthdepth values.

Q3 in Prolog Q3 is the query which has the most complex formulation in Prolog. First, the query finds all pairs of classes and its superclasses and then splits them in superclasses and classes list via superclasses and classes predicates. Then the set of all root elements of the inheritance trees are determined via the init predicate. Then all inheritance lists (up to its first cycle) are determined with the gen predicate, finally the maximal length of those lists is computed via the max_length predicate (see Listing 6 in Appendix A).

The execution time of Q3 For query Q3 an evaluation of the query expressed in Prolog takes shorter, although non–linear time (Fig. 4).

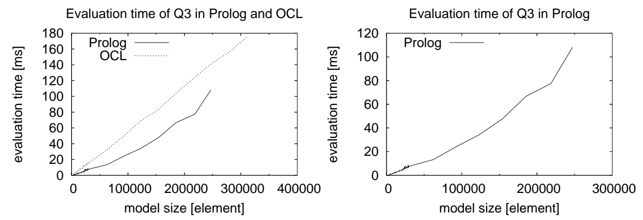

Figure 4. The execution time of Q3

The query in OCL is relatively fast despite the fact of recursive calls of the method DITantiCycle(...). This is because of two reasons. First, the method is not complex and fast. And secondly, the number of recursive calls is limited by the depth of the inheritance tree and in fact quite low.

4.4 Model load

Beside the time required for query evaluation one should also analyse time required to load a model from a file to a memory. Depending on a modus of query evaluation, this preceding action can be more or less crucial. If the model analysis take place in a modeling tool and models are already in an appropriate form (i.e. EMF for OCL queries) then the load time is not relevant. If a model needs to be loaded only once at the beginning and after that all queries are evaluated then the load time is not critical. Otherwise, the load time has to be included in the evaluation time of queries.

Figure 5 illustrates loading time in both frameworks. However, order of time is similar for both interpreters (3 seconds for the biggest model loaded in the Prolog frameworks), there are some interesting observations related to individual frameworks.

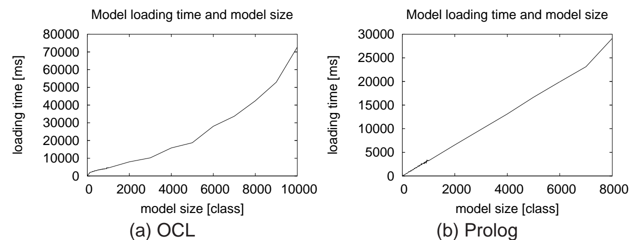

Figure 5. Model load time depending on number of classes

Observations specific for OCL An important observation is that the model load from the EMF representation in

a file to its representation in memory is non–linear. One could expect that it should be proportional to the number of elements. We assume that the deviation from linear dependency is a result of additional time required to create relations between elements. To create relations proper elements must be selected from model elements and this operation takes more time for bigger models. Moreover, due to their big size, models have to be partially stored in swap memory and this fact slows down the load process.

Observations specific for Prolog The local stack, used to store the execution environments of procedure invocations, the global stack, used to store terms created during Prolog's execution and the trail stack, used to store assignments during execution, are limited to 128 Mbytes on 32 bit processors [15]. On 64 bit machines these stacks are practically unrestricted[12]. Therefore the modelsize in our internal representation mentioned above which can be handeled is restricted to roughly 32 MBytes. In our test configuration this corresponds to models with roughly 8000 classes (250.000 model elements). This does not limit the practical usage of Prolog because according to our experiences it is always possible to restrict model size by a projection on its relevant parts for specific queries. We will further implement projection operations in future implementations of our Prolog framework.

5 Conclusion

Fast querying of models is important for comprehension, analysis and improvement of models. Fast feedback is desired for modeling activities performed by humans, but in particular for automated improvement of models using techniques such as Search–Based Software Engineering.

To study the performance characteristics of two different querying techniques, OCL and Prolog, we conducted a laboratory experiment. This paper reports on the experiment. We have developed an experiment environment to generate UML models, evaluate queries, collect, parse and analyse the results. Below we discuss limitations and the results of this preliminary study and point to directions for future experiments.

Discussion There are two important issues: the choice of queries was subjective and the level of query optimisation was determined by authors' command of the query languages. The evaluation time of queries can be shortened by improvements in their implementation or by their optimisation through an interpreter. We tried to implement queries to achieve good performance and we assumed that this level of optimisation is realistic.

[12] Indeed they are restricted to $2^{59} bytes$

Results The main observation is that the list representation of models, as used for Prolog, is more effective for queries collecting or selecting elements based on their direct properties (like Q1). On the other hand model representation reflecting original structure of models together with navigation abilities of a query language, as in case of OCL, enables faster evaluation of queries based on properties of relationships between elements (like Q2). More detailed results are presented in sections dedicated to evaluation of particular queries and model load time. Our experimental material is accessible on–line to enable future replication and extension of this study.

Future work Future studies on performance and expressiveness of both languages could provide application guidelines for OCL and Prolog queries in model analysis. Such guidelines could help tool vendors to decide which representation and evaluation of models and queries is better suited for types of analysis to be implemented.

Acknowledgement

We would like to thank Harald Störrle for his contribution to the optimisation of Prolog expressions.

References

[1] D. H. Akehurst and B. Bordbar. On Querying UML data models with OCL. In *UML 2001: Modeling Languages, Concepts and Tools*, pages 91–103, October 2001.

[2] D. H. Akehurst et al. Supporting OCL as part of a Family of Languages. In T. Baar, editor, *Proc. of the MoDELS'05 Conf. Workshop on Tool Support for OCL and Rel. Formalism*, LGL-REPORT-2005-001, pages 30–37. EPFL, 2005.

[3] T. Baar et al. Tool support for OCL and rel. formalisms - needs and trends. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 1–9. Springer, 2005.

[4] P. A. Bernstein et al. A vision for management of complex models. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(4):55–63, 2000.

[5] J.-H. Chang and A. M. Despain. Semi-intelligent backtracking of prolog based on static data dependency analysis. In *SLP*, pages 10–21, 1985.

[6] J. Chimiak-Opoka and C. Lenz. Use of OCL in a model assessment framework: An experience report. *Electronic Communications of the EASST*, 5, 2006.

[7] C. Escalante. A simple model of prolog's performance: extensional predicates. In *Proc. of CASCON'93*, pages 1119–1132. IBM Press, 1993.

[8] M. Harman. The current state and future of search based software engineering. In *FOSE'07*, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.

[9] C. F. J. Lange. *Assessing and Improving the Quality of Modeling – A Series of Empirical Studies about the UML.* Phd thesis, Eindhoven Univ. of Tech., The Netherlands, 2007.

[10] L. Lengyel et al. A visual control flow language for model transformation systems. In *IASTED Conf. on Software Engineering*, pages 194–199. IASTED/ACTA Press, 2006.

[11] OMG. Object Constraint Language. OMG Available Specification. Version 2.0, 2006.

[12] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.

[13] H. Störrle. A PROLOG-based Approach to Representing and Querying UML Models. In P. Cox et al., editors, *Proc. of the VL/HCC'07 Workshop*. IEEE, 2007.

[14] J. Warmer and A. G. Kleppe. *The Object Constraint Language—Precise Modeling with UML.* first edition, 1999.

[15] J. Wielemaker. *SWI-Prolog 5.6 Reference Manual*, 2007.

[16] C. Wohlin et al. *Experimentation in Software Engineering—An Introduction*. Kluwer Academic Publishers, 2000.

Appendix A Source code of model queries

Below source code for all queries expressed in OCL and Prolog is provided.

Listing 1 (Q1 expressed in OCL)

```
package uml context Package
def: modelSize() : Integer =
    Class.allInstances() -> size()
    + Slot.allInstances() -> size()
    + Association.allInstances() -> size()
    + InstanceSpecification.allInstances() -> size()
    + PrimitiveType.allInstances() -> size()
    + Class.allInstances().ownedElement -> size()
    + Slot.allInstances().ownedElement -> size()
    + 1  -- Model itself
endpackage
```

Listing 2 (Q1 expressed in Prolog)

```
size(Num) :-
  findall(Id,me(_,_),IdList),
  length(IdList,Num).
```

Listing 3 (Q2 expressed in OCL)

```
package uml context Class
def: getAssociationsFromClass() : Set(Association) =
   self.ownedElement
   -> asSet()
   -> select(a | a.oclIsTypeOf(Property))
      .oclAsType(Property)
      .association
   -> select(a | not (a=null))
   -> asSet()
endpackage

package uml context Package
def: getAllClasses() : Set(Class) =
   self.packagedElement
     -> asSet()
     -> select ( t |t.oclIsKindOf(Class))
        .oclAsType(Class)
     -> asSet()

def: getClassesWithNumberOfAssociations(NoA:Integer)
          : Set(Class) =
  self.getAllClasses()
   -> asSet()
   -> select( c | c.getAssociationsFromClass()
   -> size() = NoA)
endpackage
```

Listing 4 (Q2 expressed in Prolog)

```
classes(CNList) :-
  findall( ClsId, get_me(end-ClsId, assoc#_,_), ACList),
  get_dup(ACList, ClsIdList),
  findall( CN,( get_me(name-CN, class#Id, _Atts),
                memberchk(Id, ClsIdList) ), CNList).

get_dup(List, Result):-
   msort(List, SortedList),
   list_to_set(SortedList, Set),
   sublist(dup(SortedList), Set, Result).
```

Listing 5 (Q3 expressed in OCL)

```
package uml context Class
def: DITantiCycle(list:Set(Class)) : Integer =
if self.hasGeneralization() then
  if list->includes(self) then 0
  else 1 +
   (self.generalization.general.oclAsType(Class).
     DITantiCycle(list->union(Set{self}))
     -> asSet()
     -> asOrderedSet()
     -> at(1))
  endif
else 0
endif
endpackage

package uml context Package
def: getDepthsOfTheInheritanceTree() : Set(Integer) =
  self.getAllClasses()
    -> collect(c : Class | c.DITantiCycle(Set{}))
    -> asSet()

def: getMaximalDepthOfTheInheritanceTree() : Integer =
   self.getDepthsOfTheInheritanceTree()
     -> iterate(elem:Integer; result:Integer = 0
              | result.max(elem))
endpackage
```

Listing 6 (Q3 expressed in Prolog)

```
depth(Num) :-
  findall(GId-CId, get_me(ge-GId, class#CId, _), CSList),
  superclasses(CSList, SList),
  classes(CSList, CList),
  init(SList, CList, [], initList),
  gen(initList, CSList, [], Chains),
  max_length(Chains, 0, Num).

init([], _, CList, CList).

init([X|GList], CList, CList, IList) :-
 member(X, CList) -> init(GList, CList, CList, IList);
 init(GList, CList, [X|CList], IList).

gen([], _, Chains, Chains).

gen([X|List], PairList, CollectChains, Chains) :-
  transitiveList(X,X, PairList, [], Chain),
  gen(List, PairList, [Chain|CollectChains], Chains).
```