A PROLOG-based Approach to Representing and Querying Software Engineering Models.

Conference Paper · January 2007

Source: DBLP

CITATIONS
19

READS
321

1 author:

Harald Störrle
QAware
103 PUBLICATIONS **1,336** CITATIONS

SEE PROFILE

A PROLOG-based Approach to Representing and Querying Software Engineering Models

Harald Störrle *
mgm technology partners GmbH
Munich, Germany

Abstract

Striving toward the vision of Model Driven development (MDD), we face many open questions connected to the elementary tasks involved in working with models. Probably the most basic task is querying models for properties, elements, and submodels. Current tools and interfaces for model querying are either restricted in their expressiveness or require a high level of expertise in the underlying metamodels and/or query languages. As the application of MDD is gaining more widespread acceptance and more and more developers are involved with MDD efforts, this state is becoming a bottleneck. In this paper, we propose a Prolog-based model representation and query interface for models to overcome this bottleneck.

Keywords: Model Driven Development (MDD), Query-View- Transformation (QVT), Knowledge Based Software Engineering (KBSE), Industrial Applications

1 Introduction

The MoMaT approach has been developed over the last years, partially in an academic setting, and partially in two very large scale industrial projects with German federal and state agencies. In these projects, very large models have been created and demand for advanced model operations soon became pressing. We have thus turned to MDD/MDA technology.

Model Driven Development/Architecture (MDD/MDA, [6, 17]) has been proposed as a measure to raise the level of abstraction in software development, and thus to increase developer productivity. In a MDA setting, models are programs, thus modeling languages are programming languages (cf. [20]). Today, there are many different practically relevant modeling languages, most of which are predominantly visual modeling languages. Examples are the Unified Modeling Language (UML, [16]), Entity-Relationship-Diagrams (ERD, [4]), Event Process Chains (EPCs, [5]), Integration Definition for Function Modeling (IDEF, [13]), or Use Case Maps (UCM, [3]). In principle, such models may be used for a multitude of purposes, such as reporting, model transformations, model consistency checking, formal analysis, code generation, pattern detection, versioning, size measurement and so on. See Figure 1 for a synopsis of model operations.

However, one of the basic tasks in an MDD setting is querying models for properties, elements, and submodels. This task is executed, on the one hand, by developers

*Harald.Stoerrle@mgm-tp.com

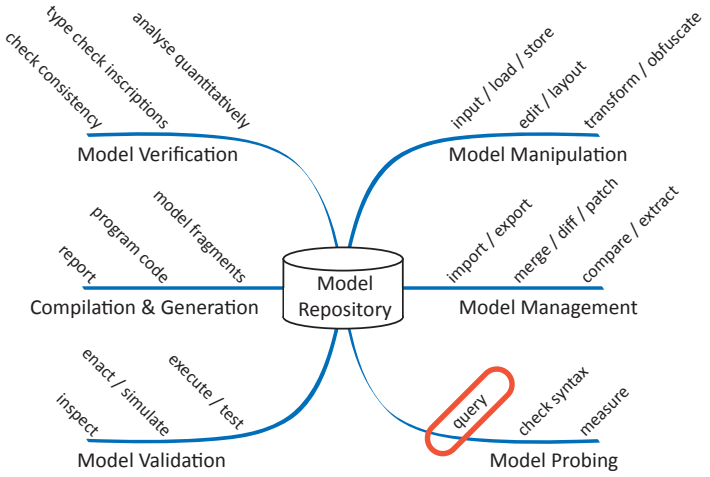


Figure 1: Classes of model operations (potentially) relevant to industrial modeling

working on the models. It is, on the other hand, also a basic building block underlying most other more complex functionalities like model transformations and model measurements. However, current tools and interfaces for model querying are either restricted in their expressiveness or require a high level of expertise in the underlying metamodels and/or query languages, both of which reduce their versatility. As the application of MDD is gaining more widespread acceptance and more and more developers are involved with MDD efforts, accessing models in an effective way is becoming a bottleneck.

Over the last years, we have created a system called the Model Manipulation Toolkit (MoMaT) which allows us to experiment with models in general. In MoMaT, models are imported into a Prolog-based model repository by a set of transformers for several modeling languages like UML, ARIS/EPC, and Use Case Maps (in this paper, we will only focus on UML models, though). Figure 2 provides an overview of MoMaT. In other papers, we have reported on using MoMaT for model version management operations (see [23, 24]).

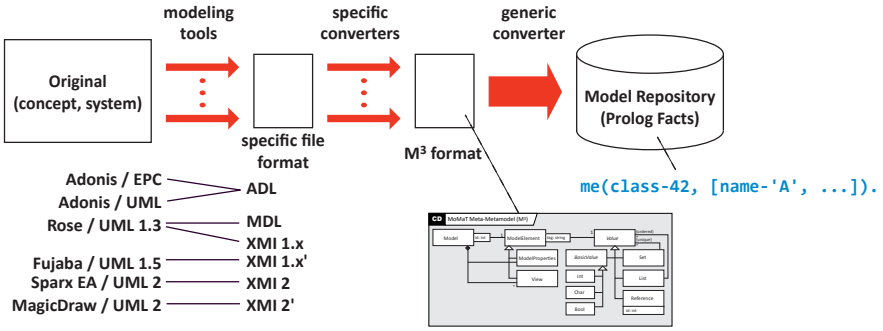


Figure 2: Schematic overview of MoMaT.

Our approach is derived from our experiences in two very large scale industrial projects with the German federal Tax Authority and the German Public Pension Authority. In these projects, very large models have been created and it soon turned out that without adequate query facilities they turn into "black holes" swallowing information but not giving it away again.

1.1 Related work

The related work may be subdivided into textual and visual query languages, and approaches that are specific to certain languages and/or tools or generally applicable (see the following schema).

	textual query language	visual query language
schema specific	query interfaces/ APIs, OCL [15]	QVT [19] Implementations like ATL, UMT [14], MOLA [9]), Porres' toolkit [18]
general	SQL, QBE, XPATH, SHORE [12]	Visual OCL [2], Constraint Diagrams [10], Query Models [22]

Most CASE tools provide APIs or predefined queries to allow users access to the models. Valuable as such facilities may be for the working software engineer, they are restricted to the specific settings in which they are implemented; application to other languages, tools, or data structures is difficult if not impossible. The OCL [15] is somewhat more general in that, theoretically, it should fit to any UML model/tool combination. In practical settings, this is not true, however. Also, OCL is extremely difficult to read and write[1] and there is very scarce tool support.

The OMG's Query-View-Transformations standard (QVT, [19]) has been created with similar goals as MoMaT. There are several implementations of QVT like the Atlas Transformation Language (ATL, `http://www.eclipse.org/m2m/atl/`), the UML Model Transformation Tool (UMT, see `http://umt-qvt.sourceforge.net/` and [14]), and the Model Transformation Language and tool MOLA, see [9]). All QVT implementations are based on the UML metamodel as their underlying data structure, which effectively excludes their application to non-UML languages. It also ties the respective tools to a particular version of the UML. The framework proposed in [18]) is a non-QVT system based on Tcl which the authors themselves deem applicable only for small and medium sized systems. None of these approaches are implemented in and using the facilities of Prolog. Also, to the best of our knowledge, none of them has been used successfully over a longer term in industrial applications.

When models are stored in a relational database, traditional relational query languages like SQL or QBE may be use to access them.[2] For XML data structures or databases, XPATH and similar approaches provide APIs with query facilities. The SHORE system (see [12]) is an approach to storing software design documents in a XML database using Prolog as the query language.

Visual OCL [2], Constraint Diagrams [10], and Query Models [22] each use a modeling language to specify queries for this language. By analogy, these approaches may thus used for other modeling languages. It is not clear, however, how such queries might be executed, much less, if used for a different language. While this paper does not yet fill this gap, it outlines a path toward this goal.

Besides this comparison scheme, Gruhn and Laue [7] present an approach where Prolog is used to access and query software engineering models, but also to represent them: they encode EPCs (in a rather ad hoc way) into Prolog facts which they then use

[1] A more detailed comparison between SQL, OCL, and MoMaT has to be deferred until we have introduced MoMaT.

to check some consistency conditions. This approach is limited to EPCs, unfortunately. There are also several approaches to encode programming language code in Prolog, e. g. for the Java language (cf. [21]), then define particular properties as Prolog rules and then check these properties by evaluating respective goals.

2 Model Representation

In MoMaT, models are represented as Prolog facts. More or less, every individual model element is represented as a single fact. The encoding into this representation is done in two steps (see Figure 2).

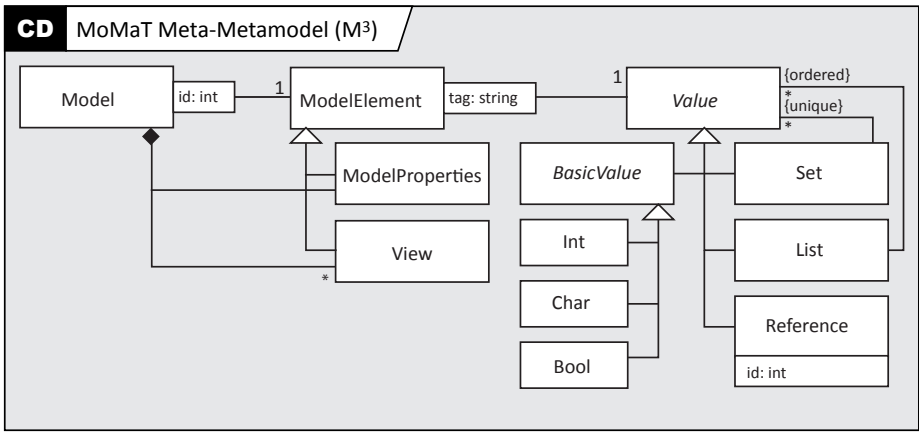


Figure 3: A Meta-metamodel as a normal form for arbitrary modeling languages.

First, specific formats are converted into a common format in order to accommodate different tools, languages, and formats. For instance, many modeling tools are capable of creating models in different languages, or of exporting models in different file format versions. Also, most tools will interpret standards in significantly different ways, will contain different specific bugs and so on, such that providing specific converters is inevitable for different tools. The common format is described in terms of a minimal, unifying meta-metamodel called the MoMaT meta-metamodel, or M^3 for short (see Figure 3). It can be seen as a least common denominator for a wide range of modeling languages.

For UML, the mapping into M^3 is simple indeed, since UML comes with a meta-model. All UML metaclasses are mapped to ModelElement. All meta-attributes and meta-associations a are mapped to tags, their types are mapped to the corresponding subclasses of Value (Reference for object types). The top level package of a UML model is mapped to Model.

The second step now simply interprets models in M^3 format as Prolog facts (see Figure 4). In MoMaT, each model element—that is, each instance of the class ModelElement of M^3—is represented as a Prolog clause of the form

```
me(type-id, [tag-value, ...]).
```

where `type` is the metaclass in the source language (such as “Feature” or “Class” in the

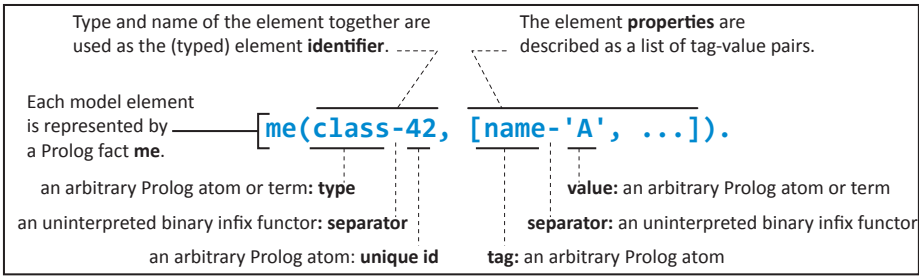


Figure 4: Representing model elements in Prolog

UML metamodel), `id` is an arbitrary unique identifier, `tag` is any atom representing a property of the model element, and `value` is the value for this tag. For instance, an abstract class with object identifier 42 and name "x" would be represented as

```
me(class-42, [name-x, isAbstract-true]).
```

This encoding is visualised in Figure 4. The value of an attribute may be an arbitrary Prolog-term.

A model is then simply a named container for a set of model elements. We use Prolog's built in module mechanism to represent models. In Prolog, modules may be defined just as well at compile time or at run time. Additional information pertaining to the model as such may be represented by a `model/2` clause. Similarly, views may be defined inside models with `view/2` clauses. The arguments of `view` and `model` are similar to those of `me`.

Figure 5 shows an example. Here, a model `m1` is defined. It is an analysis level model authored by user stoerrle and has reached the quality assurance status approved. It contains ten model elements and one view named `m1`. The view presents all model elements of the model.

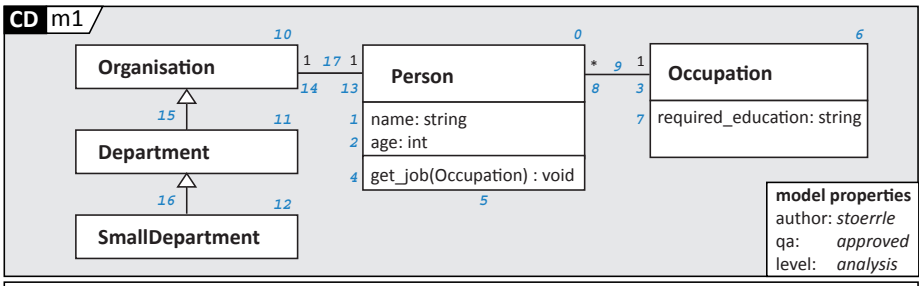
By using the Prolog module mechanism for representing models, all features of modules may be used for models as well, including nesting models, importing and exporting models, dynamic and static (i. e. compile time and run time) definition of models and so on.

3 Model queries

Simple queries select one or more model elements or their attributes based on basic selection criteria like identifier, name, or a complex combination of features. Based on the representation defined above, this may be achieved by Prolog queries like the following.[2]

```
1 ?- m1:me(METACLASS-0, VAL).
  METACLASS = class
  VAL = [name-'Person', attributes-ids([1,2,3]), operations-id(4)]
```

[2]This is actually a transcript of the SWI-Prolog top level slightly edited for readability. It is executed on the model described in Fig. 5. Recall that in Prolog, identifiers starting with an upper case letter are variables. The underscore is the don't-care-variable. Lists are enclosed in square brackets, In SWI-Prolog, `?-` is the top-level prompt. The Prolog idiom `me/3` declares that the predicate `me` is binary (and of course similar for all other attributes and arities).




```
:-module(m1, [me/2, view/2, model/2]).
   model(m1,  [level-analysis, author-stoerrle, qa-approved]).
   view(cd-1, [type-class_diagram, name-'Sample model',language-'UML 2',
               version-'2.1.1',elements-ids([0,1,2,3,4,5,6,7,8,9])]]).
   me(class-0,           [name-'Person',attributes-ids([1,2,3,14]),
                          operations-id(4)]).
   me(feature-1,         [name-name, type-string]).
   me(feature-2,         [name-age, type-int]).
   me(feature-3,         [multiplicity-1, type-id(6)]).
   me(operation-4,       [name-'get_job', parameters-ids([5]), result-void]).
   me(parameter-5,       [type-id(6)]).
   me(class-6,           [name-'Occupation', attributes-ids([7, 8])]).
   me(feature-7,         [name-'required_education', type-string, default-""]).
   me(feature-8,         [type-id(0),multiplicity-'*']).
   me(association-9,     [ends-ids([3, 8])]).
   me(class-10,          [name-'Organisation', attributes-ids([13])]).
   me(class-11,          [name-'Department']).
   me(class-12,          [name-'SmallDepartment']).
   me(feature-13,        [multiplicity-1, type-id(0)]).
   me(feature-14,        [multiplicity-1, type-id(10)]).
   me(generalization-15, [from-id(11), to-id(10)]).
   me(generalization-16, [from-id(12), to-id(11)]).
   me(association-17,    [ends-ids([14, 15])]]).
```

Figure 5: A sample transformation from a model containing a UML class diagram (top) into a Prolog module with clauses for each model element (bottom). The extra numbers in the class diagram refer to the identifiers of the Prolog code.

```
2 ?- findall(ID, m1:me(class-ID,_), IDS).
  IDS = [0,6,10,11,12]

3 ?- findall(ID-VAL, m1:me(class-ID,VAL), RES).
  RES = [0-[name-'Person', attributes-ids([1,2,3]), operations-id(4)],
         6-[name-'Occupation', attributes-ids([7,8])]]
```

The first query identifies the type and property set of element `0` inside model `m0`. The second query identifies all elements of type `class` using Prolog's built-in `findall/3` to obtain all solutions to the second argument with a single call. The third query selects all classes of model `m1`.

For the remaining examples, we need to introduce the predicates `get_me/4`, `part_of/4`, and `get_neighbours/4`, which are typical examples for predicates of the query-API of MoMaT.[3]

```
get_me(Model, Tag-Val, METACLASS-ID, VAL):-  % matches all elements of
  Model:me(METACLASS-ID, VAL),               % model containing the
  memberchk(Tag-Val,VAL).                    % Tag-Val pair
```

Using these predicates, the fourth query identifies the class named `Occupation` in `m1`. The fifth query identifies all features of type `string` in `m1`.

```
4 ?- get_me(m1, name-'Occupation', METACLASS-ID, _).
  METACLASS = class
  ID = 6

5 ?- findall(ID, get_me(m1,type-string,feature-ID,VAL), RES).
  RES = [1,7] ;
```

Query no. six identifies all elements related directly to element 6 by relationships of type `association`. There are very similar predicates for other relationships like containment, association, or generalization.

```
6 ?- get_neighbours(m1, association, 6, N).
  N = [0]
```

The predicate `get_neighbours/4` computes all neighbours of a given element that are related to this element by a particular kind of relationship as follows.

```
part_of(Model, Kind, SUPER, SUB):-             % gets ids of Kind-parts
  get_me(Model, Kind-ids(Parts), _-SUPER, _),  % of SUPER as SUB
  member(SUB, Parts).

get_neighbours(Model, Kind, Element, Neighbours):-
  get_me(Model, ends-ids(Features), Kind-_, _),
  get_me(Model, attributes-ids(ATTRS), class-Element, _),
  intersection(ATTRS, Features, [_|_]),!,
  maplist(part_of(m1,attributes),Containers,Features),
  select(Element, Containers, Neighbours).
```

Query no. 7 counts the number of dynamic and static features in a model, providing an example of how complex queries may be defined ad hoc on the command line.

```
7 ?- findall(OP, get_me(m1,operations-ids(OP),class-_,_), _OPs),
     findall(AT, get_me(m1,attributes-ids(AT),class-_,_), _ATs),
     count([_OPs, _ATs], Number_of_Features).
  Number_of_Features = 6
```

[3]The remaining MoMaT predicates are defined in the appendix. All other predicates are standard Prolog library predicates.

Of course, the queries presented so far have been rather straightforward. However, using Prolog we may execute also much more complex queries like "Identify all superclasses of a given class (transitively)" (see query 8 below).

```
8 ?- pre_closure(rels(m1,generalization),[12],[],C).
  C = [10, 11]
```

Similar queries may be used to identify all elements (transitively) related to a given element by a certain kind of relationships such as associations (query 9) and dependencies, e. g. in order to determine change impacts. A different query using similar techniques may select the shortest path of associations in a set of of classes, or the shortest path of DataFlowEdges between actions in an activity diagram. For lack of space, we cannot present the latter two queries in detail here.

```
9 ?- pre_closure(get_neighbours(m1, association),[6], [], N).
  N = [0, 10]
```

An interesting class of complex query are queries concerning more than one model. Our first example is the detection of design patterns, which can be implemented simply by a sub-model operation (query 10): all detectable patterns must be described structurally, in exactly the same way as our example is described. In this case, there is no occurrence of the composite pattern. Another frequent query is to determine all references from (elements of) one model to (elements of) another model (query 11), in order to trace change impacts across model boundaries.

```
10 ?- submodel(m1, patterns:composite, Mapping).
   Mapping = []

11 ?- findall(ID, Model:me(ID-_,_), IDS),  references(m1, REFS),
      subtract(REFS, IDS, EXT).
   REFS =[]
   IDS = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]
   EXT = []
```

As our last example, consider query 12, where a sequence of models is scanned for the first model in which a given element is defined. Such a query might be used to track the introduction of errors related to some model element.

```
12 ?- sublist(exists(M_ID, 14),[m0, m1, m2], Exists),
      head(Exists,First).
   Exists = [m1, m2]
   First = m1
```

The predefined predicate `sublist/3` returns in its third parameter the sublist of the second parameter such that all the sublist's elements satisfy the predicate provided as the first parameter. Since the ordering of the original list is maintained, the first element of the resulting list is the first model in which model element `14` appears. `exists/2` is defined as `exists(ME_ID, M_ID):- M_ID:me(_-ME_ID,_)`..

4 SQL and OCL as alternative query languages

One may argue that the Prolog code necessary for implementing the queries proposed above is not very readable, and, in fact, Prolog as a language is too uncommon to

be considered for practical applications. There are mainly three alternatives for query languages: tool-specific predefined queries, APIs, SQL, and OCL.

Many commercial tools provide selections of predefined queries (e. g. Telelogic Tau, BOC Adonis). While sufficient in many situations, this approach is not (easily) extensible. Query APIs, on the other hand, are proprietary and may thus not be used in other tools.

SQL [4] is much more popular and widespread than Prolog. In fact, it is *the* paradigmatic query language. Thus, many tools storing models in (object-) relational database tables provide SQL-like facilities for querying (e. g. Aonix StP, BOC Adonis). Most of the queries we have presented above may be expressed in SQL3 (cf. [11]), but many database products do not implement this standard completely and faithfully. So, the following SQL-query is equivalent to query 8 above, but will not terminate on IBMs DB/2 or Oracle 11.

```
WITH RECURSIVE CLOSURE(ClassId, GeneralClassId ) AS
  ( SELECT
    FROM   CLASSES
    WHERE  ClassId = '1'
    UNION ALL
    SELECT cla.GeneralClassId, clos.ClassId
    FROM   CLOSURE clos, CLASSES cla
    WHERE  clos.GeneralClassId = cla.ClassId )
SELECT DISTINCT ClassId
FROM   CLOSURE;
```

Another obvious alternative model query language is Object Constraint Language (OCL, [15]). However, even simple OCL queries tend to be even more convoluted and less readable than Prolog code. See the following OCL equivalents of queries 2, 7, and 8.

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

7) package uml context Package
def: getAllGeneralizations() : Set(Element) =
   self.getAllClasses().ownedElement.oclAsType(Element)
      ->asSet() -- Property (Association, Attribute), Generalization
endpackage

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

The most compelling advantage Prolog has over OCL is of course the much better tool support available for Prolog, including a range of IDEs, debuggers, visualisation tools, efficient compilers and so on. For OCL, there are just a few tools like [8, 1].

5 Evaluation

5.1 Applicability and usability

While the roots of MoMaT lie in academia, it has been applied successfully in industrial settings (though only by expert modelers in touch with the author). The main benefit of our approach is in its technical simplicity and the high-level declarative style of programming in an interactive environment that supports an explorative mode of work.

We have applied MoMaT in a number of different settings concerning languages, formats, and tools:

- a large UML 2 subset (class, object, activity, assembly, and use case models) using ADONIS with proprietary ADL and XML formats;
- class and use case models using Fujaba and MagicDraw with different XMI formats;
- EPCs using ADONIS with the proprietary ADL format.

Models from all these sources may be processed using MoMaT, and we are very confident that we will have no problems processing any other type or format of models similarly. In fact, our approach seems to be unique in that it is applicable also to visual languages other than UML.

MoMaT is targeted at expert modelers and has been used by such people in industrial contexts successfully. First feedback by said users indicated that OCL would have been too complicated to be used. Of course, such subjective reports by people personally acquainted to the author are not representative. Proper evaluations comparing the usability of MoMaT with competing approaches are an open issue.

5.2 Performance comparison

Although we can not provide a complete evaluation of our approach im comparison to all other approaches mentioned in section 1.1, we have some inital measurements. A competing research group from our department is implementing an Eclipse/EMF-based OCL interpreter. In a model query shootout with them, we agreed on a set of eight simple queries[4], created a range of synthetic class models, and executed the queries on the models in both tools.

Classes	10	100	1,000	10,000
Model Elements	325	3,431	29,250	312,584
XMI file size of model [Mbyte]	0.06	0.61	5.42	56.5

The class models contained between 10 and 10,000 classes, each of which had a number of attributes (ranging randomly from one to nine). Over the whole range of models and queries, MoMaT had a consistent performance advantage of about factor ten. A detailed study into this rather unexpected finding has not yet been conducted. In particular, we have not yet evaluated memory consumption in detail. However, it seems that current OCL tools and Java's XML-libraries generally require significant

[4]The queries were: determine number of model elements, depth of inheritance tree, ratio of abstract classes, set of classes participating in exactly two associations, existence of a class with a specified name, number of instances of a given class, number of associations a given class participaates in, and number of neighbours associated to a given class.

resources. We have not yet compared MoMaT with with other OCL tools on the market but would be surprised to find significantly different results.

6 Conclusions

MoMaT provides a powerfull textual query facility for models expressed in arbitrary languages, provided there is a mapping from the languages conceptual design to the M^3. While MoMaT is textual in nature, it opens up a path to defining visual queries as well: as soon as there are is a facility of transforming a (incomplete) model from any given modeling tool into the MoMaT format, such model fragments may be used to find matching submodels using MoMaT. Thus, models may be used as queries in MoMaT, so if there is a tool to create models, there is also a tool to create queries. Of course, for practical appications we would need an integrated work bench, but that is just an implementation task.

References

[1] Dresden OCL toolkit. `http://dresden-ocl.sourceforge.net`.

[2] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A Visualisation of OCL using Collaborations. In Martin Gogolla and Chris Kobryn, editors, *Proc. 4^{th} Intl. Conf. on the Unified Modeling Language (<<UML>>'01)*, number 2185 in LNCS. Springer Verlag, 2001. available at `tfs.cs.tu-berlin.de/vocl`.

[3] Ray J. A. Buhr. *Practical Visual Techniques in System Design. With Applications to Ada*. Prentice Hall, 1990.

[4] Chris J. Date. *An Introduction to Database Systems*. Addison-Wesley, 6^{th} edition, 1995.

[5] Rob Davis. *Business Process Modelling with ARIS: A Practical Guide*. Springer Verlag, 2001.

[6] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. OMG Press, 2003.

[7] Volker Gruhn and Ralf Laue. Validierung syntaktischer und anderer EPK-Eigenschaften mit PROLOG. In Markus Nüttgens, Frank J. Rump, and Jan Mendling, editors, *Proc. 5. GI Ws. Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (EPK 2006)*, volume 224 of *CEUR Workshop Proceedings*, pages 69–85, Bonn, December 2006. Gesellschaft für Informatik.

[8] Christian Hein, Tom Ritter, and Michael Wagner. Open source Library for OCL (OSLO). `http://oslo-project.berlios.de/`.

[9] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model Transformation Language MOLA. In Uwe Aßmann, editor, *Proc. 2^{nd} Working Conf. Model Driven Architecture: Foundations and Applications (MDAFA 2004)*, pages 12–26, 2004. available at `www.ida.liu.se/~henla/mdafa2004`.

[10] Stuart Kent. Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. In *Proc. Intl. Conf. Object-Oriented Programming Object Oriented Programming, Systems, and Languages 1997 (OOPSLA'97)*, pages 327–341. ACM Press, 1997.

[11] Jim Melton. *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features*. Elsevier Science Inc., New York, NY, USA, 2002.

[12] Michael Meyer and Helge Schulz. SHORE: Ein Werkzeug für übergreifende Vernetzung und Auswertung von Dokumenten. In Franz Ebert, Jürgen und Lehner, editor, *Proc. Ws. Software-Reengineering*. Gesellschaft für Informatik e.V. May 1999.

[13] National Institute of Standards and Technologies (NIST). Integration Definition for Function Modeling. Technical report, Computer Systems Laboratory, National Institute of Standards and Technologies (NIST), 1993. available at `www.omg.org/techprocess/sigs.html`, see also `www.idef.com`.

[14] Jon Oldevik. UML Model Transformation Tool (UMT). Overview and user guide, v0.8. Technical report, SINTEF, 2004.

[15] OMG. UML 2.0 OCL Specification (OMG Final Adopted Specification, ptc/2003-10-14). Technical report, Object Management Group, October 2003. available at `www.omg.org`, downloaded at December 2^{th}, 2004.

[16] OMG. UML 2.1.1 Superstructure Specification (formal/ 2007-02-03). Technical report, Object Management Group, February 2007. available at `www.omg.org`, downloaded at May 25^{th}, 2007.

[17] MDA Guide Version 1.0.1. Technical report, Object Management Group, June 2003. available at `www.omg.org/mda`, document number omg/2003-06-01.

[18] Ivan Porres. A Toolkit for Model Manipulation. *Intl. J. Software and Systems Modeling*, 2(4), 2003.

[19] QVT-Partners. Revised submission for MOF 2.0 Query/ Views/ Transformations RFP (version 1.1, 2003-08-18). Technical report, August 2003. available at `www.omg.org/mda`, download at November 1^{st}, 2004, see also `umt-qvt.sourceforge.net` and `qvtp.org`.

[20] Bran Selic. The pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, September/October 2003.

[21] Daniel Speicher, Robias Rho, and Günther Kniesel. JTransformer – eine logikbasierte Infrastruktur zur Codeanalyse. In Rainer Gimnich, Volker Riediger, and Andreas Winter, editors, *Proc. 9. Ws. Software-Reengineering (WSR 2007)* , volume 27, pages 21–22, May 2007.

[22] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Query Models. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proc.* 7^{th} *Intl. Conf. Unified Modeling Language (<<UML>>'04)*, number 3273 in LNCS, pages 98–112. Springer Verlag, 2004.

[23] Harald Störrle. A approach to cross-language model versioning. In Udo Kelter, editor, *Proc. Ws. Versionierung und Vergleich von UML Modellen (VVUU'07)*. Gesellschaft für Informatik, May 2007. appeared in Softwaretechnik-Trends 2(27)2007.

[24] Harald Störrle. A formal approach to the cross-language version management of models. In Ludwik Kuzniarz, Miroslaw Staron, and Tarja Systa, editors, *Proc. Nordic Ws. Models Driven Engineering (NW-MODE'07)*. IT University of Göteborg, August 2007. to appear.

A Selection of MoMaT library predicates

```
closure(P, X, Closure):-
  pre_closure(P, [X], [], Closure1),!,
  union([X], Closure1, Closure2),
  sort(Closure2,Closure).
pre_closure(_, [], SoFar, SoFar):-!.
pre_closure(P, Args, SoFar, Closure):-!,
  maplist(P, Args, P_of_Args1),
  flatten(P_of_Args1, P_of_Args),
  subtract(P_of_Args, SoFar, New),
  append(New, SoFar, Next),
  pre_closure(P, New, Next, Closure).

external_references(Model, EXTERNALS):-
  findall(ID, Model:me(ID-_,_), IDS),
  references(m1, REFS),
  subtract(REFS, IDS, EXTERNALS).

references(Model, EID, REFS):-
  Model:me(_-EID,VAL),
  maplist(collect_ids, VAL, REFS0),
  flatten(REFS0, REFS1),
  list_to_set(REFS1, REFS).
references(Model, REFS):-
  findall(REF, refs(Model, REF), REF_LIST),
  flatten(REF_LIST,REFS).
refs(Model, REF):-
  Model:me(_-_,VAL),
  maplist(collect_ids, VAL, REF).

collect_ids([],[]):-!.
collect_ids([_-id(ID)|Rest], [ID|RID]):-
  collect_ids(Rest, RID).
collect_ids([_-ids(IDS)|Rest], ALL_IDS):-
  collect_ids(Rest, RID),
  append(IDS, RID, ALL_IDS).

rel(Model, Kind, From, To):-
  get_me(Model, from-id(From), Kind-_, VAL),
  memberchk(to-id(To), VAL).
rels(Model, Kind, From, Targets):-
  findall(To, rel(Model, Kind, From, To), Targets).
```