MQL: A MAPPING MANAGEMENT LANGUAGE FOR MODEL-BASED DATABASES

Valéry Téguiak[1], Yamine Ait-Ameur[2], Stéphane Jean[1] and Éric Sardet[3]

[1]*LIAS/ISAE-ENSMA, ENSMA and Poitiers Univerty, Futuroscope, France*

[2]*IRIT-ENSEEIHT, INPT-ENSEEIHT, Toulouse, France*

[3]*CRITT Informatique, Futuroscope, France*

teguiakh@ensma.fr, yamine@enseeiht.fr, jean@ensma.fr, sardet@ensma.fr

Keywords: mapping, meta-modeling, model transformation, ontology engineering, query languages.

Abstract: Nowadays model mapping plays a crucial role in applications manipulating various heterogeneous sources (data integration and exchange, datawarehouse, etc.). Users need to query a given data source and still obtain results from other mapped sources. If many model management systems have been proposed that support high-level operators on model mappings, a more flexible approach is needed supporting the querying of mapping models and the propagation of queries through mappings. As a solution, we present in this paper a mapping-based query language called MQL (Mapping Query Language). This language extends the SQL language with new operators to exploit mappings. We show the interest of this language for the multi-model ontology design methodology proposed in the ANR DaFOE4App project.

1 Introduction

In order to deal with various heterogeneous models used to represent the same real word domain, several mapping languages (Bouquet et al., 2003; Horrocks et al., 2004) or frameworks (Jouault et al., 2008; Melnik et al., 2003; Moha et al., 2010) have been proposed. These frameworks support either model mappings or model transformations. (Bouquet et al., 2003; Horrocks et al., 2004) allow users to express correspondences between models and (Jouault et al., 2008; Melnik et al., 2003; Moha et al., 2010) describe model transformations. Both approaches aim at performing instance migration. Most of these languages run in central memory and do not address scalability when dealing with huge amount of data.

Moreover, with the emergence of the Web, the amount of models and instances is growing drastically. Managing mappings in such a context often requires writing more and more undesirable complex queries. Therefore, offering solutions for managing such mappings and instances in a convenient way becomes a necessity if one wants to address real sized problems.

Before year 2000, mappings were implemented by programs, then (Bernstein et al., 2000) introduced the notion of *Model Management* that aimed at reducing the amount of programming needed for the development of metadata-intensive applications. More precisely, (Bernstein, 2003) has provided model management operators (e.g, *compose, diff, merge, match, etc*) allowing to manipulate and to manage models and mappings as objects. However, to understand and to use mappings established between source models, designers need to query and to exploit them in order to express a query on a data source and to obtain data results from other sources. Thus, a more flexible approach is needed for supporting the querying of mapping model and the propagation of queries through mappings. As a solution, we propose in this paper a mapping-based query language named MQL (Mapping Query Language). This language is an extension of traditional SQL query language with new operators to exploit mappings such as crossing or filtering mappings. The interest of this language is shown on a real use case extracted from the ANR DaFOE4App project.

This paper is organized as follows. Section 2 describes the use case set up to show the interest of our proposition. This use case is an ontology design methodology based on a multi-models approach. Section 3 discusses related work. After presenting our requirements for a new query language in section 4, we present in details in section 5 and 6 our proposition i.e. the MQL language. Finally section 7 concludes this paper and gives some perspectives of this work.

2 Case Study

The proposition of the MQL language has been motivated by the need of the ANR DaFOE4App project. This project consists of a new platform for the design of ontologies (a demonstration of this platform is available at `http://testcritt.ensma.fr/dafoe/demo/dafoeV1.zip`). This platform proposes to build an ontology starting from text and using intermediate models. Mappings are established between the different used models. This section presents the approach followed by the DaFOE platform to model and to store these mappings.

2.1 Ontology design in the DaFOE platform

The DAFOE platform provides a stepwise methodology for building ontologies from text analysis. The first step is dedicated to linguistic analysis (Terminology step) in which users manage linguistic information (terms and relations between terms) extracted with natural language processing tools. Then, a step for terms disambiguation (TerminoOntology step) is performed. Finally, a formalization step (Ontology step) allows users to create *classes* and *properties* of the ontologies and to populate the created classes. Each of these steps is autonomous and has its own model respectively presented in Figure 1, 2 and 3. Furthermore, two bridges, encoded by explicit model mappings, have been identified for switching between models: a first one for producing termino-ontology concepts from texts and a second one for producing ontology concepts from termino-ontology concepts.

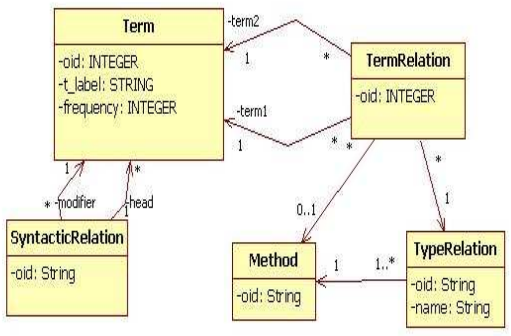


Figure 1: A subset of the Terminology model.

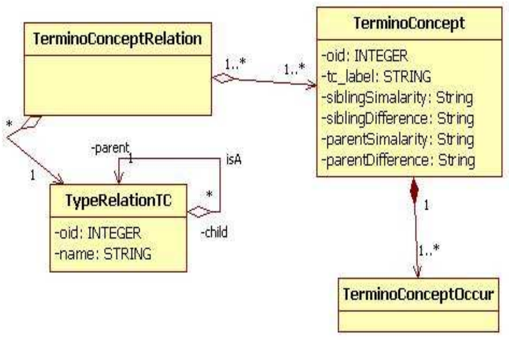


Figure 2: A subset of the TerminoOntology model.

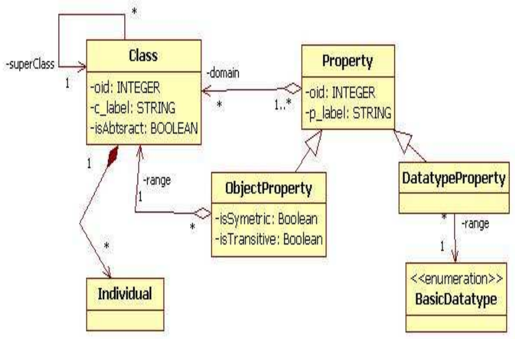


Figure 3: A subset of the Ontology model.

2.2 Modeling and Persistence of mappings

In (Téguiak et al., 2012), we argued that model-based databases (MBDB) are well adapted for handling mappings in a database context. Furthermore, because the underlying architecture (called OntoDB (H. Dehainsala, 2007)) that inspired our work does not explicitly handle mappings, we have extended it in a previous work (Téguiak et al., 2012). In that proposal, we have extended the meta-schema part of the OntoDB MBDB with a repository for mapping representations. More precisely, as illustrated in Figure 4 we have introduced mapping constructors in the OntoDB meta-model. In this resulting meta-model (named core meta-model) where models are defined by their entities and their attributes, three main constructors for creating correspondences are available. The first one, called *mLink*, is used to establish correspondences between models. The second one, called *eLink*, allows the user to establish correspondences between entities of models and finally, the *aLink* constructor is used to establish correspondences between attributes of entities. According to its arity (1:n), the *aLink* constructor uses an *expression* to write the target attribute in term of the sources attributes.

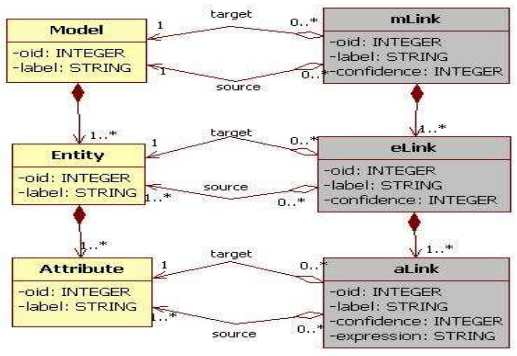


Figure 4: Core metamodel.

2.2.1 Terminology to TerminoOntology step

Considering both *Terminology* and *TerminoOntology* models, a simplified mapping between these models consists in:

- links between models. Creation of a *mLink* between the *Terminology* model and the *TerminoOntology* model;

- links between entities. Creation of a *eLink* from the *Term* entity and the *TerminoConcept* entity to express that instances of the *Term* entity will be transformed into instances of the *TerminoConcept* entity. A *eLink* between the *TermRelation* entity of the *Terminology* model and the *TerminoConceptRelation* entity of the *TerminoOntology* model is also available;

- links between attributes. Creation of a *aLink* expressing that an instance of the *TerminoConcept* entity has the same *label* as the one of its corresponding instances of the *Term* entity, prefixed by *'tc_'*. Another *aLink* expresses that the *rate* of an instance of *TerminoConcept* entity, equals to the *frequency* of the corresponding instance in the *Term* entity divided by 100.

2.2.2 TerminoOntology to Ontology step

For *TerminoOntology* and *Ontology* models, a simplified mapping consists in:

- links between models. Creation of a *mLink* between the *TerminoOntology* model and the *Ontology* model;

- links between entities. In the context of the previous created *mLink* between models, a *eLink* is created between the *TerminoConcept* entity and the *Class* entity to express that instances of the *TerminoConcept* entity will be transformed into instances of the *Class* entity. A *link* from *TerminoConceptRelation* of the *TerminoOntology* model and the *Property* entity of the Ontology model is also available;

- links between attributes. Creation of a *aLink* expressing that an instance of the *Class* entity has the same *label* as the one of its corresponding instance in the *TerminoConcept* entity. Another *aLink* expresses that the *relevance factor* of an instance of *Class* entity, equals to the *rate* of the corresponding instance in the *TerminoConcept* divided by 10.

As an illustration, assume that instances of the Ontology, TerminoOntology and Terminology models are respectively represented by Tables 1, 2 and 3. Thanks to mappings, a user who queries the *Class* entity through of the Ontology model would want to query both TerminoConcept of the TerminoOntology model and Term of Terminology model (Cf. Table 4).

Table 1: Ontology model: Class.

Class			
oid	c_label	relevance	isAbtract
1000	tc_car	0.01	true
1001	tc_wheel	0.002	false
1002	motor	0.008	true
1003	electric_motor	0.04	false

Table 2: TerminoOntology model: TerminoConcept.

TerminoConcept			
oid	tc_label	definition	rate
600	tc_car	...	0.1
601	tc_wheel	...	0.02
602	motor	...	0.08
603	motorcycle	...	0.8

Table 3: Terminology model: Term.

Term		
id	t_label	frequency
300	car	1
301	wheel	2
302	bicycle	30
303	handlebar	1

Putting these mappings all together results in a stepwise design methodology for a database recording (Cf. Figure 5) manipulated data and produced

Table 4: Query based on the Ontology model.

Results			
id	label	relevance	isAbstract
1001	tc_car	0.01	true
1002	electric_motor	0.04	false
603	motorcycle	0.08	null
302	bicycle	0.03	null
303	handlebar	0.01	null
...	...	...	...

to build an ontology from texts according to the process defined in the DaFOE4App project. The resulting database is a MOF-like repository where M_{i+1}/M_i means that the M_i level is represented as instances of the M_{i+1} level.

3 Related work

There are multiple areas related to database systems with the definition of a query language for both defining, manipulating or querying models and mappings between models. In this section we divide these approaches in three categories.

3.1 Transformation languages

A model transformation language is a vocabulary and a grammar with well-defined semantics for performing mappings. In the context of model-driven engineering, there are model transformation languages, that take as input models conforming to a given metamodel and produce as output models conforming to another metamodel. Such languages are often declarative to provide a cleaner and simpler implementation for simple mappings. Imperative constructs are also provided so that some mappings that are too complex to be handled declaratively are still describable.

However, in these languages, models and metamodels need to be explicitly loaded and stored in computer memory. As consequence, incremental model transformation is often not supported, so the whole source model is read and the target model is created when the transformation is executed. These languages lack of a sub-language for querying models, mappings and data by interpreting mappings created between models.

Among this kind of languages, we quote representative ones like Query/View/Transformation (QVT) (Kurtev,), the standardized language of the Object Management Group (OMG), ATL (Jouault et al., 2008) a hybrid model-to-model transformation language that supports both declarative and imperative constructs, Kermeta (Moha et al., 2010) a general purpose modeling and imperative programming language that offers EMF-based metamodeling, constraints, checks, transformation and behavior support.

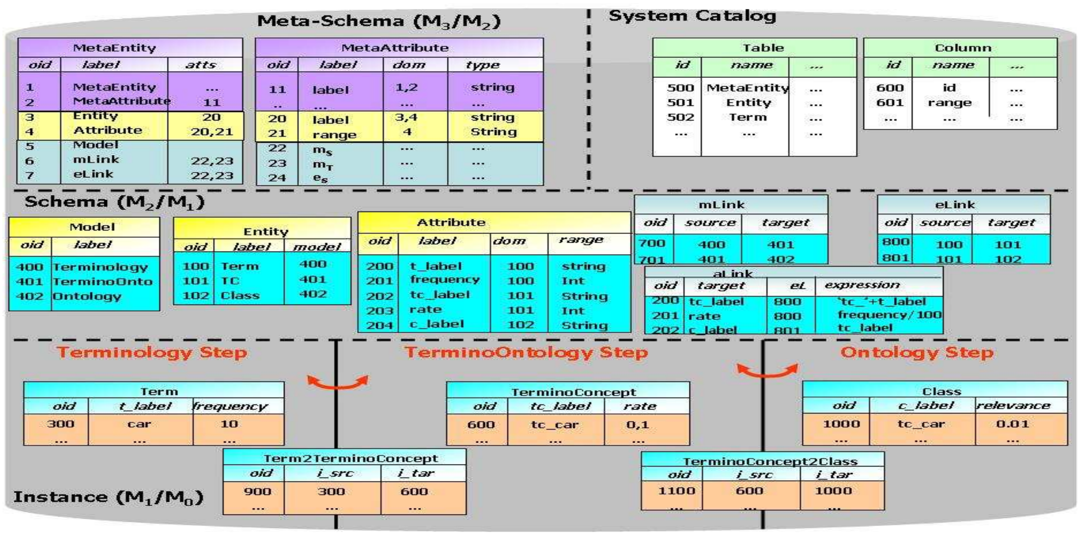
3.2 Languages for metadata repository systems

Metadata repository systems manage metadata commonly represented as models or meta-models. In order to facilitate repository application development, a dedicated query language, addressing the specific capabilities of such systems is required. The common approach is to build a MOF-based query language that provides capabilities to manipulate both data and meta-data. Representative query languages among this type of language are SchemaSQL (Laks V. S. Lakshmanan, 2001), MSQL (John Grant, 1993), SQL/M (Kelley et al., 1995), mSQL (Petrov and Nemes, 2008) dedicated for multi-/federated databases. Other proposals are encountered in the domain of ontology-based application with OntoQL (Jean et al., 2006) that exploits its underlying metamodeling database architecture called OntoDB (H. Dehainsala, 2007), SparQL (Konstantinos et al., 2010) dedicated for RDFS metamodeling database [1], etc. In general, this type of query language does not handle mappings between models because mappings are often neither explicitly represented nor exploited.

3.3 Mapping oriented languages

By mapping oriented query language, we mean languages that explicitly represent mappings between models and offer capabilities to exploit these mappings when querying data. Among these languages, we quote languages like SparQL (Konstantinos et al., 2010). Indeed in OWL-based languages, ontologies can be interconnected using *equivalent class constructors* and SparQL allows a user to query the resulting graph of interconnected classes. Another mapping oriented languages is Metaweb Query Language dedicated to query the FreeBase repository system [2]. We also have some model management platform such as Rondo (Melnik et al., 2003) that provides high level mapping operators for model management (*diff, compose, merge, etc*) but that lacks a query language embedding mappings. As limitation, these mapping oriented languages and frameworks do not allow a user to customize the mapping exploitation process. In many cases, the exploitation process is hidden to the

[1]http://www.w3.org/TR/rdf-schema/
[2]http://www.freebase.com/



MetaEntity

oid	label	atts
1	MetaEntity	...
2	MetaAttribute	11
3	Entity	20
4	Attribute	20,21
5	Model	
6	mLink	22,23
7	eLink	22,23

MetaAttribute

oid	label	dom	type
11	label	1,2	string
..	...	...	...
20	label	3,4	string
21	range	4	String
22	m_S	...	...
23	m_T	...	...
24	e_S	...	...

Table

id	name	...
500	MetaEntity	...
501	Entity	...
502	Term	...
...	...	...

Column

id	name	...
600	id	...
601	range	...
...	...	...

Model

oid	label
400	Terminology
401	TerminoOnto
402	Ontology

Entity

oid	label	model
100	Term	400
101	TC	401
102	Class	402

Attribute

oid	label	dom	range
200	t_label	100	string
201	frequency	100	Int
202	tc_label	101	String
203	rate	101	Int
204	c_label	102	String

mLink

oid	source	target
700	400	401
701	401	402

eLink

oid	source	target
800	100	101
801	101	102

aLink

oid	target	eL	expression
200	tc_label	800	'tc_'+t_label
201	rate	800	frequency/100
202	c_label	801	tc_label

Term

oid	t_label	frequency
300	car	10
...	...	...

TerminoConcept

oid	tc_label	rate
600	tc_car	0,1
...	...	...

Class

oid	c_label	relevance
1000	tc_car	0.01
...	...	...

Term2TerminoConcept

oid	i_src	i_tar
900	300	600
...	...	...

TerminoConcept2Class

oid	i_src	i_tar
1100	600	1000
...	...	...

Figure 5: Mapping management in the DaFOEApp project.

user and all the graph of interconnected database is used even if the user wants to use only a sub-part of this graph. Furthermore, in these languages or frameworks, the representation of mappings is static and can not be extended.

As illustrated above in our case study, our proposed database structure (Cf. Figure 5) is a MOF-like database that also handles mappings between models. However, this database is a bit more complex to manage using classical SQL queries. Indeed the extensions that we have proposed has introduced a new data structure (i.e. mappings) and therefore, as it is the case for abstract datatype descriptions, it requires its own management operators that encapsulate the structure of the datatype and therefore hide the mapping structure for the user. This drawback brings us to design a query language bypassing limitations of languages presented above and that makes easier querying databases (using mappings between models) by providing high level operators that hide the internal mapping representation in the database. Next section discusses the requirements for such a query language.

4 Requirements for a mapping-based query language

(Wakeman and Jowett, 1993; OMG, 2003; Patrascoiu, 2004; Petrov and Nemes, 2008) have investigated requirements for higher-level query languages managing both data and metadata (e.g, models). Here we summarize them and introduce new requirements specific to mappings exploitation.

4.1 High-level requirements

1. *Uniform treatment of data and metadata*. The language should allow users to manage data and metadata interchangeably, since metadata seems as data in a more abstract point of view. Hence query language constructors must be as much as possible invariant with respect to repository objects from different abstraction levels.
2. *Usability and expressiveness*. The language must be easy to use and expressive. These features refer for example to the compatibility with the commonly used SQL language.
3. *Reflect the repository features*. The language should consider the characteristics of the underlying repository system, in terms of metadata organization, data model, etc.

4.2 Mappings exploitation requirements

In a classical database, data retrieval is performed by SQL queries of the form "SELECT ... FROM ... WHERE ...". This type of queries does not take into account the notion of mapping because they do not exploit possible existing mappings between models. In practice, a user handles explicitly the notion of mappings when writing his queries. This situation raises the following other requirements.

4.2.1 Handling complex queries

Considering the example of section 2 and assuming that a user needs to write a query whose results are given in Table 4. Two situations may occur.

On the one hand, if the user knows the mappings characteristics, so he/she can manually write the appropriate following SQL queries:
(1) SELECT c_label, relevance FROM *Class*,
(2) SELECT tc_label, rate/10 FROM *TerminoConcept*,
(3) SELECT concat('tc_',t_label) (frequency/100)/10 FROM *Term*,

Queried attributes and entities of these three queries are defined according to the description of each model. For the *Ontology* model, *c_label* and *relevance* are retrieved, while *tc_label* and *rate* (respectively *t_label* and *frequency*) are retrieved for the *TerminoOntology* model (respectively the *Terminology* model) according to the defined mappings characteristics between these models.

On the other hand, because mappings characteristics may be evolved dynamically (new mappings may be created while existing one may be deleted or updated, just as in a peer to peer system (Iraklis and Joemon, 2003; Halevy et al.,), one needs firstly to query mappings network for characteristics retrieval, and then write the appropriate queries based on these characteristics. Such a query requires to access a repository which represented instances are model and mappings between models.

4.2.2 Handling mapping navigation

Considering more closely the second situation of the requirement presented in section 4.2.1 where users need to query mappings repository in order to retrieve mappings characteristics. One can ask itself how to handle transitivity with query languages such as SQL for example. This issue refers to the needs to dynamically navigate through the mappings hiding the exploitation of these mappings. So a policy for a transitive subqueries propagation through chains of arbitrarily huge mapped models is required because these models may contain huge amount of data.

4.2.3 Providing persistent mappings

This requirement refers to the problem of *memory saturation*, that means avoiding loading into central memory big amount of data whose models are mapped together. Indeed, the mappings repository may become very huge and therefore expensive (in response time and memory consumption) for navigation purposes because, as we presented in this paper, new models (says news modeling steps) could be created dynamically according to the needs of a particular user. Thus, a persistent-based approach is required.

5 Our approach

In this section, we present an overview of the MQL (Mapping Query Language), our mapping-based query language proposal for handling mappings according to previous quoted requirements. This language is highly coupled with the Model Based Database (MBDB) persistence approach proposed in (Téguiak et al., 2012). For each part (meta-schema, schema, instance) of the MBDB, the MQL language provides operators to define, manipulate and query its content. To keep this paper in reasonable size, we present in more details only the instance management part of MQL (Cf. Section 6). Moreover a complete description of the query processor is available in (Téguiak et al., 2011).

5.1 MetaModel Management

Since our core metamodel is not static and can be extended, the elements of this level of information must not be encoded as keywords of the MQL language. To define the syntax of the MQL language, we have chosen to adapt SQL to our core metamodel. Thus, the MetaModel Management operators allows users to create, modify and delete concepts of the core metamodel using a syntax similar to the SQL manipulation user-defined types (CREATE, ALTER, DROP).

Example 1. Construction of the core metamodel: Create the concept Model of the core metamodel.

CREATE MetaENTITY Model (oid Integer, label String).

Example 2. Construction of the core metamodel: Add the constructor for handling correspondences between model.

CREATE MetaENTITY mLink (target REF(Model), source REF(Model)).

This statement adds the *mLink* concept to our core metamodel. This concept is created with two attributes, *target* and *source*, whose datatypes are model identifiers. Indeed, the *REF syntax* has the same semantics as the *Foreign key* semantics in SQL. Notice that, for readability of queries, *label* of concept are used instead of *object identifier (oid)* (*REF(Model)* instead of *REF(5)* for example).

As a result of these previous queries, rows 5 and 6 are inserted in table *MetaEntity* (Cf. Figure 5) and two tables (*Model* and *mLink*) are automatically created in the schema part.

5.2 Model Management

To create, modify or delete the concepts of a created model, we have defined Model Management operators. In the same way as for the data definition language, we have adapted the SQL data manipulation language (INSERT, UPDATE, DELETE) to the data model of this part.

Example 3. Create a mapping between Terminology and TerminoOntology models.

INSERT INTO mLink(oid, label, confidence, source, target) VALUES (001, "Terminology to TerminoOntology", 0.9, "Terminology", "TerminoOntology")

As a result, row 700 is inserted in table *mLink* of Figure 5.

Analogously, we have defined a Model Querying Language that allows the user to query the metamodel level.

Example 4. Retrieve all the mappings in which the Terminology model is involved.

SELECT label
FROM mLink
WHERE mLink.source = 'Terminology'
OR mLink.target = 'Terminology'

5.3 Instances Management

The MQL language provides instances management capacities. These instances management capacities are handled by two sub-languages. The first one, namely DQL (Data Query Language), provides statements for retrieving instances of created models. The second one, namely DML (Data Manipulation Language) is dedicated for inserting, deleting, updating instances of a created model. An illustration of the DML is presented in the next section where we motivated extensions we made to the SQL language.

6 DML: Querying instances and mappings together

Let's consider the example below and its corresponding query.

Example 5. Retrieve all the classes whose relevance factor is high than 0.01.

SELECT C.c_label as label, C.relevance
FROM Class C
WHERE C.relevance $\geq$ *0.01*

The result of this query does not take into account that in the DaFOE platform a *Class* is also a *TerminoConcept* which in turn is a *Term*. In this section, we firstly give a motivating example that illustrates the lack of an explicit representation of mapping in queries. Then we present our statements proposal for a query language that allows the user to write queries that navigate through the network of mappings between models.

6.1 A motivating example

Querying instances and mappings together refers to the need of embedding both instance and mapping levels in the same query. Let's consider again the Example 5.

To answer this example, the previous query (noted $Q_{Ontology}$) should retrieve additional data both from the Ontology model and from models directly or indirectly (by transitivity of mappings) mapped with the Ontology model such as the TerminoOntology model, the Terminology model, etc. To simplify, we assume that additional data should be retrieved from the TerminoOntology model.

Because the user does not know mapping characteristics between the Ontology and TerminoOntology models, he needs to write a mapping level query to retrieve these characteristics.

Q_1) *Retrieve the Ontology model.*

SELECT Model.oid
FROM Entity E, Model M
WHERE E.label= "Class" AND E.model= M.oid

Q$_2$) *Retrieve mLink where the Ontology model is involved as target of a mapping.*

SELECT mLink.oid
FROM mLink
WHERE mLink.target in Q_1

Q$_3$) *Retrieve the entities mapped to Class entity.*

SELECT eLink.source
FROM eLink,Entity E
WHERE eLink.mL in (Q_2)
AND eLink.oid= E.oid AND E.label= "Class"

Q$_4$) *Retrieve correspondences between entities where the Class entity is involved as target of a mapping.*

SELECT eLink.oid
FROM eLink
WHERE eLink.mL in (Q_2)

Q$_5$) *Retrieve mapped entities and mapped attributes (through their expression).*

SELECT E.label,aLink.exp
FROM Entity E, Attribute A, aLink
WHERE E.oid in (Q_3)
AND aLink.eL in (Q_4)
AND aLink.target= A.oid
AND A.dom= E.oid

Q$_5$ query returns all information for translating a query based on the *Class* entity of the Ontology model into a query based on the *TerminoConcept* entity of the TerminoOntology model. This query produces the results of Table 5. These results are exploited to write, for the TerminoOntology model, the Q$_{TerminoOntology}$ query, representing the translation of the Q$_{Ontology}$ query on the TerminoOntology model.

Q$_{TerminoOntology}$)
SELECT TC.tc_label as label, TC.rate/10 as relevance
FROM TerminoConcept TC
WHERE TC.rate/10 $\geq$ 0.01

Table 5: Mapping level results.

E.label	aLink.exp
TerminoConcept	tc_label
TerminoConcept	rate/10

As we can observe above, the process of unfolding queries on target models is not easy and may become complex if one needs to integrate the complete network of mappings. In this case, the user handles by himself the transitivity capabilities of mappings. A commonly approach to deal with this situation consists in writing a query translator. So, the user writes a query (Q$_{Ontology}$ for example) and the translator generates unfolded queries for target queried models. This queries generation process is hidden to the user and made implicit. In other words, this approach assumes that the user does not know any mappings characteristics usable for example to customize the queries generation process.

6.2 The MQL statements proposal

To address the requirements mentioned in section 2 and to deal with the limitations illustrated in the previous section, we propose to extend the classical "SELECT ... FROM ... WHERE ... " queries by the following clauses. In other words, our approach is and hybrid one that can be used even if a user knows mappings characteristics or not. As the main purpose of MQL is to facilitate navigation through mappings, we introduced some optional statements useful for navigation and query propagation in order to get compact syntactic queries. The proposed statements are exploited by the translator in the queries translation process to customize this process.

MATCH. Specify the target models in which the MQL query is propagated at runtime.

FILTER. When propagating a MQL query from a model m_1 to another model m_2, an entity of m_1 may correspond to several entities of m_2. In this case, one may want to restrict the translation so that it applies only to part of these entities. Such a restriction is described using the FILTER clause.

CONFIDENCE. Confidence degrees are often assigned to mappings in order to handle fuzzy mappings. This clause restricts the propagation of the MQL query for the models that satisfy the specified confidence degree. When specified, this clause is used as a threshold to be respected.

With closure. If specified, the propagation of the query is achieved through the mappings repository using the *transitive closure* in the way that, instances are retrieved according to the transitivity of available mappings.

DEPTH. When a MQL query uses the *With* *closure* clause, it may result in a memory saturation or a bad response time according to the size of the graph of mappings. The DEPTH clause specifies the depth exploration of the graph of mappings. For example, *"DEPTH 4"* means that the MQL query will be propagated transitively on four consecutive mappings at most .

mWHERE. Unlike the classical WHERE clause of a SQL query, the mWHERE clause allows users to specify predicates to filter correspondences. In other words, the mWHERE clause is comparable to a SQL WHERE clause, but it is dedicated to mapping level.

Furthermore, all the previous clauses allow the user to specify parameters available in the *core mapping model*. More concretely, let's consider the case of modeling fuzzy mappings. The classical representation of fuzzy mappings is to assign a *confidence* value between 0 and 1 (as usually modeled in the literature) for each described correspondence. However, assume that a user wants to provide another way for handling fuzzy mappings by defining a *quality* property for which values could be *"weak, average, good, excellent"*. In the context of mappings, it is required to extend mapping constructors (*mLink, eLink, aLink*) by creating a new property (called *quality* for example) that will be valuated with one of *weak, average, good, best* values. This extension is made possible thanks to the meta-schema part of the underlying MBDB. Once these mapping constructors have been extended, it is possible to use them in a MQL query using the mWHERE clause. For example, the MQL query "SELECT ... mWHERE *quality* = *"good"* will propagate the MQL query through the graph of mappings using only *"good"* mappings.

Table 6 shows the global MQL syntax for querying data including constraint on mappings.

Table 6: Full DQL of MQL query statement.

SELECT <attributes: VARCHAR [ ]>
FROM <entity_name: VARCHAR [ ]>
WHERE <criteria: PREDICATE [ ]>
MATCH <mapped_models: VARCHAR [ ]>
FILTER <mapped_entities_name: VARCHAR [ ]>
CONFIDENCE <conf: INTEGER>
DEPTH <dep: INTEGER>
With closure
mWHERE <mCriteria: mPREDICATE [ ]>

6.3 MQL in action

In this section, we present how MQL can be applied to answer the Example 5.

mQ$_1$)
SELECT C.c_label, C.relevance
FROM Class C
WHERE C.relevance $\geq$ 0.01

Applied in the Ontology model, the mQ$_1$ query returns (Cf. Table 7) data extracted only from the Ontology model (no mapping statement is specified in this query). In other words, this query is a classical SQL query. For readability, all the result records are prefixed by the name of the model from where they have been retrieved.

Table 7: Results of the mQ$_1$ MQL query.

Ontology(tc_car, 0.01)
Ontology(electric_motor, 0.04)
...

mQ$_2$)
SELECT C.c_label, C.relevance
FROM Class C
WHERE C.relevance $\geq$ 0.01
***MATCH** TerminoOntology*

Applied in the Ontology model, the mQ$_2$ query returns (Cf. Table 8) data extracted from both the Ontology and the TerminoOntology models (the *Match* statement for this query has been set to TerminoOntology).

Table 8: Results of the mQ$_2$ MQL query.

Ontology(tc_car, 0.01)
Ontology(electric_motor, 0.04)
TerminoOntology(motorcycle, 0.08)
...

mQ$_3$)
SELECT C.c_label, C.relevance
FROM Class C
WHERE C.relevance $\geq$ 0.01
***MATCH** **
***FILTER** **
***DEPTH** 1*
With closure

Applied in the Ontology model, the mQ$_3$ query returns (Cf. Table 9) data extracted from both Ontology and TerminoOntology models (the *Match* statement for this query has been set to all models using the * symbol). However due to the *Depth* statement, the results are limited to 1 transitive propagation. Only the TerminoOntology model is reachable from the Ontology model with 1 propagation.

Table 9: Results of the mQ_3 MQL query.

Ontology(tc_car, 0.01)
Ontology(electric_motor, 0.04)
TerminoOntology(motorcycle, 0.08)
...

mQ_4)
SELECT C.c_label, C.relevance
FROM Class C
WHERE C.relevance $\geq$ 0.01
***MATCH** **
***FILTER** **
With closure

Applied in the Ontology model, the mQ_4 query returns (Cf. Table 10) data extracted from both the Ontology, TerminoOntology and Terminology models. In other words thanks to the * symbol of the *Match* statement and with no *Depth* limitation, mQ_4 propagates to any model transitively reachable from the Ontology models.

Table 10: Results of the mQ_4 MQL query.

Ontology(tc_car, 0.01)
Ontology(electric_motor, 0.04)
TerminoOntology(motorcycle, 0.08)
Terminology(bicycle, 0.03)
...

Furthermore, using same considerations as presented in this section, we have built for the MQL language a Data Manipulation Language (DML) allowing to perform insert, delete and update operations. Analogously to data querying, the DML of the MQL language is an extension of SQL. Each operator (INSERT INTO ..., UPDATE ..., DELETE FROM...) of the SQL language has been enriched by the same clauses of mapping used for querying data in order to propagate the execution on other models along the network of interconnected schemas. Table 11 shows the DML operators.

Table 11: Full DML of MQL query statement.

INSERT INTO <entity_name: VARCHAR>   <attributes: VARCHAR [ ]>   VALUES <attributes_values: OBJECT [ ]>   **MATCH** <mapped_models: VARCHAR [ ]>   **FILTER** <mapped_entities_name: VARCHAR [ ]>   **CONFIDENCE** <conf: INTEGER>   **DEPTH** <dep: INTEGER>   **With closure**   **mWHERE** <mCriteria: mPREDICATE [ ]>
UPDATE <entity_name: VARCHAR>   SET <attributes_name: VARCHAR> =   <attributes_value: OBJECT>   WHERE <criteria: PREDICATE[ ]>   **MATCH** <mapped_models: VARCHAR [ ]>   **FILTER** <mapped_entities_name: VARCHAR [ ]>   **CONFIDENCE** <conf: INTEGER>   **DEPTH** <dep: INTEGER>   **With closure**   **mWHERE** <mCriteria: mPREDICATE [ ]>
DELETE FROM <entity_name: VARCHAR>   WHERE <criteria: PREDICATE[ ]>   **MATCH** <mapped_models: VARCHAR [ ]>   **FILTER** <mapped_entities_name: VARCHAR [ ]>   **CONFIDENCE** <conf: INTEGER>   **DEPTH** <dep: INTEGER>   **With closure**   **mWHERE** <mCriteria: mPREDICATE [ ]>

7 Conclusion

In this paper, we have presented a mapping-based query language called MQL that makes easier querying data thanks to available mappings between models. This language has a knowledge part based on a core metamodel allowing to represent both models and mappings between models. One of the main features of our approach is that this knowledge part can be extended by evolving the core metamodel. This extension capabilities is possible thanks to a more abstract level (called the metametamodel level) where the core metamodel itself is represented. The MQL language has been implemented for model-based databases persistent context, where both instance level, metamodel level and metametamodel are persisted in a single database. Furthermore, the MQL language allows the user to manipulate the knowledge and the instance levels independently or together (in a single query).

However, as illustrated in the context of the ANR DaFOE4App project for the process of building ontologies starting for texts where our approach has been applied, all these models were stored in a single database. As a perspective of this work, we are working on improving our approach so that it can be used for engineering data exchange where models are often stored in several databases and where good response time is required.

ACKNOWLEDGEMENTS

The authors would like to thanks the partners of the ANR DaFOE4App project for their contribution.

REFERENCES

Bernstein, P. A. (2003). Applying model management to classical meta data problems. In *Proceedings of the International Conference on Innovative Data Systems Research*.

Bernstein, P. A., Halevy, A. Y., and Pottinger, R. A. (2000). A vision for management of complex models. *SIGMOD Rec.*, 29:55–63.

Bouquet, P., Giunchiglia, F., Harmelen, F. V., Serafini, L., and Stuckenschmidt, H. (2003). C-owl: Contextualizing ontologies. In *ACM SIGIR'03*.

H. Dehainsala, G. Pierra, L. B. (2007). Ontodb: An ontology-based database for intensive applications. In *Proceedings of the International Conference on Database Systems for Advanced Applications*, pages 497–506.

Halevy, A. Y., Ives, Z. G., Madhavan, J., Mork, P., Suciu, D., and I. Tatarinov, T. .

Horrocks, I., Patel-Schneider, P., Boley, H., Tabet, S., Grosof, B., and Dean., M. (2004). Swrl: a semantic web rule language combining owl and ruleml.

Iraklis, K. and Joemon, J. (2003). An architecture for peer-to-peer information retrieval. In *ACM SIGIR'03*.

Jean, S., Ait-Ameur, Y., and Pierra, G. (2006). Querying ontology based database. the ontoql proposal. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*.

John Grant, Witold Litwin, N. R. T. S. (1993). Query languages for relational multidatabases. In *Proceedings of the International Conference on Very Large Data Bases*.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). Atl: a model transformation tool. In *Science of Computer Programming*.

Kelley, W., Gala, S., Kim, W., Reyes, T., and Graham, B. (1995). Schema architecture of the unisql/m multidatabase system. In *Modern Database Systems*.

Konstantinos, M., Nektarios, G., Nikos, B., and Stavros, C. (2010). Ontology mapping and sparql rewriting for querying federated rdf data sources. In *Proceedings of the International Conference on On the move to Meaningful Internet Systems*.

Kurtev, I. State of the art of qvt: A model transformation language standard, applications of graph transformations.

Laks V. S. Lakshmanan, Fereidoon Sadri, S. N. S. (2001). Schemasql: An extension to sql for multidatabase interoperability. In *Journal of Transactions on Database Systems*.

Melnik, S., Rahm, E., and Bernstein, P. A. (2003). Developing metadata-intensive applications with rondo. In *Journal of Semantic Web*.

Moha, N., Sen, S., Faucher, C., Barais, O., and Jézéquel, J.-M. (2010). Evaluation of kermeta for solving graph-based problems. In *Journal of Software tools for Technology Transfer*.

OMG (2003). Uml 2.0 ocl specification. omg document final/03-10-14.

Patrascoiu, O. (2004). Yatl: Yet another transformation language - reference manual version 1.0. In *Proceedings of the 1st European MDA Workshop*, pages 83–90.

Petrov, I. and Nemes, G. (2008). A query language for mof repository systems. In *Proceedings of the International Conference on On the move to Meaningful Internet Systems*, pages 354–373.

Téguiak, H. V., Ait-Ameur, Y., and Sardet, E. (2012). Use of persistent meta-modeling systems to handle mappings for ontology design. In *Proceedings of the International Conference on Models and Ontology-based Design of Protocols, Architectures and Services*, page To appear.

Téguiak, H. V., Ait-Ameur, Y., Sardet, E., and Bellatreche, L. (2011). MQL: an extension of SQL for mappings manipulation. Technical report, LISI/ENSMA.

Wakeman, L. and Jowett, J. (1993). *PCTE: the standard for open repositories*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.