A Language for Ontology-Based Metamodeling Systems

Stéphane Jean, Yamine Aït-Ameur, and Guy Pierra

LISI-ENSMA and University of Poitiers
BP 40109, 86961 Futuroscope Cedex, France
{jean,yamine,pierra}@ensma.fr

Abstract. Nowadays, metamodeling techniques and ontologies play a central role in many computer science problems such as data exchange, integration of heterogeneous data and models or software reuse. Yet, if many metamodeling repositories and ontology repositories have been proposed, few attempts have been made to combine their advantages into a single repository with a dedicated language. In this paper, we present the capabilities of the OntoQL language for managing the various levels of information stored in a metamodeling systems where (1) both data, models and metamodels are available and (2) semantic descriptions are provided using ontologies. Some of the main characteristics of OntoQL are: management of the metamodel level using the same operators as the ones applied to data and to model levels, semantic queries through ontologies and SQL compatibility. We report several applications where this language has been extensively and successfully used.

1 Introduction

In the last two decades, with the appearance of model driven engineering techniques where both data, models and mappings between models are represented as data, metadata play a major role and become central concepts manipulated by these techniques. They are used in the definition of computer based solutions for interoperability, data exchange, software reuse, model transformation and so on. At the origin, databases with their system catalogs and functional languages with continuations (like in Lisp) were the first systems that permitted the explicit manipulation of these metadata.

When designing information systems, the introduction of metadata leads to information systems whose architecture is composed of three levels representing: data, models and metamodels. Such systems are named metamodel based systems (MMS). Three domains present relevant examples putting into practice MMS systems: 1) ontologies, 2) database and 3) software engineering.

1. In the semantic web, ontology models are used for representing ontologies. Yet, these ontologies are themselves models of their instances. As a consequence, managing these data actually requires the three levels of information.
2. In database systems, data are instances of schemas whose structure is represented by instantiating metadata stored in the system catalog.

3. In software engineering, UML environments are designed on the basis of the MOF architecture [1] that also uses three levels of information.

A major problem that arises when managing MMS is the growing number of evolving metamodels. Coming back to our three examples, we notice that:

1. many ontology models are used: OWL [2], RDFS [3] or PLIB [4]. Moreover, most of these models are evolving regularly to satisfy new user needs;
2. each database system has its own system catalog structure supporting common features, with specificities not available in other system catalogs;
3. each UML environment supports a specific subset of the UML standard.

To support metamodels evolution, the OMG proposal consists in introducing a fourth upper level: the metametamodel [1]. Thus new metamodels are created as instances of this upper level. If this approach has been followed in software engineering and databases with systems such as ConceptBase [5] or Rondo[6] and languages such as mSQL [7] or QML [8], it is not generalized to ontology-based applications. These systems and languages could be used to represent ontologies but they would not exploit specific features of ontologies such as universal identification of concepts using namespaces/URI or multilingual descriptions.

The aim of this paper is to highlight capabilities of the OntoQL language for managing the various levels of information stored in a semantic MMS i.e, supporting semantic descriptions through ontologies. We have introduced this language in [9, 10] by presenting its capabilities to query ontologies and instances with its formal definition. In this paper, we show that this language can be used to access and modify the metamodel level in addition to its capability to express semantic queries and its compatibility with SQL. To illustrate our proposals, we consider a concrete example of semantic MMS: databases embedding ontologies and their instances namely Ontology Based DataBases (OBDBs). But, the reader shall notice that this approach is generic. In particular, the core metamodel defined for designing the OntoQL language managing OBDBs can be set up for other MMSs. Indeed, as shown later in the paper, the structure as well as the semantics of this core metamodel can be extended according to the targeted application domain. We report several applications where this feature of OntoQL has been extensively and successfully used.

This paper is structured as follows. In the next section, we introduce an example showing the need of modifying the metamodel level in OBDBs, i.e. the ontology model. This example is used throughout this paper. In section 3, we present the OBDB context we have used to illustrate our approach. The capability of the OntoQL language to uniformly manipulate the three levels of an OBDB is then presented in section 4. Its implementation on the OntoDB OBDB [11] and application in several projects are described in section 5. Finally, our conclusion, as well as some challenges for future work, are given in section 6.

2 A Motivating Example

When an ontology must be designed, an ontology model is required. This choice is difficult not only because of the numerous existing ontology models but also because they all have their own particularities that may be simultaneously needed for the addressed problem. For example, in the context of the e-Wok Hub project[1] whose aim is to manage the memory of many engineering projects on the CO2 capture and storage, experts must design an ontology on this domain. They must choose an ontology model knowing that on the one hand they have to precisely and accurately represent the concepts of this domain (by defining their physical dimensions associated with their units of measure and the evaluation context). This need suggests to use an ontology model such as PLIB. On the other hand, constructors introduced by ontology models borrowed from Description Logics such as OWL are also required to perform inferences in order to improve the quality of documents search. This example shows the need for manipulating the ontology model used in order to take advantages of each model.

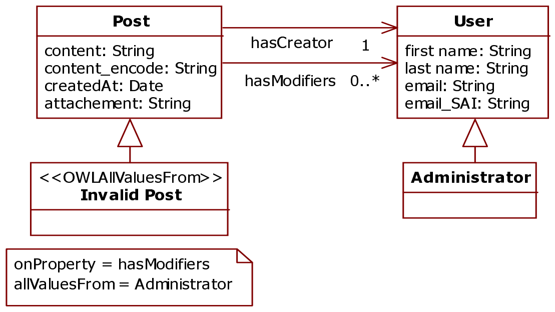

Fig. 1. An extract of an ontology for online communities

To illustrate the OntoQL language, we use an example of ontology. Instead of illustrating the use of OntoQL on a complex ontology related to the CO2 capture and storage domain, we have chosen an ontology example describing the on-line communities domain. This example, depicted in Figure 1, is influenced by the SIOC ontology (*http:// sioc-project.org/*). This ontology defines concepts such as `Community`, `Usergroup` or `Forum`. In this example we have identified the `User` and `Post` classes specialized by the `Administrator` and `InvalidPost` classes. The `InvalidPost` class is defined as an OWL restriction (`OWLAllValuesFrom`) on the `hasModifiers` property whose values must be instances of the `Administrator` class. Thus, an invalid post is a post that may be modified by administrators only. Since the UML notation does not allow us to represent this constructor, we have used a stereotype and a note to represent the `InvalidPost` class. Also notice that the UML notation prevents us to show the whole classes and properties description of this ontology (e.g. synonymous names or associated documents).

[1] http://www-sop.inria.fr/acacia/project/ewok/index.html

Before developing our proposal, we overview some systems that support ontologies management within database systems.

3 Context of the Study: OBDBs

Our proposal has been designed to manage MMS. However, to provide a concrete example of the interest of the proposed language, we consider a particular type of MMS: databases that store both ontologies and their instances, namely OBDBs. This section gives an overview of existing OBDBs and their associated languages.

3.1 Different Storage Structures for OBDBs

The most simple schema to store ontologies in a database is a three columns table `(subject, predicate, object)` [12] inspired from RDF triples. In this representation all the information (both ontologies and instances) are encoded by triples. This representation supports ontology model storage and modification. However, the performances of this representation led the research community to propose new solutions [13].

The second representation approach for OBDB consists in separating ontologies from their instance data. Data are stored either by 1) a triple representation [14], 2) or by a vertical representation [14, 15] where each class is represented by an unary table and each property by a binary table 3) or by a horizontal representation [11] where a table with a column for each property is associated to each class. Notice that whatever is the chosen representation for data, these systems use a fixed logical model depending on the used ontology model to store ontologies. By fixed, we mean a logical model that cannot evolve.

Finally, to our knowledge, OntoDB [11] is the only OBDB which separates ontologies and data and proposes a structure to store and to modify the underlying ontology model (PLIB). The representation of this part, namely the *metaschema*, allows a user to make the PLIB ontology model evolve or extend it with constructors from other ontology models if required.

As this section shows, the triple representation and the explicit representation of the metamodel are the only ways to enable the dynamic evolution of the ontology model used in an OBDB. In the next section we show the limits of existing languages for OBDBs to exploit this capability.

3.2 Existing Languages for OBDBs

Many languages have been proposed for managing OBDBs outlined in the previous section. This section overviews some of these languages according to the exploited ontology model.

Languages for RDFS

The RQL language [16] supports queries of both ontology and instances represented in RDFS. Many other languages with similar features have been proposed (see [17] for a survey). In RQL, all constructors of the RDFS ontology model are encoded as keywords in its grammar. For example the keyword `domain` is a function returning the domain of a property. As a consequence every evolution of this model requires a modification of the language grammar.

To manage the diversity of ontology models, the data model of the RQL language is not fully frozen. Indeed this data model can be extended by specializing the built-in entities `Class` and `Property`. However new entities can not be added except if they inherit from these two entities. As a consequence, PLIB constructors for associating documents to concepts of an ontology or the OWL constructor `Ontology` for grouping all the concepts defined in an ontology can not be represented. The data model of RQL can also be extended with new attributes. This extension may be useful to characterize concepts of an ontology (e.g., by a version number or an illustration). However such an extension requires to use the property constructor. So, the modification of the metamodel level also impacts a modification of the model level. Finally, even if these (partial) capabilities are offered by RQL, they are not supported or at least not explicitly on the OBDB RDF-Suite [15] and Sesame [14] which support this language.

Languages for RDF

SPARQL [18] is a recommendation of the W3C for exploiting RDF data. As a consequence it considers both ontologies and instances as triples. Thus, differing from RQL, SPARQL does not encode the constructors of a given ontology model as keywords of its grammar. This approach gives a total freedom to represent data, ontologies and the ontology model used as a set of triples. However, since ontology constructors such as class definition or subsumption relationship are not included in this model, SPARQL does not define any specific semantics for them. So SPARQL does not provide operators to compute the transitive closure of the subsumption relationship for given classes or operators to retrieve properties applicable on a class (those defined or inherited by this class). The result of a SPARQL query is dependent of the set of triples on which it is executed.

Langages independent of a given ontology model

To our knowledge, the SOQA-QL language [19] is the only language available to query ontologies and instances independently of the used ontology model. This capability has been provided by defining this language on a core ontology model (SOQA Ontology Meta Model) containing the main constructors of different ontology models. However this model is frozen. As a consequence evolutions of ontology models as well as addition of specific features are not supported.

As the previous analysis shows, each category of existing languages presents some limitations to support an uniform and shared manipulation of the three levels of an OBDB. These limits can be summarized as follows.

- Languages of the RQL category offer operators to exploit the semantics of the ontology model RDFS but extensions of this model are limited and mix the different levels of an OBDB.
- Langages of the SPARQL category offer a total freedom when manipulating the three levels of an OBDB considering all the information as triples. How-

ever, they do not offer operators supporting the exploitation of the usual semantics of ontology models.
– The SOQA-QL language supports the manipulation of data of an OBDB whatever is the used ontology model, provided that the used constructors are in the core ontology model on which this language is defined. However constructors not included in this core model can not be added.

This analysis leads us to propose another language. We design this language, named OntoQL, to manage the three levels of a MMS encoded in an OBDB.

4 OntoQL: Managing the Three Levels of MMS

We first present the data model targeted by OntoQL. Then, we present the OntoQL operators that manage each information level.

4.1 Data Model

The OntoQL language shall fulfill two requirements (1) enabling the modification of the metamodel level and (2) ensuring that these modifications match the semantics of MMSs: an instance of a metamodel (level named M_2 in the MOF) defines a valid model (M_1) which is itself expected to represent populations instances (M_0). These two requirements are fulfilled in the following way. (1) Contrary to usual DBMS where the metamodel level is fixed, OntoQL assumes the availability of a metametamodel level which is itself fixed (M_3). So instantiating this metametamodel modifies the metamodel level ensuring the flexibility of the metamodel level. (2) The metamodel level is composed of a predefined core model associated to an operational semantics. We describe this core model in the next section and then we show how it can be extended.

Core Metamodel

In an OBDB, the metamodel corresponds to the ontology model used to define ontologies. Contrariwise to the SOQA-QL language, we have chosen to include in this core model only the shared constructors of the main ontology models used in our application domain (engineering): PLIB, RDFS and OWL. However, the possibility to extend this model remains available.

Figure 2 presents the main elements of this core ontology model as a simplified UML model. We name *entities* and *attributes* the classes and properties of this UML model. The main elements of this model are the following.

– An ontology (`Ontology`) introduces a domain of unique names also called *namespace* (`namespace`). It defines concepts which are classes and properties.
– A class (`Class`) is the abstract description of one or many similar objects. It has an identifier specific of the underlying system (`oid`) and an identifier independent of it (`code`). Its definition is composed of a textual part (`name`, `definition`) which can be defined in many natural languages (`MultilingualString`). These classes are organized in an acyclic hierarchy linked by a multiple inheritance relationship (`directSuperclasses`).

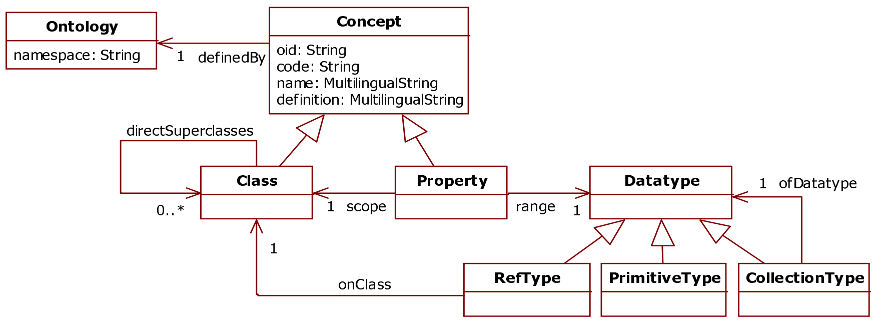

Fig. 2. Extract of the core metamodel

– Properties (`Property`) describe instances of a class. As classes, properties have an identifier and a textual part. Each property must be defined on the class of the instances it describes (`scope`). Each property has a range (`range)` to constraint its domain of values.
– The datatype (`Datatype`) of a property can be a simple type (`primitiveType`) such as integer or string. A property value can also be a reference to an instance of a class (`refType`). Finally, this value can be a collection whose elements are either of simple type or reference type (`collectionType`).

This core metamodel contains some specific features of ontologies (namespaces, multilinguism, universal identifier). A MMS based on this model supports ontology management and uses the defined ontologies to semantically describe other models. Moreover, since it includes the core components of standard metamodels (i.e, class, property and datatype), it may also be used as a kernel for numerous applications requiring a three-level architecture (e.g., enrichment of DBMS system catalog). Section 5 presents two examples issued from a real-world project.

Extensions of the Core Metamodel

The core metamodel can be extended with new entities and new attributes. When an extension is made by specialization, new entities inherit the behavior of their super entities. This behavior is defined in the operational semantics of the core metamodel. Thus, every specialization of the entity `Class` defines a new category of classes which supports by inheritance the usual behavior of a class. Every instance of the entity `Class` (or one of its specialization) defines a container which may be associated with instances. The name of this container is generated by a concretization function to access it through its representation at the meta level (M_2). Also every specialization of the entity `Property` defines relationships associating instances of classes to domain of values. The name of these relationships is also derived from the instances of properties which define them. Finally the specializations of the entity `Datatype` define domains of values that can be associated to properties.

4.2 OntoQL Operators for this Data Model

According to the data model defined, the creation of an element at the level M_i is done by instantiating the level M_{i+1}. However this process is not always practical. Indeed, in an usual DBMS, it would consist in creating a table using insert statements on the system catalog tables. So syntactic equivalences are systematically defined between insert statements (`INSERT`) at the level M_{i+1} and containers creation (`CREATE`) at the level M_i. These two syntactic constructions are valid but in general the second one is more compact. Figure 3 gives an overview of the statements available at the different levels as well as their meanings according to the target data structure. As shown on this figure, two equivalent syntactic constructions are provided for, on the one hand, inserting at the level M_3 or creating at the level M_2 and, on the other hand, inserting at the level M_2 or creating at the level M_1. In the following sections we describe more precisely the available statements.

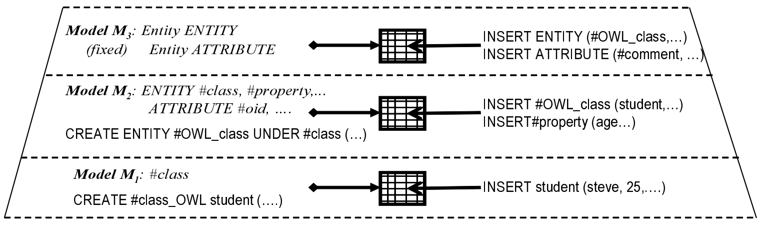
Model M3: Entity ENTITY
(fixed) Entity ATTRIBUTE
INSERT ENTITY (#OWL_class,...)
INSERT ATTRIBUTE (#comment, ...)
Model M2: ENTITY #class, #property,...
ATTRIBUTE #oid, ...
CREATE ENTITY #OWL_class UNDER #class (...)
INSERT #OWL_class (student,...)
INSERT#property (age...)
Model M1: #class
CREATE #class_OWL student (....)
INSERT student (steve, 25,....)

Fig. 3. Overview of the OntoQL language

Metamodel Definition Level and Model Manipulation/Querying

The OntoQL language provides operators to define, manipulate and query ontologies from the core metamodel presented in the previous section. Since this core metamodel is not static but can be extended, the elements of this level of information must not be encoded as keywords of the OntoQL language. So we have defined a syntactic convention to identify in the grammar a metamodel element. The convention we have chosen is to prefix each element of this model by the character `#`. This prefix indicates that this element definition must be inserted or searched in the metamodel level of the used MMS system.

To define the syntax of the OntoQL language, we have chosen to adapt SQL to the data model we have defined. Thus, the data definition language creates, modifies and deletes entities and attributes of the metamodel using a syntax similar to the manipulation of SQL user-defined types (`CREATE`, `ALTER`, `DROP`).

In this paper, we give examples to demonstrate the use of OntoQL only. However a more formal definition (an algebra) can be found in [10].

Example. Add the `AllValuesFrom` constructor OWL to the core metamodel.

```
CREATE ENTITY #OWLAllValuesFrom UNDER #Class (
    #onProperty REF(#Property),
    #allValuesFrom REF(#Class) )
```

This statement adds the `OWLAllValuesFrom` entity to our core metamodel as subentity of the `Class` entity . This entity is created with two attributes, `on-Property` and `allValuesFrom`, which take respectively as value identifiers of properties and identifiers of classes.

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

Example. Create the class named `InvalidPost` of our example ontology.

```
INSERT INTO #OWLRestrictionAllValuesFrom
         (#name[en], #name[fr], #onProperty, #allValuesFrom)
  VALUES ('InvalidPost', 'Post invalide', 'hasModifiers', 'Post')
```

This example shows that values of multilingual attributes can be defined in different natural languages (`[en]` for English and `[fr]` for French). It also shows that the names of classes and properties can be used to identify them. Indeed the value of the `onProperty` attribute is `hasModifiers`: the name of a property (and not its identifier). Notice that thanks to the syntactic equivalences previously presented, we could have written this statement with the `CREATE` syntax closer to the creation of user-defined types in SQL. It is also often more concise because properties can be created in the same time.

Finally a query language allows users to search ontologies elements stored in the OBDB. By designing this language starting from SQL we benefit from the expressivity of the object-oriented operators introduced in this language by the SQL99 standard. Thus the OntoQL language provides powerfull operators such as path expressions, nesting/unnesting collections or aggregate operators. The following example shows a path expression.

Example. Search the restrictions defined on the `hasModifiers` property with the class in which the values of this property must be taken.

```
SELECT #name[en], #allValuesFrom.#name[en]
  FROM #OWLRestrictionAllValuesFrom
 WHERE #onProperty.#name[en] = 'hasModifiers'
```

This query consists in a selection and a projection. The selection retrieves the restrictions on the property named in English `hasModifiers`. The used path expression in this selection is composed of the `onProperty` attribute which retrieves the identifier of the property on which the restriction is defined and of the `name` attribute which retrieves the name in English of this property from its identifier. The projection also applies the `name` attribute to retrieve the name of the restriction and the path expression composed of the `allValuesFrom` and `name` attributes to retrieve the name of the class in which the property implied in the restriction must take its values.

Model Definition and Data Manipulation/Querying

At this level, OntoQL defines, manipulates and queries instances of ontologies. The corresponding OntoQL syntax has again been defined starting from SQL to get an uniform syntax and remain compatible with existing database models. Thus each class can be associated to an *extent* which stores its instances. This data model generalizes the existing link between a user-defined type in SQL and a typed table which stores its instances. Indeed, contrary to a database schema which prescribes the attributes characterizing the instances of a user-defined type, an ontology only describes the properties that may be used to characterize the instances of a class. As a consequence the extent of class is only composed of the subset of properties which are really used to describe its instances. Thus, the OntoQL construct to create an extent of a class is similar to an SQL statement to create a typed table from a user-defined type. The only difference is that this statement precises the set of used properties.

Example. Create the extent of the `Administrator` class assuming that an administrator is described by his last name and his first name only.

```
CREATE EXTENT OF Administrator ("first name", "last name")
```

This statement creates a table with two columns to store instances of the `Administrator` class knowing that only the `first name` and `last name` properties describe them. This structure could also be a view on a set of normalized tables.

The semantics of OntoQL has been defined to search values of an instance for each property defined on its belonging class. When this property is not used to describe this instance, the `NULL` value is returned.

Example. Retrieve the users whose email address is known.

```
SELECT first_name, last_name FROM User WHERE email IS NOT NULL
```

The previous query will not return any administrator since the property `email` is not used on this class.

Another particularity of ontologies is that classes and properties have a namespace. The `USING NAMESPACE` clause of an OntoQL statement indicates in which namespace classes and properties must be searched. If this clause is not specified and no default namespace is available, OntoQL processes this clause as an SQL statement and thus it is compatible with SQL.

Example. Retrieve the content of the invalidate posts knowing that the class `InvalidPost` is defined is the namespace `http://www.lisi.ensma.fr`.

```
SELECT content FROM InvalidPost
USING NAMESPACE 'http://www.lisi.ensma.fr'
```

In this example the `InvalidPost` class and the `content` property are searched in the ontology which defines the namespace `http://www.lisi.ensma.fr`. In OntoQL many namespaces can be specified in the `USING NAMESPACE` clause and used to express queries composed of elements defined in different ontologies.

The next example illustrates a third particularity of ontologies: classes and properties have a textual definition that may be given in several different natural languages. This particularity is used in OntoQL to refer to each class and to each property by a name in a given natural language. The same query can be written in many natural languages.

Example. Retrieve the first and last names of users using a query in English (`7a`) and in French (`7b`).[2]

```
7a. SELECT "first name", "last name"  <=> 7b. SELECT prénom, nom
      FROM User                                  FROM Utilisateur
```

The query `7a` must be executed when the default language of OntoQL is set to English while the query `7b` requires the French default value.

Model and Data Querying

Queries on ontologies take as input/output parameter a collection of elements of the model level (e.g. classes or properties) while queries on instances take as input/output parameter a collection of instances of these elements (instances of classes and values of properties). The link between these two levels can be established in the two ways described in the next sections. (1) *From ontology to instances*, and (2) *from instances to ontology.*

From ontology to instances. Starting from classes retrieved by a query on ontologies, we can get the instances associated to these classes. Then we can express a query on data from this collection of instances. To query instances, following the approach of usual databases languages, OntoQL proposes to introduce an iterator `i` on the instances of a class `C` using the `C AS i` construct. The class `C` is known before executing the query. Moreover, OntoQL extends this mechanism with iterators on the instances of a class identified at run-time.

Example. Retrieve instances of the classes whose name ends by `Post`.

```
SELECT i.oid FROM #class AS C, C AS i WHERE C.#name[en] like '%Post'
```

In this query, the `Post` and `InvalidPost` classes fulfill the condition of selection. As a consequence this query returns instances identifiers of these two classes. And since `InvalidPost` is a subclass of `Post`, instances identifiers of the `InvalidPost` class are returned twice.

From instances to Ontology. Starting from instances retrieved by a query on data, we can retrieve the classes they belong to. Then a query on ontologies starting from this collection of classes can be expressed. OntoQL proposes the operator `typeOf` to retrieve the *basic class* of an instance, i.e. the minorant class for the subsumption relationship of the classes it belongs to.

[2] In the following examples, the default namespace is `http://www.lisi.ensma.fr`

Example. Retrieve the English name of the basic class of `User` instances.

```
SELECT typeOf(u).#name[en] FROM User AS u
```

This query iterates on the instances of the `User` class and on those of the `Administrator` class as well. For each instance, the query returns `User` or `Administrator` according to its member class.

4.3 Exploitation of the Capability of the OntoQL Language

Building Defined Concepts or Derived Classes

OntoQL can be used to compute the extent of a *defined concept* i.e., a concept defined by necessary and sufficient conditions in terms of other concepts. It also corresponds to a derived class in UML terminology.

Example. Build the extent of the class `InvalidPost`.

```
CREATE VIEW OF InvalidPost AS
SELECT p.* FROM Post AS p WHERE NOT EXISTS
  (SELECT m.* FROM UNNEST(p.hasModifiers) AS m
    WHERE m IS NOT OF REF(Administrator))
```

This query iterates on the instances `p` of the `Post` class. For each instance a subquery determines if this post has been modified only by administrators using the operator `UNNEST` (unnest a collection) and `IS OF` (type testing).

Model Transformation

The OntoQL language can also be used for model transformation. For example, an ontology represented in a source ontology model can be transformed in another one in a target ontology model. It requires to encode with the OntoQL operators, the defined equivalences between the source and target models.

Example. Transform the OWL classes into PLIB classes knowing that the attribute `comment` in OWL is equivalent to the attribute `remark` of PLIB.

```
INSERT INTO #PLIBClass (#oid, #code, #name[fr], #name[en], #remark)
SELECT (#oid, #code, #name[fr], #name[en], #comment) FROM #OWLClass
```

This statement creates a PLIB class for each OWL class. The value of the `remark` attribute of the created PLIB classes is defined with the value of the `comment` attribute of the OWL source classes. Other complex transformations may be written within this approach. We are currently extending OntoQL to allow a complete transformation language.

5 Implementation and Cases Study

OntoQL is implemented on the OntoDB OBDB. We briefly describe this implementation in next section, and then overview two case studies where this implementation has been put into practice.

5.1 Implementation of the OntoQL Engine

Two approaches could be followed to implement the OntoQL language on an OBDB. The first method consists in translating OntoQL into SQL. This approach has been followed in the RDF-Suite [15] system. The second method, followed by Sesame [14], consists in translating an OntoQL statement into a set of API-functions call that return a set of data of the OBDB.

The first solution benefits from the important effort on SQL optimization. However it has the drawback to be dependent of the OBDB on which the language is implemented. The second solution ensures the portability of the implementation on different OBDBs since it is possible to provide different implementation of the API for each OBDB. However in this case optimization is delegated to the implemented query engine.

In our implementation we have mixed these two methods. OntoQL is translated into SQL using an API which encapsulates the specificities of the OBDB. It follows 3 main steps: (1) generation of OntoQL algebra tree from a textual query (2) transformation of this tree into a relational algebra tree using an API (3) generation of SQL queries from this tree.

In addition to the implementation of the OntoQL Engine we have developed the following tools to ease the exploitation of this language.

- *OntoQL*Plus*: an editor of OntoQL statements similar to SQL*Plus provided by Oracle or isql provided by SQLServer.
- *OntoQBE*: a graphical OntoQL interface that extends the QBE interface such as the one provided by Access.
- *JOBDBC*: an extension of the JDBC API to execute OntoQL statements from the JAVA programming language.

Demonstrations and snapshots of these tools are available at *http://www.plib.ensma.fr/plib/demos/ontodb/index.html*.

5.2 Case Study: CO2 Capture and Storage

The capabilities of the OntoQL language to manipulate the metamodel level have been exploited in various projects [20–22] and especially in the eWokHub project we have introduced in section 2. Two extensions we have realized are outline below: (1) annotation of engineering models and (2) handling user preferences.

Engineering Models Annotation

The CO2 capture and storage rely on various engineering models. Engineers have to face several interpretation difficulties due to the heterogeneity of these models. To ease this process, we have proposed to annotate these models with concepts of ontologies [20]. However, the notions of annotations and engineering models were not available at the metamodel level. Thus, OntoQL was used to introduce these notions as first-order model concepts using a stepwise methodology. First, elements of the engineering models were created with the `CREATE ENTITY` operator. Then, an association table was defined to annotate the engineering models by a class of an ontology. Once the metamodel was extended, OntoQL has been used to query the engineering models departing from the ontology concepts.

User Preferences Handling

When the amount of ontological data (or instances) available becomes huge, queries return a lot of results that must be sorted by a user in order to find the relevant ones. This requirement raised from the eWokHub project where a huge amount of documents and engineering models were annotated by concepts and/or instances of ontologies. As a solution, we have enriched the metamodel to handle user preferences when querying the OBDB. Our proposition is based on a model of user preferences [21] defined at the metamodel level and stored in the OBDB using the `CREATE ENTITY` operator of OntoQL. This preference model is linked to the ontology model by associating preferences to classes or properties of ontologies (`ALTER ENTITY`). Finally, OntoQL has been extended with a `PREFERRING` clause interpreting preferences when querying the OBDB.

6 Conclusion

This paper presents the OntoQL language that combines capabilities of metamodeling languages i.e, a uniform manipulation of the three levels composing MMS systems with those of ontology query languages i.e, queries at the knowledge (semantic) level. This language is based on a core metamodel containing the common and shared constructors of most of the usual ontology models. Thus this core metamodel is used to define ontologies. Moreover, it can be extended in order to take into account specific features of a particular metamodel as shown in section 5. An implementation of the ontoQL language has been developed on top of the OntoDB ontology based database together with a suite of tools similar to those existing for SQL. This language has been used in the application domain of the CO2 capture and storage to annotate engineering models, to handle preferences and to semantically index Web Services [22].

Currently, we are studying new mechanisms to extend the core metamodel with new operators and behaviors. The challenge is to enable the automatic definition of operators behavior without any interactive programming but by exploiting a metamodel for behavioral modeling. More precisely, we are working on the possibility to dynamically add new functions that could be triggered during query processing preserving persistance of the models and their instances.

References

1. Object Management Group: Meta Object Facility (MOF), formal/02-04-03. (2002)
2. Dean, M., Schreiber, G.: OWL Web Ontology Language Reference. W3C Recommendation 10 February 2004. (2004)
3. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. World Wide Web Consortium. (2004)
4. Pierra, G.: Context Representation in Domain Ontologies and its Use for Semantic Integration of Data. Journal Of Data Semantics (JODS) **X** (2007) 34–43

5. Jeusfeld, M.A., Jarke, M., Mylopoulos, J.: Metamodeling for Method Engineering. MIT press (2009)
6. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: a programming platform for generic model management. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003). (2003) 193–204
7. Petrov, I., Nemes, G.: A Query Language for MOF Repository Systems. In: Proceedings of the OTM 2008 Conferences (CoopIS 2008). (2008) 354–373
8. Kotopoulos, G. Kazasis, F.C.S.: Querying MOF Repositories: The Design and Implementation of the Query Metamodel Language (QML). In: Digital EcoSystems and Technologies Conference (DEST 2007). (2007) 373–378
9. Jean, S., Aït-Ameur, Y., Pierra, G.: Querying Ontology Based Database Using OntoQL (an Ontology Query Language). In: Proceedings of Ontologies, Databases, and Applications of Semantics (ODBASE'06). (2006) 704–721
10. Jean, S., Aït-Ameur, Y., Pierra, G.: An Object-Oriented Based Algebra for Ontologies and their Instances. In: Proceedings of the 11th East European Conference in Advances in Databases and Information Systems (ADBIS'07). (2007) 141–156
11. Dehainsala, H., Pierra, G., Bellatreche, L.: Ontodb: An ontology-based database for data intensive applications. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07). (2007) 497–508
12. Harris, S., Gibbins, N.: 3store: Efficient bulk rdf storage. In: Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems. (2003)
13. Theoharis, Y., Christophides, V., Karvounarakis, G.: Benchmarking Database Representations of RDF/S Stores. In: Proceedings of the Fourth International Semantic Web Conference (ISWC'05). (2005) 685–701
14. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Proceedings of the First International Semantic Web Conference (ISWC'02). (2002) 54–68
15. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The ics-forth rdfsuite: Managing voluminous rdf description bases. In: Proceedings of the Second International Workshop on the Semantic Web (SemWeb'01). (2001)
16. Karvounarakis, G., Magkanaraki, A., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M., Tolle, K.: Querying the Semantic Web with RQL. Computer Networks **42**(5) (2003) 617–640
17. Bailey, J., Bry, F., Furche, T., Schaffert, S.: Web and Semantic Web Query Languages: A Survey. In: Reasoning Web. (2005) 35–133
18. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation 15 January 2008. (2008)
19. Ziegler, P., Sturm, C., Dittrich, K.R.: Unified Querying of Ontology Languages with the SIRUP Ontology Query API. In: Proceedings of Business, Technologie und Web (BTW'05). (2005) 325–344
20. Mastella, L.S., Aït-Ameur, Y., Jean, S., Perrin, M., Rainaud, J.F.: Semantic exploitation of persistent metadata in engineering models: application to geological models. In: Proceedings of the IEEE International Conference on Research Challenges in Information Science (RCIS 2009). (2009) 147–156
21. Tapucu, D., Diallo, G., Aït-Ameur, Y., Ünalir, M.O.: Ontology-based database approach for handling preferences. In: Data Warehousing Design and Advanced Engineering Applications: Methods for Complex Construction. (2009) 248–271
22. Belaid, N., Ait-Ameur, Y., Rainaud, J.F.: A semantic handling of geological modeling workflows. In: International ACM Conference on Management of Emergent Digital EcoSystems (MEDES'09). (2009) 83–90