An Object-Oriented Based Algebra for Ontologies and their Instances

Stéphane Jean, Yamine Aït-Ameur, Guy Pierra

Proc. of Advances in Databases and Information Systems (ADBIS'07), Varna, Bulgaria, Sept 29 - Oct 3, 2007.

An Object-Oriented Based Algebra for Ontologies and their Instances

Stéphane JEAN, Yamine AIT-AMEUR, and Guy PIERRA

Laboratoire d'Informatique Scientifique et Industrielle
LISI - ENSMA and University of Poitiers
BP 40109, 86961 Futuroscope Cedex, France
{jean,yamine,pierra}@ensma.fr

Abstract. Nowadays, ontologies are used in a lot of diverse research fields. They provide with the capability to describe a huge set of information contents. Therefore, several approaches for storing ontologies and their instances in databases have been proposed. We call Ontology Based Database (OBDB) a database providing such a capability. Several OBDB have been developed using different ontology models and different representation schemas to store the data. This paper proposes a data model and an algebra of operators for OBDB which can be used whatever are the used ontology model and representation schema. By extending the work done for object oriented databases (OODB), we highlight the differences between OODB and OBDB both in terms of data model and query languages.

Key words: Ontology, Database, Query Algebra, OWL, PLIB, RDF-S

1 Introduction

Nowadays, ontologies are used in a lot of diverse research fields including natural language processing, information retrieval, electronic commerce, Semantic Web, software component specification, information systems integration and so on. In these diverse domains, they provide with the capability to describe a huge set of information contents. Therefore, the need to manage ontologies as well as the data they describe in a database emerged as a crucial requirement.

We call Ontology Based Database (OBDB) a database that stores data together with the ontologies defining the semantics of these data. During the last decade, several OBDB have been proposed. They support different ontology models such as PLIB [1], RDFS [2] or OWL [3] for describing ontologies and they use different logical schemas for persistancy: unique table of triples [4], vertical representation [5] or table-like structure [6, 7] for representing the huge sets of data described by the ontologies.

In parallel to this work, ontology query languages like SPARQL [8] for RDF, RQL [9] for RDF-Schema, or OntoQL [10] for PLIB and a subset of OWL have been defined. Because of the lack of a common data model for OBDBs, dealing

with the heterogeneity of OBDB data models in order to implement these languages on top of OBDBs, is a major concern of current research activities. In this paper, we propose a data model for OBDBs which can be used whatever are the used ontology models and the representation schema.

Our work started by trying to answer to the following question: since ontologies use extensively object oriented concepts, why don't we use Object Oriented Database (OODB) models as the kernel of such a model? The answer to this question is that the OODB model is not usable without a necessary tuning effort. This answer led us to (1) highlight the existing differences between OODBs and OBDBs either from the conceptual or from the structural points of view and (2) propose another algebra of operators (to provide OBDBs with a formal semantics) extending the algebra defined for OODBs and (3) clarify the differences between OODBs and OBDBs query languages.

The objective of this paper is two-fold. On the one hand, we study and show the differences between OODB and OBDB models. For building our comparison and proposal, we have chosen *ENCORE* [11] as the OODB data model and its corresponding algebra of operators. On the other hand, we propose a generic algebra of operators defining a generic formal semantics for OBDB and show how these operators are used to describe queries of specific OBDB languages. Three languages based on different OBDB models illustrate this work: OntoQL [10], RQL [9] and SPARQL [8].

Compared to the OntoQL definition presented in [10], where we have presented the concrete syntax of OntoQL and its use on the OntoDB OBDB model [6], this paper presents a generic algebra of operators for OBDB models plus extensions and contributions like:

- support of the multi-instanciation paradigm;
- discussions of the differences between OODB and OBDB models and the corresponding query languages;
- the capability of the proposed model and algebra to overcome the heterogeneity of OBDBs data models;
- presentation of the query algebra at the different querying levels enabled by an OBDB: data, ontology and both data and ontology.

This paper is structured as follows. Next section presents a formal representation of the OBDB data model proposed in this paper. The differences between this data model and the OODB data model are highlighted as well. Section 3 presents the *ENCORE* algebra for OODBs and shows its insufficiencies to manage OBDBs. Then, an algebra based on the OBDB data model is presented as an extension of the *ENCORE* algebra. Section 4 discusses related work. Finally, section 5 concludes and introduces future work.

2 Data Models

2.1 The *ENCORE* Formal Data Model for OODBs

Formally, an OODB in the *ENCORE* data model is defined as a 8-tuple $<$ `ADT`, `O`, `P`, `SuperTypes`, `TypeOf`, `PropDomain`, `PropRange`, `Value` $>$, where:

- $\texttt{ADT}$ is a set of available abstract data types. It provides with atomic types ($\texttt{Int}$, $\texttt{String}$, $\texttt{Boolean}$), a global super type $\texttt{Object}$ and user-defined abstract data types;
- $\texttt{O}$ is the set of objects available in the database or that can be constructed by a query. All objects have an unique identifier;
- $\texttt{P}$ is the set of properties used to describe the state of each object;
- $\texttt{SuperTypes} : \texttt{ADT} \rightarrow 2^{\texttt{ADT}}$[1] is a partial function. It associates a set of super types to a type. This function defines a lattice of types. Its semantics is inheritance and it ensures substitutability;
- $\texttt{TypeOf} : \texttt{O} \rightarrow \texttt{ADT}$ associates to each object the lower (strongest) type in the hierarchy it belongs to;
- $\texttt{PropDomain} : \texttt{P} \rightarrow \texttt{ADT}$ defines the domain of each property;
- $\texttt{PropRange} : \texttt{P} \rightarrow \texttt{ADT}$ defines the range of each property;
- $\texttt{Val} : \texttt{O} \times \texttt{P} \rightarrow \texttt{O}$ gives the value of a property of an object. This property must be defined for the datatype of the object.

This data model supports collections of objects by providing the parameterized $\texttt{ADT}$ named $\texttt{Set}[\texttt{T}]$. $\texttt{Set}[\texttt{T}]$ denotes a collection type of objects of type $\texttt{T}$. $\{\texttt{o}_1, \ldots, \texttt{o}_n\}$ denotes an object of this type where the $\texttt{o}_i$'s are objects of type $\texttt{T}$. Another parameterized $\texttt{ADT}$, called $\texttt{Tuple}$, is provided to create relationships between objects. A $\texttt{Tuple}$ type is constructed by providing a set of attribute names ($\texttt{A}_i$) and attribute types ($\texttt{T}_i$). $\texttt{Tuple}[< (\texttt{A}_1, \texttt{T}_1), \ldots, (\texttt{A}_n, \texttt{T}_n) >]$ denotes a type tuple constructed using the $\texttt{A}_i$'s attribute's name and $\texttt{T}_i$'s attribute types. $< \texttt{A}_1 : \texttt{o}_1, \ldots, \texttt{A}_n : \texttt{o}_n >$ denotes an object of this type where the $\texttt{o}_i$'s are objects of the corresponding type $\texttt{T}_i$. The $\texttt{Tuple}$ type is equipped with the $\texttt{Get\_A}_i\texttt{\_value}$ functions to retrieve the value of a $\texttt{Tuple}$ object $\texttt{o}$ for the attribute $\texttt{A}_i$. The application of this function may be abbreviated using the dot-notation ($\texttt{o.A}_i$). The $\texttt{Tuple}$ type construct is fundamental for building new data types. In particular, it is useful for describing new data types that are not available in the core database schema and that may be built by expressions of the algebra.

2.2 Definition of the OBDB Data Model

The OBDB data model is based on the definition of two main parts : ontology and content. Instances are stored in the content part while ontologies are stored in the ontology part. The description of these two parts use extensively object-oriented database features. Let us describe these two parts and their relationships.

Ontology. The ontology part, can be formally defined like an OODB by a 7-Tuple as $< \texttt{E}, \texttt{OC}, \texttt{A}, \texttt{SuperEntities}, \texttt{TypeOf}, \texttt{AttDomain}, \texttt{AttRange}, \texttt{Val} >$. Here, abstract data types ($\texttt{ADT}$) are replaced by entities ($\texttt{E}$), properties ($\texttt{P}$) by attributes ($\texttt{A}$) and objects ($\texttt{O}$) by concepts of ontologies ($\texttt{OC}$). To define the built-in entities and attributes of this part, we have considered the constructors shared by the standard ontology models PLIB [1], RDF-Schema [2] and OWL [3]. Thus, in

[1] We use the symbol $2^{\texttt{C}}$ to denote the power set of $\texttt{C}$.

addition to atomic types, the global super type `ObjectE`, and the parameterized types `Tuple` and `Set`, `E` provides the predefined entities `C` and `P`. Instances of `C` and `P` are respectively the classes and properties of the ontologies. Each class and each property has an identifier defined in the context of a namespace. This is represented by the attribute $\texttt{namespace} : \texttt{C} \cup \texttt{P} \rightarrow \texttt{String}$. Entity `C` also defines the attribute $\texttt{SuperClasses} : \texttt{C} \rightarrow \texttt{SET}[\texttt{C}]$ and entity `P` defines the attributes $\texttt{PropDomain} : \texttt{P} \rightarrow \texttt{C}$ and $\texttt{PropRange} : \texttt{P} \rightarrow \texttt{C}$. The description of these attributes is similar to the definition given for an OODB. Moreover, a global super class `ObjectC` is predefined and the parameterized types `Tuple` and `Set` are also available for `C`. Thus, an ontology is similar to an OODB schema. However, an ontology gives a precise definition of concepts which means that many more attributes (name, comment, version ...) are available to describe classes and properties of ontologies. These predefined entities and attributes constitute the kernel of the ontology models we have considered. User-defined entities (restriction, objectProperty ...) and attributes (isSymetric, unionOf, remark ...) may be added to this kernel in order to take into account specific features of a given ontology model. This capability is illustrated in the following example.

Fig. 1. An illustration of our data model based on the MOF architecture

An example of the ontology part of an OBDB. Using the UML notation, figure 1 presents the data model defined for handling our ontologies. Let us comment this figure in a top-down manner. The upper part presents the subset of the data model defined to describe the used ontology model. In the MOF terminology [12], this part is the *metametamodel*.

The middle part presents the used ontology model. This part corresponds to the level M_2 of the MOF namely the *metamodel*. Each element of this *metamodel* is an instance of an element of the *metametamodel*. In this part, we have added to the predefined entities `C`, `P`, and `ObjectE`, some specific constructors of the OWL ontology model: `OWLRestrictionAllValuesFrom`, `OWLObjectProperty` and `OWL-DataProperty`. Due to space limitations, we did not represent the whole OWL ontology model. However, this can be handled using the OWL metamodel proposed by [13].

Finally, the lower part presents a toy example of an ontology inspired from the SIOC Ontology (`http://sioc-project.org/`). This part constitutes the level M_1 of the MOF. Each concept of the ontology is an instance of the ontology model defined at level M_2. The SIOC ontology describes the domain of online communities by defining concepts such as `Community`, `Usergroup` or `Forum`. In this example, we have represented the classes `User` and `Post` and refined them by the classes `Administrator` and `InvalidPost`. The class `InvalidPost` is defined as an `OWLRestrictionAllValuesFrom` on the property `hasModifiers` whose values must be instances of the class `Administrator`. Thus, an invalid post is a post which has been modified by an administrator only. Notice that the UML notation doesn't allow us to show the whole description of the classes and of the properties of this ontology (labels, comments, documents ...).

Content. The content part allows to store instances of ontology classes. Different representation schemas have been proposed and used by OBDBs (see [14] for a survey). The simplest and more general one uses an unique table of triples [4] where the data is stored in triples of the form (subject, predicate, object). Another representation schema is the vertical structure [5] where a binary table is used for each property and a unary table for each ontology class. Recently, horizontal approaches have been proposed [6, 7]. They associate, to each class, a table which columns represent those properties that are valued for at least one instance of the class. Our formalization is based on this latter approach.

In the proposed data model for OBDBs, the content part is a 5-tuple $<$ `EXTENT`, `I`, `TypeOf`, `SchemaProp`, `Val` $>$ where:

- `EXTENT` is a set of extensional definitions of ontology classes;
- `I` is the set of instances of the OBDB. Each instance has an identity;
- `TypeOf` : $\mathtt{I} \rightarrow 2^{\mathtt{EXTENT}}$ associates to each instance the extensional definitions of the classes it belongs to;
- `SchemaProp` : $\mathtt{EXTENT} \rightarrow 2^{\mathtt{P}}$ gives the properties used to describe the instances of an extent (the set of properties valued for its instances);
- `Val` : $\mathtt{I} \times \mathtt{P} \rightarrow \mathtt{I}$ gives the value of a property occurring in a given instance. This property must be used in one of the extensional definitions of the class

the instance belongs to. Since `Val` is a function, an instance can only have a unique value (which can be a collection) for a given property. Thus, if the same property is defined on different classes the instance belongs to, this property must have the same value in each extent associated to these classes.

Relationship between each part. The relationship between ontology and its instances (content) is defined by the partial function $\texttt{Nomination} : \texttt{C} \rightarrow \texttt{EXTENT}$. It associates a definition by intension with a definition by extension of a class. Classes without extensional definition are said *abstract*. The set of properties used in an extensional definition of a class must be a subset of the properties defined in the intensional definition of a class ($\texttt{propDomain}^{-1}(\texttt{c}) \supseteq \texttt{SchemaProp}(\texttt{nomination}(\texttt{c}))$).

An example of the content part of an OBDB. Figure 2 illustrates the OBDB data model on the content part. The horizontal representation of the extents of the four classes of our toy ontology are presented on figure 2(A). As shown on this example, some of the properties of an ontology class may not be used in class extent. For example, the property `content_encoded`, described in the ontology as "the encoded content of the post, contained in CDATA areas", is not used in the extent of the class `Post`. This example also demonstrates that properties used in the extent of a class may not be used in the extent of one of its subclasses. This is the case for the properties `first_name` and `last_name` which are used in the extent of `User` but not in the extent of `Administrator`.

On figure 2(B), the two main representations proposed for the content of an OBDB, i.e. the vertical and triple schemas are showed. Because of space limitations, the vertical representation corresponding to the extent of the class `User` is solely represented.

Fig. 2. Example of the content of an OBDB

A generic approach. Although, we have been using this approach on specific OBDBs with specific query languages (like OntoQL, RQL or SPARQL in this paper), this approach is generic and can be applied to other OBDBs thanks to the following capabilities.

1. The *metametamodel* offers the capability to add other attributes encoding specific information of a given ontology model.
2. The content or extent is associated to a class whatever is the logical model used to represent it (vertical, table-like, etc). A view can be associated to represent the extent of a class in order to hide the specific logical model.
3. From an implementation point of view, the operators of the algebra, we are discussing in this paper, define a generic Application Programming Interface allowing to access any logical model for contents provided that an implementation of these operators is supplied.

From these three features, we notice that our approach is flexible enough to handle different ontology models either from a conceptual modeling point of view (1,2) or from an implementation point of view (2, 3).

2.3 Differences between OODB and OBDB Data Models

This section describes the differences between the OODB and OBDB data model. Indeed, one can ask why another database model. Below, we describe a list of main identified differences from a structural and logical point of views.

Ontology part. From a functional point of view, the OBDB data model differs from the OODB model in the sense that the former stores not only the logical database model but it also stores the ontology which provides (1) data definition and descriptions independently of any context of use, while the latter stores the logical model of a database for a given application and (2) a formal theory which allows to check some level of consistency and to perform some level of automatic reasoning over the ontology-defined concepts.

Incomplete information. An extent of an ontology class is similar to a typed table associated to a user-defined type in the relational-object data model or to an extent of an object class in the ODL object-oriented data model. However, there is a crucial difference between ontologies and conceptual models of classical databases. Indeed, while a conceptual model *prescribes* which attributes *shall* be represented for each instance of each particular class, an ontology just *describes* which properties *may* be associated with instances of a class. Thus, the extent of an ontology class gathers only the set of properties valued for its instances.

Subsumption relationships. As we have pointed out in the previous section, applicable properties are distinguished from used properties in the OBDB data model. If the applicable properties are inherited through the subsumption relationship as in the object-oriented data model, this is not the case for used properties. Ontology classes may be linked by a subsumption relationship without implicit inheritance of valued properties (partial inheritance).

Universal identification of classes and properties. The identifier of classes and properties are defined in the context of a namespace allowing to universally refer to this concept from any other environment, independently of the particular system where this concept was defined.

3 Query Algebras

On top of the proposed data model, an algebra which can be the basis of an ontology query language whatever is the used ontology model is built. From the listing of the previous differences, it appears that the algebras defined for OODBs are not fully adequate for OBDBs although, as stated previously, the OBDB data model uses extensively OODB features. As a consequence, we have chosen to tune, specialize, extend and reuse the operators of the *ENCORE* algebra in order to get benefits of their properties (e.g., closure, completeness and equivalence rules). Next subsection reviews the main operators of this algebra and subsection 3.2 presents the *OntoAlgebra* algebra we propose.

3.1 Main Operators of the *ENCORE* Query Algebra

Each operator of the *ENCORE* algebra takes a collection of objects whose type is an `ADT T` and returns a collection of objects whose type is an `ADT Q`. Thus, the signature of such an operator is $\texttt{Set[T]} \rightarrow \texttt{Set[Q]}$. Following our formalisation, we use the signature $\texttt{ADT} \times 2^{\texttt{O}} \rightarrow \texttt{ADT} \times 2^{\texttt{O}}$ in order to record the datatype and its set of objects. Let us briefly describe the main operators of the *ENCORE* algebra.

Image. The *Image* operator returns the collection of objects resulting from applying a function to a collection of objects. Its signature is $\texttt{ADT} \times 2^{\texttt{O}} \times \texttt{Function} \rightarrow \texttt{ADT} \times 2^{\texttt{O}}$. `Function` contains all the properties in `P` and all properties that can be defined by composing properties of `P` (path expressions). It is easy to extend the domain of `PropDomain`, `PropRange` and `Val` from `P` to `Function`. So, this operator is defined by:

$$\texttt{Image}(\texttt{T}, \{\texttt{o}_1, \ldots, \texttt{o}_\texttt{n}\}, \texttt{f}) = (\texttt{PropRange}(\texttt{f}), \{\texttt{Val}(\texttt{o}_1, \texttt{f}), \ldots, \texttt{Val}(\texttt{o}_\texttt{n}, \texttt{f})\}) \ .$$

Project. The *Project* operator extends *Image* allowing the application of more than one function to an object. The result type is a `Tuple` whose attribute names are taken as parameter. Its signature is $\texttt{ADT} \times 2^{\texttt{O}} \times 2^{\texttt{String} \times \texttt{Function}} \rightarrow \texttt{ADT} \times 2^{\texttt{O}}$ and it is defined by:

$$\begin{aligned} &\texttt{Project}(\texttt{T}, \texttt{O}_\texttt{t}, \{(\texttt{A}_1, \texttt{f}_1), \ldots (\texttt{A}_\texttt{n}, \texttt{f}_\texttt{n})\}) = \\ &\quad (\texttt{Tuple}[< (\texttt{A}_1, \texttt{PropRange}(\texttt{f}_1)), \ldots, (\texttt{A}_\texttt{n}, \texttt{PropRange}(\texttt{f}_\texttt{n})) >], \\ &\qquad \{< \texttt{A}_1 : \texttt{Val}(\texttt{o}, \texttt{f}_1), \ldots, \texttt{A}_\texttt{n} : \texttt{Val}(\texttt{o}, \texttt{f}_\texttt{n}) > | \texttt{o} \in \texttt{O}_\texttt{t}\}) \ . \end{aligned}$$

Select. The *Select* operator creates a collection of objects satisfying a selection predicate. Its signature is $\texttt{ADT} \times 2^{\texttt{O}} \times \texttt{Predicate} \rightarrow \texttt{ADT} \times 2^{\texttt{O}}$ and it is defined by:

$$\texttt{Select}(\texttt{T}, \texttt{O}_\texttt{t}, \texttt{pred}) = (\texttt{T}, \{\texttt{o} | \texttt{o} \in \texttt{O}_\texttt{t} \wedge \texttt{pred}(\texttt{o})\}) \ .$$

OJoin. The *OJoin* operator creates relationships between objects of two input collections. Its signature is $\texttt{ADT} \times 2^{\texttt{O}} \times \texttt{ADT} \times 2^{\texttt{O}} \times \texttt{String} \times \texttt{String} \times \texttt{Predicate} \rightarrow \texttt{ADT} \times 2^{\texttt{O}}$ and it is defined by:

$$\texttt{OJoin}(\texttt{T}, \texttt{O}_\texttt{t}, \texttt{R}, \texttt{O}_\texttt{r}, \texttt{A}_1, \texttt{A}_2, \texttt{pred}) =$$
$$(\texttt{Tuple}[< (\texttt{A}_1, \texttt{T}), (\texttt{A}_2, \texttt{R}) >], \{< \texttt{A}_1 : \texttt{t}, \texttt{A}_2 : \texttt{r} > | \texttt{t} \in \texttt{O}_\texttt{t} \wedge \texttt{r} \in \texttt{O}_\texttt{r} \wedge \texttt{pred}(\texttt{t}, \texttt{r})\}) \ .$$

The definition of this operator is modified when it takes a type `Tuple` as parameter. Indeed, it becomes necessary to flatten the resulting nested tuples in order to preserve composition. In this case, the flattening operation allows the preservation of the associativity of this operator.

In addition to these main operators, the *ENCORE* algebra includes set operations (*Union*, *Difference*, and *Intersection*) and collection operations (*Flatten*, *Nest* and *UnNest*). All these operators define an algebra allowing to query OODBs. Next, we show how the definitions of these operators could be reused and extended for querying an OBDB.

3.2 *OntoAlgebra*: Adaptation of Encore Query Algebra to the OBDB Data Model

Signatures of the operators defined on the OBDB data model are $(\texttt{E} \cup \texttt{C}) \times 2^{\texttt{OC} \cup \texttt{I}} \rightarrow (\texttt{E} \cup \texttt{C}) \times 2^{\texttt{OC} \cup \texttt{I}}$. The main operators of this algebra are *OntoImage*, *OntoProject*, *OntoSelect* and *OntoOJoin*. Next subsections present the semantics of these operators for each part of the OBDB data model.

Ontology part. Signatures of the operators defined on the ontology part of the OBDB data model are restricted to $\texttt{E} \times 2^{\texttt{OC}} \rightarrow \texttt{E} \times 2^{\texttt{OC}}$. Since the data model of this part is similar to the OODB data model, the semantics of the operators of the *ENCORE* algebra is well adapted to the *OntoAlgebra* operators on this part. To illustrate the *OntoAlgebra* operators, we show how several queries are decomposed into calls to the operators of this algebra. These queries are expressed in OntoQL (a), RQL (b) and SPARQL (c). To simplify, the namespaces used in these examples are not explicitly specified.

Example 1. Retrieve the superclasses of the class named `Administrator`.

```
a. SELECT c.#superClasses FROM #class c WHERE c.#name = 'Administrator'
b. SELECT superClassOf($C) FROM $C WHERE $C = 'Administrator'
c. SELECT ?csup WHERE { ex:Administrator rdfs:subClassOf ?csup }
```

$\texttt{ext}^* : \texttt{E} \rightarrow 2^{\texttt{OC}}$ denotes the function returning the instances of an entity. Using this notation and the lambda notation, this query is expressed by applying the following *OntoAlgebra* operators:

$$\texttt{ClassAdministrator} := \texttt{OntoSelect}(\texttt{C}, \texttt{ext}^*(\texttt{C}), \lambda \texttt{c}_\bullet \texttt{c.name} =' \texttt{Administrator}')$$
$$\texttt{Result} := \texttt{OntoImage}(\texttt{ClassAdministrator}, \lambda \texttt{c}_\bullet \texttt{c.superClasses})$$
$$\texttt{ResultSPARQL} := \texttt{OntoFlatten}(\texttt{Result})$$

The *OntoSelect* operator is applied to find the class named `Administrator`. Thus, the type of `ClassAdministrator` is `SET`[`C`]. Then, the *OntoImage* operator applies the attribute `superClasses` to this class. The type of `Result` is `SET`[`SET`[`C`]]. Contrariwise to OntoQL and RQL, SPARQL doesn't support collections. Thus, we need to flatten the result using the *OntoFlatten* operator defined by:

$$\texttt{OntoFlatten}(\texttt{Set}[\texttt{T}], \texttt{O}_{\texttt{setT}}) = (\texttt{T}, \{\texttt{r} | \exists \texttt{t} \in \texttt{O}_{\texttt{setT}} \wedge \texttt{r} \in \texttt{t}\}) \ .$$

As a consequence, the type of `SPARQLResult` is `SET`[`C`].

Example 2. List the properties with their domain.

```
a. SELECT p, c FROM #propery as p, #class as c WHERE p.#scope = c.#oid
b. SELECT @P, $C FROM @P, $C WHERE domain(@P)=$C
c. SELECT ?p, ?c WHERE { ?p rdfs:domain ?c }
```

This query is expressed by applying the *OntoOJoin* operator:

$$\texttt{Result} := \texttt{OntoOJoin}(\texttt{P}, \texttt{ext}(\texttt{P}), \texttt{C}, \texttt{ext}(\texttt{C}), \texttt{p}, \ \texttt{c}, \ \lambda \texttt{p} \ \lambda \texttt{c}_\bullet \texttt{p}.\texttt{propDomain} = \texttt{c})$$

The type of `Result` is `SET`[`Tuple` $<$ (`p` : `P`), (`c` : `C`) $>$].

Content part. Signatures of the operators defined on the content part of the OBDB data model are restricted to $\texttt{C} \times 2^{\texttt{I}} \rightarrow \texttt{C} \times 2^{\texttt{I}}$. The data model of the content part presents some particularities which impose to redefine the *ENCORE* operators on this part.

OntoImage. Contrariwise to the OODB data model, one of the properties occurring in the function parameter may not be valued in the extensional definition of the class. Thus, we can not use the `Val` function to define the semantics of this operator as it is done in the definition of the *Image* operator. It becomes necessary to extend its domain to the properties defined on the intensional definition of a class but not valued on its extensional definition. This novelty of our algebra requires the introduction of the `UNKNOWN` value. We call `OntoVal` the extension of `Val`. It is defined by:

$$\texttt{OntoVal}(\texttt{i}, \texttt{p}) = \texttt{Val}(\texttt{i}, \texttt{p}), \text{ if } \exists \texttt{e} \in \texttt{TypeOf}(\texttt{i}) \text{ such that } \texttt{p} \in \texttt{SchemaProp}(\texttt{e})$$
$$\text{else, } \texttt{UNKNOWN} \ .$$

`UNKNOWN` is a special instance of `ObjectC` like `NULL` is a special value for SQL. Whereas `NULL` may have many different interpretations like value unknown, value inapplicable or value withheld, the only interpretation of `UNKNOWN` is value unknown, i.e., there is a value, but we don't know what it is. To preserve composition, `OntoVal` applied to a property whose value is `UNKNOWN` returns `UNKNOWN`. So, *OntoImage* is defined by:

$$\texttt{OntoImage}(\texttt{T}, \{\texttt{i}_1, \ldots, \texttt{i}_\texttt{n}\}, \texttt{f}) =$$
$$(\texttt{PropRange}(\texttt{f}), \{\texttt{OntoVal}(\texttt{i}_1, \texttt{f}), \ldots, \texttt{OntoVal}(\texttt{i}_\texttt{n}, \texttt{f})\}) \ .$$

Example 3. List the first names of users.

```
a. SELECT u.first_name FROM User u
b. SELECT fn FROM User{u}.first_name{fn}
c. SELECT ?fn WHERE { ?u rdf:type ex:User .
                      OPTIONAL { ?u ex:first_name ?fn } }
```

This query is expressed by applying the *OntoImage* operator:

$$\texttt{Result} := \texttt{OntoImage}(\texttt{User}, \texttt{ext}^*(\texttt{User}), \lambda \texttt{u}_\bullet \texttt{u.first\_name})$$

The type of `Result` is `SET[String]`. Since the property `first_name` is not valued for the class `Administrator`, this expression returns the value `UNKNOWN` for each administrator. In the SPARQL vocabulary, the variable is said *unbound*. This is not the case for the RQL query because this language doesn't allow to express optional patterns. As a result, this query doesn't return a value for administrators.

OntoProject. *Project* is also extended to *OntoProject* using the `OntoVal` operator previously defined :

$$\begin{aligned}\texttt{OntoProject}(\texttt{T}, \texttt{I}_\texttt{t}, \{(\texttt{A}_1, \texttt{f}_1), \ldots (\texttt{A}_\texttt{n}, \texttt{f}_\texttt{n})\}) = \\ (\texttt{Tuple}[< (\texttt{A}_1, \texttt{PropRange}(\texttt{f}_1)), \ldots, (\texttt{A}_\texttt{n}, \texttt{PropRange}(\texttt{f}_\texttt{n})) >], \\ \{< \texttt{A}_1 : \texttt{OntoVal}(\texttt{i}, \texttt{f}_1), \ldots, \texttt{A}_\texttt{n} : \texttt{OntoVal}(\texttt{i}, \texttt{f}_\texttt{n}) > | \texttt{i} \in \texttt{I}_\texttt{t}\}) \ .\end{aligned}$$

OntoSelect. The semantics of *OntoSelect* is similar to the one of *Select*:

$$\texttt{OntoSelect}(\texttt{T}, \texttt{I}_\texttt{t}, \texttt{pred}) = (\texttt{T}, \{\texttt{i} | \texttt{i} \in \texttt{I}_\texttt{t} \wedge \texttt{pred}(\texttt{i})\}) \ .$$

If the predicate taken as parameter of *OntoSelect* contains function applications, then `OntoVal` must be used. So, operations involving `UNKNOWN`, that may appear in a predicate, must be extended to handle this value. Because `UNKNOWN` is often interpreted by `NULL`, the same semantics as `NULL` is given to `UNKNOWN`. Thus, arithmetic operators like $\times$ or $+$ applied to `UNKNOWN` return `UNKNOWN`, and comparing `UNKNOWN` to any instance using a comparison operator like $=$ or $>$ returns `UNKNOWN`.

Example 4. List the posts created by an user whose email end with '@ensma.fr'.

```
a. SELECT p FROM Post p WHERE p.hasCreator.email LIKE '%@ensma.fr'
b. SELECT p FROM Post{p}.hasCreator.email{e} WHERE e LIKE '%@ensma.fr'
c. SELECT ?p WHERE { ?p rdf:type ex:Post . ?p ex:has_creator ?c .
                     ?c ex:email ?e . FILTER (?e LIKE '%@ensma.fr') }
```

This query is expressed by applying the *OntoSelect* operator:

$$\begin{aligned}\texttt{Result} := \texttt{OntoSelect}(\texttt{Post}, \texttt{ext}^*(\texttt{Post}), \\ \lambda \texttt{p}_\bullet \texttt{p.hasCreator.email LIKE } '\%@\texttt{ensma.fr}')\end{aligned}$$

The type of `Result` is `SET[Post]`. For each post created by an administrator, the value `UNKNOWN` is returned for the property `email`. As a consequence, only post created by users who are not administrators may be returned as result.

OntoOJoin. The semantics of *OntoOJoin* is similar to the one of *OJoin*:

$$\texttt{OntoOJoin}(\texttt{T}, \texttt{I}_\texttt{t}, \texttt{R}, \texttt{I}_\texttt{r}, \texttt{A}_1, \texttt{A}_2, \texttt{pred}) =$$
$$(\texttt{Tuple}[< (\texttt{A}_1, \texttt{T}), (\texttt{A}_2, \texttt{R}) >], \{< \texttt{A}_1 : \texttt{t}, \texttt{A}_2 : \texttt{r} > | \texttt{t} \in \texttt{I}_\texttt{t} \wedge \texttt{r} \in \texttt{I}_\texttt{r} \wedge \texttt{pred}(\texttt{t}, \texttt{r})\}) \ .$$

The predicate taken in parameter of *OntoOJoin* is treated as for *OntoSelect*.

Operator *. In the *ENCORE* algebra, a class `C` refers to instances of `C` and instances of all subclasses of `C`. The *ENCORE* algebra doesn't supply a built-in operator to write a non polymorphic query. Thus, we define the explicit polymorphic operator, named *, to distinguish between local queries on instances of a class `C` and instances of all the classes denoted $\texttt{C}^*$ subsumed by `C`. We denote $\texttt{ext} : \texttt{C} \rightarrow 2^\texttt{I}$ the function which returns the instances of a class and we overload the function $\texttt{ext}^*$ for the signature $\texttt{C} \rightarrow 2^\texttt{I}$ to denote the deep extent of a class. If `c` is a class and $\texttt{c}_1, \ldots \texttt{c}_\texttt{n}$ are the direct sub-classes of `c`, `ext` and $\texttt{ext}^*$ are derived recursively[2] in the following way on the OBDB data model:

$$\texttt{ext}(\texttt{c}) = \texttt{TypeOf}^{-1}(\texttt{Nomination}(\texttt{c})) \ .$$
$$\texttt{ext}^*(\texttt{c}) = \texttt{ext}(\texttt{c}) \cup \texttt{ext}^*(\texttt{c}_1) \cup \ldots \cup \texttt{ext}^*(\texttt{c}_\texttt{n}) \ .$$

The `ext` and $\texttt{ext}^*$ make it possible to define the * operator as $^* : \texttt{C} \rightarrow \texttt{C} \times 2^\texttt{I}$ where $^*(\texttt{T}) = (\texttt{T}, \texttt{ext}^*(\texttt{T}))$.

Support of the multi-instanciation paradigm. In the *ENCORE* algebra, the *Image* operator can only be applied with a property defined on the class taken in parameter. Because of the multi-instanciation paradigm, an instance of a class can provide a value for a property not defined on this class but on an another class the instance belongs to. As a consequence, this paradigm raises the need to extend the *OntoImage* operator. We denote *OntoImage'* the definition of *OntoImage* when this operator is applied to a property not defined on the class taken in parameter. When *OntoImage'* is applied to a class $\texttt{C}_1$ and a property `p` not defined on $\texttt{C}_1$ but defined on the class $\texttt{C}_2$, this operator is defined by:

$$\texttt{OntoImage}'(\texttt{C}_1, \texttt{I}_{\texttt{C}_1}, \texttt{p}) =$$
$$\texttt{OntoImage}(\texttt{OntoOJoin}(\texttt{C}_1, \texttt{I}_{\texttt{C}_1}, \texttt{C}_2, \texttt{ext}^*(\texttt{C}_2), \lambda \texttt{i}_{\texttt{c}_1} \ \lambda \texttt{i}_{\texttt{c}_2 \bullet} \texttt{i}_{\texttt{c}_1} = \texttt{i}_{\texttt{c}_2}), \texttt{i}_{\texttt{c}_2}.\texttt{p}) \ .$$

The other operators of *OntoAlgebra* are extended in the same way to handle the multi-instanciation paradigm.

Example 5. List the file size of the posts.

```
a. Not supported
b. SELECT f FROM Post{p}.file_size{f}
c. SELECT ?p WHERE { ?p rdf:type ex:Post . ?p ex:file_size ?c }
```

[2] To simplify notation, we extend all functions `f` by $\texttt{f}(\emptyset) = \emptyset$

Let's suppose that the property `file_size` is defined on a class `ExtResource`. This query is expressed by applying the *OntoImage'* operator:

$$\begin{aligned}\texttt{Result} :=& \texttt{OntoImage}'(\texttt{Post}, \texttt{ext}^*(\texttt{Post}), \texttt{file\_size})\\ :=& \texttt{OntoImage}(\texttt{OntoOJoin}(\texttt{Post}, \texttt{ext}^*(\texttt{Post}),\\ & \texttt{ExtResource}, \texttt{ext}^*(\texttt{ExtResource}), \texttt{p}, \texttt{e}, \lambda\texttt{p}\ \lambda\texttt{e}_\bullet\texttt{p} = \texttt{e}), \texttt{e.file\_size})\end{aligned}$$

Ontology and content parts. *OntoAlgebra* provides the capability to query simultaneously ontology and content parts.

Example 6. For each ontology class, whose name contains the word `Post`*, list the properties applicable (defined and inherited) on this class and the values of the instances of this class for these properties.*

```
a. SELECT p.#name, i.p
     FROM #class as C, C as i, unnest(C.#properties) as p
    WHERE C.#name like '%Post%'
b. SELECT @P, V FROM {i;$C}@P{V} WHERE $C like '%Post%'
c. applicable properties can not be expressed
```

This query is expressed by applying the following *OntoAlgebra* operators:

$$\begin{aligned}\texttt{ClassPost}:=& \texttt{OntoSelect}(\texttt{C}, \texttt{ext}(\texttt{C}),\ \lambda\texttt{c}_\bullet\texttt{c.name like }'\%\texttt{Post}\%')\\ \texttt{ClassInst}:=& \texttt{OntoOJoin}(\texttt{ClassPost}, *(\texttt{ObjectC}), \texttt{C}, \texttt{i}, \lambda\texttt{C}\ \lambda\texttt{i}_\bullet\texttt{i} \in \texttt{ext}^*(\texttt{C}))\\ \texttt{ClassPropInst}:=& \texttt{OntoProject}(\texttt{ClassInst}, \lambda\texttt{ci}_\bullet\\ & < (\texttt{C}, \texttt{ci.C}), (\texttt{i}, \texttt{ci.i}), (\texttt{p}, \texttt{ci.C.properties}) >\\ \texttt{UClassPropInst}:=& \texttt{OntoUnNest}(\texttt{ClassPropInst}, \texttt{p})\\ \texttt{Result}:=& \texttt{OntoProject}(\texttt{UClassPropInst}, \lambda\texttt{cip}_\bullet\\ & < (\texttt{n}, \texttt{cip.p.name}), (\texttt{v}, \texttt{cip.i.(cip.p)}) >\end{aligned}$$

The first selection finds the classes whose names contain the word `Post`. The result is `ClassPost` of type `Set`[`C`]. The *OntoOJoin* operator is then used to join the classes of `ClassPost` and all the instances, i.e the polymorphic instances of the root class ($*$(`ObjectC`)). The result of this operation is `ClassInst` of type `SET`[`Tuple` $<$ (`c` : `C`), (`i` : `ObjectC`) $>$]. The function `properties` is then applied to the classes contained in `ClassInst` using the *OntoProject* operator. The function `properties` returns the applicable properties of a class as a set. As a consequence, the result of this step is `ClassProp` of type `SET`[`Tuple` $<$ (`c` : `C`), (`i` : `ObjectC`), (`p`, `SET`[`P`]) $>$]. The next step consists in unnesting the set of properties contained in each of the tuple of `ClassProp`. This is achieved using the `OntoUnNest` operator defined by:

$$\begin{aligned}&\texttt{OntoUnNest}(\texttt{Tuple}[< (\texttt{A}_1, \texttt{T}_1), \ldots, (\texttt{A}_\texttt{i}, \texttt{SET}[\texttt{T}_\texttt{i}]), \ldots, (\texttt{A}_\texttt{n}, \texttt{T}_\texttt{n}) >], \texttt{I}_\texttt{t}, \texttt{A}_\texttt{i}) =\\ &\qquad (\texttt{Tuple}[< (\texttt{A}_1, \texttt{T}_1), \ldots, (\texttt{A}_\texttt{i}, \texttt{T}_\texttt{i}), \ldots, (\texttt{A}_\texttt{n}, \texttt{T}_\texttt{n}) >],\\ &\qquad \{< \texttt{A}_1 : \texttt{s.A}_1, \ldots, \texttt{A}_\texttt{i} : \texttt{t}, \ldots, \texttt{A}_\texttt{n} : \texttt{s.A}_\texttt{n} > | \texttt{s} \in \texttt{I}_\texttt{t} \wedge \texttt{t} \in \texttt{s.A}_\texttt{i}\})\ .\end{aligned}$$

The result of this operation is `UClassProp` of type `SET[Tuple` $<$ `(c : C), (i : ObjectC), (p, P)` $>$`]`. Finally the result is obtained by applying the *OntoProject* operator to retrieve, for each tuple, the name of the property referenced by the attribute name `p` and to apply this property to the instances referenced by the attribute name `i`. The final result is of type `SET[Tuple` $<$ `(name : String), (value : ObjectC)` $>$`]`.

3.3 Differences between OODB and OBDB Languages

In this section, we describe a list of the identified main differences between the query languages issued from the *ENCORE* and *OntoAlgebra* algebras.

Two levels language. An OBDB query language offers the capability to query ontology, data and both ontology and data. Each of these querying levels corresponds to a specific need. Querying ontology may be useful to discover concepts of an ontology. Querying data from the concepts of an ontology allows to query the data independently of the structure of the data (semantic querying). Querying both ontology and data is useful to extract a subset of an ontology together with its instances (from the ontology to the content part of an OBDB) or to discover how a given instance of an ontology class may be described by some other classes (from the content to the ontology part of an OBDB). In a number of OODB implementations, *metadata* are recorded in the system catalog. Using the object query language provided, one can query these *metadata*. However, object-oriented algebras define how to query the data of an OODB only. As a consequence, it is difficult to combine querying both *metadata* and data.

Unknown value. OBDB query languages may return a special value for properties defined on the intensional definition of a class but not used in its extensional definition (see section 3.2). Contrary to the `NULL` value, introduces in classical algebra, there is only one interpretation of this value: a value exists, but we don't know what it is. In *OntoAlgebra*, we have chosen to give the same semantics to this value as the one of the `NULL` value in order to remain compatible with classical database languages. As shown in [15], this is not the case of the SPARQL semantics which has introduced some mismatches with the processing of the `NULL` value in classical databases.

Path expression. OBDB query languages extend the capability of path expressions introduced by OODB query languages. Indeed, a path expression in an OBDB query language can be composed with a property not defined on the previous element of the path. This capability is introduced to handle the multi-instanciation paradigm. Moreover, a path expression can be composed with properties determined at runtime (generalized path expression). This capability is introduced to allow querying both ontologies and data.

Parametric language. OBDB query language may use environment variables such as the used natural language or the namespace of the ontology queried to restrict the search space in the OBDB and to allow users to define queries in different natural languages.

4 Related Work

To our knowledge, the SOQA Ontology Meta Model [16] is the only other proposition of an independent data model of a given ontology model. It incorporates constructors not supported by some ontology languages (e.g., methods or relationships) but it can not be extended. Our approach is dual, we have decided to incorporate only the shared constructors but to allow the extension of this core model thanks to the *metametamodel* level. This approach is much more flexible since it allows to represent all the constructors of a given ontology model. This capability is not offered by the SOQA Ontology Meta Model. For example, restrictions of OWL or documents of PLIB are not included in this model. As a consequence managing ontologies which use these constructors with SOQA Ontology Meta Model based tools is not possible without loss of data.

Concerning the query algebra, formal semantics defined for ontology query languages [15, 17] or more generally for an ontology model [18, 19] can be regarded as related work. Close to our work is the relational algebra for SPARQL presented in [15]. It presents a correspondence between SPARQL and relational algebra queries. Based on this analysis, author points out several differences between SPARQL and SQL semantics. With this work, we share the idea of defining ontology query languages starting from the well known algebra of classical database languages. However, we do not address the same kind of data. While the operators defined in its algebra regard RDF as triple data without schema or ontology information, our algebra proposes operators to exploit the ontology level (e.g. computation of the transitive closure of the subsumption relationship ...). Thus, while its algebra has the expressive power of the relational algebra, our algebra has the expressive power of object-oriented algebra to query the ontology, the data and both the ontology and the data of an OBDB.

5 Conclusion and Future Work

In this paper, we have formally defined a data model for OBDBs independent of the used ontology model and representation schema. Using this data model, we have discussed and shown the differences existing between classical databases and OBDBs. These formalization and comparison are a sound basis for engineers willing to implement ontology databases using classical databases.

As a second step, we have proposed a formal algebra of operators for querying OBDBs. We have built this algebra by extending the *ENCORE* algebra proposed for OODB. As a consequence, our algebra clarifies the differences between object-oriented query languages (e.g., SQL2003, OQL ...) and ontology query languages (e.g., RQL, OntoQL ...) in terms of semantics and expressive power.

For the future, we plan to use the proposed algebra to study optimization of OBDBs. By reusing the ENCORE algebra, we hope to benefit from most of the equivalence rules defined in this algebra. The main challenge is to find new equivalence rules deriving from the specific features of the OBDB data model.

References

1. ISO13584-42: Industrial automation systems and integration. Parts Library Part 42. Description methodology: Methodology for structuring parts families. Technical report, International Standards Organization (1998)
2. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. World Wide Web Consortium. (2004)
3. Dean, M., Schreiber, G.: OWL Web Ontology Language Reference. World Wide Web Consortium. (2004)
4. Harris, S., Gibbins, N.: 3store: Efficient bulk rdf storage. In: Proceedings of the First International Workshop on Practical and Scalable Semantic Systems (PPP'03). (2003)
5. 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
6. 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
7. Park, M.J., Lee, J.H., Lee, C.H., Lin, J., Serres, O., Chung, C.W.: An efficient and scalable management of ontology. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07). (2007)
8. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. World Wide Web Consortium. (2006)
9. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: Rql: a declarative query language for rdf. In: Proceedings of the Eleventh International World Wide Web Conference. (2002) 592–603
10. Jean, S., Aït-Ameur, Y., Pierra, G.: Querying ontology based database using ontoql (an ontology query language). In: Proceedings of On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated International Conferences (ODBASE'06). (2006) 704–721
11. Shaw, G.M., Zdonik, S.B.: A query algebra for object-oriented databases. In: Proceedings of the Sixth International Conference on Data Engineering (ICDE'90), IEEE Computer Society (1990) 154–162
12. Object Management Group: Meta Object Facility (MOF), formal/02-04-03. (2002)
13. Object Management Group: Ontology Definition Metamodel (ODM) Final Adopted Specification ptc/06-10-11. (2006)
14. 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
15. Cyganiak, R.: A relational algebra for sparql. Technical Report HPL-2005-170, HP-Labs (2005)
16. 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
17. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. In: Proceedings of the Fifth International Semantic Web Conference (ISWC'06). (2006)
18. Frasincar, F., Houben, G.J., Vdovjak, R., Barna, P.: Ral: An algebra for querying rdf. World Wide Web **7** (2004) 83–109
19. Gutierrez, C., Hurtado, C., Mendelzon, A.O.: Foundations of semantic web databases. In: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '04). (2004) 95–106