Soft Computing manuscript
The final publication is available at Springer via http://dx.doi.org/10.1007/s00500-015-1633-5

Ontologies in engineering: The OntoDB/OntoQL platform

Yamine Ait-Ameur · Mickaël Baron · Ladjel Bellatreche · Stéphane Jean · Eric Sardet

Abstract Ontologies have been increasingly used over the past few decades in a wide range of application domains spanning both academic and industrial communities. As ontologies are the cornerstone of the Semantic Web, the technologies developed in this context, including ontology languages, specialized databases and query languages, have become widely used. However, the expressiveness of the proposed ontology languages does not always cover the needs of specific domains. For instance, engineering is a domain for which the LIAS laboratory has proposed dedicated solutions with a worldwide recognition. The underlying assumptions made in the context of the Semantic Web, an open and distributed environment, do not apply to the controlled environments of our projects where the correctness and completeness of modeling can be guaranteed to a certain degree. As a consequence, we have developed over the last decades a specialized standard ontology language named PLIB associated with the OntoDB/OntoQL platform to manage ontological engineering data within a database. The goal of this paper is threefold: (i) to share our experience in manipulating ontologies in the engineering domain by describing their specificities and constraints, (ii) to define a comprehensive classification of ontologies with respect to three main research communities: Artificial Intelligence, Databases and Natural Language Processing and (iii) to present a persistent solution, called OntoDB, for managing extremely large semantic data sets associated with an ontological query language, called OntoQL. These objectives are illustrated by several examples that show the effectiveness and interest of our propositions in several industrial projects in different domains including vehicle manufacturing and CO2 storage.

Yamine Ait-Ameur
IRIT/ ENSHEEIHT, Toulouse, France
E-mail: yamine@enseeiht.fr

Mickal Baron, Ladjel Bellatreche, Stéphane Jean
LIAS/ISAE-ENSMA - University of Poitiers
1, Avenue Clement Ader, 86960 Futuroscope Cedex, France
E-mail: {baron,bellatreche,jean}@ensma.fr

Eric Sardet
CRITT Informatique
Futuroscope Cedex, France
E-mail: {sardet}@critt-informatique.fr

1 Introduction

The notion of ontology has initially been defined by Gruber as *an explicit specification of a conceptualization* (Gruber, 1993). An ontology is composed of a set of shared classes, properties and relationships with reasoning mechanisms. It has the ability to represent formally real-world knowledge in a shared way. These favorable characteristics of ontologies have led academicians and industrials coming from various communities to adopt them: Artificial Intelligence (Matuszek et al, 2006), Natural Language Processing (Estival et al, 2004), Databases/Data Warehouses (Sugumaran and Storey, 2006; Noy, 2004), Information Retrieval (Graupmann et al, 2005), etc. Some researchers have even pushed the idea of having a universal ontology[1].

The construction of ontologies is time consuming. As a consequence, several research efforts have been conducted to tackle this problem. These studies may be classified into three main categories: (i) *manual construction* as is the case of the Cyc Ontology (Matuszek et al, 2006), (ii) *community-based construction* such as

[1] Entity Relationship Conference 2011 Panel on New Directions for Conceptual Modeling.

Freebase[2] where volunteers are invited to contribute to the ontology definition and (iii) *automatic construction* in which the ontology is automatically extracted from the Web such as YAGO (Suchanek et al, 2008) and DBpedia (Mendes et al, 2012) or from relational databases (Sequeda et al, 2011). These ontologies can be defined for specific domains, including biology (Apweiler et al, 2004) and engineering (IEC61360-4, 1999), but can also be defined for general-purpose applications. The SUMO ontology (Niles and Pease, 2001) and the OWL-FC ontology (Maio et al, 2012) are examples of such upper-level ontologies.

Despite this wide and diverse usage of ontologies, most of them are defined with the same technologies defined in the context of the Semantic Web. These technologies include the RDF-Schema (Brickley and Guha, 2004) and OWL (Bechhofer et al, 2004) languages to define ontologies, specialized databases such as Jena (Wilkinson, 2006) to manage a large amount of Semantic Web data and the SPARQL language (Prud'hommeaux and Seaborne, 2008) to query them. Admittedly, these technologies are well adapted in the context of the Semantic Web, but may not cover the majority of domains and applications. This observation is based on our experience in the engineering domain for data integration applications. In this particular setting, we have identified two major drawbacks of Semantic Web technologies concerning the expressiveness of Semantic Web languages and their underlying assumptions. Indeed, data integration projects in the engineering domain require a precise definition of technical information characterizing components (e.g., units of measurement) and an explicit representation of the modeling context of each defined concept (e.g., the point of view taken to characterize a component). Moreover, as the Semantic Web is an open and distributed environment, the corresponding ontology languages adhere to the open world assumption, which means that any statement that is not known can be true, and do not make the unique name assumption, i.e., that if two objects have different identifiers, they are different. As pointed out in (de Bruijn et al, 2005), these assumptions can be misleading for engineers who are used to define constraints for checking the validity of the defined components and not to infer additional knowledge. If these assumptions are natural in the Semantic Web, they are less plausible in controlled environments where standard ontologies exist and where engineers adhere to a protocol which ensures a certain degree of correctness and completeness of modeling.

As existing ontology technologies were not adapted to our industrial projects in engineering, we have designed over the past two decades a set of technologies to represent and manage ontologies in the engineering domains. These technologies are based on the standard PLIB language (officially ISO 13584) to define ontologies (Pierra, 2003). The design of this language was initiated in the early 90's to develop an approach with standard models for the automatic exchange and integration of engineering component databases. This ontology language is equipped with a specialized database called OntoDB to store engineering data and their associated ontologies (Dehainsala et al, 2007). As the SQL language was not sufficient to query such a database, the OntoQL exploitation language was designed (Jean et al, 2006). This language follows the design logic of SQL by proposing sub-languages to define, manipulate and query both the data and the ontologies that describe them. The goal of this paper is to present this set of technologies and to describe several projects with large companies in various domains such as the automotive and petroleum industries where these technologies have been successfully put into practice on real-case settings.

This paper is structured as follows. Section 2 presents our point of view on the notion of ontology. As this notion has been used by different communities, all ontologies are not alike. Consequently, we propose a taxonomy of ontologies and present the onion model we use to design ontologies. In Section 3 we present the particularities of the engineering domain that have led us to design the PLIB language. This language is presented in Section 4 after showing the limitations of Semantic Web languages to fulfil the requirements previously identified for the engineering domain. As a large quantity of data can be described by ontologies, Section 5 reviews the persistent solutions that have been developed for storing ontologies and Section 6 details the OntoDB database that we have designed. This database is equipped with the OntoQL exploitation language described in Section 7. These technologies have been used in many industrial projects. Section 8 presents several use cases. Finally, Section 9 concludes and introduces future work.

2 A layered view of an ontology

From our point of view, an ontology is a *formal and consensual dictionary of categories and properties of entities of a domain and the relationships that hold among them* (Jean et al, 2007b). This definition highlights three main characteristics of an ontology.

1. **Formal.** An ontology describes a domain by defining a set of classes and properties using a formal lan-

[2] `http://www.freebase.com`

guage. This *ontology language* can be used to check the consistency of an ontology and to perform automatic reasoning on the ontology-defined concepts and instances.
2. **Consensual.** An ontology is agreed and shared by a community. Thus, contrary to a conceptual model, an ontology is not designed for a particular application. It describes the knowledge of a domain to fulfil the needs of the members of a community.
3. **Universal identification.** Each concept of an ontology has a universal identifier. Using this identifier, an ontology concept and the semantics it represents can be referenced from any environment.

All ontologies are not alike (Cullot et al, 2003; Pierra, 2003). On the one hand, *Linguistic Ontologies (LO)* represent the meaning of the words used in a particular universe of discourse, in a particular language. On the other hand, *Conceptual Ontologies (CO)* represent the categories and properties of objects available in some part of the world. The two following types of concepts exist in a CO (Gruber, 1993).

– *Primitive concepts* are those concepts "for which we are not able to give a complete axiomatic definition"; we must rely on a textual documentation and on a background of knowledge shared with the reader.
– *Defined concepts* are those concepts "for which the ontology provides a complete axiomatic definition by means of necessary and sufficient conditions expressed in terms of other concepts".

We call *Canonical Conceptual Ontologies (CCO)*, the ontologies that only include primitive concepts. They define a canonical vocabulary in which each information in the target domain is captured in a unique way. And we call *Non Canonical Conceptual Ontologies (NCCO)* the ontologies that also include defined concepts. They introduce new reasoning capabilities and equivalence operators that can be used to define mappings between different ontologies.

The three categories of ontologies introduced previously suggest a layered view of ontologies (see Figure 1), we called the *onion model* of domain ontologies (Jean et al, 2007b). In this view, a kernel CCO provides a formal foundation to represent and exchange efficiently the knowledge of a domain. A NCCO layer extends the canonical vocabulary with concepts equivalence to encompass all concepts broadly used in the domain. The NCCO concepts can be defined with different operators such as *class expressions* defined in description logic. Finally, a LO layer adds the natural language representation of the CCO and NCCO concepts for person-system and person-person communication.

The onion model is illustrated in Figure 2 with a toy ontology in the mechanics domain. The CCO part of this ontology is composed of the *Product*, *Rolling_Bearing*, *Roller_Bearing* and *Row_of_Balls* classes and properties such as *mass* or *width*. Based on this CCO, two NCCO classes and one NCCO property are defined. For instance, the *2_Rows_Ball_Bearing* class is defined as the ball bearings that use two rows of balls. The formal definition of this class is given using a syntax borrowed from description logic. The *life_length* NCCO property is defined as a function of different properties. Finally, the LO part of this ontology consists of the linguistic description of these classes and properties. For instance, the *Rolling_Bearing* class is described with two English names and a French name. In the next section, we detail the specific requirements for representing ontologies in the engineering domain.

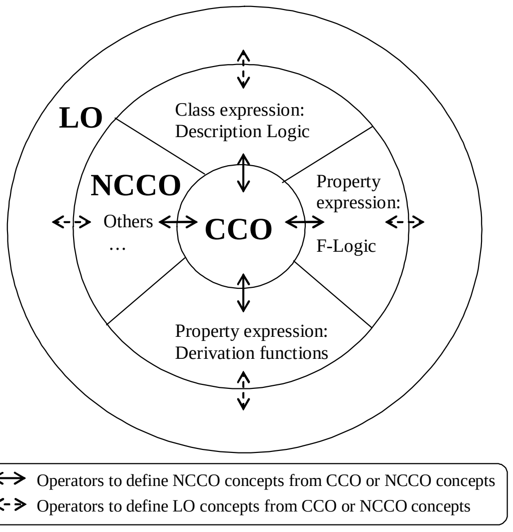


Fig. 1 The onion model of domain ontology

3 Ontologies in engineering

The engineering domain includes different fields such as aeronautics, transport, mechanics or energy. In most of these engineering fields, products to be designed are essentially assemblies of components. An important part of the engineering knowledge concerns the used components. It includes the criteria used to select a component, the conditions of component usage, the behaviour of components and the pertinent component representation for each specific discipline (Paasiala et al, 1993).

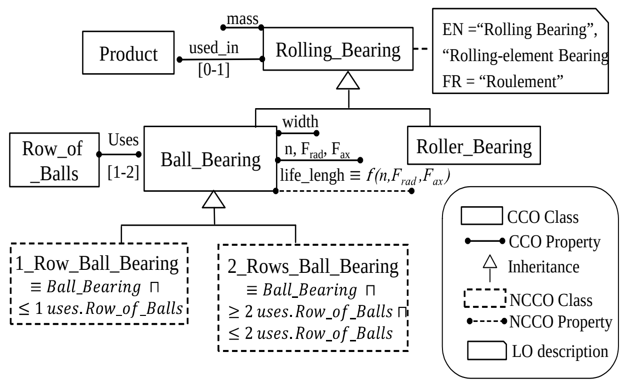

Product
used_in
[0-1]
mass
Rolling_Bearing
EN ="Rolling Bearing", "Rolling-element Bearing" FR = "Roulement"
width
Row_of _Balls
Uses
[1-2]
Ball_Bearing
n, F_{rad}, F_{ax}
life_lengh ≡ $f(n,F_{rad},F_{ax})$
Roller_Bearing
1_Row_Ball_Bearing ≡ Ball_Bearing ⊓ ≤ 1 uses.Row_of_Balls
2_Rows_Ball_Bearing ≡ Ball_Bearing ⊓ ≥ 2 uses.Row_of_Balls ⊓ ≤ 2 uses.Row_of_Balls
CCO Class
CCO Property
Inheritance
NCCO Class
NCCO Property
LO description

Fig. 2 Example of the onion model

The example presented in the next section illustrates the specificities of this domain.

3.1 Motivating example

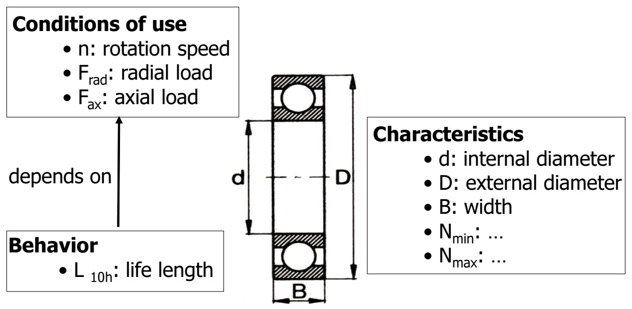


Fig. 3 Nature of properties of a component

A *rolling bearing* (or *rolling-element bearing*) is a mechanical component used to connect and to transmit load between two cylindrical shapes having the same axis but different diameters and rotational movements. The main properties of a rolling bearing are presented in Figure 3. Characteristic properties include *width*, *internal* and *external diameters*. These properties can have different values depending on the units of measure used. This scale should be explicitly represented.

Requirement 1 (*Units of measure*)

An ontology in engineering should explicitly define the units of measure of values and more generally it should describe all the technical information of a component.

The behaviour of a component is also characterized by properties. For example, the property *life length* corresponds to the length of the time period during which the bearing will behave correctly. The value of this property depends on the number of revolutions done by the bearing, and of the load it must support. Mathematically, the bearing *life-length* is a function of the *velocity* (i.e., rotational speed), the *radical load* and the *axial load.*

Requirement 2 (*Context of values*)

When a value of a property depends on a context of evaluation, this evaluation context should be explicitly defined.

Figure 3 presents a *2D representation* of a ball bearing. But other points of view of this component are used in engineering such as *3D representation* or *schematic representation* (see Figure 4). Each point of view describes a ball bearing with different characteristics.

Requirement 3 (*Point of view*)

If several perspectives are needed to describe a component, these perspectives should be explicitly defined in an ontology and the same real-world component should be described according to these different perspectives.

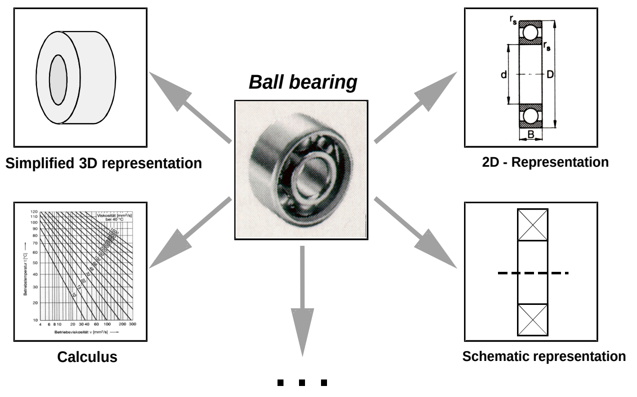


Fig. 4 Different points of view on the same component

From our experience acquired from designing ontologies in engineering (described in Section 8), it is not difficult to reach a consensus among several components suppliers on all the major properties of a rolling bearing. But it is impossible to reach a consensus on the properties that should be represented in the database of each supplier. Thus, a class should describe all its major properties. Concerning the properties, as different types of rolling bearings exist such as *ball bearings* or *roller bearings*, the diameter property should be defined at the level of circular bearing where it is meaningful.

Requirement 4 (*Context of concepts*)

The modeling context in which each class or property is defined should be made explicit and minimized. Each class should define all its major properties, at least in some broad context of a community. Each property should be defined in the context of a class: its domain of application.

Suppliers of components often need to extend a shared ontology with classes and properties that are not consensual.

Requirement 5 (*Locality of interpretation*)

Importation of resources from one ontology into another one should be possible while controlling the impact of the former over the interpretation of the latter.

In addition to these five main ontology modeling requirements, the engineering domain often relies on a set of assumptions described in the next section.

3.2 Underlying Assumptions

Open and Close World Assumptions. Under the *Close World Assumption (CWA)*, any statement that is not known to be true is false. On the contrary, under the *Open World Assumption (OWA)*, any statement that is not known can be true. These assumptions play an important role for constraints. Under the CWA, the constraints are checked; under the OWA they are used for inference. In the engineering domain, a lot of constraints are defined to describe precisely the behaviour of a component. Engineers that define these constraints expect that they will be used to check the validity of the registered components. Moreover, the CWA is adapted in the engineering context where a number of reference ontologies are already standardized or are in the standardization process either within IEC or within ISO (Pierra, 2003). These standards are rather stable and the knowledge can be considered complete. In this controlled domain, it is often possible to guarantee a certain degree of correctness of modeling. On the contrary, the OWA is often made in the Semantic Web as correct and complete modeling cannot be guaranteed on the Web. Moreover, an ontology can be intentionally under-specified expecting that others will reuse and extend it. Thus the OWA is adapted in an open context like the Web. Several studies have reported the need to use the CWA in controlled sections of the Semantic Web (de Bruijn et al, 2005; Knorr et al, 2011).

Unique Name Assumption. Under the *Unique Name Assumption (UNA)*, if two objects have different identifiers, they are different. This assumption is often made in combination with the CWA in complete and controlled domain. Contrariwise, the UNA is often not made in domains where the OWA is assumed. Indeed, without the UNA assumption, if two instances (or classes, or properties) have different identifiers, we may still derive by inference that they must be the same. From our experience, the UNA is more adapted to many engineering problems that use ontologies as solutions (Pierra, 2003). On the Web, the UNA is often considered less plausible.

Typing Assumption. Under the OWA, classification of instances is one of the key reasoning capability. For example, an instance having a value for a property p can be classified in the domain of p. Thus, a *Weak Typing Assumption (WTA)* is often made in the Semantic Web: (1) an instance may belong to any number of non connected ontology classes, (2) a property can be defined without a domain and (3) an instance can define a value for any property of the ontology. Under the WTA, a diverse set of data, ranging from structured data to unstructured data, can all be represented (Duan et al, 2011). But the cost of this flexibility is that the efficient management of such data is difficult (Aluç et al, 2014).

Instances in engineering domains are often components which are fairly structured data (relational-like). In such domain, a *Strong Typing Assumption (STA)* can be made: (1) each instance belongs to exactly one class called its basis class that is the minimum for the subsumption order of all the belonging classes of this instance, (2) each property is defined in the context of a class that defines its domain of application, and is associated with a range, and (3) only properties that are applicable in the context a class are used for describing its instances. This assumption can be used to define an efficient storage layout of engineering data as we will see in Section 6.

We have seen in this section that the engineering domain has specific requirements and assumptions. This observation has led us to define and use a specific ontology language.

4 Ontology languages

Several ontology languages have been defined in the last decade. The most well-known ontology languages are RDF-Schema (Brickley and Guha, 2004) and OWL (Bechhofer et al, 2004). These languages have been defined in the context of the Semantic Web and thus they make the OWA assumption and do not make the UNA assumption. We first present these two languages. Then we present the OWL-Flight language, which was defined mainly to eliminate some of the pitfalls in conceptual modeling with OWL induced by these assumptions. As these languages do not fulfil the five requirements introduced in the previous section, we finally present the PLIB ontology language specifically defined for the engineering domain.

4.1 RDF-Schema

RDF-Schema or *RDFS* is based on RDF. RDF is used to defined assertions as a set of triples *(subject, predicate, object)*. The subject is a URI that denotes a *resource*. The predicate is a property that characterizes the resource. The object is the value of the property, a literal value or a URI of another resource.

RDF defines few built-in predicates, mainly the typing relationship (*rdf:type*) and the property constructor (*rdf:Property*). Thus, it is not considered as an ontology language. RDFS extends RDF with a set of built-in predicates to define an ontology. The main predicates of RDFS are the following.

- **Class definition.** *rdfs:Class* and *rdfs:subClassOf* define a hierarchy of classes. Names and descriptions can be assigned to classes using *rdfs:label* and *rdfs:comment*.
- **Property definition.** *rdfs:domain* and *rdfs:range* define the domain and range of an RDF property. A hierarchy of properties can be defined using *rdfs:subPropertyOf*. As classes, properties can be described using *rdfs:label* and *rdfs:comment*.
- **Datatype definition.** *rdfs:Datatype* defines datatypes of an RDF property. The value of a property can be an instance of a class or a literal (*rdfs:Literal*). A literal can be typed using a set of allowed XML datatypes.
- **Instance definition.** Instances are defined in RDF with two kind of triples: *(i, rdf:type, C)* states that i is an instance of C and *(i, p, v)* defines v as the value of i for the p property.

RDFS defines few restrictions on the usage of these constructors. In particular, it does not separate the different levels of abstraction: instances, ontologies and ontology language. As a consequence, an instance can be a class and thus it can have instances. Likewise, a property can be a class and thus it can be the domain of another property.

RDFS does not provide non canonical constructs and thus it is oriented toward the definition of CCO ontologies. But as we have seen, it does not have specific constructors to fulfil the modeling requirements of the engineering domain identified in the previous section.

4.2 OWL

OWL extends the expressive power of RDFS. Three versions of this language have been defined : OWL Lite, OWL DL and OWL Full with an increasing expressive power (OWL Lite $\subset$ OWL DL $\subset$ OWL Full). OWL Lite and OWL DL ensure the decidability of reasoning but they are not compatible with RDFS since they restrict the usage of RDFS constructors. In contrast, OWL Full is compatible with RDFS but is not decidable. OWL DL extends RDFS with the following main constructors.

- **Ontology definition.** *owl:Ontology* defines an ontology as a set of classes and properties in a namespace. This ontology could be described by many characteristics such as its version (*owl:versionInfo*) or its compatibility with other ontologies.
- **Class definition.** *owl:Class* is the OWL class constructor corresponding to *rdfs:Class*. The *owl:equivalentClass* class axiom states that two classes have the same instances. On the contrary, two classes that do not share any instances could be defined as disjoints (*owl:disjointWith*). In addition to the usual class constructor, OWL includes NCCO class constructors. The *owl:unionOf*, *owl:intersectionOf*, *owl:complementOf* boolean operators define respectively a class as a union, intersection or complement of other classes. The *owl:allValuesFrom* (respectively, *owl:someValuesFrom*) restriction defines a class as all instances for which all values (respectively, at least one value) of a given property are members of a given class. *owl:hasValue* defines a class as all instances that have a given value for a given property. Finally, *owl:oneOf* defines a class by enumerating its instances.
- **Property definition.** Two types of properties exist: *owl:ObjectProperty* and *owl:DatatypeProperty*. They specialize the *rdf:Property* to distinguish properties whose range is a class from properties whose range is a datatype. The *owl:equivalentProperty* property axiom states that two properties have the same values. OWL introduces property characteristics. A property can be defined as symmetric (*owl:SymmetricProperty*), transitive (*owl:TransitiveProperty*), injective (*owl:InverseFunctionalProperty*), as a function (*owl:FunctionalProperty*) or as the inverse of an other property (*owl:inverseOf*). Finally, cardinality of a property can be specified using *owl:minCardinality*, *owl:cardinality* and *owl:maxCardinality*.
- **Instance definition.** An instance is defined with a set of RDF triples. The *owl:sameAs* (resp. *owl:differentFrom*) can be used to assert the equality (resp. inequality) of two instances.

Contrary to OWL Full which is fully compatible with RDFS, OWL DL imposes constraints on RDFS constructors to ensure decidability of reasoning (e.g., a constructor cannot be applied to another constructor). The last version of OWL, OWL Lite, imposes more restrictions than OWL DL to simplify the reasoning process. It forbids the usage of *owl:unionOf*, *owl:complementOf*, *owl:oneOf* and *owl:hasValue*. Moreover, *owl:minCardinality*, *owl:cardinality* and *owl:maxCardinality* could only be used with the 0 or 1 values.

Thus, OWL extends RDFS with mainly non canonical constructors to enhance the reasoning possibilities. But it does not introduce constructors to fulfil our requirements and it does not make the assumptions often made in the engineering domain.

4.3 OWL Flight

OWL Flight (de Bruijn et al, 2005) was designed to overcome some pitfalls in the OWL language for modeling specific domains. These pitfalls are mainly the result of the assumptions made in OWL: the OWA without the UNA (see Section 3). As a consequence of these assumptions, the semantics of OWL may seem counterintuitive for database and software engineers who are used to the CWA and UNA assumptions. For example, if a *2_Rows_Ball_Bearing* has more than two *Row_Of_Balls*, instead of raising an error, this knowledge can be used to infer an equality between the *Row_Of_Balls*.

As a consequence, OWL Flight is a variant of OWL based on Logic Programming that extends it with cardinality and value constraints, meta-modeling and powerful datatype support. Compared to OWL DL the constructors not included are enumerated classes, individual (in)equality assertions, complements and property restrictions in complete class definitions. Contrary to OWL DL, OWL Flight adopts the UNA and constraints are interpreted under the CWA i.e, they are used to check the data instead of being used to infer additional knowledge.

The PLIB ontology language presented in the next section was designed in the same spirit as OWL Flight. This language adopts the CWA and UNA which are more intuitive for engineers who work in domains such as mechanics or aeronautics. Moreover, this language adds important features (e.g., units of measure or points of view) which are missing in OWL for modeling precisely such domains.

4.4 PLIB

The PLIB ontology language (Pierra, 2003) is an international standard (ISO 13584) originally defined for exchanging and integrating automatically electronic catalogues of industrial components. Primitive concepts of a technical domain being rather broad and complex, the aim of this ontology model is to define precisely such concepts. PLIB can be used to define ontologies using the main following constructors.

- **Class definition.** PLIB classes are defined using the *item_class* constructor. Classes as other ontology elements are identified by a universal identifier named *BSU* (Basic Semantic Unit). They are described by a textual description (name, synonymous names, definition, note, remark) that may be given in different natural languages. This description can be completed with documents. PLIB classes can be organized into a hierarchy using two operators: *is_a* and *case_of*. *is_a* is the usual simple inheritance operator; *case_of* is a multiple subsumption relationship that does not imply inheritance of properties: the subsuming class should explicitly import the useful properties. This last operator is particularly useful for semantic integration (requirement 5). It could also be used to introduce multiple inheritance relationships in a PLIB ontology. In addition to *definition classes*, PLIB has two other types of classes: *representation and view classes. Representation classes* represent the additional properties that result from a particular point of view (requirement 3). A representation class must be associated with a definition class. Each instance of a representation class is a view of an instance of a definition class. This relationship is called *is-view-of. View classes* represent the point of view corresponding to each representation class: each representation class shall reference a view class as its modeling context.
- **Property definition.** The main property constructor of PLIB is *non_dependent_pdet*. As classes, properties are identified by a BSU and characterized by textual and/or graphical descriptions. Each PLIB property must be associated to a class that represents its domain with the *scope* constructor (requirement 4). PLIB has another property constructor named *dependent_P_DET* for defining properties that depend upon a context parameter defined by the *condition_DET* constructor (requirement 2). For example, the length of an axis depends upon its temperature. This context can also be defined as a function.

– **Datatype definition.** Primitive datatypes such are integer or string are available in PLIB. They can be associated with a unit of measure or a currency (requirement 1). Technical datatypes are available such as *level_type* used to qualify a numeric value. A class can also be used as a datatype with the *class_instance_type* constructor. Finally, PLIB supports collections and enumerated datatype.
– **Instance definition.** A PLIB instance is defined by a basis class and a set of property values. Thus, PLIB does not support multi-instanciation (i.e., that an instance belongs to different classes not linked by subsumption relationships). Instead, PLIB supports the instance aggregate mechanism: an instance may also be associated with any number of discipline-specific classes that represent points of view of a particular basis class.

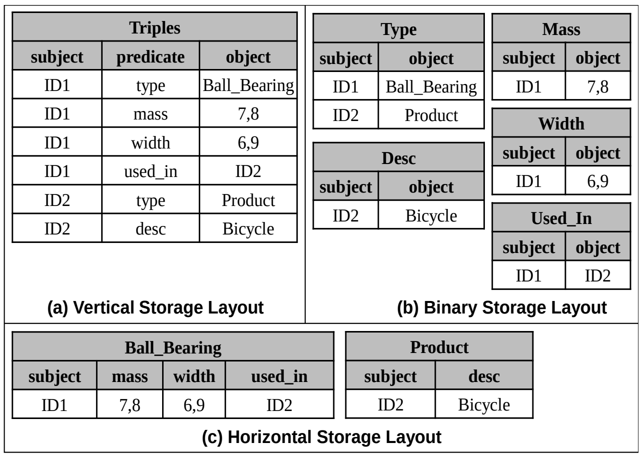
Table 1 summarizes our comparison of the studied ontology languages. As this table illustrates, RDFS focuses on defining CCOs with the assumptions of the Semantic Web domain. OWL extends this language with NCCO constructors borrowed from Description Logics. OWL-Flight is a variant of OWL that interprets constraints under the CWA. Finally, the PLIB ontology language focuses on defining precisely ontologies in the engineering domain by providing specific constructors for this domain. As we will see in Section 8, PLIB has been used to define several standard ontologies in this domain. With the development of these ontologies, a large quantity of data described by ontologies has been produced. In the next section, we describe the persistent solutions that have been developed in the last decade to manage such a large volume of ontological data.

5 Persistent solutions for ontologies

With the availability of standard ontology languages, several large ontologies have been designed. One can cite YAGO (Suchanek et al, 2008) and DBPedia (Mendes et al, 2012) in the Semantic Web and standard ontologies for Electronic Components (IEC 61360-4) or Laboratory Measuring Instruments (ISO 13584-501) in the engineering domain. As a consequence, the need to manage efficiently ontologies and their instances has appeared. Several research efforts have been proposed in the last decade to address this challenge. Two main approaches have been followed. The first one consists in natively representing the graph structure of RDF data (e.g., gStore (Zou et al, 2011)). The second approach, on which we focus in our work, uses relational database management systems (RDBMSs) to store RDF data. We call Ontology-Based Databases (*OBDBs*) these specialized databases. Examples of such systems are Oracle Spatial and Graph (Das et al, 2004), OntoDB (Dehainsala et al, 2007), Jena SDB (Wilkinson, 2006), Sesame (Broekstra et al, 2002), DLDB (Pan and Heflin, 2003), 3store (Harris and Gibbins, 2003), RStar (Ma et al, 2004), SW-Store (Abadi et al, 2007) or OntoMS (Park et al, 2007). Systems such as RDF-3X (Neumann and Weikum, 2008) or TripleBit (Yuan et al, 2013) can be considered hybrid approaches as they use native storage structures which are not graphs. Despite being studied for more than a decade, the efficient management of ontologies and their instances is still an active research topic (Aluç et al, 2014). In this section, we present the diversity of OBDBs using two orthogonal criteria: the storage layout and architecture used by OBDBs.

5.1 Storage layouts

Three main storage layouts are used for representing ontologies and/or instances in an RDBMS (Sakr and Al-Naymat, 2009): the *vertical*, *binary* and *horizontal storage layouts.* These storage layouts are detailed below and illustrated on a subset of our example of ontology (Figure 5).


Triples

subject	predicate	object
ID1	type	Ball_Bearing
ID1	mass	7,8
ID1	width	6,9
ID1	used_in	ID2
ID2	type	Product
ID2	desc	Bicycle

(a) Vertical Storage Layout

Type

subject	object
ID1	Ball_Bearing
ID2	Product

Desc

subject	object
ID2	Bicycle

Mass

subject	object
ID1	7,8

Width

subject	object
ID1	6,9

Used_In

subject	object
ID1	ID2

(b) Binary Storage Layout

Ball_Bearing

subject	mass	width	used_in
ID1	7,8	6,9	ID2

Product

subject	desc
ID2	Bicycle

(c) Horizontal Storage Layout

Fig. 5 Example of the used storage layouts

The *vertical storage layout.* This storage layout is a direct translation of the RDF data model. It consists of a single triples table with three columns *(subject, predicate, object)* (Harris and Gibbins, 2003). Since URIs are long strings, additional tables may be used to store only integer identifiers in the triples table. Three B+tree indexes are usually used (Abadi et al, 2007; Sidirourgos et al, 2008): one clustered on *(subject, property, object)* and two unclustered on *(property, object, subject)* and *(object, subject, property).* As this storage layout is a

	RDFS	**OWL**	**OWL Flight**	**PLIB**
Universal identifier	URI	URI	URI	BSU
Subsumption	subClassOf SubPropertyOf	subClassOf SubPropertyOf	subClassOf SubPropertyOf	is_a case_of
NCCO constructors	- -	Restriction Boolean operators	Restriction Boolean operators	Derivation function
Linguistic information	label/comment (multilingual)	label/comment (multilingual)	label/comment (multilingual)	name/synonym/definition note/remark (multilingual)
Technical information	- - -	- - -	- - -	specific datatypes (units, level_type) point of view (is-view-of) evaluation context
Assumptions	OWA WTA	OWA WTA	Local CWA/UNA WTA	CWA/UNA STA

Table 1 Comparison of the main ontology languages

direct translation of the RDF data model, RDF queries can be directly and easily translated into SQL queries. But these queries often require a lot of self-join operations on the triples table. RDF3X (Neumann and Weikum, 2008) and Hexastore (Weiss et al, 2008) have shown that this approach can still be employed with good performance if the triples table is replaced by a set of indexes.

The *binary storage layout.* This storage layout consists in decomposing the triples table into a set of 2-columns tables *(subject, object)*, one for each predicate (Abadi et al, 2007). In some implementations, the inheritance of classes and class membership are represented in a different way (e.g., by using the table inheritance mechanism of PostgreSQL (Broekstra et al, 2002)). Two indexes are usually used: a clustered B+ tree index on *subject* to locate them quickly and an unclustered B+ tree index on *object.* Abadi et al. have shown that this approach can be very efficient if it is implemented on a column-store DBMS (Abadi et al, 2007). By using different benchmarks, the performance of this storage layout has been analysed in several studies (Schmidt et al, 2009; Sidirourgos et al, 2008; Dehainsala et al, 2007). Two drawbacks have been identified: (1) when queries involve many properties, this storage layout requires many joins (Dehainsala et al, 2007) and (2) when properties do not appear as bound variables in queries, this storage layout requires a lot of union clauses and joins (Sidirourgos et al, 2008; Weiss et al, 2008).

The *horizontal storage layout.* This storage layout consists in denormalizing the triples table by storing them in a representation more similar to traditional relational schemas (Wilkinson, 2006; Dehainsala et al, 2007). This relational representation can be obtained either by grouping the properties that are defined together (Wilkinson, 2006) or by making some typing assumptions (Dehainsala et al, 2007). In the latter case, a table $C(p_1, \ldots, p_n)$ is created for each class C where $p_1, \ldots, p_n$ are the set of single-valued properties used by at least one instance of the class. Multivalued properties are represented by two-column tables as in the binary representation or by using the array datatype available in relational-object DBMSs. Since all instances do not necessarily have a value for all properties of the table, this representation can be sparse which can impose performance overhead. Moreover, as the binary storage layout, this storage layout does not scale well when properties do not appear as bound variables in queries.

As we can see and as several benchmarking studies have shown (Schmidt et al, 2009), each storage layout is suitable for a different type of queries and there is not a storage layout that gives the best performance for all query workloads. As a consequence, Aluç et al. have recently raised the challenge to define a workload-aware storage layout for managing ontologies and their instances (Aluç et al, 2014).

5.2 Architecture

Three main architectures called *type I*, *type II* and *type III architectures* are used by OBDBs. The type I and II architectures are illustrated in Figure 6 and an example of type III architecture is presented in Figure 7:

– *type I architecture*: as in traditional databases, this architecture uses two parts: the *data part* and the *system catalog.* The data part stores the ontology instances and the ontology schema (classes, properties, etc.). Thus, this architecture implies that the same storage layout is used for ontologies and their instances. Oracle Graph and 3store use this architecture and store ontologies and instances in a triples table (with specific indexes for Oracle Graph);

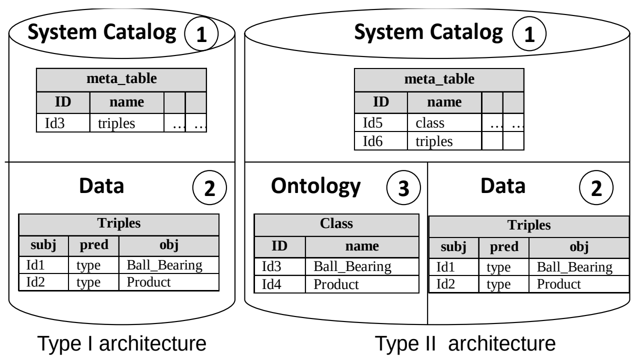


Fig. 6 Type I and type II architectures of OBDBs

– *type II architecture*: in this architecture, ontologies and their instances are stored in two different schemas. Thus, this architecture decomposes the database in three parts: the *system catalog*, the *ontology part* and the *data part*. RStar (Ma et al, 2004) or Sesame DB (Broekstra et al, 2002) are examples of OBDBs that use this architecture. In this architecture the used ontology language is hard encoded by the schema defined in the ontology part. Thus, this architecture is not adapted if the used ontology language changes regularly or if constructors from other ontology languages need to be used;
– *type III architecture*: this architecture has been proposed in OntoDB (Dehainsala et al, 2007) to be able to extend the used ontology language in the OBDB and to support different ontology languages in the same database. This need is fulfilled by proposing an architecture similar to the MOF architecture (OMG, 2002) defined for managing different metamodels. Thus this architecture is composed of 4 parts: the *system catalog*, the *ontology part*, the *data part* and the *metaschema part*. This latter part plays the same role for ontologies than the one played by the system catalog for the data. This part is used to store the used ontology language so that it can be extended. More details are given in the next section.

6 The OntoDB ontology-based database

The OntoDB architecture has been initially designed for storing PLIB ontologies and their instances. As the PLIB ontology language regularly evolves and because several other ontology languages exist, the first objective of this architecture was to support evolutions of the used ontology language. As we have seen in Section 3, instances in engineering domains are fairly structured and respect the strong typing assumption. Thus, the second objective of this architecture was to design an efficient storage layout for such data. As we have seen in the previous section, other solutions to store ontologies do not fulfil these objectives. As a consequence, we have designed the OntoDB architecture presented in the next section.

6.1 The OntoDB architecture

OntoDB is implemented on top of the PostgreSQL DBMS. It has a four-parts architecture depicted in Figure 7.

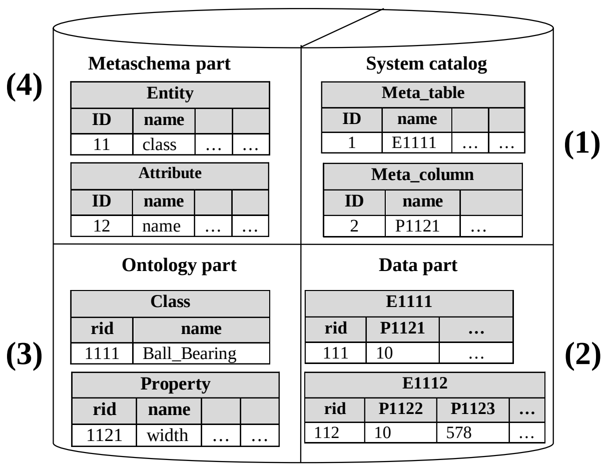


Fig. 7 Example of type III architecture: OntoDB

1. **The system catalog (1)** is a traditional part of any DBMS. It contains system tables used to manage all the data contained in the database. In OntoDB, it contains specifically the description of all the tables and columns defined in the three other parts of this architecture.
2. **The data part (2)** stores the ontology instances. An instance belongs to an ontology class and is described by a set of property values.
3. **The ontology part (3)** stores the ontologies that define the concepts of the domain covered by the database. OntoDB initially supports the PLIB ontology language.
4. **The metaschema part (4)** stores the ontology language used. For the ontology part, the metaschema part plays the same role as the one played by the system catalog in traditional DBMSs. It can be in particular used to extend the ontology part and thus to modify the used ontology language.

In the following we detail the three main parts of OntoDB.

6.2 The data part

The data part uses the horizontal storage layout. In this storage layout, a table is associated to each concrete class. Each one of these tables has a column named *rid* to identify class instances. In addition, this table has one column for each property used by at least one instance of this class. To define the link between an instance and its belonging class, the name of a table (resp. of a column) is the concatenation of *"E"* (resp. *"P"*) with the identifier of the corresponding class (resp. property).

Since properties are represented by columns, each PLIB datatype has a specific representation in PostgreSQL:

– primitive datatypes have equivalent datatypes in PostgreSQL (e.g, the PLIB *int_type* is coded as *INT8* in PotgreSQL);
– the reference type (*class_instance_type*) is used to link two instances. Because of the polymorphism, two columns are used to represent this dataype. The name of the first one is suffixed by *_rid* and provides the *identifier* of the referenced instance. The second one, suffixed by *_tablename* stores the name of the table in which the referenced instance is stored;
– the collection datatype (*aggregate_type*) is represented by the *ARRAY* type of PostgreSQL. The properties whose values are collections of instances are represented by two columns of type *ARRAY*. The first one, suffixed by *_rids*, stores the identifiers of the referenced instances. The second one, suffixed by *_tablenames*, stores the names of the tables in which these instances are stored.

An example of the data part of OntoDB for our example of ontology is depicted in Figure 8. Three tables are created for the three classes *Product*, *Ball_Bearing* and *Row_Of_Balls*. The name of a table is defined as the identifier of a class prefixed with *E*. For example, since the identifier of *Product* is *1111*, the name of the corresponding table is *E1111*. Each table has a column *rid* and columns for the used properties. Properties such as *width*, *name* or *length* are represented by a single column whose name is the identifier of the property prefixed with *P*. The *used_in* property corresponds to an association between the *Ball_Bearing* and *Product* classes. It is represented by two columns *P1124_rid* to store the identifier of the corresponding product and *P1124_tablename* to know the table in which this instance is stored (a sub-class of *Product* could be used). The same representation is used for the *uses* property. The only difference is that this property has a collection dataype and thus the ARRAY datatype of PostgreSQL is used.

E1111 (Product)	
rid	P1121 (name)
1	Bicycle

E1112 (Row_Of_Balls)	
rid	P1122 (length)
2	11
3	14

E1113 (Ball_Bearing)					
rid	P1123 (width)	P1124_rid (used_in)	P1124_table name (used_in)	P1125_rids (uses)	P1125_tables names (uses)
4	10	1	E1111	[2 ,3]	[E1112, E1112]

Fig. 8 Example of the data part of OntoDB

6.3 The ontology part

The ontology part is initially defined to store PLIB ontologies. The PLIB language is defined in the EXPRESS formalism (Schenk and Wilson, 1994), which is similar to the UML language. It defines an object-oriented model by a set of entities with inheritance relationships and attributes. The model of the PLIB language is composed of hundred of entities and attributes. Thus, instead of manually defining the storage layout for this part, a program takes as input the object-oriented representation of the PLIB language in EXPRESS and generates the set of tables of this part by applying a set of rules.

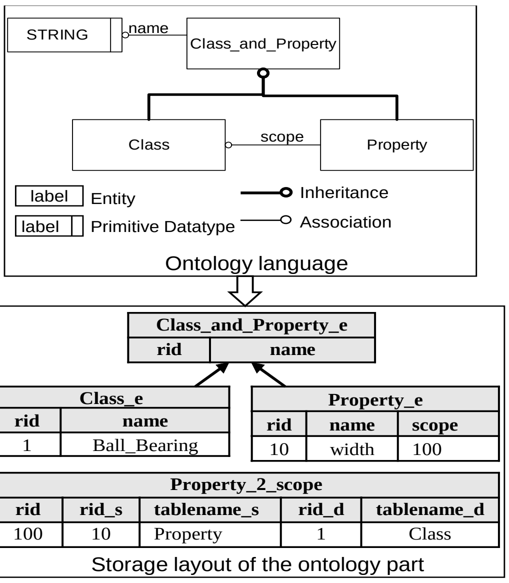
Each entity of the PLIB language is translated into a table. The name of this table is the one of the corresponding entity suffixed by *_e*. If an entity inherits from another entity, its table inherits from the table corresponding to the other entity thanks to the table inheritance mechanism available in PostgreSQL.

The associations *one-to-many* or *many-to-many* are represented by *association tables*. For an entity *A* linked to an entity *B* by an association named *a2b*, an association table named *A_2_a2b* is created. This table has the five following columns:

– *rid* identifies the association;
– *rid_s* and *tablename_s* contain respectively the identifier of an instance of the *A* entity and the table in which this instance is stored (it can be a sub-entity of *A*);
– *rid_d* and *tablename_d* play the same role as *rid_s* and *tablename_s* for the entity *B*.

Moreover, to optimize the access to an association, the table *A_e* has a column *a2b* that is a foreign key (association one-to-many) or a collection of foreign keys (association many-to-many) pointing to the primary key of the association table.

Example. Figure 9 presents a simplified ontology language and the corresponding tables in the ontology part of OntoDB. The ontology language is composed of three entities. The *Class_and_Property* entity has an attribute *name*. *Class_and_Property* is the super-entity of the entities *Class* and *Property* which represent respectively ontology classes and properties. These two entities are linked by the *scope* association that represents the domain of a property. Each entity is translated into a table in the ontology part of OntoDB (suffixed by *_e*). Each one of these tables has a column named *rid* to identify the corresponding ontology elements. The hierarchy of tables is conformed to the hierarchy of entities: the tables *Class_e* and *Property_e* inherit from the *Class_and_Property_e* table. Finally, the *scope* association is represented by the *Property_2_scope* association table. This table makes the link between a property and a class. The *scope* column of the *Property_e* table is used to join this table with the association table.



Class_and_Property_e	
rid	name

Class_e	
rid	name
1	Ball_Bearing

Property_e		
rid	name	scope
10	width	100

Property_2_scope				
rid	rid_s	tablename_s	rid_d	tablename_d
100	10	Property	1	Class

Fig. 9 Example of the ontology part of OntoDB

6.4 The metaschema part

In a database, the system catalog is used to record the tables and columns of the data part. Thanks to this part, a SQL query can be checked and the data part can be extended with new tables and columns. The same idea is applied on the ontology part in OntoDB. As a consequence, OntoDB has a new part called the *metaschema*. This part stores the ontology language used to define the ontologies. The query language of OntoDB (presented in the next section) leverages it to extend dynamically the ontology language used.

The tables of this part correspond to the formalism used to define the ontology language. In the case of OntoDB, these tables correspond to the EXPRESS language. These tables are automatically generated from the model of the EXPRESS language. The rows of these tables correspond to the constructors of the ontology language used (e.g., *class*, *property*, etc.).

An extract of the metaschema part of OntoDB for the simplified ontology language presented in Figure 9 is depicted in Figure 10. The tables *Entity* and *Attribute* store the corresponding elements of the ontology language.

Entity		
rid	name	super
1	Class_and_Property	NULL
2	Class	1
3	Property	1

Attribute			
rid	name	scope	range
11	name	1	String
12	scope	3	2

Fig. 10 Example of the metaschema part of OntoDB

6.5 Performance evaluation of OntoDB

We have presented in (Dehainsala et al, 2007) a series of performance experiments to compare the storage layout proposed by OntoDB with the vertical and binary storage layouts used by other OBDBs (see Section 5). As our work targets a specific application domain (engineering), we have developed a benchmark based on the IEC ontology (IEC61360-4, 1999), which reflects the needs of this domain. The results of these experiments show that OntoDB outperforms other storage layouts for queries in which the user specifies the class to be queried. Moreover, the difference of performance between OntoDB and the vertical and binary storage layouts increases with the number of properties used in the query. The only case where the performance of OntoDB is worse than other storage layouts is for queries in which the class to be queried is not specified and which only use a small number of properties. The interested reader can refer to (Dehainsala et al, 2006) for a complete description of these experiments.

In this section, we have presented the OntoDB architecture that we have designed for storing ontologies and their instances. The SQL language could be used to write queries on OntoDB. However, one needs to have a deep knowledge of the storage layout used by OntoDB to write such queries. As a consequence, we have defined a specific language named OntoQL to query ontologies and their instances.

7 The OntoQL exploitation language

In the last decade, a lot of Semantic Web query languages have been defined. The interested reader can refer to (Bailey et al, 2005) for a survey. SPARQL (Prud'hommeaux and Seaborne, 2008) is currently the most well known query language. SPARQL is a graph-matching query language. A SPARQL query consists of a pattern that is matched against a data source. This pattern is composed of a set of *triple patterns*, which are triples with variables. This language has been defined to query RDF data. As a consequence, it considers both ontologies and instances as triples. To take into account the semantics of RDFS or OWL ontologies, the pattern matching is defined using semantic entailment relations.

Contrary to Semantic Web query languages, our aim was to define a language (1) independent of a given ontology language (2) that exploits the different layer of an ontology (canonical, non canonical and linguistic layers) and (3) which is compatible with the SQL language. This section presents the resulting language called *OntoQL*. First, we formally define the data model targeted by this language.

7.1 Data Model

The data model addressed by OntoQL is composed of two main parts: the ontology and the content part.

7.1.1 The ontology part

The ontology part stores ontologies as instances of an ontology language. This ontology language is not hard-encoded in OntoQL. It is composed of a set of entities and attributes. Formally, this part is defined by a 8-tuple: $< E, OC, A, SuperEntities, TypeOf, AttDomain, AttRange, Val >$.

- E is a set of entities representing the ontology language. It includes predefined entities such as the constructor of classes C and properties P as well as user-defined entities.
- OC is the set of ontology elements (classes, properties ...). They have a unique identifier.
- A is the set of attributes describing each ontology element. For example, classes are described with names, which can be defined in different natural languages.
- $SuperEntities : E \rightarrow 2^E$[3] is a partial function that defines the super-entities of an entity.
- $TypeOf : OC \rightarrow E$ defines the type of an ontology element. For example *Ball_Bearing* is an instance of the C entity.
- $AttributeDomain, AttributeRange : A \rightarrow E$ define respectively the domain and range of each attribute.
- $Val : OC \times A \rightarrow OC$ defines the value that an ontology element has for a given attribute.

[3] We use the symbol 2^E to denote the power set of E.

This data model is equipped with atomic types (*Int*, *String*, *Boolean*) and with two parameterized types $Set[T]$ and *Tuple*. $Set[T]$ defines collections of elements of type T and $\{o_1, \ldots, o_n\}$ is an object of this type (the o_i's are objects of type T). The $Tuple[< (A_1, T_1), \ldots, (A_n, T_n) >]$ parameterized type creates relationships between objects. It is constructed by providing a set of attribute names (A_i) and attribute types (T_i). $Tuple[< (A_1, T_1), \ldots, (A_n, T_n) >]$ denotes a type tuple defined with A_i attribute names and T_i attribute types. $< A_1 : o_1, \ldots, A_n : o_n >$ is an object of this type (the o_i's are objects of type T_i). The *Tuple* type is equipped with the $Get\_A_i\_value$ functions to retrieve the value of a *Tuple* object o for the attribute A_i. The application of this function can be abbreviated using the dot-notation ($o.A_i$)

From the study made in Section 4, we can see that ontology languages share common constructors to define CCOs. An ontology has a namespace. It is composed of classes and properties. Classes are organized in a hierarchy using subsumption relationships. They are associated to properties whose range may be a class or a datatype. Classes and properties can be referenced using an identifier independent of the underlying system (e.g., a URI or a BSU). They are described by names and definitions that may be defined in different natural languages. As a consequence, we define a set of predefined entities and attributes. Thus, E provides the predefined entities C and P. Instances of C and P are respectively the ontology classes and properties. The entity C defines the attribute $SuperClasses : C \rightarrow SET[C]$ and the entity P defines the attributes $PropDomain : P \rightarrow C$ and $PropRange : P \rightarrow C$. The description of these attributes is similar to the definitions given for *SuperEntities*, *AttributeDomain* and *AttributeRange* replacing entities by classes and attributes by properties. Moreover, a global super class *Root* is predefined. This core ontology language can then be extended with user-defined entities and attributes.

7.1.2 The content part

The content part stores instances of ontology classes. It is formalized by a 5-tuple $< EXTENT, I, TypeOf, SchemaProp, Val >$.

- $EXTENT$ is a set of *extensional definitions* of ontology classes.

- I is the set of ontology instances. Each instance has an identifier.
- $TypeOf : I \to EXTENT$ defines the type of each instance.
- $SchemaProp : EXTENT \to 2^P$ defines the properties used to describe the instances of a class.
- $Val : I \times P \to I$ gives the value that has an instance for a given property. This property must be used in the extensional definition of the belonging class of the instance.

7.1.3 Relationship between each part

The relationship between the ontology and content parts is defined by the partial function $Nomination : C \to EXTENT$. It associates a definition by intension of a class with its definition by extension. Classes without extensional definition are said to be *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 ($propDomain^{-1}(c) \supseteq SchemaProp(nomination(c))$).

7.2 The OntoQL Query Algebra

Since our data model uses extensively object-oriented features, the OntoQL algebra, named *OntoAlgebra*, is based on the *ENCORE* algebra (Shaw and Zdonik, 1990). The signatures of the OntoQL algebra operators belong to $(E \cup C) \times 2^{OC \cup I} \to (E \cup C) \times 2^{OC \cup I}$. The main operators of this algebra are *OntoImage*, *OntoProject*, *OntoSelect* and *OntoOJoin*. For the sake of clarity, only these operators are presented and they are restricted to the signature: $C \times 2^I \to C \times 2^I$. However, the complete definition of this algebra can be found in (Jean et al, 2007a).

- OntoImage. The *OntoImage* operator applies a function to a collection of instances. Its signature is $C \times 2^I \times Function \to C \times 2^I$. *Function* contains all the properties in P and all the properties that can be defined by composing properties of P (path expressions).

Differently from the object-oriented data model, one or more of the properties occurring in the function parameter may not be valued in the extensional definition of a class. Thus, the domain of the Val function needs to be extended with the properties that are defined on the intensional definition of a class but not used in its extensional definition. This extension is called $OntoVal$ and is defined by:

$OntoVal(i, p) = Val(i, p)$
if $p \in SchemaProp(TypeOf(i))$ else, $UNKNOWN$

$UNKNOWN$ is a special instance like $NULL$ is a special value for SQL. To preserve composition, $OntoVal$ applied to a property whose value is $UNKNOWN$ returns $UNKNOWN$. With this definition of $OntoVal$, $OntoImage$ is defined by:

$$OntoImage(T, \{i_1, \ldots, i_n\}, f) = (PropRange(f), \{OntoVal(i_1, f), \ldots, OntoVal(i_n, f)\})$$

- OntoProject. The *OntoProject* operator extends *OntoImage* to apply more than one function to an instance. The result type is a *Tuple* defined by:

$$OntoProject(T, I_t, \{(A_1, f_1), \ldots (A_n, f_n)\}) = (Tuple[< (A_1, propRange(f_1)), \ldots, (A_n, propRange(f_n)) >], \{< A_1 : OntoVal(i, f_1), \ldots, A_n : OntoVal(i, f_n) > | i \in I_t\})$$

- OntoSelect. It creates a collection of instances satisfying a selection predicate. Its signature is $C \times 2^I \times Predicate \to C \times 2^I$ and it is defined by:

$$OntoSelect(T, I_t, pred) = (T, \{i \mid i \in I_t \wedge pred(i)\})$$

If the predicate *pred* contains a function, then $OntoVal$ is used.

- OntoOJoin. It creates relationships between instances of two collections. It is defined by:

$$OntoOJoin(T, I_t, R, I_r, A_1, A_2, pred) = (Tuple[< (A_1, T), (A_2, R) >], \{< A_1 : t, A_2 : r > | t \in I_t \wedge r \in I_r \wedge pred(t, r)\})$$

- Operator *. This operator is used to distinguish queries on instances of a single class C and queries on C and all its sub-classes denoted C^*. $ext : C \to 2^I$ returns the instances of a class and $ext^* : C \to 2^I$ returns the instances of a class and of its sub-classes. If c is a class and $c_1, \ldots c_n$ are the direct sub-classes of c, ext and ext^* are derived recursively[4] by:

$$ext(c) = TypeOf^{-1}(Nomination(c))$$
$$ext^*(c) = ext(c) \cup ext^*(c_1) \cup \ldots \cup ext^*(c_n)$$

With the ext and ext^* functions, the * operator is defined by $^* : C \to C \times 2^I$ where$^*(T) = (T, ext^*(T))$.

This algebra represents the semantics of the OntoQL query language. The same approach has been followed to define the semantics of the definition and manipulation language of OntoQL. These different sub-languages of OntoQL are presented in the next section.

[4] To simplify notation, we extend all functions f by $f(\emptyset) = \emptyset$

7.3 The OntoQL Language

The OntoQL language can be used to define and query ontologies and their instances. We first describe the sub-languages defined to manipulate the instances.

7.3.1 The Data Definition (DDL), Manipulation (DML) and Query (DQL) Languages of OntoQL

The DDL of *OntoQL* is used to create, alter and drop ontology elements (classes, properties ...) as well as their attributes values (name, definition ...). For example, the following statement creates the *Ball_Bearing* class as a sub-class of *Rolling_Bearing* with a name and a definition defined in different natural languages. The properties *width*, *mass* and *used_in* are defined in the same time. As the ontology language used by OntoQL is not hard encoded in its grammar, the # prefix identifies entities and attributes of the used ontology language.

```
CREATE #CLASS Ball_Bearing EXTENDS Rolling_Bearing (
DESCRIPTOR(#name[fr] = 'Roulement à billes',
           #definition[fr] = 'un type de roulement ...',
           #definition[en] = 'a type of rolling ...')
PROPERTIES(width Real, mass Real, used_in REF(Product)))
```

Once a class has been defined by intension, an extent can be attached to this class by the following statement:

```
CREATE EXTENT OF Ball_Bearing (width, mass)
```

The extent of a class is composed of a subset of the properties defined on the intensional definition of this class. In this example, the properties *width* and *mass* are the only properties used to describe the instances of the *Ball_Bearing* class.

Once an extent of a class has been defined, the DML of OntoQL can be used to insert, update and delete instances. The syntax of this language is similar to the one of the DML of SQL as the following statement shows.

```
INSERT INTO Ball_Bearing (width, mass) VALUES (6.9, 7.8)
```

The query language of OntoQL keeps a syntax close to SQL (SELECT-FROM-WHERE). Moreover, OntoQL supports the following features, all expressed by a composition of *OntoAlgebra* operators.

– *Path expressions.* Associations may be traversed using the dot notation.
– *Polymorphic query.* Queries on direct instances of a class are distinguished from queries on all instances of a class with the keyword ONLY.
– *Dependent collection.* A collection can be traversed using an iterator introduced in the FROM clause.
– *Nested queries.* Queries can be nested in the SELECT, FROM and WHERE clauses.
– *Aggregate functions.* *OntoQL* provides aggregate functions count, sum, avg, min and max.
– *Quantification.* Existential (ANY, SOME) and universal (ALL) quantification can be expressed.
– *Set operators.* Union, Intersection and Except operators are provided.

Let us give some examples of queries to illustrate the particularities of this language. The semantics of OntoQL has been defined to search values of an instance for each property defined on its belonging class. When a property is not used in the extent of this class, the NULL value is returned. The following query retrieves the width, mass and description of the products in which ball bearings are used (path expression).

```
SELECT width, mass, used_in.desc FROM Ball_Bearing
```

As the *used_in* property is not used in the extent of the *Ball_Bearing* class, the column corresponding to this property is filled with the NULL value for each resulting row.

Another particularity of ontologies is that classes and properties have a namespace. The USING NAMESPACE clause of an OntoQL statement specifies the namespace in which classes and properties must be searched. Several namespaces can be specified in the USING NAMESPACE clause if one wants to query elements defined in different ontologies. The following query specifies that the *Roller_Bearing* class and the *mass* property should be searched in the namespace http://www.lias-lab.fr.

```
SELECT mass FROM Roller_Bearing
USING NAMESPACE 'http://www.lias-lab.fr'
```

If the USING NAMESPACE clause is not specified and if there is no default namespace, the query is interpreted as an SQL statement. Thus, the OntoQL language is compatible with SQL.

The next example illustrates a third particularity of ontologies: classes and properties have a textual definition that may be given in different natural languages (linguistic layer of an ontology). This particularity is used in OntoQL to express queries in different natural languages. The next query is written in English (a.) and in French (b.).

```
a.SELECT width, mass   <=> b.SELECT longueur, masse
     FROM "Ball Bearing"       FROM "Roulement à billes"
```

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

7.3.2 The Ontology Definition (ODL), Manipulation (OML) and Query (OQL) Languages of OntoQL

Ontologies and the used ontology language can be managed with the ODL, OML and OQL of OntoQL. These languages have a syntax close to the ones of the DDL, DML and DQL. For example, the next statement, adds the `AllValuesFrom` constructor from OWL to the used ontology language of OntoQL.

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

This statement adds the `OWLRestrictionAllValuesFrom` entity to the core ontology language as a subentity of `Class`. This entity is created with two attributes `onProperty` and `allValuesFrom`. `onProperty` points to a property (`REF(#Property)`) and `allValuesFrom` points to a class (`REF(#Class)`).

An `OWLAllValuesFrom` class can then be created with the OML:

```
INSERT INTO #OWLRestrictionAllValuesFrom
        (#name[en], #onProperty, #allValuesFrom)
  VALUES ('Row_Ball_Bearing', 'uses', 'Row_of_Balls')
```

This example creates the *Row_Ball_Bearing* class as being the set of instances that only use *Row_of_Balls*.

Finally the OQL is used to search ontology elements stored in the OBDB. For example, the next query searches the `OWLAllValuesFrom` classes defined on the `uses` 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] = 'uses'
```

7.3.3 Querying both the ontologies and instances in OntoQL

By combining the DQL and OQL of OntoQL, ontologies and instances can be queried simultaneously.

From ontology to instances. Starting from classes retrieved by a query on ontologies (OQL), the instances of these classes can then be filtered (DQL). This type of queries is possible thanks to *dynamic iterators*. To query instances, an iterator `i` on the instances of a class `C` can be introduced using the `C AS i` construct. OntoQL extends this mechanism with iterators on the instances of a class identified at run-time, which are called dynamic iterators. For example, the following query retrieves the instances of all classes whose names start with `Ball`.

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

From instances to Ontology. Starting from instances retrieved by a DQL query, the description of the belonging classes of these instances can be retrieved (OQL). OntoQL proposes the `typeOf` operator to retrieve the *basis class* of an instance i.e., the minorant class for the subsumption relationship of the classes it belongs to. For example, the following query retrieves the English name of the basis class of each `Rolling_Bearing` instances.

```
SELECT typeOf(b).#name[en] FROM Rolling_Bearing AS b
```

7.4 OntoQL Query Processing

As OntoQL is implemented on top of OntoDB, an OntoQL query is translated into SQL. This process follows five main steps.

1. **OntoAlgebra query plan generation.** The OntoQL query is parsed and turned into an expression tree involving operators of its algebra in its nodes.
2. **OntoAlgebra query plan optimization.** We have identified optimization situations to reduce the OntoAlgebra query plan. These optimizations techniques are detailed in (Jean et al, 2006).
3. **OntoAlgebra query plan translation into relational algebra trees.** This translation is achieved by applying a set of rules described below.
4. **Relational algebra trees optimization.** This step consists in using the different algebraic laws that hold for the relational algebra to turn the relational trees into equivalent trees that may be executed more efficiently by the underlying DBMS.
5. **Relational algebra tree translation into SQL.** The optimized relational trees are translated into SQL queries according to the underlying DBMS and executed to get the *OntoQL* query result.

The translation between OntoQL and SQL query plan is an important step. To illustrate the set of rules defined, table 2 presents a translation rule of an *OntoAlgebra* expression to a relational algebra expression. The interested reader can refer to (Jean, 2007) for the definition of the complete set of rules. In these rules, π represents the projection operator of the relational algebra. `C` is a class and $\mathtt{p}_1, \ldots, \mathtt{p}_n$ are the properties defined on this class. Among these properties only $\mathtt{p}_1, \ldots, \mathtt{p}_u$ are used to describe their instances. The datatype of $\mathtt{p}_1$ is a collection of references and the one of $\mathtt{p}_2$ is a single-valued reference. Others properties have primitive datatypes.

Rule 1 computes the direct instances of the class `C` with their values for all the applicable properties

OntoAlgebra	Relational Algebra
$\texttt{OntoProject}\ (\texttt{C}, \texttt{ext}(\texttt{C}), \{(\texttt{p}_1, \texttt{p}_1), \ldots, (\texttt{p}_n, \texttt{p}_n)\}$	$\pi_{\texttt{Pp}_1\texttt{\_rids},\texttt{Pp}_2\texttt{\_rid},\texttt{Pp}_3,\ldots,\texttt{Pp}_u, \texttt{NULL}\rightarrow\texttt{Pp}_{u+1},\ldots,\texttt{NULL}\rightarrow\texttt{Pp}_n}(\texttt{EC})$

Table 2 Example of a translation rule

on this class. The `OntoProject` operator of *OntoAlgebra* is translated into a projection of the corresponding columns (prefixed by `P`) on the corresponding table (prefixed by `E`). The projections of properties that are not used to describe instances are translated into projections of the `NULL` values as defined in the *OntoAlgebra* semantics. The resulting column is renamed (symbol $\rightarrow$) according to the OntoDB naming convention so that other operators can reference it as a used property.

If the identifiers of *width*, *mass* and *Ball_Bearing* are respectively `1`, `2` and `3`, an example of the application of the previous rule is:

```
SELECT width, mass, used_in => SELECT P1, P2, NULL
  FROM Ball_Bearing              FROM E3
```

In this section, we have presented the OntoQL query language. This language has three main characteristics that distinguish it from other proposed languages: (1) the OntoQL language is independent of a given ontology language. Indeed, it is based on a core ontology language, which contains the constructors shared by different ontology languages and this core ontology language can be extended by the OntoQL language itself, (2) the OntoQL language exploits the linguistic information that may be associated to a conceptual ontology allowing users to express queries in different natural languages and (3) the OntoQL language is compatible with SQL. In the next section we present different applications of the OntoDB/OntoQL platform. This platform is available as open source softwares at `http://www.lias-lab.fr/forge/projects/ontodb`.

8 Applications

8.1 Standard Ontologies in Engineering

The PLIB ontology language has been used to develop a number of standard ontologies in different fields of the engineering domain such as:

- Electronic Components (IEC 61360-4);
- Process Instruments (IEC 61360-4);
- Mechanical Fasteners (ISO 13584-511);
- Measure Instruments (ISO 13584-501);
- Cutting Tools (ISO 13399);
- Bearings (ISO 23768);
- Technical Product Documentation (ISO/TC 10 NWI);
- Optics and Photonics (ISO 23584).

To show the size and the complexity of these ontologies, we detail some examples.

The *Mechanical Fasteners* ontology (ISO 13584-511) represents fasteners with their properties and domains of values as they are described in the various ISO mechanical fastener standards. These fasteners include bolts, screws, nuts, rivets, pins and washers. This ontology is composed of approximatively 250 classes and 410 properties. The definitions of these classes are given in French and in English. Several man-years were required for its definition.

The *Cutting Tools* ontology (ISO 13399) defines the terms, properties, and definitions for those portions of a cutting tool that remove material from a workpiece. Cutting items include replaceable inserts, brazed tips, and the cutting portions of solid cutting tools. This ontology is composed of approximatively 500 classes and 360 properties. As for the mechanical fasteners ontology, the definitions of these classes are given in French and in English and several man-years were required for its definition.

The PLIB language has also been used in the eCl@ss classification (`http://www.eclass.de/`). eCl@ss is a product classification and description standard for information exchange between customers and their suppliers. It describes products such as cable, wire, accumulator or battery and services such as process control system or electrical measurement. This classification is composed of approximatively 33 000 classes and described in 15 natural languages.

These example shows the interest of the PLIB ontology language. Next section describes our approach based on the OntoDB/OntoQL platform to integrate heterogenous data.

8.2 Integration a priori and a posteriori

Integrating heterogeneous, autonomous and distributed data sources is one of the favourite application domains of ontologies (Noy, 2004). They contribute on reducing syntactic and semantic heterogeneities that may exist between sources. Due to the increasing number of data sources, automatic integration processes become a necessity for companies. We have proposed data integration solutions for French companies such as Renault S.A. (French multinational vehicle manufacturer established in 1899) which needed to integrate their suppliers data sources. To perform this integration, we claim the following: if we want to avoid human intervention at integration time, mappings between sources shall be done *a priori* during the data sources design. This means that some formal shared ontology must exist, and each data source shall embed some ontological data that reference explicitly this shared ontology. Some integration systems funded by industrials and academic institutions such as the Piscel2 project funded by France Telecom for integrating Web Services (Reynaud and Giraldo, 2003) and the COIN project supported by ARPA and USAF/Rome Laboratory for exchanging financial data (Bressan et al, 2000), worked under the same assumption. Their main limitation is that once the shared ontology is defined, each source shall only use the common vocabulary. The shared ontology is in fact the integrated schema and each source has little schematic autonomy.

To overcome this limitation, we offer more schematic autonomy to data sources participating in the data integration process. To achieve this goal, three assumptions are required.

1. Each data source participating in the integration process shall contain its own ontology. This assumption refers to the need of an ontology-based database (see Section 5).
2. Each data source references a shared ontology *as much as possible*. As much as possible means that: (1) each class of a local ontology references explicitly (or implicitly through its parent class) its lowest subsuming class in the shared ontology, and (2) only properties that do not exist in the shared ontology may be defined in a local ontology; otherwise it should be imported through the *case-of* relationship. This requirement is called *smallest subsuming class reference requirement*.
3. Local ontologies may extend the shared ontology as needed by adding new concepts and properties.

Based on these assumptions, three integration scenarios have been proposed for Renault S.A called: *FragmentOnto*, *ExtendOnto* and *ProjOnto*. These scenarios are described as follows.

FragmentOnto: in this scenario, we assume that the shared ontology is complete enough to cover the needs of all data sources. This scenario is feasible for authoritative parties that can force their suppliers to use the shared ontology. The source autonomy consists in (1) selecting the relevant subset of the shared ontology (classes and properties), and (2) designing the local database schema. This approach has been presented in (Bellatreche et al, 2004). It is not well suited for integrating autonomous data sources.

ExtendOnto: in this scenario, the shared ontology is extended by each local specialization. The local instances are then integrated within the integrated system without any change.

ProjOnto: in this scenario, the shared ontology is not modified. Each data source instance is projected onto the applicable properties of its smallest subsuming class in the shared ontology, and is then added to the population of this class in the shared ontology. These scenarios are well described in the PhD thesis of Dung Nguyen Xuan (Nguyen-Xuan, 2006) and in (Bellatreche et al, 2004).

8.3 Engineering model annotation: the e-Wok Hub Project

The goal of the e-Wok Hub project was to help geologists in carrying out petroleum prospection projects. The petroleum exploration activities is based on the representation of underground reservoirs. These representations use complex and well founded geological models like stratigraphic, structural, datation, geological models, etc. Such an activity leads to a set of complex and heterogeneous models that use a huge amount of data produced either by measurement campaigns or by the corresponding computation models. By heterogeneity, we mean both heterogeneous models in terms of semantics and heterogeneous interpretations since two geologists can give different conclusions when interpreting the models and their corresponding data. Moreover, when working on such projects, engineers and geologists use a large panoply of resources such as scientific articles, reports of past projects, softwares, data files, etc.

The platform developed in the context of this project integrates these heterogeneous resources in a global architecture where ontologies play a central role for data integration, exchange and querying. In the context of this project, the OntoDB database served as a repository and the OntoQL language was used to query the integrated data.

For example, in the case of the CO2 capture and storage, engineers and geologists rely on various engineering models. They have to deal with 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 (Mastella et al, 2009). As the notions of annotations and engineering models were not available in OntoDB, the OntoQL language was used to introduce these notions. First, elements of the engineering models were created with the `CREATE ENTITY` statement of OntoQL. Then, an association table was defined to annotate the engineering models by a class of an ontology. Once this extension was done, OntoQL was used to query the engineering models departing from the ontology concepts.

Moreover, the design of the corresponding ontologies involved several ontologies issued from different domains (stratigraphy, datation, sturcural, etc.) that needed to be integrated in a single ontology without impacting their use in different areas. For this purpose, an *a posteriori case of* relationship has been used to map the different developed ontologies.

This section has shown the interest of the development presented in this paper on several examples of industrial projects. Other successful applications of the PLIB language with its associated technologies include projects with companies such as Toshiba Corporation, Philips, Siemens, Zeiss or Sandvick.

9 Conclusion

In this paper, we have highlighted the benefits of ontologies in the engineering domain. A deep analysis of the use of ontologies by research communities has brought new insights to the classification of ontologies in three main categories: *canonical ontologies*, *non-canonical ontologies* and *linguistic ontologies*. The canonical ontologies described by formalisms such as PLIB have been largely used in the engineering domain. Compared to Semantic Web languages such as RDFS or OWL, PLIB adheres to the closed-world and unique name assumptions, which are adapted to the controlled environment that we found in industrial engineering settings. This language also proceeds to represent the modeling context of the concepts defined in the ontology. This characteristic is important for data integration as an implicit modeling context is the main cause of semantic data heterogeneity (Bressan et al, 2000). This modeling context is expressed by the representation of:

- the context of each class by its set of properties and of each property by its domain of definition;
- the different point of views of the same classes (*view-of operator)*;
- the local interpretation of an ontology by the *case-of* importation mechanism;
- the context of evaluation of a property when it depends upon different parameters;
- the unit and scaling of values.

The PLIB formalism has been largely used to define several standard ontologies in different fields of the engineering domain. The increasing quantity of semantic instances pushes us to propose a persistent solution for them: the OntoDB architecture. The originality of this architecture compared to those defined for RDFS and OWL is twofold: (i) OntoDB leverages the strong typing assumption made in PLIB to propose an efficient storage layout for engineering data and (ii) OntoDB includes the *metaschema* part which has been extensively used to extend the used ontology language (e.g., to store and annotate engineering models with ontologies in the e-Wok Hub project).

The OntoDB database is equipped with the OntoQL language that can be used to define, manipulate and query ontologies and their instances. Instead of defining a language specifically for PLIB, OntoQL is based on a core ontology language composed of the main constructors of several ontology languages. And, this core ontology language can be extended with the OntoQL language itself. Another particularity of this language is that it is compatible with the SQL language: a statement without a namespace definition is considered as a SQL statement. Moreover, the syntax and semantics of OntoQL to query ontologies and their instances are close to the SQL language. Finally, OntoQL exploits the different layers of an ontology, and in particular benefits from the linguistic layer allowing users to write queries in different natural languages. The PLIB language and the OntoDB/OntoQL platform have been used in several industrial projects with companies such as Renault, the French Institute of Petroleum, Toshiba and Philips in various projects related to the problem of data integration.

The developments of the OntoDB/OntoQL platform[5] are continually evolving to satisfy the needs of our partners and of the addressed research issues. These evolutions concern several facets of our platform: (i) *database extension* such as the support of multiple ontology languages, engineering models, semantic web services and functional dependencies, (ii) *query language extension* with user preferences and behaviour semantics, (iii) *database optimization* with the selection of optimization structures such as materialized views and indexes, (iv) *personalization* with query recommendation and relaxation techniques.

Currently, we are also working on probabilistic and fuzzy models for representing uncertainly in ontologies as done in (Huang et al, 2014).

Acknowledgements This work has been initiated under the supervision of the late Professor Guy PIERRA. He has been the initiator of the OntoDB project. We are grateful for his work and his investment. We learned a lot from his experience and advices. The authors would also like to thank the persons who have contributed to the development of the OntoDB/OntoQL platform and in particular Hondjack Dehainsala for the development of OntoDB during his PhD thesis and Dung Xuan Nguyen for his PhD thesis on data integration.

[5] `http://www.lias-lab.fr/forge/projects/ontodb`

References

Abadi DJ, Marcus A, Madden SR, Hollenbach K (2007) Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07), pp 411–422

Aluç G, Özsu T, Daudjee K (2014) Workload Matters: Why RDF Databases Need a New Design. PVLDB 7(10):837–840

Apweiler R, Bairoch A, Wu CH, Barker WC, Boeckmann B, Ferro S, Gasteiger E, Huang H, Lopez R, Magrane M, Martin MJ, Natale DA, ODonovan C, Redaschi N, Yeh LS (2004) Uniprot: the Universal Protein knowledgebase. Nucleic Acids Research 32:D115–D119

Bailey J, Bry F, Furche T, Schaffert S (2005) Web and Semantic Web Query Languages: A Survey. In: Reasoning Web, pp 35–133

Bechhofer S, van Harmelen F, Hendler J, Horrocks I, McGuinness DL, Patel-Schneider PF, Stein LA (2004) OWL Web Ontology Language Reference. World Wide Web Consortium, URL `http://www.w3.org/TR/owl-ref`

Bellatreche L, Pierra G, Xuan DN, Hondjack D, Ait-Ameur Y (2004) An a Priori Approach for Automatic Integration of Heterogeneous and Autonomous Databases. In: Proceedings of the 15th International Conference on Database and Expert Systems Applications (DEXA'04), pp 475–485

Bressan S, Goh CH, Levina N, Madnick SE, Shah A, Siegel M (2000) Context Knowledge Representation and Reasoning in the Context Interchange System. Applied Intelligence 13(2):165–180

Brickley D, Guha R (2004) RDF Vocabulary Description Language 1.0: RDF Schema. World Wide Web Consortium, URL `http://www.w3.org/TR/rdf-schema/`

Broekstra J, Kampman A, van Harmelen F (2002) Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Proceedings of the First International Semantic Web Conference (ISWC'02), pp 54–68

de Bruijn J, Lara R, Polleres A, Fensel D (2005) OWL DL vs. OWL Flight: Conceptual Modeling and Reasoning for the Semantic Web. In: Proceedings of the 14th International Conference on World Wide Web (WWW'05), pp 623–632

Cullot N, Parent C, Spaccapietra S, Vangenot C (2003) Ontologies : A contribution to the DL/DB debate. In: Proceedings of the first International Workshop on Semantic Web and Database (SWDB'03), pp 109–129

Das S, Chong EI, Eadon G, Srinivasan J (2004) Supporting Ontology-Based Semantic matching in RDBMS. In: Proceedings of the 13th International Conference on Very Large Data Bases (VLDB'04), pp 1054–1065

Dehainsala H, Pierra G, Bellatreche L (2006) Managing Instance Data in Ontology-based Databases. Tech. rep., LISI/ENSMA, URL `http://www.lisi.ensma.fr/ftp/pub/documents/reports/2006/2006-LISI-003-DEHAINSALA.pdf`

Dehainsala H, Pierra G, Bellatreche L (2007) OntoDB: An Ontology-Based Database for Data Intensive Applications. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07), pp 497–508

Duan S, Kementsietsidis A, Srinivas K, Udrea O (2011) Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11), pp 145–156

Estival D, Nowak C, Zschorn A (2004) Towards Ontology-Based Natural Language Processing. In: Proceedings of the 4th Workshop on NLP and XML (NLPXML'04), pp 59–66

Graupmann J, Schenkel R, Weikum G (2005) The SphereSearch Engine for Unified Ranked Retrieval of Heterogeneous XML and Web Documents. In: Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05), pp 529–540

Gruber TR (1993) A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition 5(2):199–220

Harris S, Gibbins N (2003) 3store: Efficient Bulk RDF Storage. In: Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03), pp 1–15

Huang HD, Lee CS, Wang MH, Kao HY (2014) IT2FS-based ontology with soft-computing mechanism for malware behavior analysis. Soft Computing 18(2):267–284

IEC61360-4 (1999) Standard data element types with associated classification scheme for electric components - Part 4 : IEC reference collection of standard data element types, component classes and terms. Tech. rep., International Standards Organization

Jean S (2007) OntoQL, un langage d'exploitation des bases de données à base ontologique. PhD thesis, LISI/ENSMA and University of Poitiers

Jean S, Aït-Ameur Y, Pierra G (2006) 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), pp 704–721

Jean S, Aït-Ameur Y, Pierra G (2007a) 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), pp 141–156

Jean S, Pierra G, Ameur YA (2007b) Domain Ontologies: A Database-Oriented Analysis. In: Web Information Systems and Technologies, International Conferences, WEBIST 2005 and WEBIST 2006. Revised Selected Papers, pp 238–254

Knorr M, Alferes JJ, Hitzler P (2011) Local Closed World Reasoning with Description Logics under the Well-Founded Semantics. Artificial Intelligence 175(9-10):1528–1554

Ma L, Su Z, Pan Y, Zhang L, Liu T (2004) RStar: an RDF Storage and Query System for Enterprise Resource Management. In: Proceedings of the 30th International Conference on Information and Knowledge Management (CIKM'04), pp 484–491

Maio CD, Fenza G, Furno D, Loia V, Senatore S (2012) OWL-FC: an upper ontology for semantic modeling of Fuzzy Control. Soft Computing 16(7):1153–1164

Mastella LS, Aït-Ameur Y, Jean S, Perrin M, Rainaud JF (2009) 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), pp 147–156

Matuszek C, Cabral J, Witbrock M, Deoliveira J (2006) An Introduction to the Syntax and Content of Cyc. In: Proceedings of the 2006 AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering, pp 44–49

Mendes PN, Jakob M, Bizer C (2012) DBpedia: A Multilingual Cross-domain Knowledge Base. In: Proceedings of the 8th International Conference on Language Resources and Evaluation (LREC'12), pp 1813–1817

Neumann T, Weikum G (2008) RDF-3X: a RISC-style engine for RDF. PVLDB 1(1):647–659

Nguyen-Xuan D (2006) Intégration de base de données hétérogènes par articulation a priori d'ontologies : application aux catalogues de composants industriels. PhD thesis, LISI/ENSMA and University of Poitiers

Niles I, Pease A (2001) Towards a Standard Upper Ontology. In: Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001), pp 2–9

Noy NF (2004) Semantic Integration: A Survey Of Ontology-Based Approaches. SIGMOD Record 33(4):65–70

OMG (2002) Meta Object Facility (MOF), Specification v1.4, OMG Document formal/02-04-03

Paasiala P, Aaltonen A, Riitahuhta A (1993) Automatic Component Selection. In: Proceedings of the 9th CIM-Europe annual conference on Realising CIM's industrial potential, pp 303–312

Pan Z, Heflin J (2003) DLDB: Extending Relational Databases to Support Semantic Web Queries. In: Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03), pp 109–113

Park MJ, Lee JH, Lee CH, Lin J, Serres O, Chung CW (2007) An Efficient and Scalable Management of Ontology. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07), pp 975–980

Pierra G (2003) Context-Explication in Conceptual Ontologies: The PLIB Approach. In: Proceedings of the 10th ISPE International Conference on Concurrent Engineering (CE 2003), pp 243–254

Prud'hommeaux E, Seaborne A (2008) SPARQL Query Language for RDF. W3C Recommendation 15 January 2008 `http://www.w3.org/TR/rdf-sparql-query/`

Reynaud C, Giraldo G (2003) An Application of the Mediator Approach to Services over the Web. In: Special track Data Integration in Engineering, Concurrent Engineering (CE'2003), pp 209–216

Sakr S, Al-Naymat G (2009) Relational Processing of RDF Queries: A Survey. SIGMOD Record 38(4):23–28

Schenk D, Wilson P (1994) Information Modelling The EXPRESS Way. Oxford University Press

Schmidt M, Hornung T, Lausen G, Pinkel C (2009) SP^{2}Bench: A SPARQL Performance Benchmark. In: Proceedings of the 25th International Conference on Data Engineering (ICDE'09), pp 222–233

Sequeda J, Tirmizi SH, Corcho Ó, Miranker DP (2011) Survey of Directly Mapping SQL Databases to the Semantic Web. Knowledge Engineering Review 26(4):445–486

Shaw GM, Zdonik SB (1990) A Query Algebra for Object-Oriented Databases. In: Proceedings of the 6th International Conference on Data Engineering, pp 154–162

Sidirourgos L, Goncalves R, Kersten ML, Nes N, Manegold S (2008) Column-Store Support for RDF Data Management: not all swans are white. PVLDB 1(2):1553–1563

Suchanek FM, Kasneci G, Weikum G (2008) YAGO: A Large Ontology from Wikipedia and WordNet. Journal Web Semantics 6(3):203–217

Sugumaran V, Storey VC (2006) The role of domain ontologies in database design: An ontology management and conceptual modeling environment. ACM Transactions on Database Systems (TODS) 31(3):1064–1094

Weiss C, Karras P, Bernstein A (2008) Hexastore: Sextuple Indexing for Semantic Web Data Management. PVLDB 1(1):1008–1019

Wilkinson K (2006) Jena Property Table Implementation. In: Proceedings of the 2nd International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS'06), pp 35–46

Yuan P, Liu P, Wu B, Jin H, Zhang W, Liu L (2013) TripleBit: a Fast and Compact System for Large Scale RDF Data. PVLDB 6(7):517–528

Zou L, Mo J, Chen L, Özsu MT, Zhao D (2011) gStore: Answering SPARQL Queries via Subgraph Matching. PVLDB 4(8):482–493