# Une approche langage pour la gestion de données dans les systèmes de méta-modélisation

**Stéphane Jean — Yamine Aït-Ameur — Guy Pierra**

*LISI/ENSMA et Université de Poitiers - Téléport 2 - 1, Avenue Clément Ader*
*86960 Futuroscope - France*

*{jean,yamine,pierra}@ensma.fr*

*Catégorie Chercheurs*

RÉSUMÉ. *L'utilisation des méta-données dans les systèmes informatiques a provoqué l'émergence de systèmes gérant trois niveaux d'information : instance, modèle et méta-modèle. C'est par exemple le cas des bases de données traditionnelles, avec le niveau méta-base, et celui des bases de données contenant des ontologies que nous nommons Bases de Données à Base Ontologique (BDBO). Des langages ont été définis sur ces structures afin de permettre la manipulation des instances et des modèles. Par contre, ils ne permettent pas de manipuler le niveau méta-modèle de la même manière que les niveaux modèle et instance. Pourtant, ce besoin est devenu une préoccupation majeure car les méta-modèles sont souvent nombreux et évolutifs. Nous présentons, dans cet article, le langage OntoQL que nous avons conçu pour permettre la manipulation homogène de telles structures à trois niveaux. Pour illustrer l'intérêt de ce langage, nous le présentons dans le contexte des BDBO.*

ABSTRACT. *Usage of metadata in information systems leads to systems managing three levels of information: instance, model and metamodel. For example, it is the case of traditional databases, with the metabase level, and of databases containing ontologies that we call Ontology Based DataBases (OBDBs). Several languages have been defined for these structures in order to manage instances and models. But, they don't offer the capability to manage the metamodel level in the same manner as the instance and model levels. However, this need appeared because of the numerous existing metamodels and of their evolution. In this paper, we present the OntoQL language that we have defined to allow an uniform manipulation of three-levels structures. To illustrate the interest of this language, we consider a concrete example running in the OBDBs context.*

MOTS-CLÉS : *Ontologie, Base de données, langage d'interrogation, méta-modélisation, MOF*

KEYWORDS: *Ontology, Database, Query language, Meta-modeling, MOF*

## 1. Introduction

Les méta-données permettent de décrire la structure et/ou la sémantique des données. Depuis longtemps utilisées dans le contexte des bases de données pour constituer la méta-base, les méta-données sont de nos jours utilisées dans un nombre croissant de domaines tels que le Web Sémantique ou le génie logiciel. L'introduction de méta-données dans les systèmes informatiques amène à une architecture de systèmes dans lesquels trois niveaux de modélisation sont représentés : instance, modèle et méta-modèle. Ainsi, dans le contexte du Web Sémantique, des ontologies sont utilisées pour décrire la sémantique des données. Ces ontologies sont représentées à partir d'un modèle d'ontologie. Or, ces ontologies sont elles-mêmes modèles de leurs propres instances. La gestion simultanée des données et des ontologies, qui en définissent le sens, nécessite donc bien de représenter trois niveaux de modélisation. De même, dans les bases de données, qu'elles soient relationnelles, objets ou relationnelles-objets, les données sont des instances d'un schéma dont la structure est représentée en instanciant une méta-base. Dans le contexte du génie logiciel, l'OMG a suggéré de représenter les modèles UML en utilisant le MOF [Obj02]. A nouveau, trois niveaux de modélisation ont à être représentés dans le même système.

Le problème principal qui se pose pour la gestion de tels systèmes est que, désormais, les méta-modèles sont nombreux et évolutifs. Par exemple, dans le contexte du Web Sémantique, de nombreux modèles d'ontologie sont utilisés tels que OWL [DEA 04], RDF-Schema [BRI 04], PLIB [ISO 98] ou F-Logic [KIF 89]. De plus, la plupart de ces modèles sont soumis à des évolutions régulières afin de satisfaire les besoins des utilisateurs. De même, dans le contexte des bases de données, chaque système emploie sa propre structure de méta-base. C'est également le cas dans le contexte du génie logiciel, où chaque atelier UML implante, de façon plus ou moins complète, une des versions de la norme UML. Ainsi, les utilisateurs de systèmes à trois niveaux sont confrontés à la nécessité de s'adapter à la diversité et à l'évolution des méta-modèles. Or, peu d'outils sont aujourd'hui disponibles pour permettre leur modification. Soulignons que ce besoin de modification des méta-modèles ne porte pas seulement sur leur structure. Il porte aussi sur leur sémantique. Une instance de méta-modèle n'étant rien d'autre qu'une représentation d'un modèle, destinée donc à être instanciée, il importe que les modifications du méta-modèle n'empêchent pas celui-ci de définir des modèles instanciables, et, autant que faire se peut, permettent de définir le comportement de ces nouveaux modèles.

Pour permettre l'évolution des méta-modèles, la proposition de l'OMG consiste à ajouter un quatrième niveau : le méta-méta-modèle. Ce niveau permet de créer de nouveaux méta-modèles comme instance de ce niveau supérieur. Si cette piste a été suivie en génie logiciel afin de rendre, par exemple, les ateliers UML interopérables, elle n'est guère facile à mettre en oeuvre. De plus, elle ne permet pas de représenter la sémantique des modifications, et elle n'est généralisée, ni dans les bases de données, ni dans les applications du Web Sémantique. En conséquence, le problème de la maîtrise de la diversité et de l'évolution des méta-modèles demeure entier.

Dans les Systèmes classiques de Gestion de Base de Données (SGBD), l'outil principal de gestion est constitué d'un langage de définition et de manipulation de données. Différents langages ont ainsi été proposés selon le paradigme de modélisation utilisé par les systèmes hôtes. Mais, le point commun de ces langages est de ne permettre que de créer de nouveaux modèles, instanciant ainsi un méta-modèle figé, puis de peupler ces modèles. Ils ne permettent pas de modifier les meta-modèles. Ces langages ne répondent donc pas aux besoins des systèmes de méta-modélisation qui nécessitent également de modifier les méta-modèles.

L'objectif de cet article est de proposer une approche langage pour permettre l'exploitation uniforme et la manipulation homogène des trois niveaux de représentation de l'information existant dans les systèmes de méta-modélisation. Cette approche est présentée dans le contexte du Web Sémantique où le niveau méta, constitué du modèle d'ontologie, vise tout particulièrement à définir la sémantique des données à travers une description fine des concepts utilisés dans chaque modèle particulier. Mais, l'approche utilisée, et en particulier le langage à deux niveaux proposé et le méta-modèle noyau défini, semblent s'appliquer à l'identique aux autres systèmes de méta-modélisation, le noyau pouvant évidemment être étendu, tant dans sa structure que dans sa sémantique, selon le domaine d'application visé.

Cet article est structuré comme suit. Dans la section suivante nous introduisons un exemple montrant le besoin de modifier le niveau méta-modèle dans les systèmes de gestion de Base de Données à Base Ontologique (BDBO) dans lesquels le niveau méta représente le modèle d'ontologie. Nous utilisons ensuite cet exemple tout au long de cet article. Dans la section 3, nous présentons les structures de BDBO existantes. Dans la section 4, nous décrivons les langages qui ont été proposés jusque là pour gérer ces structures à trois niveaux. Le langage que nous proposons, OntoQL, est ensuite présenté dans la section 5 et son implantation sur la BDBO OntoDB [PIE 05, DEH 07] est décrite en section 6. Enfin, la section 7 conclut cette présentation du langage OntoQL et décrit les travaux que nous comptons mener autour de ce langage.

## 2. Un exemple motivant le problème

Lorsqu'il s'agit de développer une ontologie, il est nécessaire de choisir un modèle pour la représenter. Ce choix est difficile, non seulement parce qu'il existe de très nombreux modèles d'ontologie, mais également parce qu'ils présentent chacun des particularités qui peuvent être simultanément nécessaires pour le problème à traiter. Par exemple, dans le contexte du projet e-Wok Hub [1] , visant à gérer la mémoire de plusieurs projets d'ingénierie sur la capture et le stockage de CO2, les experts doivent développer des ontologies pour couvrir ce domaine. Ils doivent choisir un modèle d'ontologie sachant que, d'une part, il est nécessaire de modéliser les concepts du domaine physique avec précision, c'est-à-dire en définissant leurs dimensions physiques, associées à des unités de mesure et à un contexte d'évaluation. Ce besoin

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

suggère l'utilisation d'un modèle d'ontologie tel que PLIB. D'autre part, l'utilisation de constructeurs introduits par les modèles d'ontologie issus de la logique de description, tels que OWL, apparaissent également nécessaires pour permettre d'effectuer des inférences afin d'améliorer la qualité des recherches documentaires, également essentielles pour le problème visé. Cet exemple montre l'intérêt de pouvoir manipuler le modèle d'ontologie utilisé afin de pouvoir bénéficier des apports de chaque modèle.

Dans la suite de cet article nous utilisons un exemple d'ontologie pour illustrer les principes du langage OntoQL. L'exemple d'une ontologie sur le domaine du stockage du CO2 étant trop compliqué à expliquer, nous avons choisi d'utiliser un exemple abordant le domaine plus simple des communautés en-ligne. Cet exemple, présenté à la figure 1, est inspiré de l'ontologie SIOC (http://sioc-project.org/). L'ontologie SIOC décrit le domaine des communautés en-ligne en définissant des concepts tels que `Community`, `Usergroup` ou `Forum`. Dans cet exemple, nous avons représenté les classes `User` et `Post` et les avons spécialisées par les classes `Administrator` et `InvalidPost`. La classe `InvalidPost` est définie comme une restriction OWL (`OWLAllValuesFrom`) sur la propriété `hasModifiers` dont les valeurs doivent être des instances de la classe `Administrator`. Ainsi, un post invalide est un post qui n'a été modifié que par des administrateurs. La notation UML ne permettant pas de représenter ce type de constructeur, nous avons utilisé un stéréotype associé à une note pour représenter la classe `InvalidPost`. Il est à noter également que la notation UML ne nous permet pas de montrer la description complète des classes et des propriétés de cette ontologie (par exemple, les noms synonymes ou les documents associés).

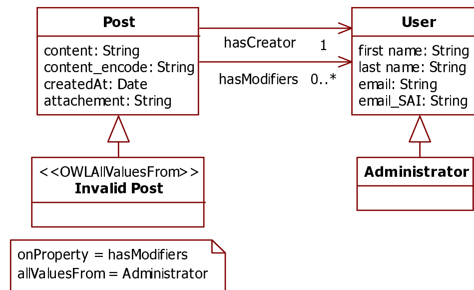


**Figure 1.** *Un extrait d'une ontologie pour les communautés en-ligne*

Dans la section suivante nous montrons les différentes structures qui ont été proposées pour stocker des ontologies dans des bases de données.

## 3. Les différentes structures de stockage des BDBO

Le schéma le plus simple permettant de stocker des ontologies dans une base de données est une table à trois colonnes `(sujet, predicat, objet)` [HAR 03]. Dans cette représentation, nommée *par triplet*, l'ensemble des informations, ontologies et données, est décomposé sous la forme de triplets. Cette représentation permet de stocker et faire évoluer le modèle d'ontologie utilisé. Cependant, les faibles performances de cette représentation ont conduit la communauté scientifique à en proposer de nouvelles [THE 05].

La seconde approche de représentation de BDBO consiste à séparer la représentation des ontologies et des données. Les données sont stockées soit par une représentation par triplet [MA 04], soit par une représentation verticale [ALE 01, BRO 02], où chaque classe est représentée par une table unaire et chaque propriété par une table binaire, soit par une représentation horizontale [PAR 07, PIE 05, DEH 07], où une table ayant une colonne pour chaque propriété est associée à chaque classe. Quelle que soit la représentation choisie pour les données, ces systèmes utilisent un modèle logique fixe dépendant du modèle d'ontologie utilisé pour stocker les ontologies.

Enfin, à notre connaissance la BDBO OntoDB [PIE 05, DEH 07] est la seule à proposer la séparation des données et des ontologies tout en proposant une structure pour stocker et faire évoluer le modèle d'ontologie utilisé (PLIB). La représentation de cette partie, nommée *méta-schéma*, permet d'adapter ce système aux évolutions du modèle d'ontologie PLIB, mais aussi, lorsque le besoin s'en fait sentir, d'étendre le modèle d'ontologie utilisé par des constructeurs provenant d'autres modèles.

Comme cette section le montre, seule la représentation par triplet ou l'ajout explicite du niveau méta-méta-modèle permet de faire évoluer dynamiquement le modèle d'ontologie utilisé dans une BDBO. Dans la section suivante, nous montrons les limites des langages usuels de BDBO pour exploiter cette capacité.

## 4. Les langages usuels des BDBO

Différents langages ont été proposés pour les BDBO. Ces langages sont fondés sur des modèles de données correspondant aux structures de représentation décrites dans la section précédente. Nous présentons dans cette section quelques-uns de ces langages en fonction du modèle d'ontologie considéré.

### 4.1. *Les langages pour RDF-Schema*

Le langage RQL [KAR 02] a été le premier langage permettant d'interroger des données et des ontologies modélisées en RDF-Schema. De nombreux autres langages présentant des caractéristiques similaires ont ensuite été proposés (voir [BAI 05] pour un état de l'art). En RQL, l'ensemble des constructeurs du modèle d'ontologie RDF-Schema sont codés dans sa grammaire comme mots clés. Par exemple, le mot clé

`domain` est une fonction permettant de retrouver le domaine d'une propriété. En conséquence, toute évolution de ce modèle nécessite de modifier la grammaire du langage.

Pour gérer la diversité des modèles d'ontologie, le modèle de données sur lequel repose le langage RQL n'est pas complètement figé. En effet, ce modèle de données peut être étendu en spécialisant les entités prédéfinies `Class` et `Property`. Cependant, il ne permet pas d'ajouter des entités si elles n'héritent pas de ces deux entités prédéfinies. Cette limitation empêche, par exemple, de représenter les constructeurs de PLIB permettant de représenter les documents décrivant les concepts d'une ontologie ou le constructeur `Ontology` qui permet, en OWL, de regrouper l'ensemble des concepts définis dans une ontologie. D'autre part, l'extension du modèle de données de RQL avec de nouveaux attributs permettant de caractériser les concepts d'une ontologie, comme par exemple, un numéro de version ou une illustration, nécessite l'utilisation du constructeur de propriétés. En conséquence, la modification du niveau méta-modèle modifie également automatiquement le niveau modèle. Enfin, même si ces capacités (partielles) sont offertes par RQL, elles ne sont pas supportées ou du moins pas explicitées sur les BDBO RDF-Suite [ALE 01] et Sesame [BRO 02] sur lesquelles il a été implanté.

### 4.2. *Les langages pour RDF*

SPARQL [PRU 06] est un langage en cours de standardisation pour des données modélisées en RDF. En conséquence, il considère l'ensemble des informations, ontologie et données, comme des triplets. Ainsi, à la différence de RQL, SPARQL ne code pas les constructeurs d'un modèle d'ontologie donné comme des mots clés de sa grammaire. Cette approche laisse une liberté totale pour représenter les données, l'ontologie et le modèle d'ontologie selon une approche par triplet. Par contre, étant donné que les constructeurs d'ontologie tels que la définition de classes et la relation de subsomption ne sont pas inclus dans ce modèle, SPARQL ne leur attribue pas de sémantique. En conséquence, SPARQL ne fournit pas d'opérateurs pour calculer la clôture transitive de la relation de subsomption pour une classe donnée ou d'opérateurs pour retrouver les propriétés applicables d'une classe, c'est-à-dire celles qui sont définies ou héritées par cette classe. Le retour d'une requête SPARQL est complètement dépendant de la base de triplets sur laquelle elle est exécutée.

### 4.3. *Les langages indépendants d'un modèle d'ontologie particulier*

Le langage SOQA-QL [ZIE 05] (projet SIRUP) est, à notre connaissance, le seul langage permettant d'interroger les ontologies et les données qu'elles décrivent indépendamment du modèle d'ontologie utilisé. Pour offrir cette capacité, ses auteurs ont basé ce langage sur un noyau de modèle d'ontologie (SOQA Ontology Meta Model) contenant les constructeurs principaux de différents modèles d'ontologie. Les auteurs ont choisi d'inclure dans ce modèle des constructeurs, mêmes si ceux-ci ne

sont pas disponibles dans certains modèles d'ontologie. Par contre, ce modèle est figé. En conséquence, les évolutions des modèles d'ontologie tout comme les spécificités non représentées dans ce modèle noyau ne peuvent pas être prises en compte par le langage SOQA-QL.

### 4.4. *La nécessité d'un nouveau langage pour les BDBO*

Comme l'analyse des trois langages précédents le montre, chaque type de langage existant présente des limitations concernant la manipulation uniforme des trois niveaux d'une BDBO. Ces limites peuvent être résumées de la façon suivante.

– Les langages du type RQL offrent des opérateurs pour exploiter la sémantique du modèle d'ontologie RDF-Schema, mais par contre, l'extension de ce modèle est limitée et mélange les différents niveaux d'une BDBO.

– Les langages du type SPARQL offrent une liberté totale de manipulation des trois niveaux d'une BDBO en considérant l'ensemble des données comme des triplets, mais par contre, ils n'offrent pas les opérateurs permettant d'exploiter la sémantique usuelle des modèles d'ontologie.

– Le langage SOQA-QL permet de manipuler les données d'une BDBO quel que soit le modèle d'ontologie utilisé pourvu que les constructeurs utilisés soit présents dans le méta-modèle sur lequel repose ce langage. Par contre les constructeurs qui ne sont pas inclus dans ce méta-modèle ne peuvent pas être ajoutés.

Cette analyse nous a conduits à proposer un nouveau langage pour les BDBO nommé OntoQL. Ce langage est décrit dans la section suivante.

## 5. Le langage OntoQL

Définir la sémantique d'un langage d'exploitation de données nécessite d'abord de définir le modèle des données traitées par le langage, puis la signification de chaque énoncé du langage en termes de ce modèle. Nous avons suivi cette démarche pour le langage OntoQL en définissant formellement le modèle de données traité ainsi que les opérateurs de ce langage par une algèbre assurant la complétude de ce langage. Pour simplifier, la description de ces éléments dans les prochaines sections est informelle.

### 5.1. *Modèle de données*

Le langage OntoQL doit satisfaire deux objectifs, (1) permettre la modification de niveau méta-modèle et (2) assurer que ces modifications respectent la sémantique des systèmes de méta-modélisation : une instance de méta-modèle (niveau appelé $\mathrm{M}_2$ dans le MOF) définit bien un modèle (niveau $\mathrm{M}_1$) lui-même destiné à représenter des populations d'instances (niveau $\mathrm{M}_0$). Ces deux objectifs sont réalisés de la façon suivante. (1) Alors que dans les SGBD usuels c'est le niveau méta-modèle qui est figé,

OntoQL suppose l'existence d'un niveau méta-méta-modèle lui même figé (niveau $\mathrm{M}_3$ du MOF). Instancier ce méta-méta-modèle permet alors de modifier le niveau méta-modèle. Ceci assure la flexibilité du niveau méta-modèle. (2) Mais, le niveau méta-modèle, contient à priori un modèle noyau prédéfini associé à une sémantique opérationnelle câblée. Nous décrivons d'abord ce modèle noyau dans la prochaine section puis décrivons la possibilité de l'étendre dans la section suivante.

#### 5.1.1. *Méta-modèle noyau*

Dans le contexte des BDBO, le méta-modèle décrit le modèle d'ontologie dans lequel les ontologies vont pouvoir être définies. Contrairement à l'approche proposée pour la conception du langage SOQA-QL, nous avons choisi de n'inclure dans ce modèle noyau que les constructeurs communs aux différents modèles d'ontologie que nous avons considérés parmi lesquels figurent les standards PLIB, RDFS et OWL. Par contre, nous avons offert la possibilité d'étendre ce modèle.

La figure 2 présente les principaux éléments de ce modèle d'ontologie noyau sous la forme simplifiée d'un modèle UML. Nous nommons *entités* et *attributs* les classes et propriétés de ce modèle UML. Le manque de place nous empêche de justifier le choix de ces constructeurs. Nous en fournissons seulement ci-dessous une description.

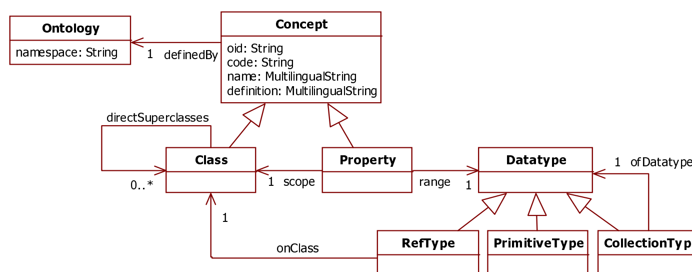


**Figure 2.** *Extrait du méta-modèle noyau*

– Une ontologie (`Ontology`) définit un domaine d'unicité des noms aussi appelé *espace de noms* (`namespace`). Elle regroupe la définition d'un ensemble de concepts qui sont des classes et des propriétés.

– Une classe (`Class`) est la description abstraite d'un ou plusieurs objets semblables. Une classe possède un identifiant interne à la BDBO (`oid`) et un identifiant indépendant de celle-ci (`code`). Sa définition comporte une partie textuelle (`name`, `definition`), qui permet de la rattacher à une connaissance préexistante de l'utilisateur. Cette partie textuelle peut être donnée dans plusieurs langues naturelles (`MultilingualString`). Ces classes sont organisées dans une hiérarchie acyclique où elles sont liées par une relation d'héritage multiple (`directSuperclasses`).

– Les propriétés (`Property`) sont des éléments qui permettent de décrire les instances d'une classe. Comme les classes, les propriétés possèdent toujours un identifiant et une partie textuelle éventuellement définie dans plusieurs langues naturelles. Chacune des propriétés doit être définie sur la classe ou sur l'une des super-classes des instances qu'elle décrit (`scope`). Chaque propriété a un co-domaine (`range`) qui permet de contraindre son domaine de valeurs.

– Le type de données (`Datatype`) d'une propriété peut être un type simple (`primitiveType`) tel que le type entier ou le type chaînes de caractères. Une propriété peut également prendre ces valeurs dans une classe en faisant référence à ses instances (`refType`). Enfin, cette valeur peut être une collection dont les éléments sont soit d'un type simple soit des références à des instances d'une classe (`collectionType`).

Soulignons que, bien qu'il ait été développé pour gérer des ontologies, ce modèle est en fait un méta-modèle standard susceptible d'être utilisé comme noyau pour de nombreuses applications nécessitant une modélisation à trois niveaux (par exemple, l'enrichissement de la méta-base d'un SGBD, la représentation d'un modèle conceptuel ou l'ingénierie dirigée par les modèles).

#### 5.1.2. *Extensions du méta-modèle noyau*

Le méta-modèle noyau peut être étendu avec de nouvelles entités et de nouveaux attributs. Lorsque cette extension est faite par spécialisation, les nouvelles entités héritent du comportement de leurs super-entités. Ce comportement est défini dans la sémantique opérationnelle du méta-modèle noyau. Ainsi, toute spécialisation de l'entité `Class` définit une nouvelle catégorie de classes, mais celles-ci supportent, par héritage, le comportement usuel d'une classe. Toute instance de l'entité `Class` (ou d'une quelconque spécialisation) définit un conteneur susceptible d'être associé à des instances. Le nom de ce conteneur est généré par une fonction dite de concrétisation qui permet d'y accéder à partir de sa représentation du niveau méta ($\mathrm{M}_2$). De même, toute spécialisation de l'entité `Property` définit des relations associant des instances de classes à des domaines de valeurs. Le nom de ces relations est également dérivé des instances de propriétés qui les définissent. Enfin, les spécialisations de l'entité `Datatype` définissent des domaines de valeurs permettant de typer des propriétés.

### 5.2. *Les constructeurs du langage OntoQL définis sur le modèle de données*

Compte tenu du modèle de données défini, la création d'un élément du niveau $\mathrm{M_i}$ se fait en instanciant le niveau $\mathrm{M_{i+1}}$. Notons qu'une telle écriture ne s'avère pas toujours très commode. En effet, cela reviendrait, dans un SGBD usuel, à créer une table en faisant des insertions dans les tables de la méta-base. Des équivalences syntaxiques sont donc définies de façon systématique entre l'insertion (`INSERT`) de niveau $\mathrm{M_{i+1}}$ et la création de conteneur (`CREATE`) de niveau $\mathrm{M_i}$. Les deux sont licites, la seconde écriture s'avérant en général plus synthétique. La figure 3 donne une vue d'ensemble des différents niveaux d'instructions disponibles ainsi que leur signification par rapport à la structure des données cibles. Comme cette figure le montre, deux syntaxes

équivalentes sont fournies d'une part pour insérer au niveau $\mathrm{M}_3$ ou créer au niveau $\mathrm{M}_2$ et, d'autre part, pour insérer au niveau $\mathrm{M}_2$ et pour créer au niveau $\mathrm{M}_1$. Nous décrivons de façon plus précise, dans les sections ci-dessous, les traitements disponibles.

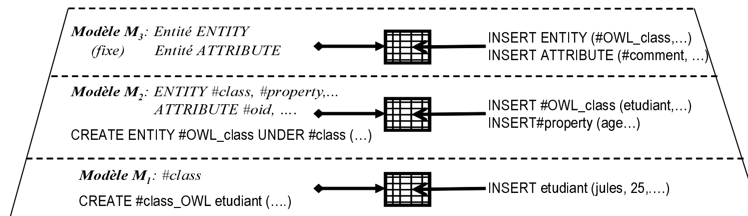


**Figure 3.** *Vue d'ensemble du langage OntoQL*

5.2.1. *Définition, manipulation et interrogation du niveau méta-modèle (ontologies)*

Le langage OntoQL permet de définir, modifier et interroger les ontologies à partir du méta-modèle noyau présenté à la section précédente. Puisque ce méta-modèle noyau n'est pas statique, mais qu'il peut être étendu, les éléments de ce niveau de modélisation ne doivent pas être codés comme des mots clés du langage OntoQL. Nous avons donc dû définir une convention permettant de reconnaître dans la grammaire un élément du méta-modèle indépendamment de son nom. La convention que nous avons choisie est de préfixer chaque élément de ce modèle par le caractère `#`. Ce préfixe permet de savoir que la définition de cet élément doit être insérée ou recherchée dans le niveau méta du système de modélisation à trois niveaux utilisé.

Notre proposition pour les BDBO étant d'enrichir les bases de données par une couche ontologique, pour définir la syntaxe du langage OntoQL, nous avons choisi d'adapter celle du langage SQL au modèle de données que nous avons défini. Le langage de définition de données permet ainsi de créer, modifier et supprimer les entités et attributs du méta-modèle en utilisant une syntaxe similaire à la manipulation de types utilisateur en SQL (`CREATE`, `ALTER`, `DROP`).

**Exemple 1.** *Ajouter le constructeur de restriction* `AllValuesFrom` *de OWL dans le modèle d'ontologie utilisé.*

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

Cette instruction permet d'ajouter l'entité `OWLAllValuesFrom` à notre méta-modèle noyau comme sous-entité de l'entité `Class`. Cette entité est créée avec les deux attributs `onProperty` et `allValuesFrom` qui prennent respectivement comme valeur des identifiants de propriétés et des identifiants de classes.

Pour créer, modifier et supprimer les éléments des ontologies, nous avons défini un langage de manipulation de données. De la même façon que pour le langage de

définition de données, nous avons adapté le langage de manipulation de données de SQL (`INSERT`, `UPDATE`, `DELETE`) au modèle de cette partie.

**Exemple 2.** *Créer la classe nommée* `InvalidPost` *de notre ontologie exemple.*

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

Dans cet exemple, nous voyons que le langage OntoQL permet d'attribuer une valeur dans différentes langues naturelles (`[en]` pour anglais et `[fr]` pour français) aux attributs multilingues. Il montre également que les noms des classes et des propriétés peuvent être utilisés pour les identifier. En effet, la valeur de l'attribut `onProperty` est définie comme étant `hasModifiers` qui permet de référencer la propriété du même nom. Notons que grâce aux équivalences syntaxiques présentées précédemment, nous aurions pu écrire cette instruction avec la syntaxe `CREATE` plus proche de celle de création de types utilisateur en SQL. Elle est également souvent plus concise, car elle permet également de créer de nouvelles propriétés et de les associer à la classe créée.

Enfin, un langage d'interrogation permet de rechercher les éléments des ontologies contenues dans la BDBO. En concevant ce langage à partir de SQL, nous profitons de l'expressivité des opérateurs orientés-objet introduits dans ce langage depuis la norme SQL99. Ainsi, le langage OntoQL permet, en particulier, d'exprimer des expressions de chemin, de grouper/dégrouper des collections ou d'utiliser les opérateurs d'agrégats. L'exemple suivant montre l'utilisation d'une expression de chemin.

**Exemple 3.** *Rechercher le nom des restrictions portant sur la propriété* `hasModifiers`. *Retourner également le nom de la classe dans laquelle les valeurs de cette propriété doivent être prises.*

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

Cette requête consiste en une sélection et une projection. La sélection permet de retrouver les restrictions portant sur la propriété dont le nom en anglais est `hasModifiers`. L'expression de chemin utilisée dans cette sélection est composée de l'attribut `onProperty` qui permet de retrouver l'identifiant de la propriété sur laquelle porte la restriction et de l'attribut `name` qui permet de retrouver le nom anglais de cette propriété à partir de son identifiant. La projection applique également l'attribut `name` pour retrouver le nom de la restriction et applique l'expression de chemin composée des attributs `allValuesFrom` et `name` pour retrouver le nom de la classe dans laquelle la propriété impliquée dans la restriction doit prendre ses valeurs.

#### 5.2.2. *Définition, manipulation et interrogation du niveau modèle (données)*

A ce niveau, dans une BDBO, le langage OntoQL permet de définir, manipuler et interroger les instances des ontologies. Pour conserver une syntaxe uniforme et une compatibilité avec les modèles de bases de données existants, nous avons également choisi, à ce niveau, de partir du langage SQL. Ainsi, chaque classe peut être liée à une *extension* qui stockent les instances de cette classe ainsi que leur caractérisation soit

au sein d'une table, soit au sein d'une vue. Ce modèle de données généralise le lien existant entre un type utilisateur en SQL et une table typée qui en stocke les instances. En effet, contrairement à un schéma de base de données qui prescrit les attributs qui caractérisent les instances d'un type utilisateur, une ontologie décrit les propriétés qui peuvent être utilisées pour décrire les instances d'une classe. En conséquence, l'extension d'une classe ne comprend que le sous-ensemble des propriétés qui sont effectivement utilisées pour en décrire les instances. La conséquence de cette particularité est que la syntaxe du langage OntoQL pour créer une extension de classe est similaire à l'instruction SQL de création d'une table typée à partir d'un type utilisateur, à la différence près qu'elle permet de préciser l'ensemble des propriétés utilisées.

**Exemple 4.** *Créer l'extension de la classe* `Administrator` *sachant qu'un administrateur n'est décrit que par son nom et son prénom.*

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

Cette instruction provoque la création d'une table à deux colonnes pour stocker les instances de la classe `Administrator` sachant que seules les propriétés `first name` et `last name` les décrivent. Notons que cette structure de stockage peut être une vue sur un ensemble de tables normalisées.

La sémantique du langage OntoQL a été définie pour permettre de rechercher la valeur d'une instance pour n'importe quelle propriété définie sur la classe de cette instance. Lorsque cette propriété n'est pas utilisée pour décrire cette instance, la valeur `NULL` est retournée. Nous avons attribué à la valeur `NULL` la même sémantique que celle définie dans la norme SQL.

**Exemple 5.** *Rechercher les utilisateurs dont l'adresse e-mail est connu.*

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

La requête précédente ne retournera aucun administrateur puisque la propriété `email` n'est pas utilisée sur cette classe.

Une autre particularité du modèle de données adressé est que les classes et propriétés sont regroupées dans un espace de noms. En conséquence, pour référencer ces éléments dans une instruction OntoQL, il est nécessaire d'indiquer l'espace de noms dans lequel ils doivent être recherchés. Ceci est possible dans la clause `USING NAMESPACE` d'une instruction OntoQL. Si cette clause n'est pas indiquée et qu'aucun espace de noms n'est défini par défaut, OntoQL interprète l'instruction comme une commande SQL. Le langage OntoQL est ainsi compatible avec le langage SQL. L'exemple suivant illustre l'utilisation des espaces de noms dans OntoQL.

**Exemple 6.** *Rechercher le contenu des posts invalides sachant que la classe* `InvalidPost` *est définie dans l'espace de noms* `http://www.lisi.ensma.fr`.

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

Dans cette exemple, la classe `InvalidPost` et la propriété `content` sont recherchées dans l'ontologie qui définit l'espace de noms `http://www.lisi.ensma.fr`. OntoQL permet de spécifier plusieurs espaces de noms dans la clause `USING NAMESPACE`. Ceci permet de faire des requêtes composées d'éléments de différentes ontologies.

Le prochain exemple illustre une troisième particularité du méta-modèle noyau. A savoir, le fait que les classes et propriétés ont une définition textuelle pouvant être donnée dans plusieurs langues naturelles. Le langage exploite cette capacité en permettant de référencer chaque classe et chaque propriété par un nom dans une langue naturelle donnée. Ceci permet d'écrire la même requête dans plusieurs langues naturelles.

**Exemple 7.** *Rechercher le nom et le prénom des utilisateurs en écrivant la requête en anglais et en français.*[2]

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

La requête `7a` doit être exécutée lorsque la langue par défaut de OntoQL est l'anglais tandis que la requête `7b` requiert que sa valeur par défaut soit le français.

#### 5.2.3. *Interrogation conjointe des niveaux méta-modèle et modèle*

Pouvoir interroger à la fois les ontologies et les données est particulièrement utile, par exemple, pour retrouver les descriptions ontologiques de l'ensemble des classes auxquelles appartient une instance. En conséquence, cette capacité est une des exigences pour le langage OntoQL.

Les requêtes sur les ontologies prennent en paramètre d'entrée et/ou de sortie une collection d'éléments du niveau méta (classe, propriété . . .) tandis que les requêtes sur les données prennent en paramètre d'entrée et/ou de sortie une collection d'instances de ces éléments (instances de classes et valeurs de propriétés). Le lien entre ces deux niveaux se fait donc dans les deux sens suivants décrits dans les prochaines sections. (1) Ontologie vers données. A partir de classes, récupérées par une requête sur les ontologies, nous pouvons obtenir les instances rattachées à ces classes. Nous pouvons alors effectuer une requête sur les données à partir de ces collections d'instances. (2) Données vers ontologie. A partir d'instances, récupérées par une requête sur les données, nous pouvons obtenir leurs classes d'appartenance. Nous pouvons alors effectuer une requête sur les ontologies à partir de ces collections de classes.

##### 5.2.3.1. Ontologie vers données

Pour interroger les données, suivant l'approche des langages usuels des bases de données, OntoQL propose d'introduire un itérateur `i` sur les instances d'une classe `C` par la syntaxe `C as i`. Dans ces requêtes, la classe `C` est connue avant l'exécution de la requête. OntoQL étend ce mécanisme pour permettre d'introduire un itérateur sur les instances d'une classe déterminée pendant l'exécution de la requête.

**Exemple 8.** *Rechercher les identifiants des instances des classes dont le nom en français se termine par* `Post`.

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

Dans cette requête, les classes `Post` et `InvalidPost` respectent la condition de sélection. En conséquence, cette requête retourne les identifiants des instances de ces

2. Dans les exemples suivants, l'espace de noms par défaut est `http://www.lisi.ensma.fr`

deux classes. Et, puisque `InvalidPost` est une sous-classe de la classe `Post`, les identifiants des instances de la classe `InvalidPost` sont retournés en double.

5.2.3.2. Données vers ontologie.

OntoQL propose l'opérateur `typeOf` pour retrouver la *classe de base* d'une instance, c'est-à-dire la classe minorante pour la relation de subsomption des classes auxquelles elle appartient. Cet opérateur est appelé avec une notation fonctionnelle pour le distinguer d'une propriété.

**Exemple 9.** *Rechercher le nom en anglais de la classe de base des instances de la classe* `User`.

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

Cette requête parcourt les instances de la classe `User` de la BDBO et donc également celles de la classe `Administrator`. Pour chacune de ces instances, la requête retourne `User` ou `Administrator` suivant la classe dont elle est membre.

### 5.3. *Exploitation des capacités du langage OntoQL*

Le langage OntoQL, présenté dans les sections précédentes, offre des opérateurs pour calculer l'extension d'un concept ou faire des transformations entre différents modèles d'ontologie. Dans cette section, nous donnons un aperçu de ces capacités.

Le langage OntoQL permet de construire l'extension d'une classe comme une vue. Cette capacité peut être utilisée pour calculer l'extension d'un *concept défini*, c'est-à-dire un concept dont les conditions nécessaires et suffisantes de reconnaissance en termes d'autres concepts de l'ontologie sont fournies.

**Exemple 10.** *Construisez l'extension de la classe* `InvalidPost` *à partir des instances de la classe* `Post`.

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

La requête principale de cette instruction parcourt l'ensemble des instances `p` de la classe `Post`. Pour chacune de ces instances, une sous-requête permet de savoir si ce post n'a été modifié que par des administrateurs. Pour cela, l'opérateur `UNNEST` est utilisé afin de dégrouper la collection d'utilisateurs ayant modifié ce post. L'itérateur `m` parcourt chacun des identifiants de cette collection et regarde, via la négation de l'opérateur `IS OF`, si il référence une instance de la classe `Administrator`.

Le langage OntoQL peut être également utilisé pour transformer une ontologie représentée dans un modèle d'ontologie source en une ontologie d'un modèle d'ontologie cible. Ceci nécessite de coder avec les opérateurs OntoQL, lorsque c'est possible, les équivalences définies entre le modèle source et le modèle cible [FAN 07].

**Exemple 11.** *Transformer les classes OWL en classes PLIB sachant que l'attribut* `comment` *en OWL est équivalent à l'attribut* `remark` *de PLIB.*

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

Cette instruction insère une classe PLIB pour chaque classe OWL. La valeur de l'attribut `remark` des classes PLIB créées est renseignée avec la valeur de l'attribut `comment` des classes OWL sources.

## 6. Implantation

Nous avons implanté le langage OntoQL, présenté dans la section précédente, sur la BDBO OntoDB. Nous présentons cette implantation dans la prochaine section. Nous décrivons ensuite les outils développés autour de ce langage.

### 6.1. *Implantation du moteur OntoQL*

Le premier problème auquel nous avons été confronté pour implanter OntoQL sur OntoDB était de choisir une méthode pour réaliser le traitement d'une requête. Deux approches ont été suivies pour implémenter un langage sur une BDBO. La première consiste à traduire une requête du langage source en une requête SQL spécifique à la BDBO. Cette approche a été suivie dans les systèmes RDF-Suite [ALE 01] et RStar [MA 04]. La seconde consiste à traduire une requête du langage source en une suite d'appels d'une API (Application Programming Interface) dont chaque méthode retourne un ensemble de données de la BDBO. Cette approche a été suivie dans le système Sesame [BRO 02].

La première solution présente l'avantage de profiter de l'important travail qui a été mené sur l'optimisation des moteurs SQL des bases de données. Elle présente cependant l'inconvénient d'être dépendante de la BDBO sur laquelle le langage est implanté. La seconde permet la portabilité sur différentes BDBO puisqu'il est possible de fournir une implémentation de l'API pour chaque BDBO. Cependant, dans ce cas, l'optimisation des requêtes est à la charge du moteur du langage implanté.

Dans notre implantation, nous avons allié ces deux méthodes. Ainsi, le langage OntoQL est traduit en une requête SQL dépendante de OntoDB. Mais, cette traduction est réalisée en passant par une API qui encapsule les spécificités de cette BDBO. La traduction suit ainsi les 3 étapes suivantes : (1) génération de l'arbre des opérateurs de l'algèbre d'OntoQL à partir d'une requête textuelle (2) transformation de cet arbre en un arbre utilisant les opérateurs de l'algèbre relationnelle en passant par une API (3) génération de requêtes SQL à partir de cet arbre selon le SGBD utilisé.

Le second problème posé par cette implantation était de permettre, en particulier, l'utilisation complète du modèle d'ontologie PLIB utilisé dans OntoDB. Pour cela, nous avons utilisé les capacités d'extension du méta-modèle noyau de OntoQL. L'utilisation de ces capacités à nécessité trois étapes : (1) représenter le méta-modèle noyau de OntoQL dans la partie méta-schéma de OntoDB (2) établir les correspondances

entre ce méta-modèle noyau et le modèle PLIB et (3) étendre le méta-modèle noyau avec les constructeurs spécifiques de ce modèle d'ontologie.

### 6.2. *Outils permettant l'exploitation du moteur OntoQL*

Pour faciliter l'utilisation du langage SQL, de nombreux outils ont été développés. Ayant le même objectif pour OntoQL, nous avons également développé des outils similaires à ceux existant pour SQL. Dans cette section, nous décrivons ces outils.

*OntoQL*Plus* est un éditeur d'instructions OntoQL en ligne de commande similaire à SQL*Plus fourni par Oracle ou isql fourni par SQLServer. Il propose une coloration syntaxique des commandes OntoQL et conserve l'historique des commandes exécutées.

*OntoQBE* est un éditeur graphique de requêtes OntoQL fourni en tant que plug-in pour le logiciel PLIBEditor qui permet de visualiser et d'éditer des ontologies PLIB. La figure 4 montre l'interface proposée. Elle étend l'interface à la QBE, telle que celle fournie par le logiciel Access, pour prendre en compte les aspects orientés-objets du langage OntoQL tels que les expressions de chemin (TAG1) et le polymorphisme (TAG2) ainsi que ses aspects ontologiques comme par exemple la description (illustration, code version ...) des propriétés utilisées dans la requête (TAG3).

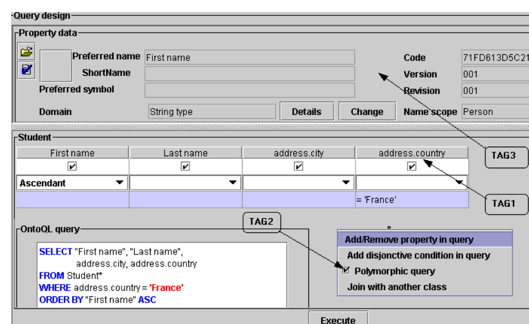


**Figure 4.** *OntoQBE : un éditeur graphique de requête OntoQL à la QBE*

*OntoAPI* est une représentation JAVA du modèle noyau considéré et présenté figure 2. Elle contient, entre autres, les interfaces `EntityClass` et `EntityProperty` représentant les classes et les propriétés. Lorsque de nouveaux éléments sont ajoutés au modèle noyau, cette API peut être regénérée automatiquement pour les prendre en compte. Cette API implante le concept de chargement à la demande ou chargement paresseux proposé, par exemple, dans le framework hibernate (www.hibernate.org). Le chargement à la demande consiste à ne charger un objet à partir de la base de données que lorsqu'un utilisateur y accède via un accesseur. Ainsi, une classe manipulée par l'interface `EntityClass` est initialement chargée avec son nom, son code, sa version ... Puis, lorsque la méthode `getProperties` est appelée, ses propriétés sont

chargées par l'appel d'une requête OntoQL de manière transparente pour l'utilisateur.

*JOBDBC* est une autre API permettant d'exécuter des requêtes OntoQL depuis le langage de programmation JAVA. Cette interface étend l'API JDBC. Par exemple, l'interface `OntoQLResultSet` étend `ResultSet` en proposant des méthodes telles que `getEntityClass` ou `getEntityProperty` pour permettre d'accéder à un élément du résultat d'une requête dont le type est une classe d'OntoAPI. De même, l'interface `OntoQLResultSetMetaData` étend `ResultSetMetaData` pour permettre d'obtenir la description ontologique (nom en plusieurs langues, définition, illustration . . .) d'une colonne d'un `OntoQLResultSet` qui référence une propriété.

L'ensemble de ces outils permet d'utiliser le langage OntoQL de manière similaire à ce qui est actuellement proposé par les SGBD pour le langage SQL. Des démonstrations et captures d'écrans sur ces outils sont disponibles à l'adresse *http://www.plib.ensma.fr/plib/demos/ontodb/index.html*.

## 7. Conclusion

Dans cet article, nous avons présenté le langage OntoQL permettant de définir, manipuler et interroger les données d'une BDBO. Même si notre présentation a été illustrée sur les BDBO, ce langage est également adapté à d'autres systèmes de méta-modélisation tels que les bases de données traditionnelles. Notre travail est motivé par l'absence de langage qui permette de manipuler uniformément les trois niveaux d'une telle structure. Le besoin d'un tel langage se fait ressentir lorsque les méta-modèles sont variés et évolutifs. C'est notamment le cas des modèles d'ontologie.

Le résultat de ce travail est un langage qui s'appuie sur un méta-modèle noyau, contenant les constructeurs communs à la plupart des méta-modèles usuels et en particulier aux différents modèles d'ontologie, qui peut être modifié pour prendre en compte les spécificités d'un modèle d'ontologie particulier. Ce langage a été conçu en étendant le modèle de données, la syntaxe et la sémantique du langage SQL. Il exploite les particularités des BDBO pour permettre l'interrogation des données dans plusieurs langues naturelles. Nous avons implanté ce langage sur la BDBO OntoDB et développé des outils similaires à ceux existants pour les bases de données traditionnelles permettant d'exploiter ce langage. Des démonstrations de ces outils sont accessibles sur *http://www.plib.ensma.fr/plib/demos/ontodb/index.html*.

Les travaux sur OntoQL menés actuellement consistent à étudier de nouveaux mécanismes permettant d'étendre notre méta-modèle noyau sans avoir à coder manuellement toute la sémantique des opérateurs ajoutés. Nous travaillons actuellement à la possibilité d'ajouter des types utilisateurs, à la manière de SQL99, auxquels des primitives, comme par exemple, les opérateurs de comparaison, seraient associées.

**Remerciements.** Ce travail a été partiellement supporté par le projet ANR *e-WokHub* (ANR05RNTL02706).

## 8. Bibliographie

[ALE 01] ALEXAKI S., CHRISTOPHIDES V., KARVOUNARAKIS G., PLEXOUSAKIS D., TOLLE K., « The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. », *Proceedings of the Second International Workshop on the Semantic Web (SemWeb'01)*, 2001.

[BAI 05] BAILEY J., BRY F., FURCHE T., SCHAFFERT S., « Web and Semantic Web Query Languages: A Survey. », *Reasoning Web*, 2005, p. 35-133.

[BRI 04] BRICKLEY D., GUHA R. V., « RDF Vocabulary Description Language 1.0: RDF Schema », World Wide Web Consortium, 2004.

[BRO 02] BROEKSTRA J., KAMPMAN A., VAN HARMELEN F., « Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema », *Proceedings of the First International Semantic Web Conference (ISWC'02)*, July 2002, p. 54–68.

[DEA 04] DEAN M., SCHREIBER G., « OWL Web Ontology Language Reference », World Wide Web Consortium, 2004.

[DEH 07] DEHAINSALA H., PIERRA G., BELLATRECHE L., « OntoDB: An Ontology-Based Database for Data Intensive Applications », *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07) (To appear)*, 2007.

[FAN 07] FANKAM C., AÏT-AMEUR Y., PIERRA G., « Exploitation of Ontology Languages for both Persistence and Reasoning Purposes: Mapping PLIB, OWL and Flight ontology models », *Proceedings of the Third International Conference on Web Information Systems and Technologies (WEBIST'07) (To Appear)*, 2007.

[HAR 03] HARRIS S., GIBBINS N., « 3store: Efficient bulk RDF Storage. », *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems(PPP'03)*, 2003.

[ISO 98] ISO13584-42, « Industrial Automation Systems and Integration. Parts Library Part 42. Description methodology: Methodology for Structuring Parts families », rapport, 1998, International Standards Organization.

[KAR 02] KARVOUNARAKIS G., ALEXAKI S., CHRISTOPHIDES V., PLEXOUSAKIS D., SCHOLL M., « RQL: a declarative query language for RDF. », *Proceedings of the Eleventh International World Wide Web Conference*, 2002, p. 592-603.

[KIF 89] KIFER M., LAUSEN G., « F-Logic: A Higher-Order language for Reasoning about Objects, Inheritance, and Scheme. », *Proceedings of the 1989 ACM SIGMOD international conference on Management of data (SIGMOD'89)*, 1989, p. 134-146.

[MA 04] MA L., SU Z., PAN Y., ZHANG L., LIU T., « RStar: an RDF storage and query system for enterprise resource management », *Proceedings of the thirteenth international conference on Information and knowledge management (CIKM'04)*, 2004, p. 484–491.

[Obj02] Object Management Group, « Meta Object Facility (MOF), formal/02-04-03 », 2002.

[PAR 07] PARK M. J., LEE J. H., LEE C. H., LIN J., SERRES O., CHUNG C. W., « An Efficient and Scalable Management of Ontology », *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, 2007.

[PIE 05] PIERRA G., DEHAINSALA H., AÏT-AMEUR Y., BELLATRECHE L., « Base de Données à Base Ontologique : principes et mise en œuvre. », *Ingénierie des Systèmes d'Information*, vol. 10, n° 2, 2005, p. 91–115, Lavoisier.

[PRU 06] PRUD'HOMMEAUX E., SEABORNE A., « SPARQL Query Language for RDF », World Wide Web Consortium, 2006.

[THE 05] THEOHARIS Y., CHRISTOPHIDES V., KARVOUNARAKIS G., « Benchmarking Database Representations of RDF/S Stores. », *Proceedings of the Fourth International Semantic Web Conference (ISWC'05)*, 2005, p. 685-701.

[ZIE 05] ZIEGLER P., STURM C., DITTRICH K. R., « Unified Querying of Ontology Languages with the SIRUP Ontology Query API. », *Proceedings of Business, Technologie und Web (BTW'05)*, 2005, p. 325–344.