Contribution à la définition d'une méthode de conception de bases de données à base ontologique

Chedlia Chakroun

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Laboratoire d'Informatique et d'Automatique pour les Systèmes

THESE

pour l'obtention du Grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AEROTECHNIQUE

(Diplôme National — Arrêté du 7 août 2006)

Ecole Doctorale : Sciences et Ingénierie pour l'Information, Mathématiques
Secteur de Recherche : INFORMATIQUE ET APPLICATIONS

Présentée par :

Chedlia CHAKROUN

****

Contribution à la définition d'une méthode de conception de bases de données à base ontologique

****

Directeurs de Thèse : **Yamine AIT-AMEUR** et **Ladjel BELLATRECHE**

Soutenue le 2 Octobre 2013
devant la Commission d'Examen

JURY

Rapporteurs :	**Djamal BENSLIMANE**	Professeur, Université Claude Bernard / LIRIS, Lyon
	Arnaud GIACOMETTI	Professeur, Université François – Rabelais / LI , Tours
Examinateurs :	**Claude GODART**	Professeur, Université de Lorraine / LORIA, Nancy
	Sofian MAABOUT	Maître de conférences, Université Bordeaux 1 / LaBRI, Bordeaux
	Yamine AIT-AMEUR	Professeur, INPT-ENSEEIHT / IRIT, Toulouse
	Ladjel BELLATRECHE	Professeur, ISAE-ENSMA / LIAS, Futuroscope

Merci à

Yamine AIT-AMEUR, mon directeur de thèse, pour m'avoir guidée pendant ces années de thèse. Je lui suis également reconnaissante pour le temps conséquent qu'il m'a accordé, ses qualités pédagogiques et scientifiques, sa franchise et sa sympathie. J'ai beaucoup appris à ses côtés et je lui adresse ma gratitude pour tout cela ;

Ladjel BELLATRECHE, mon co-directeur de thèse, qui a guidé mes premiers pas dans le monde de la recherche et sans qui ce travail n'aurait jamais vu le jour. Je le remercie d'avoir eu la gentillesse de me transmettre un peu de son savoir-faire ;

Djamal BENSLIMANE et **Arnaud GIACOMETTI** de m'avoir fait l'honneur d'être rapporteurs de cette thèse et à qui je souhaite exprimer ma profonde reconnaissance ;

Claude GODART et **Sofian MAABOUT** pour avoir accepté d'être membres du jury en tant qu'examinateurs. Je suis très honorée de l'intérêt qu'ils ont porté à mes travaux ;

Guy PIERRA, **Yamine AIT-AMEUR** et **Emmanuel GROLLEAU**, directeurs du LISI et du LIAS, pour m'avoir accueillie au sein du laboratoire ;

tout le personnel du LIAS pour leur présence et leur soutien cordial. Je pense en particulier à **Frédéric CARREAU** et **Claudine RAULT** pour leurs aides techniques et administratives respectives ;

mes ami(e)s **Yassine**, **Youness**, **Houcem**, **Ameur**, **Wiem**, **Emeni**, **Amira**, **Hana**, **Youcef**, **Abir**, **Hallouma**, **Ilyes**, **Valéry**, **Selma**, **Ahcene**, **Idir**, **Nacima**, **Dilek**, **Loé**, **Christian** et **François** pour leurs encouragements et leur bonne humeur.

mon père, ma mère, mon frère et ma sœur pour leur soutien au cours de ces années et sans lesquels je n'en serais pas là aujourd'hui.

Assem, mon mari, qui a tout fait pour m'aider, qui m'a soutenue et surtout supportée dans tout ce que j'ai entrepris.

A tous ceux qui me sont chers :

mon père, ma mère,

mon frère, ma sœur,

mes ami(e)s,

Assem.

Table des matières

Partie II Contributions

Introduction générale

Contexte et problématique

Les ontologies ont été introduites dans les systèmes d'information comme des modèles de connaissances qui fournissent des définitions et des descriptions des concepts d'un domaine d'application cible. Elles ont connu un réel succès dans différents domaines (Web sémantique, ingénierie, e-commerce, etc.). Par conséquent, la quantité de données ontologiques manipulées et générées par différentes applications dans divers domaines a considérablement augmentée. Ceci est principalement dû à trois principaux facteurs : (i) le développement des ontologies de domaine, (ii) la disponibilité d'outils logiciels pour la construction, l'édition et le déploiement des ontologies et (iii) l'existence d'un ensemble de formalismes de modèle d'ontologies (OWL, PLIB, etc) qui ont considérablement contribué à l'émergence des applications basées sur les ontologies. L'augmentation du volume des données ontologiques a rendu leur gestion difficile voire incompatible avec le traitement en mémoire centrale. Face à cette situation, la gestion, le stockage, l'interrogation de ces données et le raisonnement sur eux a nécessité le développement d'outils logiciels capables de gérer ces activités. En outre, en raison de la large utilisation de ces données, la nécessité d'un processus de développement systématique et cohérent gérant à la fois la persistance, l'évolutivité et la performance des applications est apparue. Par conséquent, un nouveau type de bases de données, nommé bases de données à base ontologique ($\mathcal{BDBO}$) est défini.

Une $\mathcal{BDBO}$ est une base de données dédiée au stockage et à la gestion à la fois des ontologies et des données ontologiques référencées par ces modèles au sein de la même base de données. Plusieurs architectures de $\mathcal{BDBO}$ ont été proposées dans la littérature. Chaque $\mathcal{BDBO}$ admet une implémentation adoptant une représentation codée en dur (schéma ad-hoc ou table de triplets) pour (a) la représentation des concepts ontologiques et (b) pour la représentation des instances. Le modèle logique, prédéfini, est représenté automatiquement dans la $\mathcal{BDBO}$ cible. Les données à base ontologique sont chargées directement dans les tables figées qui leur sont associées. Ces tables sont créées suivant des règles figées par l'implémentation du système. Aucune méthodologie de conception de $\mathcal{BDBO}$ n'est proposée. Par conséquent, des anoma-

lies de conception peuvent être détectées dans la base de données. La redondance est tolérée dans la majorité des $\mathcal{BDBO}$ et des données inconsistantes peuvent être définies sans le moindre contrôle. Ainsi, tel qu'illusté dans la figure 1.(A), le cycle de vie des $\mathcal{BDBO}$ se résume en un chargement direct de l'ontologie et ses instances dans un modèle de stockage figé sans tenir compte d'aucune approche de conception.

Les principales études sur les $\mathcal{BDBO}$ ont été principalement concentrées sur la phase de modélisation physique des $\mathcal{BDBO}$. Les phases conceptuelle et logique ont été partiellement traitées. Cette situation est semblable à ce que nous avons vu lors de l'élaboration de la technologie de base de données. En effet, dans les années 70, quand le Professeur Edgar Frank Codd [Codd, 1970] a proposé son modèle relationnel, le cycle de vie des base de données ne comptait que deux phases: les phases logique et physique. Dans la phase logique, le schéma de base de données peut être normalisé en utilisant les dépendances fonctionnelles définies sur les propriétés afin de réduire la redondance et assurer la qualité du schéma final de la base de données. En 1975, quand Peter Chen [Chen, 1975] a proposé son modèle entité-association, il a contribué à son tour à prolonger le cycle de vie des bases de données à travers l'ajout de la phase de modélisation conceptuelle. Ainsi, grâce au travail de Chen, la technologie des bases de données a pu avoir son cycle de vie. De plus, en fournissant un cadre commun, le modèle relationnel s'est avéré d'une grande valeur en tant que véhicule pour la recherche et pour la communication parmi les chercheurs. Il a été utilisé dans plusieurs domaines tels que l'architecture des systèmes de base de données, les machines de base de données, la théorie de la concurrence, la mise à jour des vues, la décomposition des requêtes (en particulier dans les systèmes distribués) et l'équivalence des données. Le modèle relationnel a été mis en œuvre dans un certain nombre de SGBD. Plusieurs méthodes de conception ont été proposées en se basant sur le cycle de vie des bases de données relationnelles. On peut citer la méthode MERISE [Rochfeld, 1986], Rational Unified Process [1], etc. Ces méthodologies ont été supportées par des outils industriels et académiques tels que PowerAMC [2] et Rational Rose [3] ce qui a largement participé au succès qu'ont connu les bases de données relationnelles. Ainsi, en se basant sur ces propos, et afin de garantir un succès similaire au celui connu par les bases de données relationnelles, les $\mathcal{BDBO}$ doivent développer une certaine maturité à travers le développement (a) d'une (des) méthodologie(s) de conception incluant les principales phases d'un cycle de vie de base de données et (b) la mise à disposition d'un ensemble d'outils d'aide à la conception traitant les différentes étapes d'une telle méthodologie. Celle-ci devrait identifier la redondance intégrée dans l'ontologie et minimiser sa représentation.

Néanmoins, aucune méthodologie de conception de $\mathcal{BDBO}$ n'a été définie. C'est l'objectif principal des travaux présentés dans cette thèse.

Plus précisément, ce travail de thèse vise principalement à proposer une approche pour concevoir une $\mathcal{BDBO}$ à partir d'une ontologie de domaine qui:

1. http://www-01.ibm.com/software/rational/rup/
2. `http://www.sybase.com/products/modelingdevelopment/poweramc`
3. `http://www-01.ibm.com/software/awdtools/developer/rose/`

– s'appuie sur les méthodologies de conception des bases de données relationnelles et les théories associées : la théorie des dépendances et la théorie de la normalisation ;
– exploite les richesses des ontologies pour s'appuyer sur de telles théories ;
– génère un modèle logique de données normalisé ;
– propose aux $\mathcal{BDBO}$ un premier cycle de vie ;
– améliore la qualité de la donnée représentée dans la base de données cible ;
– redonne au concepteur des $\mathcal{BDBO}$ sa place initiale dans le processus de conception.

De plus, vue la diversité des architectures des $\mathcal{BDBO}$ et des modèles de stockage adaptés par ces dernières, cette méthodologie de conception doit être accompagnée par une approche permettant son déploiement dans les différents types de $\mathcal{BDBO}$. Cette approche a pour objectif d'offrir à la personne en charge de cette tâche un ensemble de modèles de déploiement lui permettant de choisir la solution la plus pertinente.

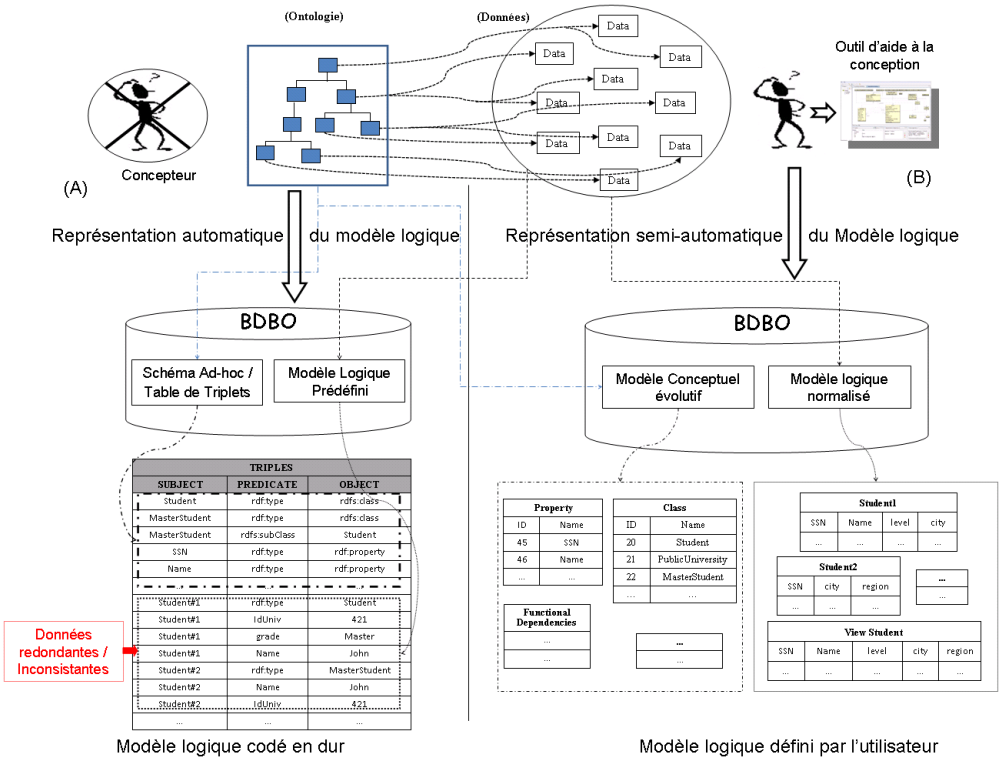

FIGURE 1 – Présentation générale de notre proposition

Notre proposition

A travers cette thèse, nous proposons une méthodologie de conception et de déploiement dédiée à la persistance des ontologies et de leurs instances dans une base de données à base

ontologique qui répond aux objectifs précédents. Notre approche suppose l'existence d'une ontologie de domaine (globale) intégrant les dépendances entre ses concepts (dépendances entre propriétés et dépendances entre classes). Ces relations de dépendance sont en partie identifiées à partir des définitions des concepts ontologiques. Notre approche est inspirée des méthodologies définies pour la conception des bases de données relationnelles. Elle admet trois phases de modélisation: conceptuelle, logique et physique. La première phase consiste à définir, à partir de l'ontologie de domaine choisie, une ontologie locale jouant le rôle de modèle conceptuel. A travers cette phase, le concepteur intègre le processus de conception. La phase de modélisation logique permet de définir un modèle logique de données en troisième forme normale en s'appuyant sur les théories des dépendances et de la normalisation et leur adaptation aux concepts ontologiques. Afin de préserver la complétude du cycle de vie des bases de données à base ontologique, nous proposons une solution de déploiement des bases de données à base ontologique. Différentes options de déploiement sont identifiées selon deux principaux facteurs: (1) les architectures des $\mathcal{BDBO}$ et (2) les modèles de stockage adaptés pour la persistance des concepts et des données ontologiques. Dans le but de faciliter l'application de notre approche et d'accompagner le concepteur dans la phase de modélisation, un outil d'aide à la conception, implémentant notre méthodologie, est proposé. La figure 1.(B) illustre le schéma global de notre proposition.

Structure du mémoire

Ce mémoire est organisé en deux parties.

La première partie comporte deux chapitres décrivant les motivations pour la définition d'une méthodologie de conception et de déploiement de bases de données à base ontologique.

Le chapitre 1 introduit la génération des ontologies. Une analyse du concept d'ontologie dans une perspective d'utilisation pour les bases de données est présentée. Après avoir introduit le concept d'ontologie, nous présentons ses principaux modèles de définition, ses éditeurs ainsi que ses principaux modèles formels. Ensuite, nous présentons une classification des ontologies à travers un modèle en couches, nommé modèle en oignon, permettant la combinaison des différentes catégories d'ontologies. Enfin, nous étudions les travaux qui explicitent les dépendances fonctionnelles au niveau ontologique.

Le chapitre 2 expose le besoin de stockage des données à base ontologique et propose une nouvelle génération de bases de données dédiée au stockage et à la manipulation des ontologies appelées bases de données à bases ontologique ($\mathcal{BDBO}$). Une étudie détaillée d'une panoplie de $\mathcal{BDBO}$ existantes est proposée. A travers cette étude, nous proposons une classification de ces systèmes de stockage selon deux principaux critères (les architectures adaptées et les modèles de stockages utilisés) ainsi que l'identification de leurs limites quant au processus de persistance des données.

Les résultats obtenus au cours de cette analyse sont utilisés dans la conclusion de ce chapitre pour dégager la nécessité d'enrichir le cycle de vie des bases de données à base ontologique par la proposition d'une méthodologie de conception et de déploiement des *BDBO* inspirée du processus de conception des bases de données relationnelles.

A partir des constats identifiés précédemment, la deuxième partie expose notre contribution à la définition d'une méthode de conception et de déploiement des *BDBO*.

Le chapitre 3 présente une étude approfondie des ontologies montrant l'importance de ses caractéristiques inexploitées lors du processus de conception des *BDBO*. Nous montrons que des relations de dépendance entre les concepts ontologiques peuvent être définies et réutilisées lors de la modélisation logique. Ensuite, nous présentons le rôle du concepteur dans le processus de conception des bases de données et l'impact de son absence dans le cycle de vie des *BDBO*. Puis, nous présentons les grandes lignes de nos contributions à travers la présentations générale de notre processus de conception des bases de données à base ontologique exploitant ainsi les relations ontologiques identifiées.

Le chapitre 4 présente la modélisation des dépendances ontologiques entre les propriétés de données et leur exploitation dans le processus de conception des *BDBO*. Nous proposons une première approche de conception de *BDBO* exploitant ce type de dépendance et nous présentons sa mise en œuvre dans un environnement de bases de données à base ontologique.

Le chapitre 5 présente la modélisation des dépendances ontologiques entre les classes et leur exploitation dans le processus de conception des *BDBO*. Nous proposons une approche de conception de *BDBO* exploitant ces dépendances pour l'identification des classes ontologiques selon leur type et l'association du traitement spécifique à chacun de ces types de classes afin d'améliorer la qualité des données représentées dans la base de données cible. Ensuite, nous présentons sa mise en œuvre dans un environnement de base de données à base de modèles.

Le chapitre 6 présente l'enrichissement de la méthodologie de conception proposée par un mécanisme de placement de classes dans la hiérarchie de subsomption et la proposition d'une approche de déploiement de la méthodologie définie dans un environnement de bases de données à base de modèles.

Le chapitre 7 est consacré aux développements menés autour du langage OntoQL (langage pour OntoDB) pour la validation de notre méthodologie sous OntoDB et la présentation de notre outil d'aide à la conception des *BDBO*.

Nous terminons cette thèse par une conclusion générale et par un exposé des perspectives ouvertes par les travaux réalisés.

Première partie

État de l’art

Chapitre 1

Introduction aux ontologies : genèse

Sommaire

Résumé. L'objectif de ce chapitre est de présenter un état de l'art sur les ontologies. Le but de cette étude est d'identifier les caractéristiques d'un tel concept, d'étudier ses langages de définitions et d'analyser leurs pouvoirs d'expression.

1 Introduction

Dans les années 70, les premiers travaux de modélisation ont été proposés. Le Prof. Edgar Frank Codd [Codd, 1970] a proposé son modèle relationnel traitant deux phases de modélisation: logique et physique. Dans la phase logique, le schéma de la base de données peut être normalisé en utilisant les dépendances fonctionnelles définies sur les propriétés et ce dans le but de réduire la redondance et d'assurer la qualité du schéma final de la base de données. Grâce à ces travaux, le premier cycle de vie des bases de données (BD) a été défini.

En 1975, en proposant le modèle entité-relation, Peter Chen [Chen, 1975] a contribué à l'extension du cycle de vie des BD par l'ajout de la phase conceptuelle (modèle conceptuel). Grâce à cette contribution, un cycle de vie complet dédié à la conception des BD a été tracé. En se basant sur ce dernier, plusieurs méthodologies de conception ont été proposées telles que la méthode MERISE [Rochfeld, 1986], Unified Process [4], Rational Unified Process [5], etc. Ces méthodologies ont été accompagnées par des outils académiques et industriels tels que Sybase PowerAMC [Soutou and Brouard, 2012], Rational Rose [Soutou and Brouard, 2012], etc.

Grâce au succès de ces méthodologies, les BD ont été largement utilisées dans plusieurs domaines. La quantité des données stockées est devenue importante et répartie sur plusieurs sources. Dans le but d'exploiter en même temps ces différentes données hétérogènes issues de diverses sources, un nouveau système de stockage a été proposé. Appelés systèmes d'intégration, ces structures de stockage ont permis de fournir une unique source uniformisée en dépit de l'hétérogénéité structurelle des données stockées (nature de données, modèles de stockages utilisés) d'une source à une autre. Ainsi, ayant l'impression d'utiliser un système homogène, l'utilisateur exploite ces données issues des différentes sources essentiellement utilisées lors du processus d'aide à la décision. Avec l'apparition des mécanismes d'intégration de données, le modèle conceptuel a subi une évolution pour donner naissance à un nouveau schéma appelé schéma global. Ce schéma, dédié à la conception des systèmes d'intégration, est défini comme étant une vue sur les schémas locaux des différentes sources. En effet, deux principales approches ont été proposées. La première, appelée GaV (global-as-view) [Calì et al., 2002], consiste à définir à la main (ou de façon semi-automatique) le schéma global en fonction des schémas des sources de données à intégrer ensuite à le connecter aux différentes sources. La deuxième approche, appelée LaV (local-as-view) [Lenzerini, 2002], suppose l'existence d'un schéma global et consiste à définir les schémas des sources de données à intégrer comme des vues du schéma global. Ainsi, le schéma global a permis de résoudre les conflits structurels, mais, malheureusement, il n'a pas su apporter de solutions concernant les conflits sémantiques. Notons que ces derniers concernent essentiellement les conflits de nommage, les conflits de contexte et les conflits de mesure. Dans le but de résoudre ces problèmes et d'ajouter la couche sémantique aux modèles de représentation de données, un nouveau schéma de modélisation a été défini. Appelées ontologies, ces modèles permettent de conceptualiser un domaine en termes

4. http://fr.wikipedia.org/wiki/Unified_Process
5. http://www-01.ibm.com/software/rational/rup/

de classes et de propriétés d'une manière consensuelle.

Dans ce chapitre, nous proposons une étude détaillée des ontologies. Dans la section qui suit, nous introduisons les ontologies en présentant ses principaux définitions, caractéristiques, langages de définition, éditeurs et modèles formels. Ensuite, nous établissons une étude comparative des ontologies et des modèles conceptuels. Après avoir dégagé les similitudes et les différences entre ces deux modèles, nous présentons une classification des ontologies en se basant sur la nature de ce qu'elles décrivent. Puis, nous présentons un ensemble de travaux exploitant les ontologies dans la conception des bases de données. Avant de conclure ce chapitre, nous étudions les travaux remontant les dépendances fonctionnelles au niveau ontologique.

2 Introduction aux ontologies

Dans cette partie, nous définissons la notion d'ontologie tout en dégageant ses caractéristiques. Nous présentons ses principaux langages de définition, ses éditeurs ainsi qu'un ensemble de formalismes dédiés à sa définition. Enfin, une étude comparative entre ontologies et modèles conceptuels est présentée.

2.1 Définition et caractéristiques

Dans cette section, nous présentons deux principales définitions d'ontologie : celle de Gruber et celle de Pierra et al.. Dans [Gruber, 1993a], Gruber définit une ontologie comme suit: *'An ontology is an explicit specification of a conceptualization'*. Dans ses travaux, Gruber a insisté sur la standardisation et le partage de l'ontologie, ce qui n'est pas mentionné dans sa définition. Pierra et al. ont complété cette définition [Pierra, 2008] [Jean et al., 2006b] comme suit: 'a formal and consensual dictionary of categories and properties of entities of a domain and the relationships that hold among them'. Pour Pierra et al. *entities* désigne tout élément ayant une existence (matérielle ou immatérielle) dans le domaine à conceptualiser. Le terme *dictionary* souligne le fait que toute entité et propriété d'entités décrites dans une ontologie de domaine peuvent être référencées directement, au moyen de symboles, pour en permettre l'usage dans n'importe quel modèle et dans tout contexte. Ainsi, en se basant sur ces définitions, nous relevons cinq caractéristiques importantes décrivant les ontologies de domaine :

1. formelle : basées sur des théories formelles, les ontologies permettent à une machine de vérifier automatiquement certaines propriétés de consistance et/ou de faire certains raisonnements automatiques sur les ontologies et leurs instances ;
2. explicite: les catégories de concepts du domaine et leurs différentes propriétés sont décrites explicitement, ce qui permet de lever toute sorte d'ambiguïté ;
3. consensuelle : une ontologie est une conceptualisation validée par une communauté. C'est un accord sur une conceptualisation partagée ;

4. référençable : chaque concept d'une ontologie est associé à un identifiant permettant de le référencer à partir de n'importe quel environnement, indépendamment de l'ontologie dans laquelle il a été défini .
5. réutilisable : une ontologie est un composant pouvant être réutilisé pour différentes tâches.

Après avoir introduit les ontologies, nous présentons, dans la section suivante, les principaux modèles proposés pour leur définition.

2.2 Modèles d'ontologies

La représentation d'une ontologie nécessite un ensemble de primitives permettant l'expression des axiomes, des instances, des entités et des relations qu'elle contient et les opérateurs nécessaires pour leur traitement. Ces primitives sont modélisées à travers un modèle d'ontologies. En examinant la littérature, plusieurs modèles d'ontologies ont été définis. Ils mettent en œuvre la logique de description, les bases de données, les réseaux sémantiques, etc. Ces modèles peuvent être classés selon deux domaines d'utilisation : le domaine de l'ingénierie et celui du Web Sémantique. Dans notre étude, nous avons choisi d'étudier deux principaux modèles : le modèle OWL [Horrocks et al., 2003] (Ontology Web Langage) relatif au domaine du Web sémantique et le modèle PLIB [Pierra, 2003] (Parts LIBrary) utilisé dans le domaine de l'ingénierie.

2.2.1 Modèles d'ontologies orientés Web Sémantique

Dans le domaine du Web Sémantique, plusieurs langages d'ontologies ont été proposés. Parmi les plus connus, nous citons RDF [6] (Resource Description Framework), RDF-Schema [7], OIL [Fensel et al., 2001] (Ontology Inference Layer), DAML [8] (DARPA Agent Markup Language) et OWL [9]. Ces derniers offrent des constructeurs permettant la représentation des informations de l'univers de discours, afin de faciliter le partage et l'échange des ontologies et des instances associées. Dans cette partie, nous proposons l'étude du modèle OWL, langage de définition des ontologies manipulées dans ce manuscrit.

– **OWL.** C'est un standard du W3C [10] pour traiter et indexer, à l'aide d'ontologies, les informations sur le Web. Basé sur RDFS et inspiré de DAML+OIL, OWL offre un vocabulaire supplémentaire avec une sémantique formelle pour la description des propriétés et des classes, les relations entre classes, les cardinalités, les équivalences conceptuelles, les caractéristiques des propriétés (par exemple la symétrie), les classes énumérées, etc. Ainsi, OWL offre la possibilité de raisonner sur les données. L'un des objectifs de ce modèle

6. http://www.w3.org/RDF/
7. http://www.w3.org/TR/rdf-schema/
8. http://www.daml.org/
9. http://www.w3.org/TR/owl-features/
10. World Wide Web Consortium. http://www.w3.org

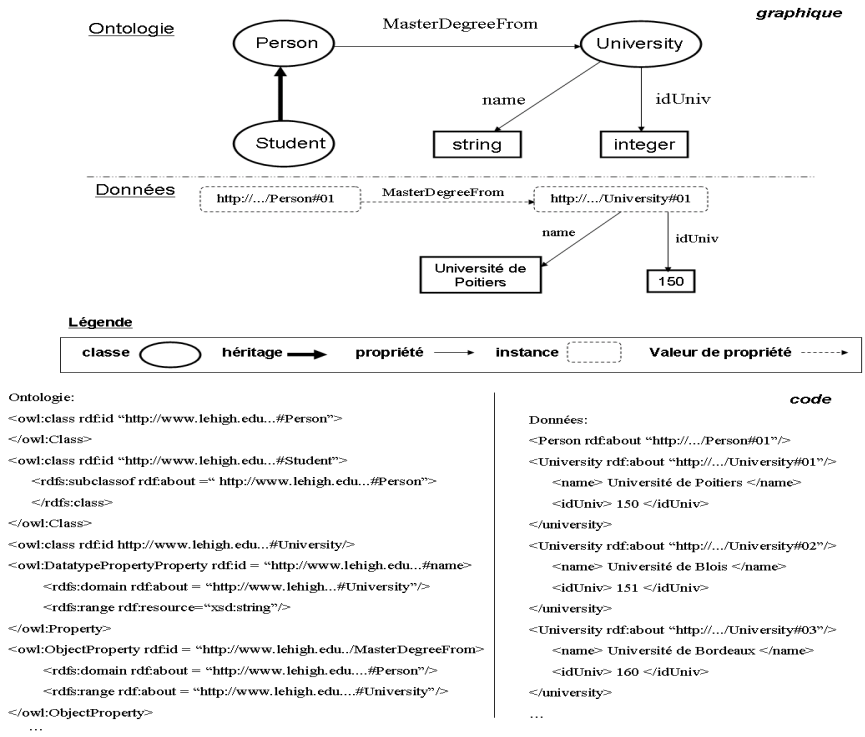

FIGURE 1.1 – Exemple d'ontologie exprimée en OWL

d'ontologies est que ses raisonnements soient décidables, ce qui veut dire qu'il est possible de concevoir un algorithme pouvant garantir la détermination, dans un temps fini, si oui ou non les définitions fournies par une ontologie OWL peuvent être déduites à partir de celles d'une autre ontologie [Horrocks et al., 2003]. L'impossibilité de proposer un langage à la fois compatible avec RDF-Schema et offrant des raisonnements décidables a poussé les concepteurs de OWL à en spécifier trois versions : OWL Lite, OWL DL et OWL Full. Ces langages sont caractérisés respectivement par leur expressivité croissante (OWL Lite $\subset$ OWL DL $\subset$ OWL Full). A l'encontre des deux premières versions, seule OWL Full est compatible avec RDF Schéma. Les deux premières versions restreignent les capacités des constructeurs de RDF Schéma. Par exemple, avec RDFS et OWL Full, une classe peut être une instance d'une autre classe ce qui n'est pas le cas avec OWL Lite et OWL DL.

Dans ce qui suit, nous introduisons les principaux constructeurs offerts par OWL Lite : les constructeurs de classes, de propriétés, d'individus ainsi que les types de données. Nous préciserons par la suite les limitations de ce langage par rapport à ses constructeurs ainsi que les libertés permises par OWL DL et OWL Full.

– **Constructeurs de classes.** Avec OWL Lite, une classe peut être déclarée comme étant une classe nommée ou une expression de classes. Dans le premier cas, la classe est spécifiée par un URI (Uniform Resource Identifier) en utilisant le constructeur *owl:Class*.

Dans le deuxième cas, plusieurs expressions sont proposées.

1. Les restrictions de propriétés. Elles précisent le co-domaine de certaines propriétés uniquement pour les classes où elles sont spécifiées. Deux constructeurs décrivant une restriction de propriété sont identifiées:
   – *owl:allValuesFrom* (∀ P.C) : la restriction d'une propriété à valeur littérale avec une quantification universelle définit une classe comme étant l'ensemble des instances ne prenant comme valeur d'une propriété que des instances d'une classe donnée.
   – *owl:someValuesFrom* (∃ P.C) : la restriction d'une propriété à valeur littérale avec une quantification existentielle permet de définir une classe comme étant l'ensemble des instances prenant comme valeurs d'une propriété au moins une instance d'une classe donnée.
2. Les restrictions de cardinalité. Une restriction de cardinalité restreint le nombre de valeurs distinctes qu'une instance peut prendre localement pour une propriété P. Deux principaux types de restrictions sont proposés: (a) les restrictions non typées (*owl:minCardinality*, *owl:maxCardinality*, *owl:Cardinality*) et (b) les restrictions typées (*owl:minCardinality Q*, *owl:maxCardinality Q*, *owl:Cardinality Q*). Contrairement à la deuxième catégorie, le type du co-domaine n'est pas spécifié dans la première catégorie.
3. Le constructeur booléen de classes. OWL Lite offre un constructeur *owl:intersectionOf* permettant de définir une classe comme étant l'intersection de plusieurs classes. Ainsi, si une classe C_j est définie tel que $C_j \equiv C_1 \cap C_2 \cap .. \cap C_{n-1} \cap C_n$, et si c est une instance de C_j, alors c est une instance de chacune des classes C_1, C_2,..., C_{n-1} et C_n.

De plus, OWL Lite offre les constructeurs nécessaires pour l'organisation des classes dans une hiérarchie de subsomption tel que:
– *rdfs:subClassOf* de RDFS-Schéma. Ce constructeur est équivalent à la subsomption en logique de description (C ⊆ D où C et D sont deux classes)
– *owl:equivalentClass*: C ≡ D signifie que C ⊆ D ∧ D ⊆ C.

– **Constructeurs de propriétés.** OWL Lite met à disposition des constructeurs pour la définition à la fois (1) des propriétés de type simple (*owl:datatypeProperty*) dont le co-domaine est un type de données issu de la spécification XML Schema et (2) des propriétés de type objet (*owl:ObjectProperty*) ayant pour co-domaine une classe ontologique. Ces propriétés peuvent être associées à une classe par la spécification de leur domaine (*rdfs:domain*). Si le domaine n'est pas spécifié, la propriété est attribuée à la classe racine (*owl:Thing*). De la même manière, une propriété peut préciser son co-domaine à l'aide du constructeur *rdfs:range*. Comme pour les classes, OWL Lite permet l'organistation des propriétés en hiérarchie de subsomption à l'aide des constructeurs *rdfs:subPropertyOf* et *owl:equivalentProperty*. De plus, il permet d'associer aux propriétés des caractéristiques mathématiques tel que la transitivité (*owl:TransitiveProperty*),

la symétrie (*owl:SymmetricProperty*) et l'injectivité (*owl:InverseFunctionalProperty)*. Une propriété peut être aussi qualifiée comme étant une fonction (*owl:FunctionalProperty*) ou l'inverse d'une autre (*owl:inverseOf*).

– **Constructeurs d'individus.** Une instance OWL est définie à l'aide du constructeur *owl:individual*. Des assertions sur ces instances peuvent être définies tel que suit:
  – Le constructeur *rdf:type* permet de déclarer qu'un individu appartient à une classe C ($i \in C$).
  – Le constructeur *owl:sameAs* permet de définir une égalité entre deux individus i_1 et i_2 ($i_1 \equiv i_2$).
  – Le constructeur *owl:differentFrom* permet de différencier deux individus i_1 et i_2 ($i_1 \neq i_2$).
– **Les types de données.** OWL Lite permet de supporter les types de données primitifs issus de la spécification XML Schema (xsd:boolean, xsd:string, etc.) ainsi que les types de données RDFS intégrés (rdfs:Literal).

OWL Lite est la version la plus simplifiée d'OWL. Il offre aux utilisateurs une base simple de fonctionnalités du langage tout en offrant la possibilité de raisonner sur les concepts et les individus. Mais, son pouvoir d'expression est limité. Il ne permet d'exprimer qu'un ensemble de contraintes simples. On peut citer à cet égard l'exemple de la contrainte de cardinalité limitée aux valeurs 0 et 1. De plus, l'expression de l'opérateur booléen *union* ne peut être traduite avec ce langage. Pour pallier ces insuffisances, une deuxième version d'OWL notée OWL DL a été proposée. Basé sur la logique de description SHOIN, cette version a la particularité d'être décidable mais incompatible avec RDFS. Elle inclut tous les constructeurs d'OWL Lite excepté quelques restrictions telles que la cardinalité limitée par les valeurs 0 ou 1. OWL DL permet par exemple de définir une classe comme étant :

– l'union de plusieurs classes à l'aide du constructeur *owl:unionOf* ;
– le complémentaire d'une autre classe à l'aide du constructeur *owl:complementOf* ;
– l'ensemble des instances ayant une certaine valeur pour une propriété donnée à l'aide du constructeur *owl:hasValue* ;
– une énumération d'instances à l'aide du constructeur *owl:oneOf*.

De plus, il permet d'exprimer la contrainte de cardinalité sans se limiter aux valeurs 0 et 1. Même si OWL DL a enrichi le pouvoir d'expression d'OWL, il reste toujours incompatible avec RDFS. Par conséquent, une troisième version notée OWL Full a été spécifiée. Celle-ci est destiné aux utilisateurs voulant combiner à la fois le pouvoir d'expression qu'OWL procure aux capacités de méta-modélisation de RDF. Avec la proposition de cette version, une instance peut être traitée comme étant une classe. De plus, la distinction entre les propriétés de données et les propriétés d'objets n'est pas nécessaire avec ce langage. Notons qu'OWL Full reste peut utilisé à cause de son niveau élevé d'expressivité le rendant indécidable.

Dans la figure 1.1, nous présentons un extrait de l'ontologie Lehigh University Benchmark (LUBM [11]) [Guo et al., 2005] décrivant le domaine universitaire.

2.2.2 Modèles d'ontologies orientés ingénierie

En vue de répondre aux besoins de l'ingénierie et de simplifier le travail de modélisation, plusieurs standards de modélisation d'ontologies ont été proposés (PLIB, STEP [Pierra, 2000](Exchange of Product model data), etc.). Dans la section suivante, nous proposons d'introduire le modèle PLIB.

1. **PLIB.** Initié en 1987, PLIB (séries de ISO 13584) est un modèle d'ontologies facilitant le dialogue des industriels et le regroupement des fournisseurs des composants. Il permet de modéliser les catalogues de composants en décrivant toute la connaissance sur le comportement de ces composants ainsi que leurs critères de sélection pour une application donnée. PLIB repose sur un langage formel EXPRESS pour la description des catalogues sous forme d'instances de modèle. Un tel catalogue contient une ontologie décrivant les catégories et les propriétés qui caractérisent un domaine donné et les instances de cette ontologie.

   Nous présentons ci-dessous les principaux constructeurs du modèle d'ontologies PLIB.
   - **Constructeurs de classes.** De la même manière qu'OWL, PLIB offre des constructeurs permettant la définition des classes et leur organisation dans une hiérarchie de subsomption. Une classe PLIB est décrite par un identifiant unique 'BSU' (Basic Semantic Unit). Elle peut être définie comme étant:
     - une classe de définition à l'aide du constructeur *item_class*.
     - une classe de point de vue à l'aide du constructeur *functional_view_class*.
     - une classe de représentation à l'aide du constructeur *functional_model_class*.

     Pour organiser les classes dans une hiérarchie, deux opérateurs sont définis. Le premier est l'opérateur *is_a*. Il permet la définition d'une hiérarchie simple avec l'héritage de toutes les propriétés par les classes subsumantes. Le second opérateur, nommé *case_of*, permet la définition d'une hiérarchie avec importation explicite d'une partie des propriétés. Cet opérateur permet ainsi la construction modulaire d'ontologies de domaine.
   - **Constructeurs de propriétés.** PLIB offre des constructeurs pour la déclaration des propriétés et leur organisation dans une hiérarchie de subsomption. D'une manière similaire aux classes, une propriété PLIB est identifiée par un BSU. Celle-ci peut être définie comme étant une propriété autonome dont la valeur est indépendante de toute autre propriété (*non_dependent_p_det*) ou inversement (*dependent_p_det*). Elle peut aussi être définie comme un paramètre de contexte (*condition_det*) décrivant le contexte dans lequel est situé l'objet (par exemple, le poids d'un objet dépend de l'altitude). Contrairement à RDFS et OWL, PLIB adopte le typage fort des propriétés, c'est à dire chaque propriété doit définir son champ d'application nommé domaine (*name_scope*)

11. http://www.lehigh.edu/ zhp2/2004/0401/univ-bench.owl

ainsi que son domaine de valeurs ou co-domaine. En pratique, plusieurs classes issues de différentes branches de la hiérarchie des classes peuvent définir le domaine d'une propriété. Afin de répondre à ce besoin, PLIB propose de qualifier les propriétés tel que suit:
  - A l'aide du constructeur *is_a*, une propriété est définie sur une classe C au plus haut niveau de la hiérarchie. Ainsi, elle est visible pour toutes les sous-classes de C.
  - Dans le cas où la propriété est visible sur une classe C, elle peut être applicable sur cette classe, c'est à dire toute instance doit présenter une valeur représentant cette propriété.
- **Les types de données.** PLIB permet de représenter les types de données simples, énumérés, des données qualifiées tel que les unités (int_measure_type, etc.) et les aggrégats (array_type, list_type, etc.).
- **Constructeurs d'individus.** PLIB offre les constructeurs nécessaires pour la définition des instances. En effet, une instance PLIB est définie par sa classe de base et l'ensemble des valeurs attribuées aux propriétés de cette classe. Etant donné que PLIB n'offre pas la multi-instanciation, il met en place un mécanisme d'agrégation d'instance. A l'aide du constructeur is_view_of, une instance peut appartenir non seulement à sa classe de base mais également aux classes de représentation attachées à cette classe de base. Afin d'éviter toute redondance au niveau des valeurs des propriétés des instances, les propriétés définies sur la classe de base et sur les classes de représentation sont déclarées disjointes, à l'exception des propriétés définies comme étant communes pour établir la jointure.

La figure 1.2 présente un exemple de l'ontologie décrivant le domaine universitaire en EXPRESS.

2.2.3 Synthèse

Après avoir examiné les modèles d'ontologies, nous remarquons qu'un ensemble de points communs est partagé. En effet, ils permettent tous de conceptualiser l'univers de discours à l'aide des classes et des propriétés. Notons que ces concepts peuvent être organisés dans une hiérarchie de subsomption. De plus, chaque concept est associé à un identifiant unique lui permettant d'être identifié de n'importe quel système ou environnement. Nous précisons que la notation de cet identifiant varie d'un langage à un autre (BSU pour PLIB, URI pour OWL, etc.). Enfin, aucun de ces modèles ne permet de définir des relations de dépendance entre les concepts ontologiques explicitement. Aucune dépendance ne peut être définie de manière explicite ni entre les propriétés, ni entre les classes. Ce point sera détaillé dans le chapitre 3. Des différences entre ces modèles sont aussi relevées. Elles concernent essentiellement les constructeurs offerts pour la description des concepts. Certains de ces modèles offrent un langage atomique (pas de redondance de données) tandis que d'autres offrent des constructeurs permettant la description de concepts définies et ce au travers d'expressions de données ou d'opérateurs

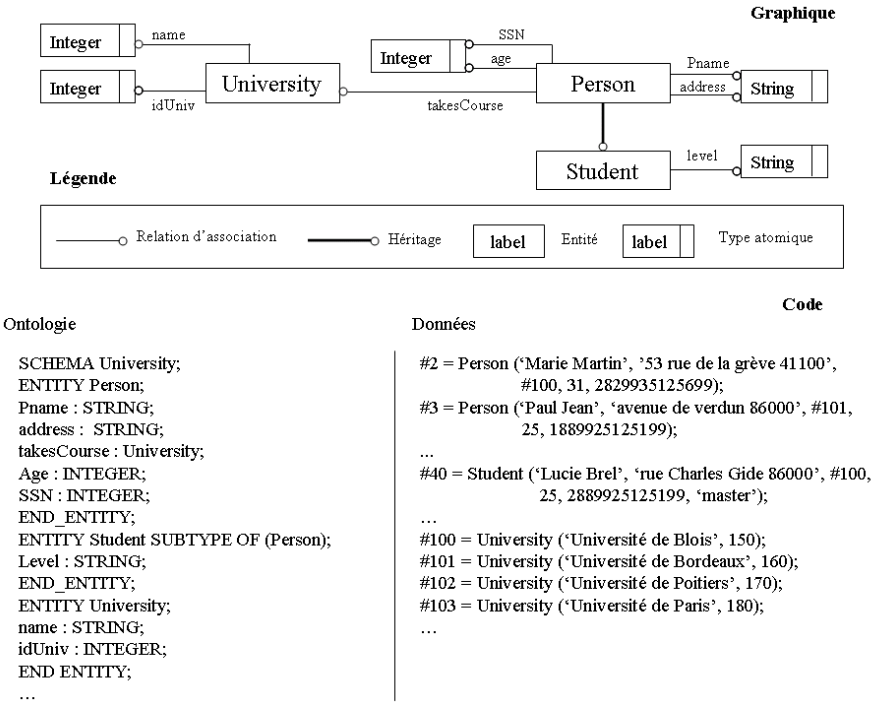

FIGURE 1.2 – Exemple d'ontologie exprimée en EXPRESS.

ensemblistes, etc.

Après avoir introduit les principaux modèles d'ontologies, nous présentons dans la section suivante les outils permettant leur manipulation.

2.3 Éditeurs d'ontologies

Un éditeur d'ontologies est une plateforme permettant de construire et de modéliser une ontologie pour un domaine donné et de définir des formulaires d'entrée de données offrant la possibilité d'acquérir des données sous forme d'instances de cette ontologie. Plusieurs éditeurs d'ontologies ont été proposés dans la littérature. Nous citons à titre d'exemple OntoEdit [Sure et al., 2002], OilEd [12], PLIBEditor [13] et Protégé [14]. Dans cette section, nous proposons l'étude de l'éditeur Protégé. Ce choix est justifié par le grand succès qu'a connu cet éditeur dû à son soutien par une forte communauté de développeurs et d'universitaires dans différents domaines.

12. http://www.xml.com/pub/r/861
13. www.plib.ensma.fr
14. http://protege.stanford.edu/

2.3.1 Protégé

Protégé est un logiciel open source initialement développé par le Stanford Center for Biomedical Informatics Research. Son utilisation a ensuite dépassé le cadre de la médecine et de la biologie pour couvrir plusieurs domaines. Désormais, Protégé est devenu un éditeur d'ontologie parmi les plus utilisés. Il comporte une librairie Java pouvant être étendue pour créer de véritables applications à base de connaissances. Protégé a mis en place des plugins pour les langages RDF, DAML+OIL et OWL. Ainsi, il est utilisé comme éditeur d'ontologies pour ces différents langages. De plus, il offre la possibilité de raisonner sur les ontologies en utilisant un moteur d'inférence général tel que JESS [15] ou des outils d'inférence propres au web sémantique basés sur des logiques de description tels que RACER [16].

Après avoir présenté les outils dédiés à la modélisation et la manipulation des ontologies, nous proposons dans la suite d'étudier ce modèle en profondeur. Nous présentons, dans la section qui suit, quelques formalisations définies pour la description de ce dernier.

2.4 Formalisation

En examinant la littérature, un ensemble de modèles formels d'ontologie a été identifié [Pierra, 2003][Bellatreche et al., 2003][Horrocks and Patel-Schneider, 2003]. Dans OWL [17], une ontologie O est définie par un ensemble de faits, d'axiomes et d'annotations, défini en termes de graphes RDF et de déclarations. En effet, le contenu principal de O est traduit à travers ses axiomes et ses faits fournissant ainsi les informations sur ses classes, ses propriétés et ses individus. Les annotations sur les ontologies OWL sont généralement utilisées pour spécifier les informations associées à une ontologie tel que les importations des autres ontologies, etc. La figure 1.3 décrit le diagramme d'ontologie OWL. Dans ce diagramme, une ontologie O est représentée par la classe *OWLOntology* qui est une sous classe de *RDFSResource*. Elle est en association avec la classe *OWLGraph* reliant une ontologie à un ou plusieurs graphes contenant les déclarations la définissant. En effet, un graphe RDF est un ensemble de triplets RDF. L'ensemble des nœuds d'un graphe RDF est l'ensemble des sujets et des objets de triplets représentés dans le graphe. Notons que les graphes RDF ne sont pas tous des graphes OWL, cependant, la classe *OWLGraph* spécifie le sous-ensemble de graphes RDF représentant les graphes OWL valides. Une association *StatementForOntology* est définie entre les classes *OWLOntology* et *RDFStatement*. Celle-ci relie une ontologie à une ou plusieurs déclaration RDF qui représente une expression ou un sous-graphe, contenant un sujet, un prédicat et un objet en RDF.

Dans PLIB, une ontologie O est définie par un ensemble d'éléments (class-and-property-elements) décrivant des classes (Class) et des propriétés (Property-DET) organisées dans une

15. http://www.jessrules.com/
16. http://www.sts.tu-harburg.de/ r.f.moeller/racer/
17. http://www.omgwiki.org/

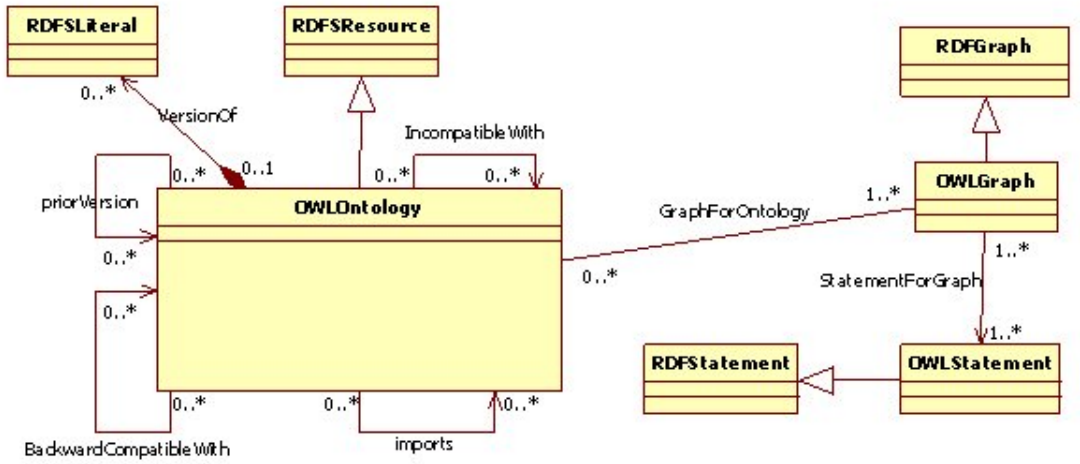

FIGURE 1.3 – Description du diagramme OWL.

hiérarchie de subsomption et liés au travers d'un ensemble de relations. Deux types de subsomption entre les classes sont définis. La première représente la relation d'héritage classique. La deuxième décrit la relation d'importation (is-case-of) où la classe se déclarant un cas d'une autre (item-class-case-of), importe explicitement les propriétés nécessaires de la classe dont elle est un cas. La figure 1.4 décrit le méta-modèle simplifiée de PLIB.

Dans [Bellatreche et al., 2003], les auteurs ont proposé un modèle formel simplifié dans lequel ils définissent une ontologie de domaine O par le quadruplet : O : <C, P, Sub, Applic>, avec :

- C: ensemble des classes utilisées pour décrire les concepts d'un domaine donné. Chaque classe est associée à un identifiant globalement unique.
- P : ensemble des propriétés utilisées pour décrire les instances de l'ensemble des classes C par des valeurs appartenant soit à des types simples, soit à d'autres classes [18].
- Sub : C $\rightarrow 2^C$; la relation de subsomption [19] qui, à chaque classe c_i de l'ontologie, associe ses classes subsumées directes [20]. Sub = OOSub $\cup$ OntoSub où :
  - OOSub décrit la subsomption entre les classes internes de la même ontologie avec l'héritage usuel (is-a).
  - OntoSub représente l'opérateur de spécialisation d'une classe avec importation sélective de propriétés. Il permet de redéfinir entièrement la structure des classes en fonction des besoins. La caractéristique de cette relation est (i) de préciser quelles sont les pro-

18. Le terme propriété regroupe les deux notions parfois distinguées sous les noms d'attribut et de relation (ou rôle)

19. c_1 subsume c_2 ssi : $\forall$ x $\in c_2$, x $\in c_1$

20. c_1 est une subsumante directe de c_2 dans une ontologie O ssi la relation de subsomption entre c_1 et c_2 ne peut se déduire par transitivité, des autres relations de subsomption définies entre les différentes classes de O

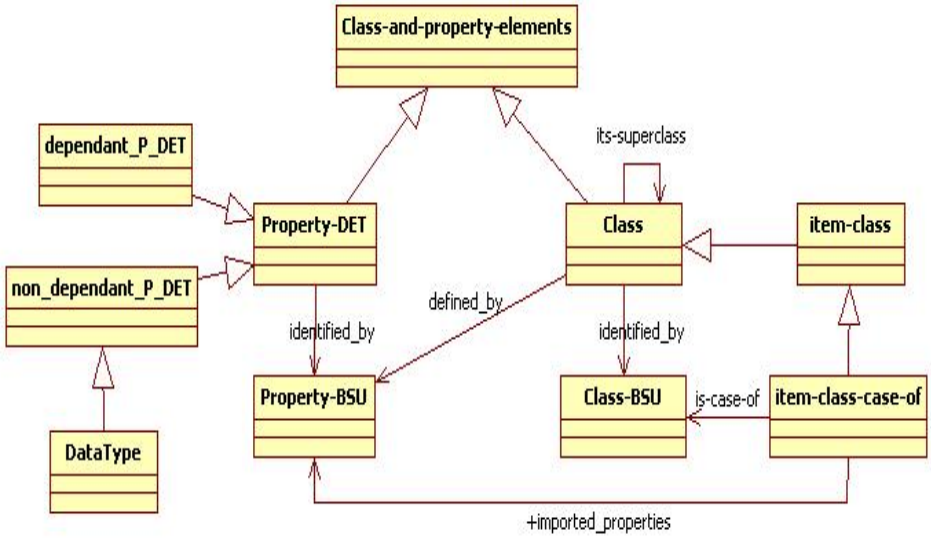

FIGURE 1.4 – Méta-modèle simplifiée de PLIB

priétés importées et (ii) d'être considérée comme une relation d'interfaçage entre des ontologies différentes, et non comme une relation interne à une ontologie.

– Applic : C $\rightarrow 2^P$; la fonction qui associe à chaque classe c_i de l'ontologie les propriétés qui sont applicables pour cette classe, c'est-à-dire, qui peuvent être utilisées pour en décrire les instances. Seules les propriétés ayant pour domaine une classe, ou l'une de ses super-classes, sont applicables à cette classe. Si le domaine d'une propriété n'est pas spécifié, la racine de l'ontologie est implicitement considérée.

Notons que la subsomption, relation ontologique, ne doit pas être confondue avec l'héritage, relation logique introduite dans les langages à objets. Une ontologie ne prescrit en aucun cas les propriétés devant être valuées pour chaque instance. Le fait qu'une propriété soit applicable pour une classe n'impose en aucun cas que toutes les instances de cette classe (ou d'une de ses sous-classes) possèdent une valeur pour cette propriété.

Précisons que dans le domaine de la logique de description, d'autres formalisations d'ontologie ont été proposées [Horrocks and Patel-Schneider, 2003]. Cependant, ces dernières ne sont pas traitées dans cette partie car nous nous intéressons plus aux formalisations définies de point de vue bases de données.

2.5 Ontologie vs. Modèle conceptuel

Dans nos travaux de recherche, nous nous intéressons à la conception des bases de données stockant à la fois les ontologies et ses instances. Ce processus nécessite l'étude de la phase de

la modélisation conceptuelle. En se basant sur la définition des ontologies et de ses caractéristiques, nous avons identifié un ensemble de similitudes ainsi que des différences relevées entre les ontologies et les modèles conceptuels. Dans le but de définir un modèle conceptuel dédié à ce type de base de données, nous présentons dans ce paragraphe une étude comparative des ontologies et des modèles conceptuels [Spyns et al., 2002, Fankam et al., 2009]. En effet, comme les modèles conceptuels (MC), les ontologies *conceptualisent l'univers du discours au moyen de classes hiérarchisées et de propriétés caractéristiques*. Par contre, ils diffèrent dans leur *objectif de modélisation*. Les MCs prescrivent les informations devant être représentées dans un système informatique particulier afin de répondre à un cahier des charges applicatif donné. Par contre, les ontologies visent à décrire, en se basant sur un consensus, l'ensemble des informations permettant la conceptualisation des domaines d'applications indépendamment de toute application et de tout système dans lequel elles peuvent être utilisées. Une deuxième différence identifiée concerne *la structure de modélisation et l'identification des concepts*. En effet, dans un modèle conceptuel de données, chaque concept a un sens dans le contexte du modèle dans lequel il est défini. Au contraire, dans une ontologie, chaque concept est identifié de façon individuelle et constitue une unité élémentaire de connaissance [Fankam et al., 2009]. Ainsi, un MC peut être défini à partir d'une ontologie en extrayant uniquement les concepts pertinents répondant à un besoin applicatif donné. La troisième différence traite la consensualité. Les concepts définis dans un MC ne sont pas réutilisables à l'extérieur de ce modèle. Au contraire, dans une ontologie, les classes et les propriétés sont associées à des identifiants universels leur permettant d'être référencées d'une manière unique à partir de n'importe quelle structure. En conséquence, ces concepts sont réutilisables à chaque fois qu'on a besoin. De plus, grâce à cette caractéristique, l'intégration sémantique de tous les systèmes basés sur une même ontologie pourra facilement être réalisée à condition que les références à l'ontologie soient explicitées. Une quatrième différence concerne la non canonicité des informations représentées. Contrairement aux MC dans lesquels les informations du domaine sont décrites par un langage minimal et non redondant, les ontologies utilisent à la fois des concepts atomiques ainsi que des concepts définis fournissant des accès alternatifs à la même information. Une dernière différence concerne le raisonnement. En effet, contrairement aux MC, il existe des mécanismes d'inférence dédiés aux ontologies permettant de raisonner sur ces modèles pour la classification de ses concepts, la vérification de la cohérence des spécifications, etc.

Cette étude montre que les ontologies peuvent procurer aux concepteurs les mêmes fonctionnalités présentées par les modèles conceptuels. En effet, une ontologie peut être vue comme un modèle conceptuel global aidant les concepteurs à définir leur propre modèle pour une application donnée. De plus, leur prise en compte dans le processus de conception des bases de données pourrait permettre de réduire de façon significative un ensemble de problèmes tels que l'intégration des données, etc.

2.6 Synthèse

Les ontologies sont des modèles formels explicites d'une conceptualisation partagée d'un domaine donné. Elles sont essentiellement utilisées pour raisonner sur les objets du domaine concerné. Elles permettant ainsi des raisonnements automatiques ayant pour objet soit d'effectuer des vérifications de consistance, soit d'inférer de nouveaux faits. Plusieurs modèles d'ontologies ont été proposés. Ces derniers mettent à disposition des constructeurs pour la définition des concepts, des axiomes ainsi que des relations de subsomption entre les classes, les propriétés, etc. Afin de faciliter la phase de raisonnement, des éditeurs d'ontologies munis de moteurs d'inférence ont été proposés. Ceci montre l'intérêt majeur du processus de raisonnement sur les ontologies et les dispositifs offerts pour la réalisation de ce dernier. Dans la communauté des bases de données, les ontologies ont été considérées comme étant des modèles conceptuels glaubaux procurant aux concepteurs les primitifs pour la définition du modèle conceptuel de données pour une application donnée dans un domaine spécifique.

3 Classification des ontologies

Les modèles d'ontologies offrent différents constructeurs pour la définition des ontologies. Cependant, ces ontologies ne sont pas toutes identiques. Elles peuvent différer selon la nature de la chose décrite. En d'autres termes, les ontologies peuvent décrire soit des concepts ou des termes, d'où la classification en ontologies conceptuelles et ontologies linguistiques. Notons que dans cette thèse, seules les ontologies conceptuelles sont manipulées. Cependant, dans cette partie, les ontologies linguistiques ne sont pas traitées.

3.1 Ontologies conceptuelles

Une ontologie conceptuelle est une ontologie visant à décrire un objet, une situation ou un phénomène. Dans une ontologie conceptuelle, deux types de concepts peuvent être identifiés : les concepts primitifs et les concepts définis. Les concepts dits primitifs ou canoniques sont les concepts qui ne peuvent pas avoir une définition axiomatique complète [Gruber, 1993b]. Ils permettent de définir les frontières du domaine conceptualisé en décrivant chacun un consensus parmi une communauté. La définition de tels concepts ne peut se réduire avec d'autres concepts, mais elle représente une fondation sur laquelle d'autres concepts peuvent être définis d'où la notation de concepts définis. Un concept défini ou non canonique est un concept « pour lequel une ontologie fournit une définition axiomatique complète au moyen de conditions nécessaires et suffisantes exprimées en termes d'autres concepts primitifs ou eux-mêmes définis » [Gruber, 1993b]. Les concepts non canoniques ne permettent pas d'introduire de nouveaux concepts mais plutôt d'enrichir le vocabulaire les représentant et ce en introduisant, à l'aide de constructeurs offerts par les modèles d'ontologies, des alternatives de désignation pour des

concepts que l'on pouvait déjà décrire. La distinction entre la nature des concepts a engendré une deuxième classification tel que suit:

1. Les Ontologies Conceptuelles Canoniques (OCC). Les OCC sont les ontologies dont les définitions ne contiennent aucune redondance. Elles permettent de définir un langage conceptuel canonique en décrivant chaque concept du domaine d'une manière unique. Par exemple, elles ne contiennent que des classes canoniques, c'est à dire celles «pour lesquelles nous ne sommes pas capable de donner une définition axiomatique complète et dont la compréhension repose sur une connaissance préalable du lecteur» [Gruber, 1993a]. Par conséquent, les OCC ne comportent que des concepts atomiques (primitifs). Ces concepts sont décrits d'une manière précise à l'aide de constructeurs orientés OCC permettant, à titre d'exemple, de spécifier le contexte dans lequel chaque élément ontologique est défini [Pierra, 2003]. Par contre, aucun opérateur pour la définition des équivalences conceptuelles n'est fourni par cette catégorie d'ontologies. Un exemple d'une OCC définie pour le commerce électronique est présentée dans [IEC61360-4, 1999].
2. Les Ontologies Conceptuelles Non Canoniques (OCNC). Dans ce paragraphe, nous introduisons les OCNC puis nous présentons un ensemble de constructeurs orientés OCNC.
   (a) *Définition.* Les OCNC sont des ontologies dont les définitions contiennent de la duplication de données. Elles contiennent à la fois les concepts primitifs et les concepts définis. Avec l'introduction des concepts définis, les équivalences conceptuelles sont définies. Celles-ci permettent d'exprimer différemment des concepts identiques à l'aide d'une panoplie de constructeurs tels que les opérateurs booléens, etc (voir paragraphe suivant). En effet, la définition de telles équivalences offre, en particulier à la communauté Intelligence Artificielle, une possibilité d'effectuer des raisonnements en appliquant le mécanisme d'inférence. Pour la communauté base de données, ces équivalences sont plutôt comparées aux mécanismes de vues avec une théorie formelle offrant des capacité d'inférence. Grâce à la redondance qu'elles introduisent, le nombre de concepts en termes desquels il est possible d'exprimer une requête donnée augmente. De plus, les constructeurs OCNC facilitent la définition des mappings entre différentes ontologies ainsi que les tâches d'intégration. Par contre, une perte de précision dans la définition des concepts primitifs est relevée. Une telle définition se limite généralement à un libellé et un commentaire pour la description des classes et des propriétés.
   (b) *Constructeurs de non canonicité.* Plusieurs constructeurs ont été proposés pour la définition des équivalences conceptuelles. Ces constructeurs peuvent s'appliquer:
      i. sur les classes d'une ontologie tel que l'union ($C_j \equiv C_1 \cup C_2 \cup ... \cup C_n$), l'intersection ($C_j \equiv C_1 \cap C_2 \cap ... \cap C_n$), le complément (le complément d'une classe C_i sélectionne tous les individus du domaine de discours qui n'appartiennent pas à cette classe, $C_j \equiv \neg C_i$), etc.
      ii. sur les propriétés à travers la définition des restrictions. Trois principales restrictions sont identifiées. La première concerne la restriction du codomaine d'une

propriété. Cette restriction peut agir soit sur la valeur du codomaine ($\exists P.C$, $\forall P.C$ où P et C décrivent respectivement une propriété et une classe ontologiques, etc.) ou sur le nombre des valeurs du co-domaine ($\geq nP.C$, $\leq nP.C$, $= nP.C$ où n $\in \mathbb{N}$, etc.). La deuxième restreint plutôt les valeurs du domaine de la propriété. En ce qui concerne la troisième restriction, elle spécifie la valeur exacte d'une propriété donnée ($\exists P.\{x\}$ où x spécifie une valeur donnée de la propriété P).

(c) *Ajout des termes.* Un concept peut être exprimé au travers de différents termes utilisés dans la description langagière d'un domaine donné. Ces termes introduisent des relations linguistiques telles que la synonymie, etc.

Après avoir présenté la classification des ontologies, nous étudions dans la section suivante, l'ensemble des relations entre les différentes catégories identifiées.

3.2 Modèle en oignon

Afin d'identifier les relations entre les différentes catégories d'ontologies, un modèle en couches appelé modèle en oignon [Jean et al., 2006b] a été proposé tel qu'illustré dans la figure 1.5.(A). Il se compose de trois couches comme suit:

1. *une couche canonique* fournissant une base formelle pour modéliser et échanger efficacement la connaissance d'un domaine.

2. *une couche non canonique* fournissant les mécanismes permettant la liaison des différentes conceptualisations établies sur ce domaine. Parmi les opérateurs d'expression de concepts ontologiques non canonique, nous citons les expressions de classes, les fonctions de dérivation, les expressions de propriété, etc.

3. *une couche linguistique* fournissant une représentation en langage naturel des concepts de ce domaine, éventuellement dans les différents langages où ces concepts sont significatifs. Parmi les opérateurs d'expression de concepts ontologiques linguistiques, nous citons la synonymie, la traduction en un langage donné, etc.

La figure 1.5.(B) présente un exemple d'instanciation du modèle en oignon. Par exemple, le concept non canonique *PublicUniversity* désignant une université publique est défini par le biais d'expression de propriété à partir du concept canonique University comme étant une université ayant un statut public.

Dans la section suivante, nous examinons les principales approches d'exploitation des ontologies conceptuelles dans le processus de conception des bases de données.

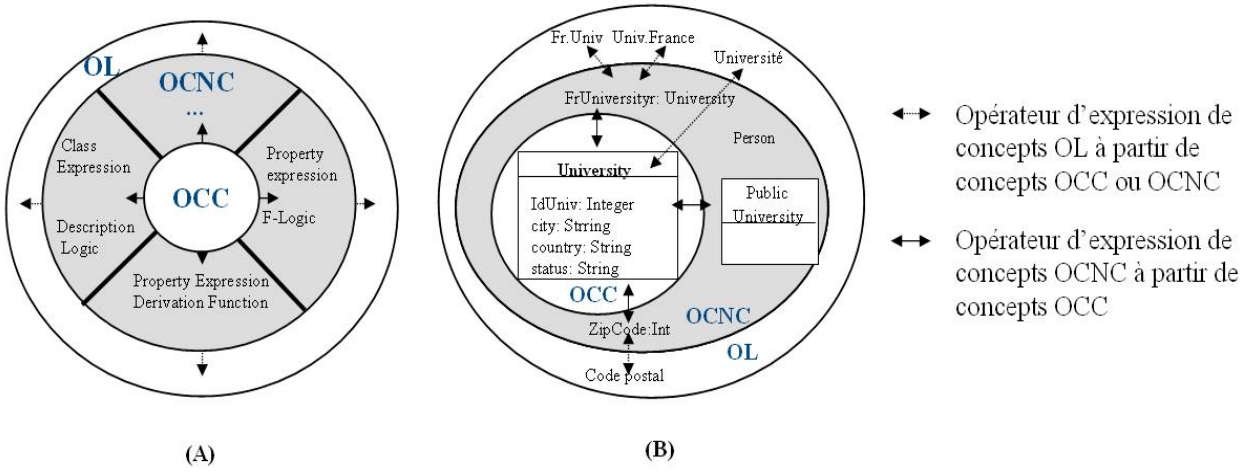

FIGURE 1.5 – Classification des ontologies : modèle en oignon

4 Exploitation des ontologies dans la conception des bases de données

Vues les similarités relevées entre les ontologies et les modèles conceptuels, nous proposons, dans cette section, d'étudier les principaux travaux de rapprochement des ontologies et des bases de données. Différentes approches basées sur l'exploitation des ontologies pour la définition d'un modèle conceptuel de données sont représentées. Dans cette section, nous nous intéressons aux approches conceptuelles. Deux principales approches sont étudiées : celle de [del Mar Roldán García et al., 2005] et celle de [Fankam et al., 2009].

Dans [del Mar Roldán García et al., 2005], Roldan-Garcia et ses collègues ébauchent une méthodologie pour la conception des bases de données en se basant sur une ontologie de domaine prédéfinie. Dans cette approche, ils proposent (a) d'étendre, au besoin, l'ontologie de domaine avec de nouveaux concepts (classes et propriétés) manquants et nécessaires pour le système à mettre en place, (b) d'extraire le sous-ensemble de classes de l'ontologie ainsi étendue qui couvre les besoins du système en cours. Ce sous-ensemble constitue le modèle conceptuel de données et sera utilisé dans la suite du processus de conception de la base de données. Cette approche vise à la fois à faciliter la conception de la base de données en réutilisant la sémantique formalisée du domaine telle qu'elle est définie dans les ontologies, et à permettre, par la suite, son accès à partir de l'ontologie du domaine utilisée. Cette méthodologie reste très préliminaire vu qu'aucune correspondance n'est définie entre les différentes ontologies et les modèles conceptuels de données générés. De plus, elle ne facilite pas ou peu l'intégration, car le lien entre données et ontologie est indéfini.

Dans [Fankam et al., 2009], les auteurs proposent une approche de conception de base de

données fondée sur les ontologies de domaine. Nommée SISRO (Spécialisation, Importation Sélective et Représentation des Ontologies), cette approche est basée sur quatre étapes. La première consiste à la définition de l'ontologie locale. Durant cette étape, le concepteur extrait, par spécialisation des ontologies de domaine existantes, les concepts pertinents pour son application, importe de façon sélective les propriétés nécessaires et étend si besoin l'ontologie par les concepts et/ou les propriétés nécessaires qui ne figuraient pas dans l'ontologie. Ainsi, le concepteur a (i) la possibilité de réutiliser la connaissance du domaine formalisée dans l'ontologie et (ii) la liberté de structurer les concepts importés de l'ontologie globale. La deuxième étape concerne la définition du modèle conceptuel. Le concepteur définit le MC à partir du sous-ensemble canonique de l'ontologie locale. Il choisit les classes et les propriétés canoniques constituant son MC. La troisième étape traite la définition des modèles logique (MLD) et physique (MPD) de données. Le modèle conceptuel est traduit, en se basant sur le processus de normalisation, en un modèle logique normalisé. Notons que durant cette étape, le concepteur peut définir des vues pour représenter les concepts non canoniques de l'ontologie locale s'il le souhaite. La dernière étape consiste à fournir l'accès au niveau ontologique. Deux méthodes sont proposées pour permettre ce type d'accès: l'accès par vues et l'accès par représentation explicite des ontologies. Dans la première méthode, une vue est associée à chaque concept (canonique ou non canonique) de l'ontologie locale et pour les concepts de l'ontologie de domaine faisant l'objet d'une relation de subsomption. Ainsi, les données stockées dans la base de données deviennent accessibles au niveau connaissance en termes ontologiques. La deuxième méthode représente à la fois dans la base de données l'ontologie locale et son articulation avec l'ontologie partagée. Ainsi, les identifiants de concepts peuvent être exploités pour lier toute donnée et le concept ontologique qui en définit le sens. Les étapes de cette approche sont décrites dans la Figure 1.6. Cette approche offre au concepteur une large autonomie pour la définition du modèle conceptuel de données. De plus, elle facilite l'intégration future des bases de données conçues à partir de la même ontologie de domaine. Par contre, étant donné que les dépendances fonctionnelles ne sont pas gérées par cette approche, offrir un schéma normalisé reste une tâche difficile pour le concepteur.

Dans la section suivante, nous proposons d'étudier les principaux travaux portant sur les dépendances fonctionnelles au niveau ontologique.

5 Ontologies et dépendances fonctionnelles

Depuis les années 70, les dépendances fonctionnelles ont été largement étudiées dans la théorie des base de données. Ces dépendances, définies généralement sur les attributs, ont été spécialement exploitées dans le processus de conception des BD. Elles sont utilisées pour modéliser les relations entre les attributs d'une relation, calculer les clés primaires, définir le modèle logique normalisé, réduire les données redondantes, etc.

Dans le domaine de la logique de description (DL), les dépendances fonctionnelles ont

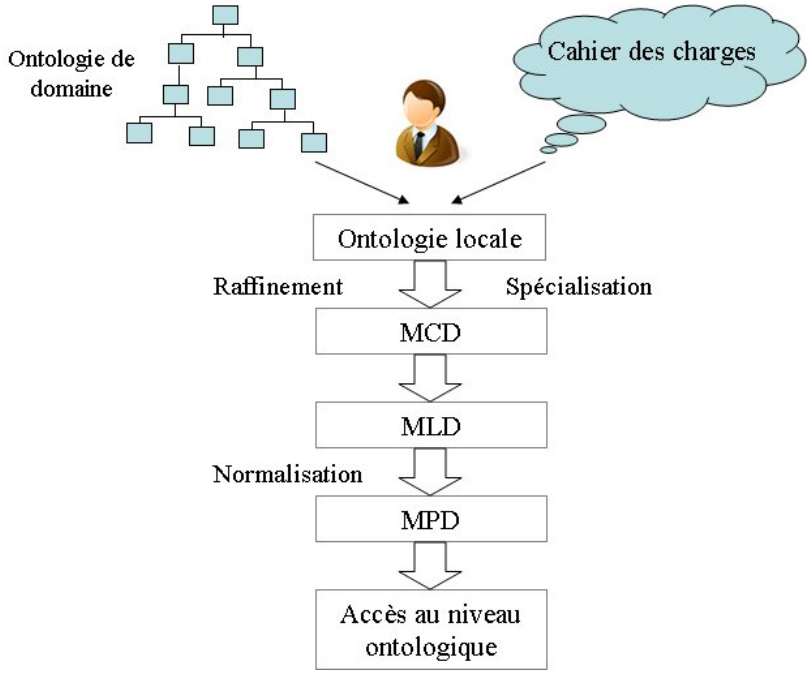

FIGURE 1.6 – La méthode SISRO.

également fait l'objet de plusieurs recherches [Borgida and Weddell, 1997, Calvanese et al., 2001, Calvanese et al., 2008, Motik et al., 2009, Romero et al., 2009, Calbimonte et al., 2009].

Dans [Borgida and Weddell, 1997], Borgida et al. ont exprimé la nécessité d'ajouter des contraintes d'unicité pour les modèles sémantiques de données, en particulier pour la logique de description tandis que dans [Calvanese et al., 2001], les auteurs ont étudié la possibilité de les ajouter à ce langage.

Dans [Calvanese et al., 2008], Calvanese et al. ont étendu le langage $\mathcal{DLR}$ avec des contraintes d'identification et des dépendances fonctionnelles. En effet, $\mathcal{DLR}$ est une famille de logique de description particulièrement adaptée pour la modélisation des schémas de base de données. Ainsi, le DL résultant, appelé DLR_{ifd}, offre la possibilité d'exprimer ces contraintes à travers de nouveaux types d'assertions dans la TBox [21]. Par exemple, une assertion de dépendance fonctionnelle est de la forme ($fd\ R\ i_1,...,\ i_h \rightarrow j$) où R est une relation, h≥2 et $i_1,...,i_h,j$ désignent des éléments de R.

Dans [Motik et al., 2009], Motik et al. ont montré le rôle des contraintes dans les ontologies tout en traçant une comparaison entre les contraintes dans les bases de données et celles dans les ontologies.

Dans [Romero et al., 2009, Calbimonte et al., 2009], les auteurs se sont intéressés à l'étude des relations de dépendances et de leur implications dans les ontologies. Dans [Calbimonte et al., 2009], les auteurs proposent un nouveau constructeur OWL pour la définition des dépen-

21. terminologie composée d'un ensemble d'axiomes qui exprime les relations existant entre concepts

dances fonctionnelles. Ils ont défini une dépendance fonctionnelle FD par le quadruplet suivant FD = (A, C, R, f) où:

- A est un antécédent i.e. est une liste de chemins (A = $\{u_1, u_2, .., u_n\}$). Un chemin u_i est à son tour composé de rôles r_i (r_i (u_i=$\{r_{i,1}, r_{i,2}, .., r_{i,n}\}$).
- C est une conséquence composée par un unique chemin (C = $\{u\}$).
- R est un concept racine représentant le point de départ de tous les chemins pour l'antécédent et la conséquence.
- f est une fonction déterministe qui prend comme paramètres les co-domaines des derniers rôles des chemins de A.

Ces dépendances sont par la suite traduites en un ensemble de règles SWRL pour le processus de raisonnement.

Dans [Romero et al., 2009], les auteurs proposent une approche pour définir les dépendances fonctionnelles d'une ontologie de domaine en s'appuyant sur les concepts et les relations définis dans une telle ontologie. Ils définissent un algorithme d'identification de dépendances fonctionnelles exploitant les capacités de raisonnement de DL_Lite_A. Précisons que DL_Lite_A est une variante de la famille DL-Lite qui est une nouvelle logique de description spécialement adaptée pour capturer les langages basiques d'ontologie. Cet algorithme permet de calculer,dans un premier temps, l'ensemble des dépendances fonctionnelles entre les concepts ontologiques puis de calculer la fermeture transitive. En effet, Romero et al. décrivent une dépendance fonctionnelle comme étant une relation permettant de décrire les instances d'un concept comme étant dépendantes fonctionnellement des instances d'un autre concept. Pour deux concepts C_1 et C_2, les auteurs ont établi que chaque instance i_1 de C_1 permet de déterminer une unique instance de C_2 : s'il existe une propriété d'objet fonctionnelle (PO_i) et que PO_i est obligatoirement valuée pour i_1 et que cette propriété relie i_1 à une instance de C_2. Cette relation de dépendance est désignée par $C_1 \rightarrow C_2$. Ces dépendances sont exploitées par la suite pendant le processus de conception des entrepôts de données pour automatiser la définition du schéma logique.

6 Conclusion

Dans ce chapitre, nous avons réalisé une étude approfondie du modèle de représentation des données sémantiques. Appelé ontologie, ce modèle décrit à la fois les données et leur sémantique dans un domaine donné. Ces données sont décrites d'une manière formelle, consensuelle et référençable et peuvent être primitives (canoniques) ou définies (non canoniques). Dans notre étude, nous nous sommes intéressés aux modèles d'ontologies et à l'étude de leur pouvoir d'expression. Notre analyse nous a permis d'identifier que ces derniers n'offrent pas de constructeurs pour la définition des relations de dépendance entre les classes et/ou les propriétés ontologiques. En effet, ce type de dépendances est plutôt défini en vue d'être exploité dans la conception des SGBD tandis que les ontologies sont essentiellement utilisés pour raisonner à propos des objets du domaine concerné. Dans nos travaux, nous nous intéressons au stockage des ontologies dans

les structures qui leur sont dédiées. De telles structures sont appelées bases de données à base ontologique et seront présentées dans le chapitre suivant. Par conséquent, un intérêt particulier est porté à la conception de ces systèmes. Ainsi, un besoin d'enrichir les modèles d'ontologies par les relations de dépendance entre les concepts ontologiques est mis en évidence. Pour ce faire, nous proposons à travers ce manuscrit d'expliciter de telles relations et de les exploiter pour la proposition d'une méthodologie de conception des bases de données à base ontologique.

Chapitre 2

Les Bases de données à base ontologique

Sommaire

Résumé. Dans le chapitre précédent, nous avons mené une analyse du domaine des ontologies sans étudier les structures dédiées à la persistance de ces modèles. Dans ce chapitre, nous nous intéressons à l'étude de telles structures, appelées bases de données à bases ontologique ($\mathcal{BDBO}$). L'objectif de cette étude est (i) d'analyser la composition de ces systèmes, (ii) d'étudier leur processus de conception et (iii) de positionner notre travail par rapport à l'existant.

1 Introduction

Le processus classique de conception d'une base de données (BD) permet de générer trois différents modèles: le modèle conceptuel, logique et physique de la base. La structure de stockage actuelle d'une BD représente le modèle logique de données, généralement sous forme d'un schéma relationnel, et stocke les données conformément à ce modèle. Dans les premiers travaux de conception des BD [Codd, 1970], la phase de la modélisation conceptuelle a été d'abord négligée avant d'être considérée comme une étape clé lors du processus de conception [Chen, 1975]. En effet, plusieurs travaux portant sur la conception des bases de données [Chen, 1975] ont donné à cette phase sa vraie valeur au sein du processus de conception. Celle-ci permet d'offrir une vue abstraite du domaine étudié tout en facilitant à l'utilisateur final la compréhension du schéma physique de la BD et son interrogation. Dans les bases de données classiques, aucune trace du modèle conceptuel n'est sauvegardée. Seul le modèle logique est représenté dans une structure classique d'une BD. Celui-ci présente un pouvoir d'expression restreint par rapport au schéma conceptuel vues les décisions d'optimisation qu'il renferme. De plus, une perte de la sémantique est ressentie lors de la traduction du modèle conceptuel vers le modèle logique. Par conséquent, une sauvegarde du modèle conceptuel au sein de la base de données s'avère nécessaire. Elle permettrait ainsi de réduire la distance entre les modèles conceptuel et logique et de faciliter la manipulation de la BD par l'utilisateur final. Des travaux de recherches portant sur la réduction de la distance entre le modèle conceptuel et logique ont été proposés [Bancilhon et al., 1992] [Stonebraker and Moore, 1996].

Pendant la même période, une nouvelle communauté de chercheurs issue de l'ingénierie des connaissances s'est intéressée à la modélisation des concepts d'un domaine et leur représentation d'une manière explicite et consensuelle. Connues sous le nom d'ontologies, ces représentations permettent de préserver la sémantique des données et de favoriser leur échange à travers les BD. Ces dernières ont connu un réel succès dans différents domaines (Web sémantique, ingénierie, e-commerce, etc.). Ainsi, la quantité de données à base ontologique a considérablement augmenté, ce qui a rendu leur gestion difficile voire incompatible avec le traitement en mémoire centrale.

Compte tenu des (1) similarités relevées entre les modèles conceptuels et les ontologies, (2) le besoin de représenter à la fois les modèles conceptuel et logique au sein de la même base de données, et (3) la nécessité de satisfaire les besoins de performance requis pour de nombreuses applications et le besoin de persister la grande masse de données ontologiques, une nouvelle structure de stockage, appelée base de données à bases ontologiques ($\mathcal{BDBO}$), a été proposée. Ce type de base de données permet à la fois de stocker les données ontologiques et le modèle les référençant (l'ontologie). La figure 2.1 illustre la construction des $\mathcal{BDBO}$. Plusieurs $\mathcal{BDBO}$ ont été proposés dans la littérature incluant Jena [Carroll et al., 2004, Wilkinson et al., 2003], Oracle [Chong et al., 2005], Sesame [Broekstra et al., 2002], SOR [Lu et al., 2007], DLDB [Pan and Heflin, 2003], OntoDB [Dehainsala et al., 2007], OntoMS [Park et al., 2007], RDFSuite [Alexaki et al., 2001b], etc. Ces $\mathcal{BDBO}$ se différencient principalement par (a) leurs

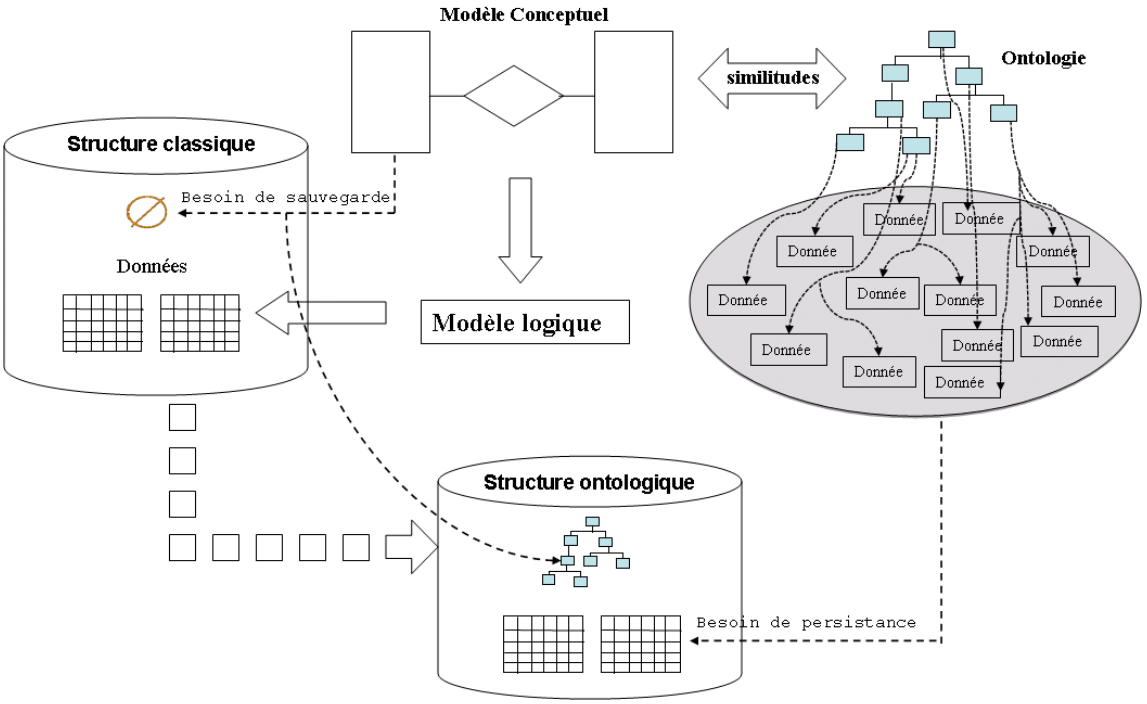

FIGURE 2.1 – Traduction du besoin des bases de données à base ontologique.

architectures, (b) leurs modèles de stockage, (c) les mécanismes utilisés pour définir le lien entre les instances et les concepts ontologiques et (d) les langages d'exploitation (RQL, SPARQL, OntoQL, etc.). En revanche, elles implantent toutes le modèle logique "en dur" (modèle logique figé) sans tenir compte de la phase de modélisation conceptuelle.

Dans ce chapitre, notre objectif est de bien positionner notre travail par rapport à l'état de l'art sur les bases de données à base ontologique et leur processus de conception.

Dans la section qui suit, nous commençons par représenter les données à base ontologique. Dans la section 3, nous introduisons les structures de persistance de ces données, appelées bases de données à bases ontologique. Une classification des $\mathcal{BDBO}$ est proposée dans la section 4. Dans la section 5, nous présentons une étude détaillée d'un ensemble de $\mathcal{BDBO}$ existantes. Avant de conclure, nous décrivons les processus de conception des bases de données à base ontologique de la littérature tout en positionnant nos travaux à travers la section 6.

2 Données à base ontologique

Nous appelons donnée à base ontologique toute donnée dont la signification est définie explicitement par référence à une ontologie. Étant donnée qu'une ontologie est un formalisme de modélisation à base de *classe*, *propriété* et *instance*, une donnée à base ontologique peut être

décrite comme étant la totalité ou une partie d'un individu appartenant à une classe ontologique caractérisée par un ensemble de propriétés applicables de la classe dont il est instance. Les références aux classes sont établies à l'aide d'identifiants universels fournis par les formalismes d'ontologies.

Ces données étaient, initialement, gérées en mémoire centrale par les systèmes informatiques. Durant cette dernière décennie, nous avons assisté à un accroissement considérable du volume des données à base ontologique qui fait que leur gestion en mémoire centrale devient le talon d'Achille des applications. Le besoin d'une structure de données offrant à la fois la performance du système et la persistance des données s'avère nécessaire. Une telle structure doit offrir un ensemble de fonctionnalités telles que suit :

- persister à la fois les ontologies et les données ;
- être doté d'un langage d'interrogation ;
- stocker une masse importante de données et permettre le passage à l'échelle ;
- offrir une certaine flexibilité permettant l'évolution du modèle d'ontologie par l'ajout de constructeurs, de type de données, etc ;
- définir des mécanismes d'inférence permettant le raisonnement sur les données canoniques et non canoniques ;
- offrir un mécanisme de gestion des données à caractère canonique et non canonique ;
- être conçue selon un processus de conception éliminant toute forme de redondance ;
- offrir des mécanismes d'optimisation.

Dans le but de répondre à de telles exigences, un effort considérable a été fourni par un ensemble de chercheurs et d'industriels proposant ainsi une nouvelle structure permettant la représentation des données à base ontologique dans des systèmes de gestion de bases de données. Celle-ci offre à la fois la performance du système et la persistance de ses données comme cela est défini dans la section suivante.

3 Bases de données à base ontologique

Dans cette section, nous introduisons les bases de données à base ontologiques ($\mathcal{BDBO}$). D'abord, nous présentons leur définition. Ensuite, nous décrivons leurs architectures et les modèles de stockage qu'elles utilisent pour la persistance des données. Enfin, leurs principaux langages d'interrogation sont identifiés.

3.1 Définition

Une base de données à base ontologique est une base de données contenant à la fois des ontologies, un ensemble de données et des liens entre ces données et les éléments ontologiques qui en définissent le sens [Pierra et al., 2005] [Hondjack, 2007]. Le modèle de la base de données dans cette nouvelle structure offre un socle commun où cohabitent plusieurs modèles, essentiel-

lement le modèle logique et le modèle conceptuel de données. La principale nouveauté de cette structure de stockage est de représenter à la fois l'ontologie et les données référencées par cette ontologie au sein de la même base de données. Généralement, cette représentation peut différer d'une $\mathcal{BDBO}$ à une autre selon deux principaux critères: l'architecture de la base de données et les modèles de stockage utilisés. Nous détaillons ces derniers dans les deux sections qui suivent.

3.2 Architecture d'une base de données à base ontologique

Nous appelons architecture d'une base de données à base ontologique l'organisation des modèles les composant. Quatre principales composantes peuvent définir une $\mathcal{BDBO}$: (a) la composante méta-base, (b) la composante données, (c) la composante ontologie et (d) la composante méta-schéma.

3.2.1 La composante Méta-base

Cette composante est une partie traditionnelle des bases de données classiques. Elle contient l'ensemble des tables systèmes permettant la gestion et le bon fonctionnement de l'ensemble des données contenues dans la base. Appelée aussi *system catalog*, elle représente une composante essentielle de toute architecture de base de données.

3.2.2 La composante Données

Elle représente généralement les instances de classes de l'ontologie et les valeurs de propriétés ontologiques. Cette composante peut, dans certaines architectures de $\mathcal{BDBO}$, stocker à la fois les instances ontologiques et la description de l'ontologie les référençant.

3.2.3 La composante Ontologie

Cette partie permet de stocker un ensemble d'ontologies représentant la sémantique des différents domaines couverts par la base de données. Elle inclut la représentation de la description des classes, des propriétés et de leur hiérarchies respectives. Notons que dans certains systèmes, les composantes ontologie et données peuvent être fusionnées (voir section 4).

3.2.4 La composante Méta-schéma

Cette partie joue le même rôle pour la partie ontologie que la partie méta-base pour la partie données. Elle permet de décrire le modèle d'ontologies et d'adapter ce modèle aux évolutions quand le besoin se présente. Notons que cette partie peut être absente dans certaines $\mathcal{BDBO}$.

L'architecture d'une base de données à base ontologique diffère d'une $\mathcal{BDBO}$ à une autre. Elle peut intégrer la totalité des composantes ou une partie d'entre elles. Une classification des $\mathcal{BDBO}$ selon leur architecture est proposée dans la section 4.

3.3 Modèles de stockage

En se penchant sur la littérature, différents modèles de stockage ont été définis pour la persistance des ontologies et des données les référençant [Agrawal et al., 2001, Alexaki et al., 2001a, Alexaki et al., 2001a, Pan and Heflin, 2003]. Trois principales représentations sont identifiées [Pan and Heflin, 2003]: (1) la représentation verticale, (2) la représentation horizontale et (3) la représentation hybride. Nous détaillons chacune d'entre elles dans les paragraphes suivants.

3.3.1 Représentation verticale

Appelée représentation générique [Alexaki et al., 2001a], cette approche admet une table verticale où chaque enregistrement correspond à un triplet RDF (sujet, prédicat, objet) tel qu'illustré dans la figure 2.2.(a). Une colonne peut identifier une propriété, une classe, une instance de propriété ou une instance de classe. Une deuxième variante de cette approche, appelée approche verticale avec ID, consiste à stocker respectivement l'identifiant (URI) des ressources et toutes les valeurs littérales dans des tables spécifiques telles que RESSOURCES et LITERAL comme indiqué dans la figure 2.2.(b). La table RESSOURCES est composée de deux colonnes. La première représente l'identifiant ID de type ENTIER et la deuxième introduit l'URI représentant l'URI des ressources. De même, la table LITERAL est constituée de deux colonnes : la colonne ID de type Entier et la colonne value pour les valeurs littérales. La colonne ID de ces deux tables est référencée dans la table TRIPLES.

Cependant, avec cette représentation, l'interrogation de la base de données est coûteuse vu le besoin d'effectuer un nombre élevé d'auto-jointures. La mise à jour d'une information nécessite, dans la plupart du temps, la modification d'un ensemble de triplets.

3.3.2 Représentation horizontale

Cette approche est similaire à la représentation traditionnelle utilisée par les SGBD relationnels. Avec cette représentation, une table horizontale par classe ontologique est créée. Les propriétés ayant pour domaine une classe particulière représentent les colonnes de la table qui lui est associée. Notons que cette représentation couvre les inconvénients de l'approche précédente puisque chaque table se rapporte à une classe distincte. Par conséquent, on a moins de perte d'espace et une réduction du coût d'interrogation. Cependant, il reste la question des propriétés dont le domaine n'est pas spécifié. Les intégrer en tant que colonnes dans toutes les

TRIPLES		
Subject	Predicate	Object
University	Type	Class
name	Type	University
idUniv	Type	property
idUniv	domain	University
name	domain	University
…	…	…
University_1	Type	University
University_1	idUniv	150
University_1	name	Université de Poitiers
…	…	…

(a) Approche verticale

TRIPLES		
Subject	Predicate	Object
j_1	Type	C_1
j_1	P_1	I_1
…	…	…

LITERAL	
ID	VALUE
I_1	Université de Poitiers
…	…

RESSOURCES	
ID	URI
C_1	University
P_1	name
P_2	idUniv
j_1	University_1
name	Type

(b) Approche verticale avec ID

FIGURE 2.2 – Illustration de l'approche verticale.

tables créées serait très coûteux.

3.3.3 Représentation hybride

Connue sous le nom de représentation spécifique [Alexaki et al., 2001a], ce modèle est composé d'une table unaire créée pour chaque classe de l'ontologie et une table binaire définie pour chaque propriété de l'ontologie. Cette approche se décline en trois variantes selon l'approche adoptée pour la représentation de l'héritage : (1) une table unique pour toutes les classes de l'ontologie, (2) une table par classe avec héritage de table (si un SGBD relationnel objet est utilisé) et (3) une table par classe sans héritage de table.

Ces différents modèles permettent le stockage des données dans les bases de données à base ontologique. Afin de les manipuler, un ensemble de langages d'interrogation a été fourni.

3.4 Langages d'interrogation

En examinant la littérature, plusieurs modèles d'ontologies ont été proposés. Ces derniers ont été accompagnés par une panoplie de langages d'interrogation, définis (i) dans le domaine du Web Sémantiques comme SPARQL [Prud'hommeaux and Seaborne,], RDQL [22] pour l'interrogation de données RDF et RQL [Karvounarakis et al., 2002] et SeRQL [Broekstra and Kampman, 2004] pour les données RDF-Schéma ou (ii) dans le domaine de l'ingénierie comme CQL [23] et OntoQL [Jean et al., 2006a] pour des données définies avec le modèle d'ontologie

22. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/
23. http://www.toplib.com/en/aboutCQL.php

PLIB. Nous passons en revue, ci-dessus, quelques langages d'exploitation de données à base ontologique.

3.4.1 SPARQL

SPARQL [24] (SPARQL Protocol and RDF Query Language) est un langage d'interrogation pour les données représentées en RDF. Il permet d'exprimer des interrogations à travers diverses sources de données (à travers les requêtes SELECT, CONSTRUCT, DESCRIBE et ASK) et de rechercher des motifs de graphe (graph patterns) sur le graphe décrit par les données RDF. Plusieurs outils [25] manipulant ce langage ont été proposés. Nous citons à cet égard l'exemple de SPARQL Software et Twinkle.

3.4.2 RQL

Avec l'apparition du modèle d'ontologie RDF-Schema, plusieurs langages d'interrogation dédiés à l'interrogation des données et des ontologies représentées avec ce modèle ont été proposés. RQL [Karvounarakis et al., 2002] a été l'un des premiers langages défini. Il est basé sur un modèle de données formel ayant la particularité de distinguer les différents niveaux manipulés: données (instances), ontologies (concepts ontologiques) et modèles d'ontologies.

3.4.3 OntoQL

OntoQL [Jean et al., 2006a] est un langage de requête pour l'exploitation des bases de donnés à base ontologique. Implémenté sur la $\mathcal{BDBO}$ OntoDB, il permet de définir, modifier et interroger les ontologies et ses données et d'étendre, si besoin, le modèle d'ontologie utilisé pour la définition des ontologies. Ainsi, OntoQL permet la manipulation directe des modèles et des concepts ontologiques tout en cachant les représentations internes.

Une multitude de langages d'interrogation a été proposée. Ces langages sont spécifiques aux modèles d'ontologies utilisés et diffèrent d'une base de données à une autre. De plus, différentes architectures et un ensemble de modèles de stockage ont été dédié aux différentes $\mathcal{BDBO}$. Dans la section suivante, nous proposons une classification des $\mathcal{BDBO}$ dirigée par les architectures.

4 Classification des bases de données à base ontologique

En se basant sur les différents niveaux d'abstraction supportés par les bases de données à base ontologique, trois principales architectures de $\mathcal{BDBO}$ ont été identifiées: (1) les $\mathcal{BDBO}$ de

24. http://www.yoyodesign.org/doc/w3c/rdf-sparql-query/#sparqlDefinition
25. http://www.w3.org/wiki/SparqlImplementations

Type$_1$, (2) les $\mathcal{BDBO}$ de Type$_2$ et (3) les $\mathcal{BDBO}$ de Type$_3$. Dans cette section, nous présentons ces différents types d'architectures tout en décrivant les possibilités d'extensions permises par ces dernières.

4.1 $\mathcal{BDBO}$ de Type$_1$

Cette architecture est similaire aux architectures de bases de données classiques. Elle est composée de deux parties illustrées sur la figure 2.3. La première partie concerne la définition de la méta-base. La seconde partie représente les données. Celle-ci impose ainsi le même schéma de stockage pour le schéma ontologique et pour les données à base ontologique. Cette architecture a été dédiée principalement au stockage des triplets RDF contenant à la fois la définition d'ontologies et d'instances. En effet, les informations sont représentées en utilisant un seul schéma composé d'une unique table de triplets composée de trois colonnes (sujet, predicat, objet) représentant respectivement le sujet, le prédicat et l'objet. Au niveau ontologie, les colonnes représentent respectivement l'identifiant d'un élément d'ontologie, un prédicat et la valeur du prédicat pouvant être soit un identifiant d'un élément d'ontologie, soit une valeur littérale. L'exemple *(grade, domain, Person)* indique que le domaine de la propriété *grade* est la classe personne (*Person*). Au niveau instance, les colonnes représentent respectivement l'identifiant d'une instance, une caractéristique de celle-ci et la valeur de cette caractéristique. L'exemple *(Dilek, grade, PhD_Student)* indique que *Dilek* a le grade de doctorante. Les principales $\mathcal{BDBO}$ dédiées au stockage de triplets RDF et utilisant un seul schéma pour stocker les ontologies et leurs instances sont 3Store [Harris and Gibbins, 2003], Jena [Carroll et al., 2004], Oracle [Chong et al., 2005] et Sward [Petrini and Risch, 2007]. Notons que la plupart de ces $\mathcal{BDBO}$ supportent la sémantique d'OWL ou de RDFS grâce à la capacité des données RDF à supporter des descriptions d'ontologies RDFS ou OWL.

L'étude de ce premier type de $\mathcal{BDBO}$ nous permet de conclure que cette architecture ne permet pas de dissocier le schéma de l'ontologie des instances ontologiques. Pour pallier cet inconvénient, une deuxième architecture a été proposée, elle est décrite dans le paragraphe suivant.

4.2 $\mathcal{BDBO}$ de Type$_2$

Cette architecture a la particularité de stocker les ontologies et leurs instances dans deux parties séparées telqu'illustré sur la figure 2.4. Les données du niveau ontologie et du niveau instances sont stockées dans deux schémas distincts. Le premier schéma, dédié au niveau ontologique, dépend du modèle d'ontologie utilisé (OWL, PLIB, RDFS, etc.). Il est composé de table(s) figée(s) pour le stockage des différents concepts ontologiques tels que les classes, les propriétés ou les relations de subsomption. De la même manière, le schéma correspondant au niveau instances est encodé en dur. Plusieurs schémas ont été proposés selon les différentes

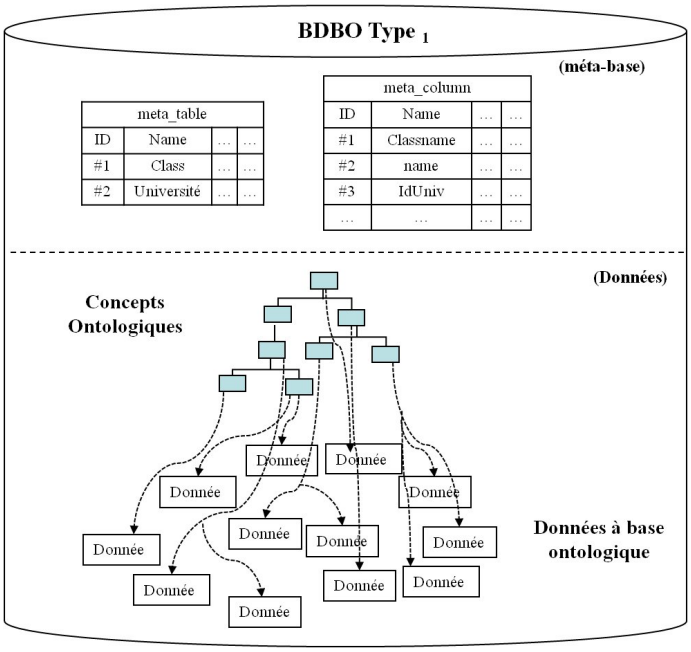

FIGURE 2.3 – Architecture $Type_1$.

$\mathcal{BDBO}$ existantes. Les unes utilisent la table verticale, d'autres exploitent la représentation binaire, etc. Parmi les principales $\mathcal{BDBO}$ de $Type_2$, nous citons RDFSuite [Alexaki et al., 2001b], Sesame [Broekstra et al., 2002], Sor [Lu et al., 2007], DLDB [Broekstra et al., 2002] et OntoMS [Park et al., 2007].

Contrairement à l'architecture de $Type_1$, cette architecture offre une séparation entre les deux schémas de stockage (modèle et instances). Mais, elle manque de flexibilité dans la mesure où elle impose un schéma d'ontologie figé ne favorisant pas l'introduction de nouveaux concepts issus d'autres modèles d'ontologies. Ainsi, une troisième architecture répondant à cette insuffisance a été proposée. Elle est décrite dans le paragraphe suivant.

4.3 $\mathcal{BDBO}$ de $Type_3$

Cette architecture de base de données a été conçue de sorte à ce qu'elle soit extensible afin de pouvoir y représenter d'autres types de modèles et ce grâce à sa partie *méta-schéma*. En effet, une $\mathcal{BDBO}$ de $Type_3$ est une $\mathcal{BDBO}$ de $Type_2$ contenant un niveau supplémentaire noté méta-schéma stockant le modèle d'ontologies dans un méta-modèle réflexif telqu'illustré dans la figure 2.5. Le méta-schéma permet (1) un accès générique aux ontologies, (2) l'évolution du modèle d'ontologies utilisé et (3) le stockage de différents modèles d'ontologies (OWL, PLIB, etc.). OntoDB [Dehainsala et al., 2007] est un exemple de $\mathcal{BDBO}$ de $Type_3$. Cette $\mathcal{BDBO}$ a la particularité d'offrir aux concepteurs la possibilité d'étendre le schéma de l'ontologie afin de représenter les nouveaux concepts non supportés par les modèles d'ontologies existants. Plu-

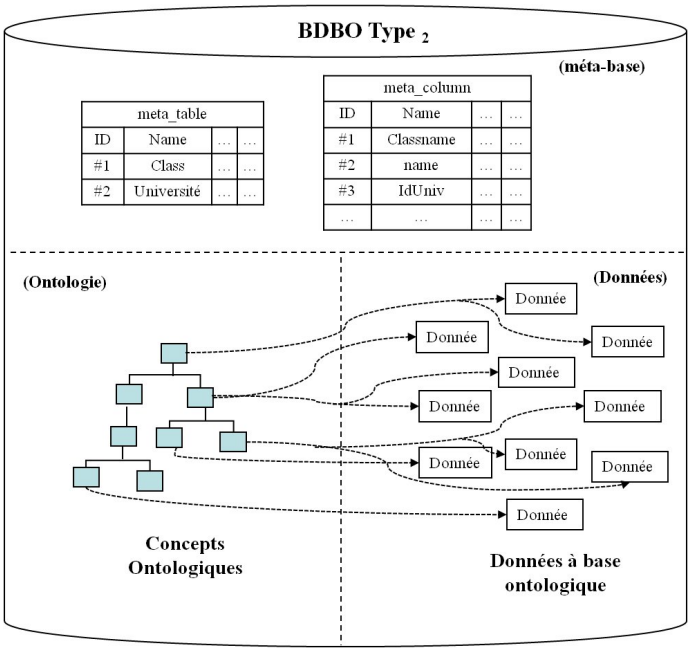

FIGURE 2.4 – Architecture $Type_2$.

sieurs travaux se sont intéressés à l'extension du méta-modèle dans ce type de base de données. Dans [Tapucu et al., 2009], les auteurs modélisent les préférences dans le but de permettre leur partage et leur réutilisation entre applications. En effet, les préférences expriment les attentes des utilisateurs qui veulent trouver dans la réalité ce qui correspond le mieux à leur souhait. Tapucu et al. proposent une approche consistant à traiter ces préférences au niveau des ontologies de domaine qui décrivent le ou les domaines abordés par ces applications. Un modèle de préférences basé principalement sur les types de préférences proposés dans les approches issues des communautés Base de Données et Web Sémantique est défini. Afin de pouvoir représenter ce modèle, ils proposent d'étendre le méta-modèle de la $\mathcal{BDBO}$ OntoDB permettant ainsi la liaison du modèle de préférences au modèle d'ontologie utilisé. Dans [Belaid et al., 2009], Belaid et. al proposent un modèle représentant les services informatiques ainsi que les concepts d'ontologies de services qui en définissent le sens. Afin de permettre la manipulation des modèles de données et des services, Belaid et. al proposent l'extension du méta-schéma d'OntoDB pour la représentation et l'exploitation du modèle proposé.

Après avoir présenté une classification des $\mathcal{BDBO}$, nous proposons, dans la section suivante, une analyse d'un ensemble de ces systèmes.

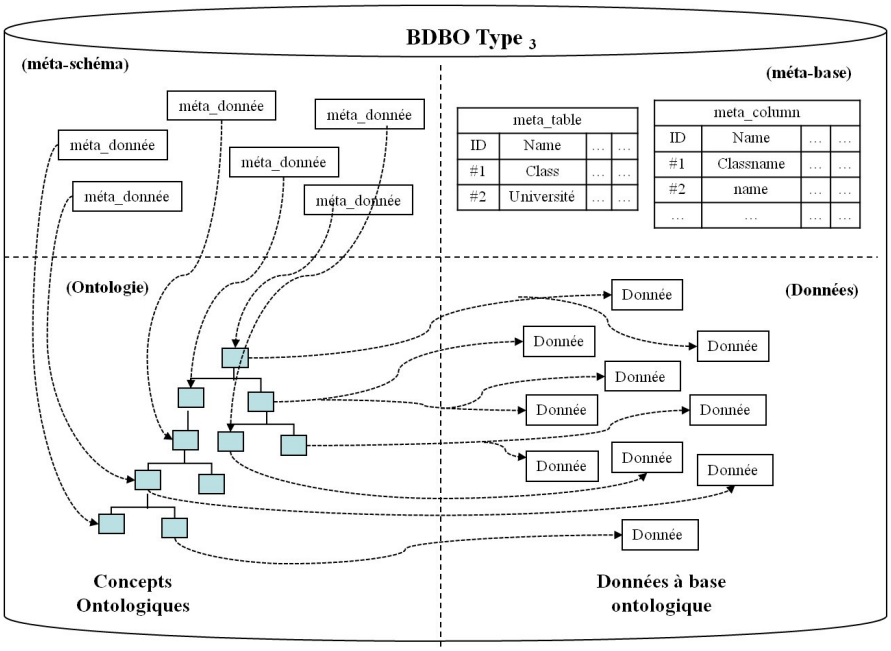

FIGURE 2.5 – Architecture $Type_3$.

5 Exemples de bases de données à base ontologique

Dans cette section, nous présentons un ensemble de systèmes de bases de données à base ontologique existants parmi une multitude de $\mathcal{BDBO}$. Nous proposons l'étude de sept $\mathcal{BDBO}$ dont deux issues du monde industriel et cinq issues du monde académique. Nous avons estimé qu'elle étaient représentatives des différentes solutions de $\mathcal{BDBO}$ proposées. L'objectif d'une telle étude est d'identifier l'architecture de la $\mathcal{BDBO}$, les modèles de stockages utilisés pour la représentation des ontologies et des données à base ontologique, le formalisme d'ontologie supporté, le langage d'interrogation dédié à la base et la flexibilité quant à la représentation et la persistance des données.

5.1 Systèmes industriels

Deux principaux systèmes ont été identifiées : la $\mathcal{BDBO}$ d'Oracle [Chong et al., 2005] et la $\mathcal{BDBO}$ d'IBM Sor [Lu et al., 2007].

5.1.1 Oracle

En 2005, dans sa version 10g, Oracle [Chong et al., 2005] a proposé son premier système de gestion des données RDF. Il offre ainsi à ses utilisateurs une base de donnés à base ontologique permettant le stockage des ontologies et des données sémantiques, leur interrogation

et l'inférence sur ces données. Dans cette $\mathcal{BDBO}$, toutes les données (ontologies et instances) sont stockées sous forme de triplets dans une unique table. Dans le but d'améliorer le modèle de stockage, cette table a subi une décomposition [Wu et al., 2008] dans laquelle les données RDF/OWL sont persistées dans un ensemble de tables dont les principales sont: *RDF_Link*$ et *RDF_Value*$. La table *RDF_Link*$ stocke les triplets sous forme normalisée à l'aide d'identifiants (ID). La table *RDF_Value*$ persiste la correspondance entre chaque ID et l'URI qui lui correspond et stocke les littéraux. Chaque littéral a une forme canonique. Elle correspond à sa première forme rencontrée. Pour faciliter les comparaisons, les formes équivalentes rencontrées par la suite référencent la forme canonique. Oracle offre aux utilisateurs une base de règles logique dont les résultats peuvent être matérialisés. Afin de faire des déductions de connaissances, ces règles sont spécifiées explicitement lors de l'interrogation des données à travers le langage SPARQL. Notons que le système Oracle permet d'interroger ces données par le biais d'une syntaxe SQL établie grâce à la fonction RDF_MATCH accédant aux données sémantiques. Autrement dit, une même requête SQL permet à la fois d'accéder aux données RDF/RDFS et aux autres données rendues persistantes dans la base de données. Ainsi, le système Oracle peut adopter deux représentations différentes pour le mécanisme de stockage. La première consiste à stocker simultanément les ontologies et ses instances à travers une représentation verticale. La deuxième consiste à séparer le stockage des ontologies et ses données en utilisant la représentation verticale pour la persistance des ontologies et les tables relationnelles usuelles pour le stockage des données.

Afin de charger l'ontologie et ses données, Oracle nécessite une phase de préparation suivant l'enchaînement d'un ensembles d'étapes :

- *Activation des modules sémantiques.* Cette étape consiste en la création d'un tablespace pour les tables système.
- *Création du réseau sémantique.* Cette étape permet d'ajouter le support de données sémantiques à une base de données Oracle. La création de ce réseau se fait en tant qu'utilisateur avec des privilèges d'administrateur, en spécifiant un tablespace valide avec un espace suffisant. Notons que ce réseau est créé une seule fois.
- *Création d'une table sémantique.* Après avoir créé le réseau sémantique, une table stockant les références aux données sémantiques est créée. Cette table doit avoir une colonne de type $SDO\_RDF\_TRIPLE\_S$ contenant les références de toutes les données associées à un modèle unique.
- *Création du modèle sémantique.* La dernière étape de la phase de préparation au chargement consiste en la création du modèle sémantique. Lors de sa création, la spécification de son nom, de la table stockant les références de ses données sémantiques et de la colonne déjà créée de type $SDO\_RDF\_TRIPLE\_S$ est obligatoire.

5.1.2 SOR

SOR (Scalable Ontology Repository) est un système proposé par IBM sur le SGBD DB2 [Lu et al., 2007] offrant aux utilisateurs le stockage, l'interrogation et le raisonnement sur des ontologies RDF/RDFS et OWL. D'une manière similaire à Sesame, SOR propose une représentation des données à base ontologique reposant sur la séparation de la représentation des ontologies et des instances. En effet, la représentation des ontologies repose sur un schéma figé dépendant du modèle d'ontologie utilisé (RDFS, OWL). Ce schéma représente un ensemble de constructeurs relatif au langage d'ontologie adapté. Par exemple, SOR définit les tables HASVALUE, INTERSECTION, SUBCLASSOF, CLASSES, etc décrivant respectivement les constructeurs de contrainte de valeur, d'intersection, de subsomption et de définition des classes. Ce système permet le stockage des classes non canoniques (CNC) dans la base de données en (a) décomposant la CNC en instanciations de constructeurs de classes OWL, (b) attribuant un nouvel ID à chaque instanciation et (c) la stockant dans la classe correspondante. Prenons l'exemple du concept FMasterStudent $\equiv$ Woman $\cap \exists$ level.master désignant les étudiantes en master. Tout d'abord, S_1 est défini pour la représentation de $\exists$ level.master dans la table HASVALUE traduisant le fait d'avoir le niveau de master. Ensuite, I_1 représentant l'intersection entre la classe Woman décrivant les femmes et S_1 est défini dans la table INTERSECTION. Finalement, les relations de subsomption FMasterStudent $\subseteq I_1$ et $I_1 \subseteq$ FMasterStudent sont représentées dans la table SUBCLASSOF. Au niveau données, la représentation hybride est adoptée. SOR admet SPARQL comme langage d'interrogation. Il est doté d'un module d'inférence constitué d'un raisonneur de logique de description (raisonneur DL) et d'un moteur d'inférence basé sur un ensemble de règles permettant ainsi le raisonnement au nivau structure qu'au niveau instances.

5.2 Systèmes académiques

Dans cette partie, nous présentons les $\mathcal{BDBO}$ Jena [Carroll et al., 2004, Wilkinson et al., 2003], Sesame [Broekstra et al., 2002], DLDB [Pan and Heflin, 2003], OntoMS [Park et al., 2007] et OntoDB [Dehainsala et al., 2007].

5.2.1 Jena

Développé initialement par les chercheurs dans les laboratoires HP en 2000, Jena [26] [Carroll et al., 2004] est un framework open-source permettant la manipulation des ontologies définies avec les langages RDF, RDFS, DAML+OIL et OWL. Il a été largement utilisé dans une grande variété d'applications du web sémantique. Il offre aux utilisateurs une API de programmation pour la gestion des données et des applications du Web sémantique, un moteur d'inférence interne et un parseur RDF/XML. L'interrogation des données est assurée par le langage de requêtes SPARQL. Jena peut être déployé sur différents SGBD relationnels tels que PostgreSQL,

26. http://jena.sourceforge.net/

MySQL, Oracle, etc. et ce via JDBC. Il adopte la représentation verticale au sein d'une seule table de triplets pour la persistance des ontologies et des instances ontologiques. Plusieurs versions de Jena ont vu le jour. La version actuelle de Jena est Jena2 [Wilkinson et al., 2003]. Dans chacune de ces versions, différents schémas de bases de données ont été proposés. Dans Jena1, la structure de la table verticale (sujet, prédicat, objet) est adoptée pour la persistance à la fois des ontologies et des données à base ontologique. Dans Jena2, l'approche utilisée est la deuxième variante de l'approche verticale avec ID. Les ressources et les valeurs des propriétés de type littéral (types simples) sont stockées séparément dans les tables *ressources* et *litteral* (voir figure 2.2.(b)).

5.2.2 Sesame

Développée par une communauté de développeurs de la société Aduna [27], Sesame [Broekstra et al., 2002] est une base de données à base ontologique dédiée à l'analyse, au stockage, à l'inférence et à l'interrogation des données et des méta-données RDF et RDFS. Elle est basée sur des normes élaborées par le W3C et est disponible sous une licence open-source libérale (BSD). Sesame est caractérisée par sa rapidité de chargement de triplets RDF et le support de plusieurs langages de requêtes RDF comme RQL, SPARQL et SeRQL. Elle offre une API (SAIL API) pouvant être connectée à toutes les solutions de stockage RDF. En effet, Sesame peut être déployée sur un ensemble de systèmes de stockage (bases de données relationnelles, systèmes de fichiers, etc) sans la nécessité de changer les modules fonctionnels et le moteur de requêtes. Quatre principaux modules sont identifiés :

1. le module 'Query model' implémente le moteur de requêtes du langage RQL ;
2. le module 'Export module' permet d'exporter les instances de la base de données en format XML-RDF ;
3. le module 'Admin module' assure l'insertion des données RDF dans la base de données ainsi que leur suppression ;
4. le module 'RDF SAIL' (RDF Storage And Inference Layer) englobe l'ensemble des interfaces conçues pour la persistance et la récuperation des données RDFS. Cette couche admet une implémentation spécifique pour chaque système de base de données. Dans [Broekstra et al., 2002], Broekstra et.al proposent deux implémentations de cette couche, chacune basée sur un schéma de base de données spécifique (PostgreSQL et MySQL).

A la différence d'Oracle, Sesame stocke séparemment les données des deux niveaux ontologique et données et ce dans deux différents schémas. Au niveau ontologique, il admet une représentation dépendante du modèle d'ontologie supporté. Le schéma adopté est composé de tables correspondantes au stockage de chaque constructeur d'ontologies tels que (i) la table 'Class' stockant les classes ontologiques en indiquant leurs identifiants externe (URI) et interne

27. http://www.openrdf.org/consulting.jsp

(id), (ii) la table 'SubClassOf' persistant la hiérarchie de ces classes en indiquant leurs super-classes, (iii) la table 'Property' destinée au stockage des propriétés, etc. D'une manière similaire aux classes, les propriétés sont associées à des identifiants. De plus, elles admettent un domaine (domain) et un codomaine (range). Au niveau instance, une représentation hybride est adoptée.

5.2.3 DLDB

DLDB [Pan and Heflin, 2003] est une base de données à base ontologique basée sur un système de gestion de base de données relationnelle étendu par des fonctionnalités supplémentaires pour le support et l'inférence des données DAML+OIL dans sa première version (DLDB1) et OWL DL [28] dans sa deuxième version (DLDB2 [Pan et al., 2008]). DLDB admet la représentation hybride pour le stockage des données. Celles-ci sont stockées par le biais d'une table unaire créée pour chaque classe et une table binaire associée à chaque définition de propriété. Seules les classes et les propriétés atomiques sont représentées. La hiérarchie des classes est stockée à l'aide des vues définies de manière récursive. En effet, une vue de classe consiste en l'union de sa table et de toutes les vues de ses sous-classes directes. Ainsi, la vue de classe contient à la fois les instances explicitement typées ainsi que celles inférées. Notons que son mécanisme de raisonnement, établi par le raisonneur Fact [29], présente des insufisances sur les inférences établies sur les constructeurs ontologiques *inverseOf*, *equivalentClass*, *transitive*, *hasValue*, etc.

5.2.4 OntoMS

OntoMS [Park et al., 2007] est le premier système de gestion de données OWL relationnel. Il stocke séparemment l'ontologie et ses instances dans deux différents schémas. Ce système propose une représentation des données à base ontologique dépendant de l'ontologie qui décrit ces données. Cette représentation, dite horizontale, consiste à stocker les instances d'une classe ainsi que ses valeurs de propriétés dans une table relationnelle. Lorsqu'une propriété est multi-valuée, ses valeurs peuvent être représentées en utilisant de nouvelles tables pour ces propriétés. OntoMS offre un mécanisme de raisonnement complet pour les instances sur les inférences établies sur les constructeurs ontologiques *inverseOf*, *symmetric* et *transitive*.

5.2.5 OntoDB

OntoDB [Dehainsala et al., 2007] est une $\mathcal{BDBO}$ développée au sein du Laboratoire d'Informatique Scientifique et Industrielle (LISI). Sa première version a été implantée sur le SGBD PostgreSQL et manipule le modèle d'ontologies PLIB. Elle permet de stocker explicitement les données et l'ontologie définissant leur sens, d'y accéder et de faire évoluer le modèle d'ontologies utilisé. En effet, l'architecture d'OntoDB se compose de quatre parties: (1) la partie

28. http://www.obitko.com/tutorials/ontologies-semantic-web/owl-dl-semantics.html
29. http://owl.man.ac.uk/factplusplus/

méta-base, (2) la partie méta-schéma, (3) la partie ontologie et (4) la partie données. Grâce à sa partie méta-schéma, OntoDB permet le support des évolutions du modèle d'ontologies PLIB et l'extension de ce modèle lorsque le besoin se présente. L'interrogation des données au niveau ontologique est assurée par le langage de requête OntoQL [Jean et al., 2006a]. OntoDB definit deux tables, nommées *Entity* et *Attribute* pour la description du modèle d'ontologie. La première table définit les entités du méta-schéma telles que Class, Property, etc. décrivant respectivement les classes et les propriétés. Tandis que la deuxième table définit les attributs tels que *name*, *scope* décrivant respectivement le nom de l'entité et son domaine. Les ontologies sont représentées dans un schéma figé définissant les constructeurs ontologiques correspondant à l'instanciation des entités et des attributs définis au niveau méta-schéma. Parmi ces tables, nous citons la table *Class* et la table *Property* persistant respectivement les classes et les propriétés de l'ontologie manipulée. Une représentation horizontale est exploitée pour la persistance des données à base ontologique. En effet, une table est créée pour chaque classe de l'ontologie dont les colonnes correspondent au sous-ensemble de propriétés applicables de la classe en question. La figure 2.6 illustre l'architecture d'OntoDB. Notons que cette $\mathcal{BDBO}$ impose le typage des propriétés (possession d'un domaine et codomaine uniques), la mono-instantiation et le typage fort des données à base ontologique.

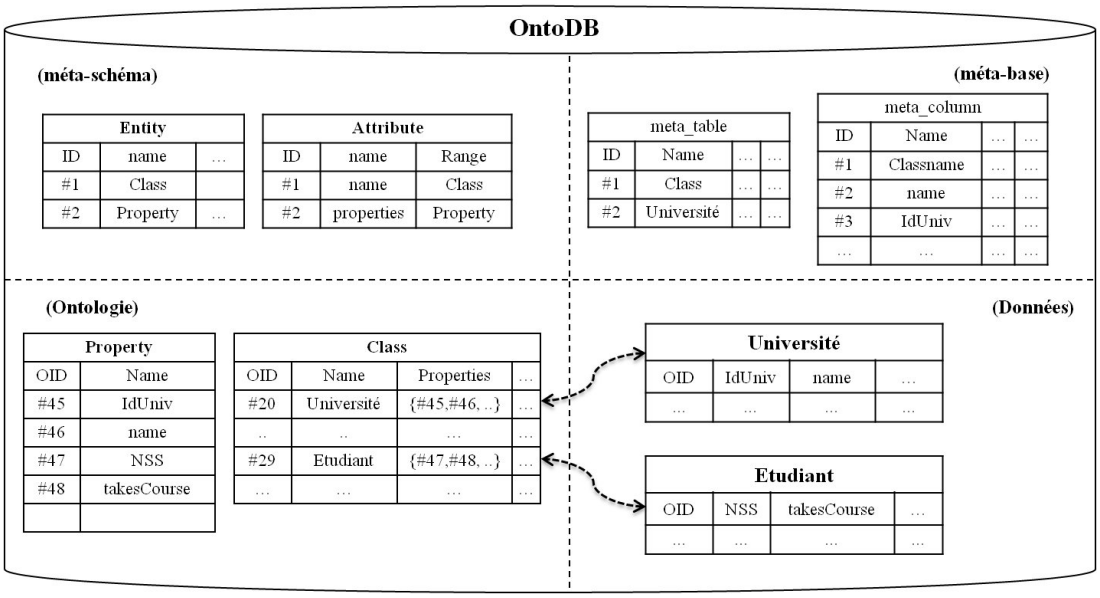

FIGURE 2.6 – Architecture d'OntoDB.

5.3 Synthèse sur les $\mathcal{BDBO}$ étudiées

Les différentes $\mathcal{BDBO}$ permettent l'analyse, la persistance et l'interrogation des ontologies et des données la référençant dans un même référentiel. Cependant, aucune $\mathcal{BDBO}$ ne permet

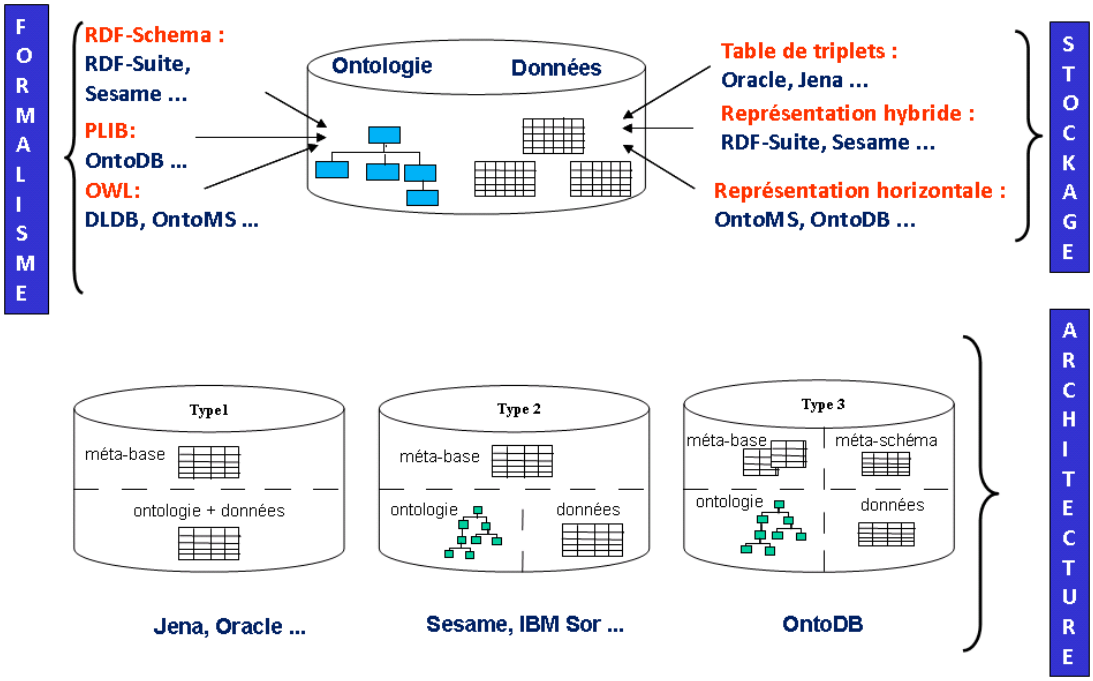

FIGURE 2.7 – Diversité dans les $\mathcal{BDBO}$

de satisfaire la totalité des besoins identifiés dans la section 2.

- Notre étude effectuée sur les $\mathcal{BDBO}$ Jena [Carroll et al., 2004, Wilkinson et al., 2003], Oracle [Chong et al., 2005], Sesame [Broekstra et al., 2002], SOR [Lu et al., 2007], DLDB [Pan and Heflin, 2003] et OntoDB [Dehainsala et al., 2007] montre que la majorité des bases de données à base ontologique utilisent un schéma d'ontologie figé. Jena et Oracle exploitent la représentation verticale tandis que Sesame, SOR et DLDB admettent un schéma d'ontologie prédéfini correspondant aux formalismes RDFS et OWL. L'API Jena ne peut guère être enrichie et ne permet l'ajout d'aucune nouvelle construction. De son côté, le système Oracle offre la possibilité de définir des bases de règles pour enrichir la sémantique des ontologies et des données.
- Excepté OntoDB, aucun système ne permet d'introduire dynamiquement un nouveau type de données et plus généralement un nouveau concept au niveau modèle. En effet, dans Jena [Carroll et al., 2004, Wilkinson et al., 2003], Sesame [Broekstra et al., 2002], SOR [Lu et al., 2007] et DLDB [Pan and Heflin, 2003], seules les données de type déjà prévu par le formalisme manipulé peuvent être représentées. Le système Oracle [Chong et al., 2005] permet en plus de supporter les types de données définies dans la base de données relationnelle. Grâce à sa partie méta-schéma, OntoDB offre la possibilité d'étendre le modèle d'ontologies et d'ajouter de nouveaux types de données.
- La représentation des données canoniques et non canoniques est assurée par un ensemble de $\mathcal{BDBO}$. Les systèmes Oracle et SOR adoptent une approche par saturation pour la représentation de telles données. Notons que cette représentation engendre un coût élevé

	$Type_1$	$Type_2$	$Type_3$
Oracle	√	√	
Jena	√		
OntoDB			√
OntoMS		√	
RDFSuite		√	
Sesame		√	
Sor		√	
3Store	√		
Sward	√		
DLDB		√	

TABLE 2.1 – Classification des $\mathcal{BDBO}$

lors du processus de mise à jour. En effet, lors de chaque modification de données, la $\mathcal{BDBO}$ Oracle effectue la mise à jour des vues associées aux différentes bases de règles. Tandis que SOR, pour maintenir la saturation de la base de données, rajoute les nouveaux faits inférés à chaque nouvelle insertion de données. Jena persiste à la fois les données canoniques et non canoniques dans la table de triplets. Dans sa première version, OntoDB n'offrait aucun support pour la représentation des données non canoniques. Dans sa deuxième version, les données non canoniques sont transformées en données canoniques et insérées dans la base de données. De plus, des vues calculant l'extension des données non canoniques sont définies. Notons que les données non canoniques ne peuvent pas toutes être représentées. Dans DLDB, les données canoniques sont persistées dans des tables selon l'approche hybride. Quant aux données non canoniques, elles sont représentées au travers des vues de classe.

– La plupart des $\mathcal{BDBO}$ sont dotées d'un mécanisme d'inférence permettant de raisonner sur les concepts et/ou instances ontologiques. Oracle, Sesame et SOR utilisent des raisonneurs internes tandis que Jena et DLDB utilisent des raisonneurs externes.
– Quelle que soit la structure utilisée, les $\mathcal{BDBO}$ étudiées admettent une approche directe qui n'offre aucune autonomie au concepteur dans la mesure où le modèle conceptuel de données (MCD) est strictement inclus dans l'ontologie utilisée.

Après avoir étudié et comparé ces $\mathcal{BDBO}$, nous remarquons que ces dernières diffèrent selon trois principaux critères comme cela est décrit dans la figure 2.7 : (1) l'architecture, (2) les modèles de stockage et (3) le formalisme. Dans le tableau 2.1, nous proposons une classification dirigée par les architectures pour un ensemble de $\mathcal{BDBO}$. Nous remarquons que la majorité de ces $\mathcal{BDBO}$ adoptent soit l'architecture deux parties ou trois parties ($Type_1$, $Type_2$). Seule la $\mathcal{BDBO}$ OntoDB utilise l'architecture quatre quarts et propose à ses utilisateurs une flexibilité au niveau du modèle d'ontologies. En ce qui concerne les modèles de stockage, trois principaux modèles sont utilisés. Le tableau 2.2 résume les représentations associées au stockage de

	Ontologie : Modèle de Stockage	**Données : Modèle de Stockage**
Oracle	représentation verticale	représentation verticale
Jena	représentation verticale	représentation verticale
OntoDB	structure ad hoc	représentation horizontale
RDFSuite	structure ad hoc	représentation hybride
Sesame	structure ad hoc	représentation hybride
Sor	structure ad hoc	représentation hybride
3Store	représentation verticale	représentation verticale
DLDB	structure ad hoc	représentation hybride

TABLE 2.2 – $\mathcal{BDBO}$ et modèles de stockage

l'ontologie et des données pour un panel de $\mathcal{BDBO}$.

Après avoir établi l'analyse d'un ensemble de $\mathcal{BDBO}$, nous proposons dans la section suivante l'étude de leur processus de conception.

6 Conception des bases de données à base ontologique

Dans les $\mathcal{BDBO}$ existantes, les ontologies ne subissent aucun traitement spécifique quant au processus de conception. L'ontologie et ses instances sont chargées directement dans la base de donnée cible. L'ontologie est stockée soit dans une table de triplets [Chong et al., 2005] [Carroll et al., 2004] [Harris and Gibbins, 2003] soit dans des tables dont la structure se rapproche de la structure du langage de définition d'ontologies de l'ontologie manipulée comme cela est décrit dans le tableau 2.2. Le schéma de ces tables est défini de façon ad hoc et diffère d'une implémentation à une autre [Lu et al., 2007] [Broekstra et al., 2002] [Pan and Heflin, 2003]. En effet, il est défini suivant les détails des informations que ces implémentations veulent capturer et du lien effectué avec la partie données. Malgré ces différences, ces schémas peuvent présenter des similarités. A titre d'exemple, pour persister les ontologies RDF-Schema suivant un schéma *ad hoc*, les tables *class*, *subclass*, *property*, *subproperty*, *domain* et *range* sont toujours implémentées tel que présenté dans la figure 2.8. De la même manière, les données à base ontologique sont chargées directement dans les tables figées qui leur sont associées. Ces tables sont créées suivant des règles figées par l'implémentation du système. Ces règles diffèrent d'une implémentation à une autre et peuvent être tel que suit: (a) chaque donnée s'écrit sous forme de triplet, (b) créer une table unaire par classe, (c) créer une table binaire par propriété, etc. Ainsi, les instances sont insérées dans une (des) table(s) figée(s) traduisant ces règles selon un modèle de stockage prédéfini. Cette insertion peut entraîner une certaine redondance dans la base de données. Par exemple, pour les $\mathcal{BDBO}$ Jena [Carroll et al., 2004] et Oracle [Chong et al.,

2005], toute instance est représentée sous forme de triplets et stockée dans la table verticale. Le même triplet peut être inséré en double sans le moindre contrôle. De plus, des inconsistances de données peuvent être détectées. Par exemple, considérons la classe des étudiants *Student* domaine des propriétés *id*, *age*, *birthdate*, *grade* et *name* décrivant respectivement l'identifiant, l'age, la date de naissance, le grade et le nom de l'étudiant. En utilisant la syntaxe DL, la classe non canonique *MasterStudent* décrivant les étudiants en master peut être définie comme une restriction OWL telle que suit: $MasterStudent \equiv Student \cap grade : Master$.

Class	
ID	Name
1	Université
2	Etudiant
3	Professeur
4	Master:Etudiant
5	…

Range	
Type	Property
xsd:string	1
xsd:integer	2
xsd:string	3
xsd:float	4
…	…

Domain	
Property	Class
1	1
2	1
3	2
4	2
5	…

Property	
ID	Name
1	nom
2	idUniv
3	adresse
4	téléphone
5	…

SubClass	
super	sub
4	2
…	…

SubProperty	
super	sub
…	…

FIGURE 2.8 – Exemple de schéma ad hoc d'une ontologie RDF Schema.

La figure 2.9 présente le stockage de la classe Student à travers une représentation verticale (table de triplets). La représentation des concepts non canoniques dans cette table présente des anomalies :

- si le grade de l'instance est modifié, le triplet (*Student*#1, type, MasterStudent) devient faux.
- si la date de naissance de l'instance est modifiée, le triplet (*Student*#1, age, 23) devient faux.

Ainsi, l'identification des concepts non canoniques est cruciale pour le développement d'applications de bases de données car elle réduit la redondance et l'incohérence des bases de données construites à partir d'ontologies. Notons que d'un point de vue de "monde ouvert", la redondance des données peut être exploitée pour raisonner sur les instances.

D'après notre étude, nous remarquons que le processus de conception des $\mathcal{BDBO}$ se résume en une seule étape principale: le chargement. Celui-ci concerne à la fois l'ontologie et les données à base ontologique. La figure 2.10 illustre un tel processus. En se référant aux bases de données traditionnelles et aux besoins identifiés dans la section 2, nous remarquons que le cycle de vie des $\mathcal{BDBO}$ est incomplet. L'ontologie manipulée subit un chargement direct selon des règles d'insertion figées par la base de données cible. Aucun processus de conception n'est défini. L'absence d'une telle phase dans le cycle de vie des $\mathcal{BDBO}$ fait apparaître des insuffisances dans le système. La redondance des données est tolérée et des formes d'inconsistances peuvent être identifiées. De plus, aucune méthodologie de déploiement ni d'optimisation de données n'est présentée. Seuls un chargement direct et une interrogation de données sont offerts. Dans cette thèse, nous proposons d'enrichir le cycle de vie des $\mathcal{BDBO}$ par l'ajout des

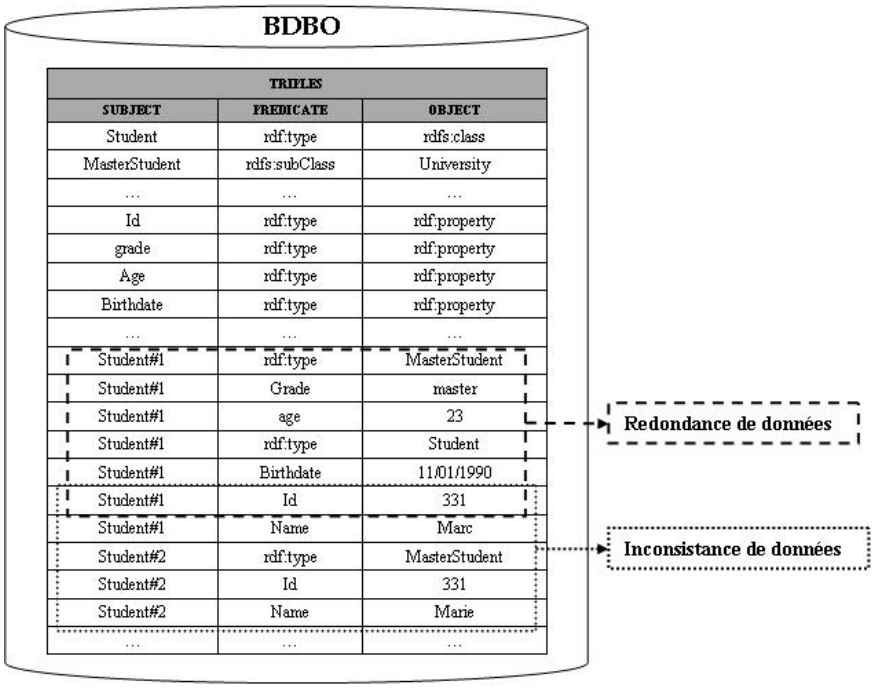

FIGURE 2.9 – Exemple d'anomalies de conception de $\mathcal{BDBO}$.

phases de conception et de déploiement.

7 Conclusion

Dans ce chapitre, nous avons fait un état de l'art sur les données à base ontologique et leurs référentiels de stockage en répondant à un ensemble de questions :

- comment stocker ces données et les concepts les référençant?
- comment les représenter?
- où les persister?
- comment les manipuler?
- comment les interroger?
- comment les déployer?

Pour cela, nous avons d'abord introduit les données à base ontologique et nous avons dégagé le besoin d'une structure pour leur gestion et leur manipulation. Ensuite, nous avons présenté les référentiels proposés pour leur persistance, nommés bases de données à base ontologique. Ce type de base de données a la particularité de stocker à la fois l'ontologie et ses instances dans le même référentiel. Elles offrent aux utilisateurs le potentiel de gérer de grandes masses de données tout en préservant la performance de la base. Dans notre étude, nous avons présenté l'architecture globale d'une $\mathcal{BDBO}$ ainsi que les principaux modèles de stockage qu'elle utilise pour le stockage des ontologies et des données à base ontologique. Afin de bien manipuler

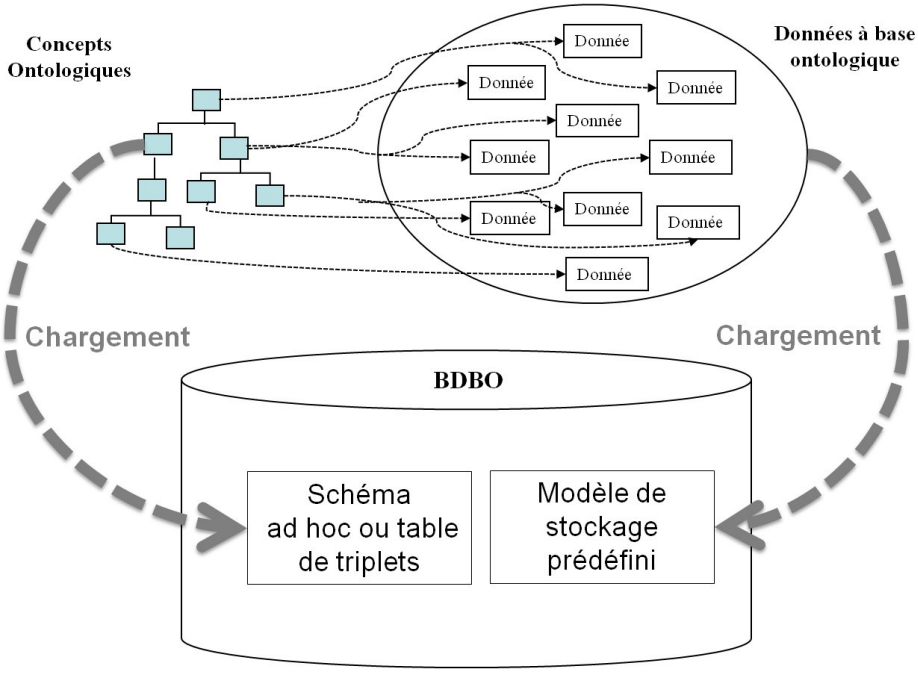

FIGURE 2.10 – Processus de conception des $\mathcal{BDBO}$.

ces données, nous avons décrit les principaux langages d'interrogation. Une analyse détaillée d'un panel de $\mathcal{BDBO}$ a été ensuite proposé. Celui-ci nous a permis de (i) classer les $\mathcal{BDBO}$ selon leur architecture et (2) d'identifier les limites de ces bases de données quant au processus de persistance des données. En effet, chaque $\mathcal{BDBO}$ admet une implémentation adoptant une représentation codée en dur pour (a) la représentation des concepts ontologiques et (b) pour la représentation des instances. Aucune méthodologie de conception de $\mathcal{BDBO}$ n'est proposée. Par conséquent, des anomalies de conception peuvent être détectées dans la base de données. La redondance est tolérée dans la majorité des $\mathcal{BDBO}$. Des données inconsistantes peuvent être définies sans le moindre contrôle. Vue l'importance de la phase de conception dans le cycle de vie des bases de données, nous proposons dans le cadre de cette thèse, une étude approfondie des ontologies et l'exploitation de leurs richesses afin de définir une approche de conception des $\mathcal{BDBO}$ et d'intégrer le concepteur dans cette dernière. Ainsi, à travers nos travaux, nous contribuons à l'enrichissement du cycle de vie des $\mathcal{BDBO}$ par l'ajout de la phase de conception dans un premier temps et la phase de déploiement dans un deuxième temps.

Deuxième partie

Contributions

Chapitre 3

Ontologies et relations de dépendance

Sommaire

Résumé. Dans le chapitre précédent, nous avons présenté une analyse détaillée d'une panoplie de *BDBO* qui nous a mené à identifier les limites de ces systèmes dûes à leur processus de conception. Dans ce chapitre, nous nous intéressons à l'amélioration d'un tel processus et au rôle du concepteur durant la phase de conception. Dans un premier temps, nous proposons d'étudier (a) les caractéristiques ontologiques pouvant être exploitées dans le processus de conception des *BDBO* ainsi que (b) le rôle du concepteur dans le déroulement d'un tel processus.

Dans un deuxième temps, nous introduisons notre proposition décrivant une approche de conception des bases de données à base ontologique intégrant le concepteur et exploitant les relations ontologiques identifiées.

1 Introduction

Plusieurs modèles d'ontologies ont été développés dans différents domaines : OWL dans le domaine du Web sémantique, PLIB dans le domaine de l'ingénierie, etc. Ces modèles offrent aux concepteurs des constructeurs, des fonctions et des procédures permettant de définir des concepts d'un domaine en termes de classes, de propriétés et de relations les reliant. En examinant les concepts ontologiques quant à leur définition, nous avons identifié un ensemble de caractéristiques. Parmi ces caractéristiques, nous nous intéressons aux caractéristiques ontologiques décrivant des relations de dépendance entre les concepts ontologiques. Deux types de dépendances sont identifiés : (1) les relations de dépendance entre les propriétés et (2) les relations de dépendance entre les classes. Ces relations de dépendance n'ont pas été exploitées dans l'enrichissement des modèles de bases de données à base ontologique. En effet, à l'image des bases de données traditionnelles, les relations de dépendance entre les concepts ontologiques peuvent jouer un rôle crucial dans le processus de conception des bases de données dédiées à la persistance des ontologies. Elles ont un impact important sur la phase de la modélisation logique et le processus de normalisation. De plus, elles permettent au concepteur de définir le schéma logique de données relatif aux concepts ontologiques et par la suite, de reprendre le contrôle du processus de conception des bases de données à base ontologique.

Dans ce chapitre, nous proposons, dans un premier temps, d'étudier les caractéristiques ontologiques inexploitées en identifiant les relations de dépendance entre les concepts ontologiques (propriétés et classes) dans les modèles d'ontologies. Ensuite, une étude concernant le rôle du concepteur dans le contrôle du processus de conception des bases de données est présentée. Dans un deuxième temps, nous proposons un processus de conception des bases de données à base ontologique inspiré de l'architecture ANSI/SPARC et exploitant les relations ontologiques inexploitées.

2 Caractéristiques d'ontologie inexploitées

Un modèle d'ontologie utilise une approche de modélisation orientée objet qui, outre les mécanismes usuels (e.g., classes, propriétés et héritage) définit un ensemble de mécanismes spécifiques (e.g., propriétés dépendantes les unes des autres en PLIB, classes définies en OWL). Compte tenu du fait que les concepts ontologiques peuvent être définis les uns des autres (concepts dérivés), des relations de dépendance entre ces derniers peuvent être déduites. Deux catégories de relations ontologiques sont identifiées : (i) les relations de dépendance entre les propriétés et (ii) les relations de dépendance entre les classes.

2.1 Relations de dépendance entre les propriétés

Dans une ontologie, les propriétés donnent la capacité d'exprimer des caractéristiques au sujet de ses classes et de leurs instances. Deux types de propriétés sont identifiés : (1) les propriétés de données et (2) les propriétés d'objet (composition, agrégation, association, etc.). Dans la première catégorie, les propriétés relient des individus à des valeurs de données, tandis que dans la deuxième catégorie, les propriétés relient des instances à d'autres instances. La définition de ces propriétés peut être décrite à partir d'autres propriétés. Par conséquent, des relations de dépendance entre propriétés peuvent être déduites : (a) des relations de dépendance entre les propriétés de données et (b) des relations de dépendance entre les propriétés d'objet.

2.1.1 Relations de dépendance entre les propriétés de données

Une propriété de données est une relation binaire entre une classe ontologique et un domaine de valeurs. Par exemple, le numéro de sécurité sociale d'une personne (NSS), son nom, son prénom, sa date de naissance et son âge sont des propriétés de données d'une classe *Personne* ayant pour domaine de valeurs un entier, une chaîne de caractères ou une date. La figure 3.1 décrit cet exemple. Dans une ontologie, les propriétés de données peuvent être définies d'une

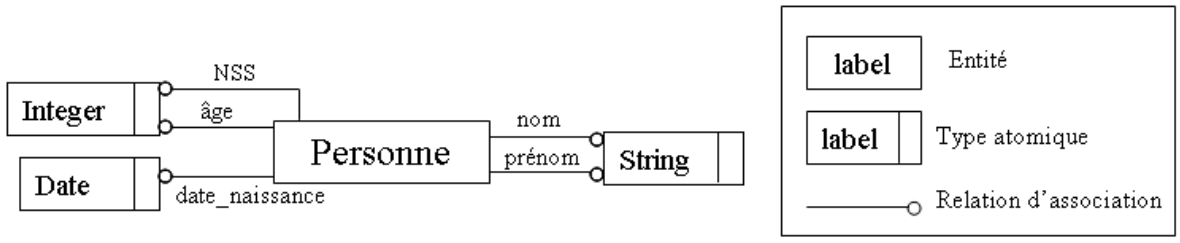

FIGURE 3.1 – Exemple de propriétés de données

manière atomique (primitive) ou dérivée. Dans le deuxième cas, ces propriétés sont calculables à partir des propriétés primitives et/ou définies. Ainsi, leur définition permet l'identification des relations de dépendance entre les propriétés de données.

1. **Exemples**

   (a) Considérons les propriétés de données *date de naissance* et *âge* ayant pour domaine la classe *Personne*. En assumant que la propriété *date de naissance* est primitive, la propriété *âge* est considérée comme une propriété dérivée étant donné que l'âge d'une personne est calculable à partir de sa date de naissance (*âge* = année actuelle - *année de naissance*). Ainsi, la connaissance de la valeur de la propriété *date de naissance* permet la connaissance de la valeur de la propriété *âge*.

   (b) Le nom complet d'une personne est calculé à partir de son prénom et de son nom

(*NomComplet = prénom + nom*). Par conséquent, une relation de dépendance entre les propriétés *nom*, *prénom* et *NomComplet* peut être relevée.

(c) Une adresse est calculée à partir de la rue, du code postal, de la ville et du pays (*adresse = rue+code postal+ville+pays*). Ainsi, la propriété *adresse* est une propriété dérivée à partir des propriétés ontologiques *rue*, *code postal*, *ville* et *pays*. Par conséquent, nous pouvons admettre que la connaissance des valeurs des propriétés *rue*, *code postal*, *ville* et *pays* permet la déduction de la valeur de la propriété *adresse*.

(d) La moyenne d'une matière est calculée à partir des notes obtenues dans cette matière et leur coefficients. Dans un contexte ontologique, la propriété de données $moyenne_{Matière}$ est considérée comme une propriété de données définie sur les propriétés de donnée atomiques $note_1$,$note_2$,.., $note_n$ et leur *coefficient* ($moyenne_{Matière} = \sum (note_i \times coefficient_i) \div \sum coefficient_i$). Ainsi, les propriétés $note_i$ et $coefficient_i$ déterminent la propriété $moyenne_{Matière}$.

(e) La moyenne générale d'un étudiant est calculée à partir de ses moyennes par matière et leur coefficients. Ainsi, la propriété $moyenne_{Générale}$ est calculable à partir des propriétés $moyenne_{Matière}$ et $coefficient_{Matière}$ définissant respectivement la moyenne pour chaque matière et le coefficient de cette dernière ($moyenne_{Générale} = \sum (moyenne_{Matière} \times coefficient_{Matière}) \div \sum coefficient_{Matière}$). Par conséquent, une relation de dépendance est établie entre les propriétés $moyenne_{Matière}$, $coefficient_{Matière}$ et $moyenne_{Générale}$.

2. **Modèles d'ontologies et relations de dépendance entre les propriétés de données**
Pour illustrer le pouvoir d'expression d'EXPRESS quant à la définition des propriétés de données dérivées, considérons la figure 3.2. Dans cet exemple, nous définissons deux entités *Personne* et *Maths* (correspondant aux deux classes ontologiques Personne et Maths). L'entité *Personne* définit un attribut dérivé *nomComplet* associé à la fonction EXPRESS (nom_prénom) et retournant la concaténation des attributs *nom* et *prénom* d'une instance de l'entité *Personne*. La moyenne en mathématiques est définie comme un attribut dérivé calculé à partir des notes du TP ($note_{TP}$), du TD ($note_{TD}$) et de l'examen ($note_{EXAMEN}$). Contrairement à PLIB, dans OWL, les propriétés de données ne peuvent pas être dérivées. OWL n'offre ni des constructeurs d'expressions, ni des fonctions permettant leur calcul à partir des propriétés déjà définies. Par conséquent, en OWL les propriétés de données sont toujours considérées indépendantes les unes des autres.

2.1.2 Relations de dépendance entre les propriétés d'objets

Une propriété d'objets est une relation binaire entre deux classes ontologiques. Par exemple, la propriété d'objets *Enseigne* dans l'entité *Professeur* décrit une relation binaire entre les classes *Professeur* et *Cours*.

```
ENTITY Personne;
NSS: NUMBER;
nom: STRING;
prénom: STRING;
DERIVE
NomComplet: STRING:= nom_prénom(Self);
End_ENTITY;
FUNCTION nom_prénom(pers: Personne): STRING;
Return (pers.nom+ ' '+ pers.prénom);
End_FUNCTION;
```

```
ENTITY Maths;
note_TD:REAL;
note_TP: REAL;
note_EXAMEN: REAL;
DERIVE
moyenne_MATHS: REAL:= (note_TD+ note_TP +
                       (2* note_EXAMEN))/4;
End_ENTITY;
```

FIGURE 3.2 – Dérivation des propriétés de données en EXPRESS

Dans une ontologie, les propriétés d'objets peuvent être primitives ou dérivées. Dans le deuxième cas, les propriétés d'objet sont calculables à partir des propriétés d'objets primitives et/ou définies tel que décrit dans la section suivante.

1. **Exemples**

   (a) Soient les propriétés d'objets *PèreDe* et $GrandPère_{Paternel}$ ayant pour domaine et co-domaine la classe *Personne*. La relation $GrandPère_{Paternel}$ peut être décrite comme une composition de la propriété *PèreDe* ($GrandPère_{Paternel}(p_i) = PèreDe \circ PèreDe\ (p_i) \mid p_i \in Personne$). Ainsi, la propriété ontologique $GrandPère_{Paternel}$ est une propriété d'objet calculable à partir de la propriété d'objet primitive *PèreDe*. Sa définition permet d'identifier une relation de dépendance entre ces propriétés.

   (b) Soient les propriétés d'objets *FilsDe*, *FrèreDe*, *SoeurDe* et *NeuveuDe* ayant pour domaine et co-domaine la classe *Personne*. La relation *NeuveuDe* est définie comme le fils de la soeur ou le fils du frère : $NeuveuDe(p_i) = FilsDe(FrèreDe(p_i)) \cup FilsDe(SoeurDe(p_i)) \mid p_i \in Personne$.

   Par conséquent, une relation de dépendance entre les propriétés primitives *FilsDe*, *FrèreDe* et *SoeurDe* et la propriété dérivée *NeuveuDe* est établie.

   (c) Soient les propriétés d'objets *TanteDe*, *CousineDe*, *OncleDe* et *FilleDe* ayant pour domaine et co-domaine la classe *Personne*. La relation *CousineDe* est définie comme la fille de la tante ou la fille de l'oncle ($CousineDe(p_i)= FilleDe(TanteDe(p_i)) \cup FilleDe(OncleDe(p_i)) \mid p_i \in Personne$). Ainsi, elle est calculable à partir des propriétés primitives *TanteDe*, *OncleDe* et *FilleDe* ce qui permet d'identifier une relation de dépendance entre ces propriétés .

   (d) Soient les propriétés d'objets $GrandPère_{Paternel}$, *GrandPèreDe* et $GrandPère_{Maternel}$ ayant pour domaine et co-domaine la classe *Personne*. La relation *GrandPèreDe* est définie comme le père de la mère ou le père du père : $GrandPèreDe(p_i) = \text{GrandPère}_{Paternel}(p_i) \cup \text{GrandPère}_{Maternel}(p_i) \mid p_i \in Personne$.

La définition de cette propriété permet de définir une relation de dépendance entre les propriétés *GrandPèreDe*, $GrandPère_{Paternel}$ et $GrandPère_{Maternel}$.

2. **Modèles d'ontologies et relations de dépendance entre les propriétés d'objets**

   PLIB offre au concepteur des fonctions pour la définition des propriétés d'objet dérivées. Pour illustrer cette fonctionnalité, considérons la figure 3.3. Dans cet exemple, nous définissons un schéma de nom famille. Ce schéma est constitué de trois entités. Les entités *Femme* et *Homme* héritent de l'entité *Personne*. L'entité *Personne* définit des attributs dérivés décrivant le grand-père paternel ($grandPere_{Paternel}$), le grand-père maternel ($grandPere_{Maternel}$), la grand-mère paternelle ($grandMere_{Paternelle}$) et la grand-mère maternelle ($grandMere_{Maternelle}$). Ces attributs sont associés respectivement aux fonctions *GrandPereP*, *GrandPereM*, *GrandMereM* et *GrandMereP* retournant les individus ayant le lien de parenté décrit. Notons que les contraintes ne sont pas représentées dans la figure 3.3. A l'opposé de PLIB, OWL ne propose aucun constructeur quant à la définition des propriétés d'objets dérivées. Celles-ci peuvent être exprimées, mais ne peuvent pas être calculées. Par conséquent, les propriétés d'objets dans OWL sont toujours définies indépendamment les unes des autres.

```
SCHEMA Famille;

ENTITY Personne;
nom : STRING;
pere : Homme;
mere: Femme;
DERIVE
grandMere_Maternelle : Femme := GrandMereM(SELF);
grandMere_Paternelle : Femme := GrandMereP(SELF);
grandPere_Paternel : Homme := GrandPereP(SELF);
grandPere_Maternel : Homme := GrandPereM(SELF);
ENd_ENTITY;

ENTITY Femme SUBTYPE OF (Personne);
END_ENTITY;

ENTITY Homme SUBTYPE OF (Personne);
END_ENTITY;

FUNCTION GrandPereP (p : Personne) : Homme;
RETURN (p.pere.pere);
END_FUNCTION;

FUNCTION GrandPereM (p : Personne) : Homme;
RETURN (p.pere.mere);
END_FUNCTION;

FUNCTION GrandMereM (p : Personne) : Femme;
RETURN (p.mere.mere);
END_FUNCTION;

FUNCTION GrandMereP (p : Personne) : Femme;
RETURN (p.mere.pere);
END_FUNCTION;


END_SCHEMA;
```

FIGURE 3.3 – Exemple de propriétés d'objects dérivées en EXPRESS

2.1.3 Synthèse

Dans une ontologie, il existe des propriétés de données et d'objets calculables à partir d'autres propriétés ontologiques (primitives et/ou définies). Ces propriétés, dites dérivées, permettent d'identifier des relations de dépendance entre les différentes propriétés. Dans le but de faciliter leur présentation et leur manipulation, nous proposons de les modéliser par un graphe

où les noeuds représentent les propriétés ontologiques et les arcs représentent les relations de dépendance entre ces propriétés. La figure 3.4 décrit les relations de dépendance déductibles entre les propriétés primitives (P_{pr}) et les propriétés définies (P_{def}) à travers les deux langages d'ontologies OWL et PLIB. Contrairement à OWL, PLIB permet de définir des propriétés dérivées à partir de propriétés primitives et définies. Et par la suite, il permet de décrire des relations de dépendance déductibles entre P_{pr} et P_{def} ou P_{def} et P_{def}. Mais, aucun de ces deux langages ne permet de déduire une relation de dépendance entre deux propriétés primitives (P_{pr} et P_{pr}). Ces caractéristiques ontologiques sont inexploitées dans la conception des modèles de bases de données à base ontologique. En effet, les dépendances entre propriétés ontologiques peuvent être exploitées, comme dans les bases de données traditionnelles, dans le processus de normalisation et la définition du schéma logique correspondant aux classes ontologiques.

Nœud 1	Nœud 2	Arc (Noeud1, Noeud2)	PLIB	OWL
P_{pr}	P_{pr}	$A_i(P_{pr}, P_{pr})$	NON	NON
P_{pr}	P_{def}	$A_j(P_{pr}, P_{def})$	OUI	NON
P_{def}	P_{def}	$A_k(P_{def}, P_{def})$	OUI	NON

FIGURE 3.4 – Expression des relations de dépendance entre propriétés dans les modèles d'ontologies PLIB et OWL

2.2 Relations de dépendance entre classes

Dans une ontologie, deux types de classes sont identifiés : les classes canoniques (primitives) et les classes non canoniques (définies). Dans le deuxième cas, les classes sont définies à partir d'équivalences conceptuelles. Celles-ci sont traduites soient (a) à partir d'expressions de classes basées sur des opérateurs élémentaires sur les ensembles ou (b) par des restrictions définies sur les propriétés d'objets et de données. Ainsi, deux types de relations entre classes sont identifiées : (1) les relations déduites à partir des opérateurs ensemblistes sur les classes et (2) les relations déduites à partir des restrictions sur les propriétés.

2.2.1 Relations déduites à partir des opérateurs ensemblistes sur les classes

Les classes ontologiques peuvent être définies à partir d'expressions de classes définies par le biais d'opérateurs élémentaires sur les ensembles comme les opérateurs d'union ($\cup$), d'intersection ($\cap$), etc. Appliqués aux classes ontologiques canoniques et/ou non canoniques,

ces opérateurs permettent la définition de nouveaux concepts ontologiques dérivés. Dans la section suivante, nous décrivons un ensemble d'exemples traduisant ce type de relation.

1. **Exemples**
   (a) Soient les classes ontologiques *Personne*, *Femme* et *Homme*. La classe *Personne* peut être définie comme l'union des classes *Femme* et *Homme*. Cela signifie que $\forall$ personne$_i$, personne$_i \in$ (*Personne* $\equiv$ *Femme* $\cup$ *Homme*) $\wedge$ (*Femme* $\cap$ *Homme* = Φ) $\Leftrightarrow$ ((personne$_i \in$ *Femme*) $\vee$ (personne$_i \in$ *Homme*)). Ainsi, une relation de dépendance est relevée entre les classes *Personne*, *Femme* et *Homme*.
   (b) Considérons les classes ontologiques *Etudiant*, *EtudiantTravailleur* et *Salarié*. La classe *EtudiantTravailleur* peut être définie comme étant l'intersection des classes *Etudiant* et *Salarié*. Cela signifie que $\forall$ étudiantTravailleur$_j$, étudiantTravailleur$_j \in$ (*EtudiantTravailleur* $\equiv$ *Etudiant* $\cap$ *Salarié*) $\Leftrightarrow$ ((étudiantTravailleur$_j \in$ *Etudiant*) $\wedge$ (étudiantTravailleur$_j \in$ *Salarié*)). Une relation de dépendance est établie entre les classes *Etudiant*, *EtudiantTravailleur* et *Salarié*.
2. **Modèles d'ontologies et opérateurs ensemblistes**
   Le langage OWL offre des constructeurs permettant de définir des expressions de classes à l'aide des opérateurs ensemblistes. Ces expressions sont introduites à travers des équivalences conceptuelles définissant des classes comme étant équivalentes à des expressions décrites à partir d'autres classes. L'exemple (a) (*Personne* $\equiv$ *Femme* $\cup$ *Homme*) traité ci-dessus est traduit en OWL à l'aide du constructeur *owl:unionOf* tel que illustré dans la figure 3.5.(a). Le langage OWL offre aussi un constructeur (*owl:intersectionOf*) permettant de définir des équivalences conceptuelles définies par le biais de l'opérateur ensembliste intersection. L'exemple (b) (*EtudiantTravailleur* $\equiv$ *Etudiant* $\cap$ *Salarié*) traité ci-dessus est illustré dans la figure 3.5.(b) à l'aide du constructeur proposé.

```
<owl:Class rdf:about="\#Personne">
 <owl:equivalentClass>
   <owl:Class>
     <owl:unionOf rdf:parseType="Collection">
         <owl:Class rdf:ID="Femme"/>
           <owl:Class rdf:about="\#Homme"/>
     </owl:unionOf>
   </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

(a)

```
<owl:Class rdf:ID="EtudiantTravailleur">
   <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
         <owl:Class rdf:about="\#Etudiant"/>
         <owl:Class rdf:about="\#Salarié"/>
      </owl:intersectionOf>
    </owl:Class>
   </owl:equivalentClass>
</owl:Class>
```

(b)

FIGURE 3.5 – Exemple d'opérateurs ensemblistes dans OWL

Dans PLIB, le langage de modélisation permet aussi l'expression des classes définies à travers des opérateurs ensemblistes.

2.2.2 Relations déduites à partir des restrictions

Dans une ontologie, une classe peut être décrite par une restriction de propriétés de données ou d'objets. Celle-ci représente un type particulier de description de classe contraignant ses individus à la satisfaction d'un ensemble de contraintes liées soit à la valeur de la propriété soit à sa cardinalité. Dans le cas d'une contrainte de valeur, une limitation est imposée sur l'image de la propriété lors de son application à cette description de classe particulière (existence de valeurs de propriétés particulières). Tandis que dans le deuxième cas, la limitation est plutôt exercée sur le nombre des valeurs prises par une propriété dans le contexte de cette description de classe. Dans la section suivante, nous décrivons un ensemble d'exemples traduisant les relations décrites à partir de ces restrictions.

1. **Exemples**
   (a) Considérons les classes ontologiques *Homme* et *Personne* et la propriété de données *Sexe* ayant pour domaine la classe *Personne*. La classe *Homme* peut être définie comme étant les individus de la classe *Personne* dont la valeur de la propriété *Sexe* est masculin. Cette définition permet donc d'identifier une relation de dépendance entre les classes *Personne* et *Homme*.
   (b) Une épouse est une femme qui a un mari. Ainsi, en considérant la propriété d'objets *aPourMari* ayant respectivement pour domaine et co-domaine les classes ontologiques *Épouse* et *Époux*, une définition de la classe *Épouse* serait d'attribuer une valeur = 1 à la cardinalité de la propriété *aPourMari*. Cette description de classe permet de relever une dépendance structurelle entre les deux classes *Épouse* et *Époux*, c'est-à-dire, la connaissance d'une épouse permet la connaissance d'un seul époux.

```
<owl:Class rdf:ID="Homme">
 <owl:equivalentClass>
  <owl:Restriction>
   <owl:onProperty>
     <owl:DatatypeProperty rdf:ID="sexe"/>
   </owl:onProperty>
   <owl:hasValue
     rdf:datatype="http://www.w3.org/2001/
     XMLSchema#string«>masculin</owl:hasValue>
  </owl:Restriction>
 </owl:equivalentClass>
</owl:Class>
```

(a)

```
<owl:Class rdf:ID="Epouse">
 <owl:equivalentClass>
  <owl:Restriction>
   <owl:onProperty>
    <owl:ObjectProperty rdf:ID=« aPourMari"/>
   </owl:onProperty>
   <owl:cardinality
     rdf:datatype="http://www.w3.org/2001/
     XMLSchema#int">1</owl:cardinality>
  </owl:Restriction>
 </owl:equivalentClass>
</owl:Class>
```

(b)

FIGURE 3.6 – Exemple de restrictions en OWL

Notons que dans le cas ou la cardinalité est égale à n, la dépendance est dite calculatoire.

2. **Modèles d'ontologies et restrictions de propriétés**

OWL offre des constructeurs permettant la définition des restrictions de propriétés restreignant le co-domaine de certaines propriétés uniquement pour la classe où elles sont spécifiées. Prenons l'exemple (a) traité ci-dessus. Avec OWL, la valeur masculin de la propriété *sexe* est spécifiée à l'aide du constructeur *owl:hasValue* tel que présenté dans la figure 3.6.(a). De plus, OWL propose des constructeurs permettant la description de restrictions de cardinalité restreignant le nombre de valeurs distinctes d'une propriété P. Parmi ces constructeurs, on peut citer *owl:minCardinality*, *owl:Cardinality* et *owl:maxCardinality*. Une illustration traduisant l'exemple (b) (en particulier *Epouse*) est présentée dans la figure 3.6.(b). Dans PLIB, le langage de modélisation permet l'expression des classes définies à travers des restrictions sur les valeurs des propriétés et leur cardinalité est devenue désormais possible. Pour illustrer un exemple en PLIB, considérons la figures 3.7. Dans cet exemple, nous reprenons la définition de la classe *Homme* décrite ci-dessus où la contrainte de valeur *"masculin"* définie sur la valeur de la propriété *sexe* est introduite à l'aide du constructeur *enumeration_constraint_Type*.

```
<?xml version="1.0" encoding="UTF-8"?>
<ontoml:constraint xsi:type="ontoml:INTEGRITY_CONSTRAINT_Type"
xmlns:ontoml="urn:iso:std:iso:13584:-32:ed-1:tech:xml-schema:ontoml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:iso:std:iso:13584:-32:ed-1:tech:xml-schema:ontoml
D:\plib\plibDoc\P32\IS\xml_schema\ontoml.xsd">

        <constrained_property property_ref="1234-678#02-SEXE#1"/>
        <redefined_domain xsi:type="ontoml:ENUMERATION_CONSTRAINT_Type">
                <subset>
                    <val:string_value xmlns:val="urn:iso:std:iso:ts:29002:-10:ed
                      1:tech:xml-schema:value">masculin</val:string_value>
                </subset>
        </redefined_domain>
</ontoml:constraint>
```

FIGURE 3.7 – Exemple d'expression de la contrainte de valeur en PLIB

2.2.3 Synthèse

Dans une ontologie, il existe des classes calculables à partir d'autres classes primitives et/ou définies. Ces classes, dites non canoniques, permettent d'identifier des relations de dépendance entre les différentes classes (canoniques et non canoniques). Afin de faciliter la compréhension et l'exploitation de ces relations, nous proposons de les modéliser à travers un graphe de dépendance entre classes où les noeuds représentent les classes ontologiques et les arcs représentent les relations de dépendance entre ces classes. La figure 3.8 décrit les relations de dépendance

déductibles entre les classes ontologiques canoniques (CC) et/ou non canoniques (CNC) à travers les deux modèles d'ontologies OWL et PLIB. Ces derniers permettent de définir des classes dérivées à partir des CC et CNC et par la suite de déduire des relations de dépendance entre les CNC entre elles ou les CC et les CNC. Ce type de dépendances n'a pas été exploité dans le processus de conception des modèles de bases de données dédiées à la persistance des ontologies et leurs instances. L'identification de telles relations peut jouer un rôle important dans le processus de conception tel que présenté dans le Chapitre 5.

3 Rôle du concepteur dans le processus de conception des BD

Dans les bases de données traditionnelles, le processus de conception passe, selon l'architecture ANSI/SPARC, par trois principales phases: (1) la modélisation conceptuelle, (2) la modélisation logique et (3) la modélisation physique. La première phase constitue une étape clé du processus de conception des bases de données. Elle donne la possibilité au concepteur de collecter les besoins de l'utilisateur et de définir un modèle conceptuel traduisant les besoins applicatifs collectés répondant à un cahier des charges donné. Une fois ce modèle défini, le concepteur définit les dépendances fonctionnelles et le traduit en un modèle logique décrivant le schéma logique des données. En se basant sur ce schéma, l'administrateur définit la manière dont la BD sera implémentée. Avec l'émergence des ontologies et des modèles de bases de données à base ontologique, le concepteur a perdu le contrôle du processus de conception. En effet, les bases de données dédiées au stockage des ontologies et leurs instances sont générées en dur à partir de l'ontologie adaptée. Leur schéma logique est défini au préalable et ne peut être modifié (figé). De ce fait, il serait intéressant, à l'image des bases de données traditionnelles où le concepteur joue un rôle important dans le processus de conception, de rendre à ce dernier sa propre valeur et l'intégrer davantage dans le processus de conception des bases de données à base ontologique.

Nœud 1	Nœud 2	Arc (Noeud1, Noeud2)	PLIB	OWL
CC	CC	impossible	NON	NON
CC	CNC	A_i(CC, CNC)	OUI	OUI
CNC	CNC	A_j(CNC, CNC)	OUI	OUI

FIGURE 3.8 – Déduction des relations de dépendance entre classes

4 Notre proposition

Dans cette section, nous relevons, dans un premier temps, un ensemble de constats sur les ontologies et leur persistance dans les bases de données à base ontologique. Ces constats nous ont mené à proposer, dans un deuxième temps, un processus de conception de bases de données à base ontologique.

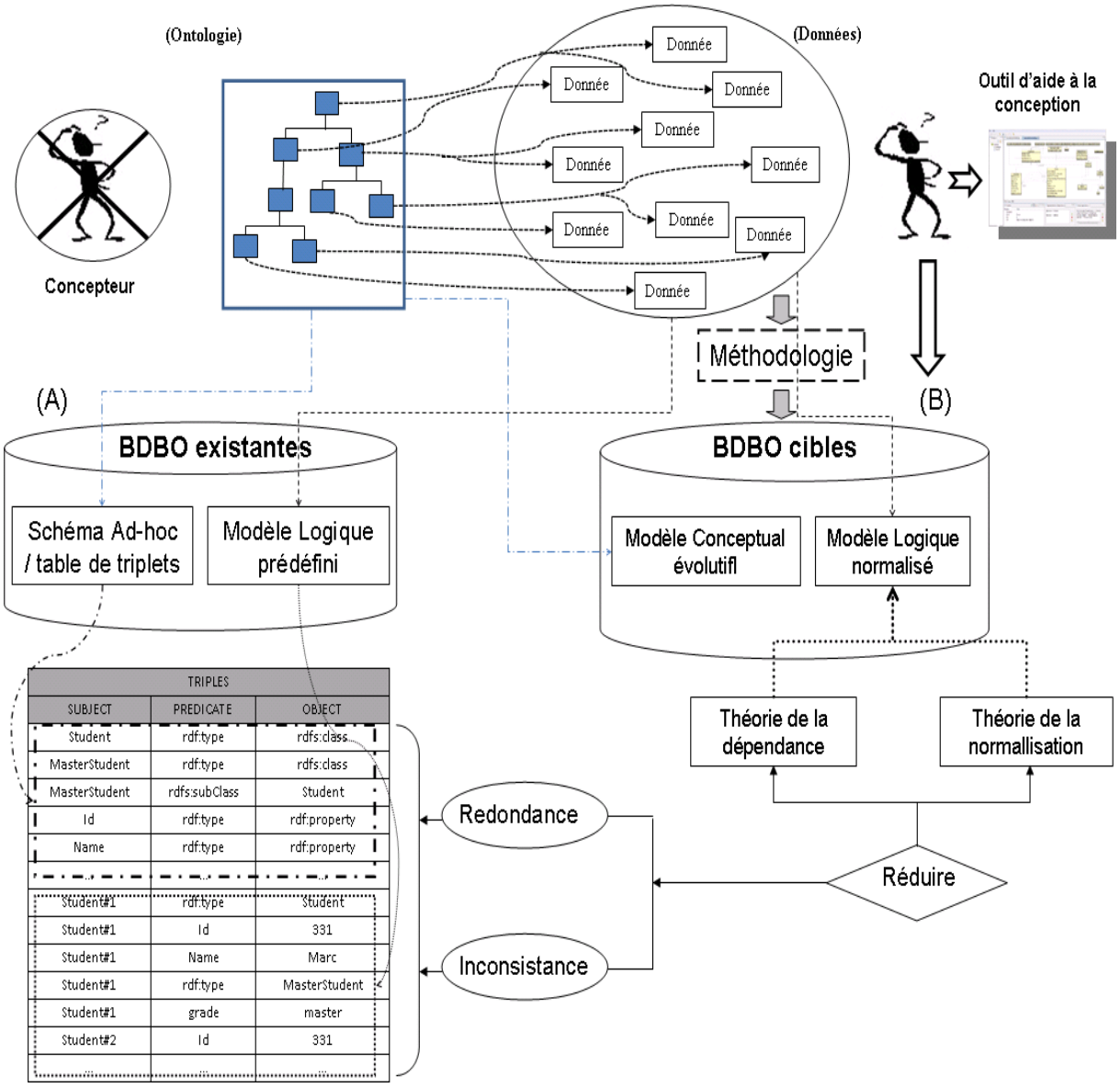

FIGURE 3.9 – Processus de conception des bases de données à base ontologique

4.1 Constats

Après une étude approfondie des ontologies du point de vue conceptuel et expressif d'une part et l'analyse de leur gestion dans les bases de données à base ontologique d'autre part, un ensemble de constats a été relevé.

- Dans la définition d'une ontologie, les propriétés peuvent être dérivées (fonctions, composition, etc.) ce qui permet d'identifier des relations de dépendance entre les propriétés ontologiques (atomiques et/ou dérivées). Ces relations peuvent être exploitées dans le processus de conception des bases de données à base ontologique.
- De la même manière que pour les propriétés, les classes ontologiques peuvent être définies à partir d'expressions de classes à l'aide d'opérateurs ensemblistes (union, intersection, etc) et/ou de restriction définie sur les propriétés d'objets et de données. Ainsi, des relations entre les classes sont identifiées. Ces relations peuvent être traduites par un ensemble de dépendances définies entre les classes ontologiques et exploitées dans le processus de conception des bases de données à base ontologique.
- Les ontologies sont stockées dans des bases de données à base ontologique où les modèles de stockage des données sont figés. Aucun processus de conception ni de déploiement des bases de données à base ontologique n'a été proposé. Par conséquent, des anomalies de redondance et d'incohérence de données peuvent apparaître dans les $\mathcal{BDBO}$.
- Le concepteur est totalement absent dans le processus de conception des bases de données à base ontologique. D'où le besoin de proposer une méthodologie de conception de $\mathcal{BDBO}$ assurant l'intégration du concepteur dans les différentes phases du processus de conception. Un tel processus est proposé dans la section suivante.

4.2 Démarche globale de conception

En se basant sur les constats relevés, nous proposons dans cette section d'introduire une nouvelle démarche dédiée à la conception des bases de données à base ontologique. Contrairement au processus de conception de $\mathcal{BDBO}$ actuel où l'ontologie et ses instances sont directement chargées dans la $\mathcal{BDBO}$ (voir figure 3.9.(A)), notre approche :

- est inspirée des méthodologies de conception des bases de données classiques. Elle traite les différentes phases traditionnelles du processus de conception : la modélisation conceptuelle, logique et physique ;
- ré-intègre le concepteur dans le processus de conception ;
- exploite les relations de dépendance entre les concepts ontologiques à différents niveaux de conception ;
- traite la redondance des données sur différents niveaux : concepts ontologiques et instances ;
- réduit l'incohérence des données représentées ;
- est fondée sur deux grandes théories : la théorie de la dépendance et la théorie de la

normalisation ;
– est accompagnée par la proposition d'un outil d'aide à la conception aidant l'utilisateur au cours des différentes phases de conception.

La figure 3.9.(B) décrit le schéma global de notre approche.

5 Conclusion

Les ontologies conceptuelles sont riches en terme d'expressivité. Elles permettent de définir des concepts dérivés à partir d'expressions d'équivalences, d'opérateurs ensemblistes, de fonctions, etc. Ces définitions nous ont permis d'identifier des relations de dépendance entre les différents concepts ontologiques. Étant donné qu'aucun processus de conception de bases de données à base ontologique n'a été proposé, nous introduisons une nouvelle démarche globale de conception de $\mathcal{BDBO}$ préservant le rôle du concepteur dans le processus de conception et exploitant les relations de dépendance ontologiques identifiées. Le descriptif de notre approche est détaillé dans les chapitres suivants. Dans le chapitre 4, nous étudions l'exploitation des dépendances entre les propriétés pour l'intégration du processus de normalisation dans la phase de conception. Dans le chapitre 5, les relations de dépendance sont d'abord étudiées au niveau des classes ontologiques puis exploitées pour l'identification de ces classes selon leur type (canonique ou non canonique) et l'association du traitement spécifique à chacun de ces deux types de classes. Le chapitre 6 déroule, dans un premier temps, les différentes étapes de notre approche en mettant l'accent sur la phase de la modélisation conceptuelle et le placement des classes dans la hiérarchie de subsomption. Dans un deuxième temps, une approche de déploiement de notre méthodologie de conception sur les différents types de $\mathcal{BDBO}$ est présenté. Enfin, un outil d'aide à la conception est présenté dans le chapitre 7.

Chapitre 4

Dépendances fonctionnelles entre propriétés ontologiques et conception des *BDBO*.

Sommaire

Résumé. Dans le chapitre précédent, nous avons montré l'existence des relations de dépendance entre les propriétés ontologiques. Toutefois, ces relations n'ont pas été exploitées dans le processus de conception des bases de données à base ontologique. Dans ce chapitre, nous proposons d'intégrer les dépendances fonctionnelles entre les propriétés des données dans le modèle d'ontologies, et de les exploiter dans le processus de conception des $\mathcal{BDBO}$. Dans un premier temps, nous proposons la modélisation de ces relations et leur intégration aux ontologies. Dans un deuxième temps, nous présentons une approche de conception de $\mathcal{BDBO}$ intégrant ces relations de dépendance dans la phase de la modélisation logique. Pour valider notre approche, une mise en œuvre dans un environnement de bases de données à base de modèles est présentée.

1 Introduction

Dans les bases de données, plusieurs problèmes peuvent provenir suite à leur mauvaise conception. Le premier problème concerne la redondance des données. En effet, certains choix de conception entraînent une répétition des données lors de leur insertion dans la base en question. Cette redondance cause souvent des anomalies provenant de la complexité des insertions. Le deuxième critère relevé traduit les anomalies liées à l'incohérence en modification. La redondance de l'information entraîne des risques en cas de modification d'une donnée répétée à différents endroits. De plus, une mauvaise conception peut causer des anomalies d'insertion et de suppression. Afin de pallier à ces problèmes, de nombreux travaux liés à la mise au point d'une théorie de conception de bases de données relationnelles ont été proposés. Ces travaux ont donné naissance à la fondation d'une nouvelle théorie : la théorie de la normalisation. Basée sur les dépendances fonctionnelles (DF), cette théorie permet de définir la structure de données de la base selon un ensemble de règles permettant la résolution des problèmes liés aux différentes anomalies décrites ci-dessus. L'adaptation de cette théorie dans le processus de conception des bases de données traditionnelles a contribué à leur grand succès.

En parallèle, avec l'émergence des ontologies, plusieurs systèmes permettant leur gestion ont été proposés. Ces systèmes, notés bases de données à base ontologique ($\mathcal{BDBO}$), permettent la persistance à la fois des ontologies et de leurs instances dans la même base de données. Malheureusement, ces $\mathcal{BDBO}$ posent les différents problèmes classiques identifiés dans les bases de données traditionnelles qui ne sont pas soumises au processus de normalisation. Afin de faire face à ce problème et d'améliorer la qualité du schéma relationnel, nous proposons dans ce chapitre, d'exploiter les relations de dépendance entre les propriétés ontologiques et d'intégrer le processus de normalisation dans la conception des bases de données à base ontologique. Nous proposons, dans un premier temps, la modélisation des dépendances ontologiques entre les propriétés des données et leur persistance dans les ontologies. Dans un deuxième temps, nous proposons une approche de conception de $\mathcal{BDBO}$ exploitant ces dépendances pour l'intégration du processus de normalisation dans la phase de conception. Enfin, nous présentons une mise en œuvre de notre approche dans les bases de données à base de modèle de Type 1 et de Type 3.

2 Modélisation des dépendances fonctionnelles entre les propriétés ontologiques

Dans cette section, nous introduisons le concept de dépendances fonctionnelles entre les propriétés ontologiques et nous proposons leur modélisation dans le but de faciliter leur compréhension et leur manipulation.

2.1 Définition

Une dépendance fonctionnelle entre deux propriétés ontologiques P_1 et P_2 ayant pour domaine une classe ontologique C_1 existe si et seulement si la connaissance d'une instance (valeur) de P_1 permet de déterminer une et une seule instance de P_2. La notion d'une telle dépendance est : $(C_1\ ;\ \{P_1\}) \rightarrow P_2$. Une dépendance fonctionnelle est une propriété définie sur toutes les instances d'une classe et pas seulement sur un individu particulier.

Définition	Dépendances Fonctionnelles
$P_2 = P_1 * 2 + P_3$	$C_1 x\ \{P_1, P_3\} \rightarrow P_2$
$P_4 = (P_2 * 30) / 100$	$C_1 x\ \{P_2\} \rightarrow P_4$
P_5 est une propriété clé	$C_1 x\ \{P_5\} \rightarrow P_1$ $C_1 x\ \{P_5\} \rightarrow P_2$ $C_1 x\ \{P_5\} \rightarrow P_3$ $C_1 x\ \{P_5\} \rightarrow P_4$

FIGURE 4.1 – Exemples de dépendances fonctionnelles entre les propriétés ontologiques

2.2 Exemple d'illustration

Soient un ensemble de propriétés $\{P_i, i \in 1..5\}$ ayant pour domaine une classe C_1. Considérons que les propriétés P_2 et P_4 sont des propriétés calculables avec : $P_2 = P_1 * 2 + P_3$ et $P_4 = (P_2 * 30) \div 100$. La propriété P_5 est considérée comme *une propriété clé* (propriété permettant d'identifier de façon unique les instances de sa classe) de la classe C_1. Notons que dans les bases de données à base ontologique, l'URI présente aussi une propriété clé. La différence est que l'URI est une propriété clé générée automatiquement par le système tandis que P_5 présente une propriété clé calculable à partir des dépendances fonctionnelles d'une classe. En se basant sur ces définitions, un ensemble de dépendances fonctionnelles entre les différentes propriétés ontologiques de la classe C_1 est défini. Par exemple, la dépendance fonctionnelle $(C_1\ ;\ \{P_2\} \rightarrow P_4)$ traduit la définition de la propriété P_4. La figure 4.1 décrit les différentes dépendances correspondantes à chacune de ces définitions.

2.3 Graphe de dépendances entre les propriétés ontologiques

Les dépendances entre le propriétés ontologiques sont modélisés à l'aide d'un graphe orienté. Soit P^O l'ensemble de toutes les propriétés primitives et définies d'une ontologie O. Notons $G : (P(P^O), A)$ le *graphe de dépendances entre les propriétés ontologiques* où $P(P^O)$ représente

l'ensemble des parties de propriétés (les nœuds) et A l'ensemble des arcs tel qu'un arc $A_k \in A$ entre une paire de propriétés p_i et p_j ($\in P^O$) existe si une dépendance fonctionnelle C x $\{p_i\}$ $\rightarrow p_j$ a été définie. Reprenons l'exemple d'illustration décrit dans le paragraphe précédent. Un graphe G de dépendances entre les propriétés ontologiques peut être défini pour la classe C_1 où P^O= $\{P_1, P_2, P_3, P_4, P_5\}$ et A=$\{A_1, A_2, A_3, A_4, A_5, A_6\}$ tel que A_i représente la relation de dépendance entre les propriétés appartenant à P^O. Par exemple, l'arc A_1 traduit la relation de dépendance C_1 x $\{P_1,P_3\} \rightarrow P_2$. La figure 4.2 illustre le graphe correspondant à cet exemple. Notons que le graphe peut contenir à la fois des nœuds isolés et des nœuds reliés les uns des

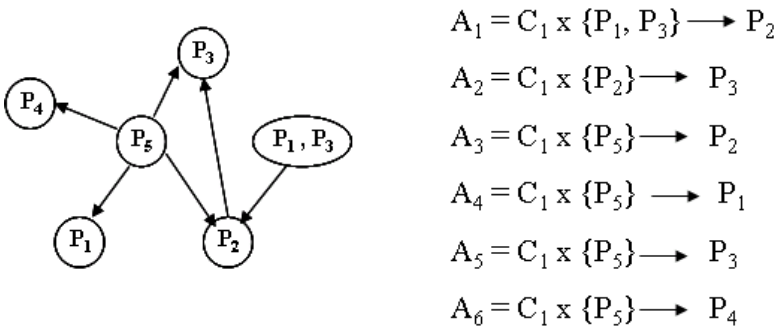

FIGURE 4.2 – Exemple de graphe de dépendances entre les propriétés des données

autres. Les noeuds isolés représentent les propriétés n'ayant aucune relation de dépendance avec le reste des propriétés ontologiques.

3 Persistance des dépendances fonctionnelles entre les propriétés dans une ontologie

Dans le chapitre précédent, nous avons montré l'existence des relations de dépendance entre les propriétés ontologiques et en particulier les propriétés des données (voir section 2.1 (Chapitre 3)). Rappelons qu'à travers les définitions des propriétés dérivées, seules des relations de dépendance entre les propriétés primitives et définies ou les propriétés définies entre elles peuvent être déduites. Aucune relation de dépendance entre deux propriétés primitives ne peut être relevée. Par exemple, considérons les deux propriétés *NSS* et *nom* déterminant respectivement le numéro de sécurité sociale et le nom d'une personne. La dépendance fonctionnelle *NSS* $\rightarrow$ *nom* ne peut être ni exprimée ni déduite à travers la description des concepts ontologiques via les modèles d'ontologies existants (PLIB, OWL, etc.). D'où le besoin d'interagir avec le concepteur pour définir explicitement de telles dépendances. Pour faire face à ce problème, nous proposons, dans cette section, une extension du modèle d'ontologies pour permettre l'intégration des dépendances fonctionnelles entre les propriétés des données et leur modélisation dans une ontologie.

3.1 Intégration des dépendances fonctionnelles entre les propriétés ontologiques dans le modèle d'ontologies

Afin d'intégrer les dépendances fonctionnelles entre les propriétés ontologiques, nous proposons d'adopter le modèle d'ontologies formalisé par Fankam et al. décrivant une ontologie O par le quadruplet $< C, \mathcal{P}, Applic, Sub >$ (voir Chapitre 1.2.4) et l'enrichir avec un nouveau concept noté PFD décrivant les dépendances fonctionnelles entre l'ensemble des parties de propriétés ontologiques tel que suit: O= $< C, \mathcal{P}, Applic, Sub, PFD >$ où :

- $PFD \subseteq C$ x $2^{Applic(C)} \rightarrow Applic(C)$ une relation d'un ensemble de propriétés $\mathcal{P}$ sur $\mathcal{P}$ représentant les dépendances fonctionnelles sur les propriétés des données applicables de $\mathcal{P}$ ayant pour domaine une classe $c_i \in C$. On note :
  - $PFD.RightPart$ la propriété $P_i \in \mathcal{P}$ décrivant la partie droite de PFD i.e. $PFD(c_i, \{P_1, P_2, .., P_n\})$.
  - $PFD.LeftPart$ l'ensemble de propriétés $\{P_1, P_2, .., P_n\} \subset \mathcal{P}$ décrivant la partie gauche (la source) de PFD i.e. $domain(PFD(c_i))$).

3.2 Exemple d'illustration

Afin d'illustrer la formalisation de l'ontologie telle que décrite dans la section précédente, considérons l'exemple suivant. Soient $C^O = \{C_1, C_2, .., C_n\}$ et $P^O = \{P_1, P_2, .., P_n\}$ respectivement l'ensemble des classes et des propriétés d'une ontologie O. Prenons le cas de la classe $C_1 \in C^O$ tel que l'ensemble de ses propriétés applicables $Applic(C_1)$=$\{P_1, P_2, P_3, P_4, P_5\}$. Supposons que l'ensemble de ces propriétés applicables sont des propriétés des données. En se basant sur les définitions des propriétés ontologiques de la classe C_1 de la figure 4.1 (voir Section 2.2), PFD(C_1)={ PF_1, PF_2, PF_3, PF_4, PF_5 } où PF_i décrit une dépendance fonctionnelle entre un ensemble de propriétés des données de la classe C_1. Par exemple, PF_1 traduit la dépendance fonctionnelle entre les propriétés P_1, P_3 et la propriété P_2: PF_1={(C_1 ; {P_1, P_3}) $\rightarrow$ P_2 }. Tandis que PF_2 décrit la dépendance fonctionnelle entre les propriétés P_2 et P_4: PF_2={ (C_1 ; {P_2}) $\rightarrow$ P_4 }.

4 Exploitation des dépendances fonctionnelles entre les propriétés dans la conception des $\mathcal{BDBO}$

Dans une ontologie, un ensemble de propriétés dépendantes les unes des autres peut être associé à chaque classe ontologique. Ces dépendances, traduisant des contraintes sur les instances, représentent la base de la théorie de la normalisation. En effet, le processus de normalisation permet aux modèles de données de vérifier la robustesse de leur conception et l'amélioration de leur modélisation à travers la génération des relations normalisées. Ainsi, il favorise l'élimi-

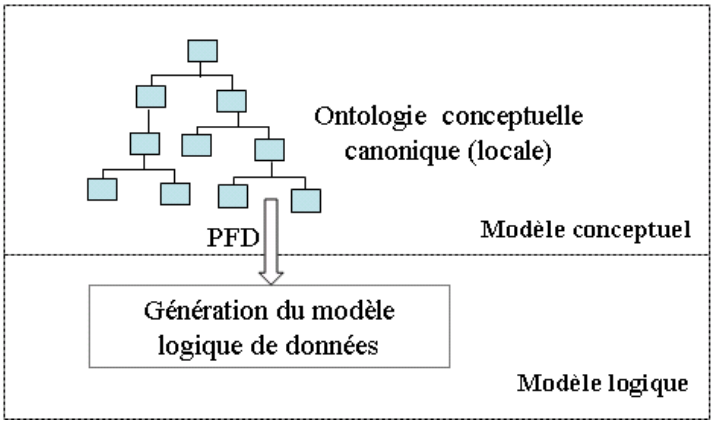

FIGURE 4.3 – Exploitation des PFD dans la conception des $\mathcal{BDBO}$.

nation de la redondance des données lors de leur insertion dans la base de données et permet d'éviter les anomalies de lecture et d'écriture. Afin d'exploiter les dépendances fonctionnelles définies sur les propriétés ontologiques et leur impact dans le processus de normalisation, nous proposons, dans cette section, une approche de conception de bases de données à base ontologique intégrant les PFD dans la phase de la modélisation logique. Nous supposons l'existence d'une ontologie couvrant les besoins de l'utilisateur et enrichie par les dépendances fonctionnelles entre les propriétés des données pour chaque classe ontologique. Nous supposons que l'ontologie traitée est une ontologie conceptuelle canonique (toutes les classes ontologiques sont des classes canoniques). Pour chacune de ses classes, nous exploitons les PFD définies sur ses propriétés pour la génération des tables normalisées en troisième forme normale comme dans les bases de données traditionnelles. Une vue relationnelle définie sur ces tables est associée à chaque classe canonique. La figure 4.3 illustre les différentes étapes de cette approche.

4.1 Modélisation conceptuelle des données

Dans cette section, nous présentons dans un premier temps, la description de l'ontologie adaptée définissant le modèle conceptuel des données. Dans un deuxième temps, nous proposons un traitement sur cette ontologie permettant sa préparation pour le processus de normalisation.

4.1.1 Description de l'ontologie

Dans cette section, nous présentons les hypothèses liées à la définition de l'ontologie. Dans notre approche, nous considérons l'existence d'une ontologie conceptuelle canonique O. Cette ontologie est enrichie par les dépendances fonctionnelles définies sur les propriétés ontologiques de données associées à chaque classe canonique $cc_i \in C^O$. Notons que certaines proprié-

tés peuvent ne pas être impliquées dans ces dépendances. Autrement dit, pour chaque $cc_i \in C^O$, les PFD sont définies sur un ensemble de propriétés P_{PFD} tel que $P_{PFD} \subseteq$ Applic(cc_i). En ce qui concernne ses propriétés héritées P_{her}, notons que $P_{her} \subseteq$ Applic(cc_i).

Prenons l'exemple de la classe canonique *University* (U) définie comme étant le domaine des propriétés *IdUniv*, *Name*, *city*, *region*, *country* et $Univ_{acronym}$ décrivant respectivement l'identifiant de l'université, son nom, la ville, la région et le pays de sa localisation et son acronyme. D'où, $Applic(U)$= {*IdUniv*, *Name*, *city*, *region*, *country*, $Univ_{acronym}$} . Considérons l'ensemble des PFD associé à la classe *University* tel que:

PFD = {PF_1, PF_2, PF_3, PF_4, PF_5, PF_5, PF_6} où:

- PF_1=(U ; {*IdUniv*}) → *Name* ;
- PF_2=(U ; {*IdUniv*}) → *city* ;
- PF_3=(U ; {*IdUniv*}) → *region* ;
- PF_4=(U ; {*IdUniv*}) → *country* ;
- PF_5=(U ; {*city*}) → *region* ;
- PF_6=(U ; {*region*}) → *country*.

Soit P_{PDF^U} les propriétés impliquées dans les dépendances fonctionnelles ontologiques PFD associées à la classe *University*. Ainsi, P_{PDF^U}= {*IdUniv*, *Name*, *city*, *region*, *country*}. Par conséquent, seule la propriété $Univ_{acronym}$ n'est pas impliquée dans P_{PDF^U} ($Univ_{acronym} \notin P_{PDF^U}$ et $P_{PDF^{PU}} \subset Applic(U)$). Ceci nous mène à nous poser la question: Comment traiter les propriétés ontologiques non impliquées dans la définition des dépendances fonctionnelles associées à une classe canonique dans la génération du schéma logique de données? Dans le but d'assurer la complétude du modèle logique de données, nous proposons une phase de préparation permettant l'extension des dépendances fonctionnelles définies sur les propriétés tel que décrit dans la section suivante.

4.1.2 Phase de préparation

La définition des dépendances fonctionnelles entre les propriétés des données d'une classe canonique (*cc*) peuvent impliquer une partie ou la totalité de ses propriétés applicables (*Applic(cc)*). Supposons que ces dernières sont toutes considérées lors de la conception de la $\mathcal{BDBO}$. Notons que les dépendances entre la clé triviale *(URI)* et les propriétés des données de la même classe ne sont pas considérées par défaut et qu'on privilégie les dépendances entre les propriétés ontologiques de données. Dans certains cas, les PFD peuvent ne pas être associées à toutes les classes canoniques. Par conséquent, trois différents cas sont identifiés :

1. $P_{PFD^{cc}} = Applic(cc)$. Les propriétés des données impliquées dans les dépendances fonctionnelles associées à la classe *cc* représente l'intégralité de ses propriétés applicables. Dans ce cas, aucun traitement spécifique n'est appliqué. Toutes les propriétés applicables seront représentées dans le schéma logique relatif à la classe *cc*.
2. $P_{PFD^{cc}} \subset Applic(cc)$. L'ensemble des propriétés des données impliquées dans les dépendances fonctionnelles de la classe *cc* représente un sous ensemble de ses propriétés ap-

plicables. Par conséquent, certaines propriétés ne seront pas représentées dans le schéma logique de cc. Etant donné que chaque concept ontologique est associé à un *identifiant unique (URI)* permettant de le référencer à partir de n'importe quel environnement, indépendamment de l'ontologie dans laquelle il a été défini, une dépendance cc; $\{URI\} \rightarrow p_i$ peut être explicitement définie $\forall\ p_i \in Applic(cc)$. Ainsi, nous proposons d'étendre l'ensemble PFD^{cc} en considérant les dépendances fonctionnelles définies sur l'URI et l'ensemble des propriétés P_{NonImp} non impliquées dans la définition des définition des PFD^{cc} tel que $P_{NonImp} = Applic(cc)$ - $P_{PFD^{cc}}$. Autrement dit, $\forall\ p_i \in P_{NonImp}$, nous considérons la dépendance fonctionnelle entre l'URI et p_i notée : cc; $\{URI\} \rightarrow p_i$.

3. $PFD^{cc} = \emptyset$. Aucune dépendance fonctionnelle entre les propriétés applicables de la classe cc n'a été définie. Nous adoptons le même traitement que pour le cas précédent en définissant explicitement les dépendances fonctionnelles définies sur l'URI et chacune des propriétés applicables de cc. Autrement dit, $\forall\ p_j \in Applic(cc)$, nous considérons la PFD: cc; $\{URI\} \rightarrow p_j$.

En reprenant l'exemple de la classe *University* décrit dans le paragraphe précédent, nous nous situons dans le deuxième cas où $P_{PFD^U} \subset Applic(U)$. Ainsi, en appliquant les règles de préparation au processus de normalisation, PFD^U est étendu par la dépendance PF_7 ={U; $\{URI\}$ $\rightarrow Univ_{acronym}$}. Ainsi, toutes les propriétés applicables de U seront représentées dans la base de données en question. Pour ce faire, une phase de modélisation logique est nécessaire. Celle-ci est définie dans le paragraphe suivant.

4.2 Modélisation logique des données

La modélisation logique des données permet de définir la structure de données dans les bases les stockant. Afin d'être conforme au processus de conception des bases de données traditionnelles, nous proposons, dans notre approche, la génération d'un modèle logique de données en troisième forme normale. Rappelons que le processus de normalisation a pour objectif de décomposer une relation en plusieurs relations normalisées en se basant sur un ensemble de règles. Afin de générer le modèle logique de données à partir d'une ontologie canonique adaptée, six principales phases ont été définies: (a) le calcul de la couverture minimale, (b) la décomposition en relation, (c) la génération des clés primaires (autre que la clé triviale, (d) la génération des clés étrangères (e) la génération des tables normalisées et (f) la génération des vues relationnelles. La figure 4.4 traduit ces différentes étapes.

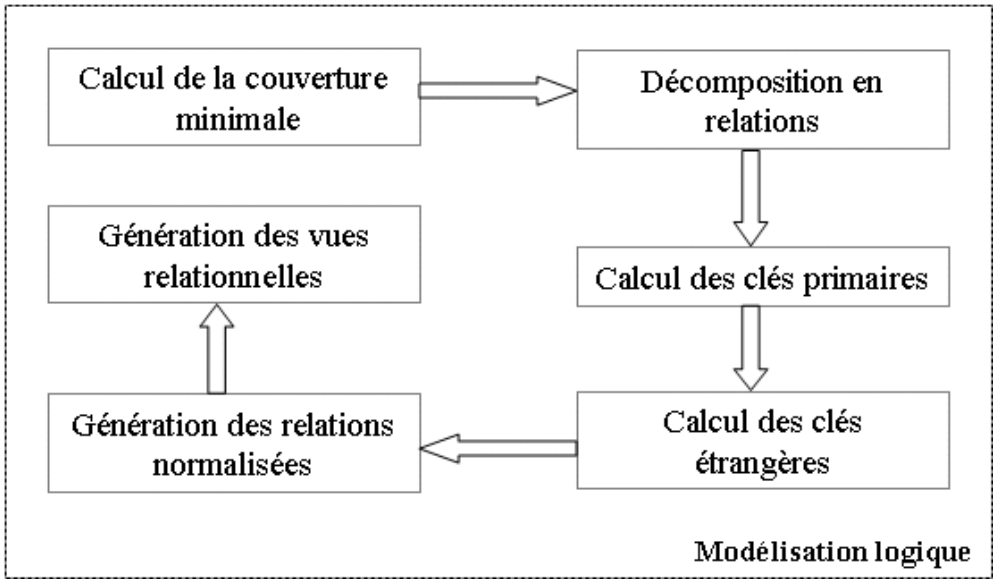

FIGURE 4.4 – Modélisation logique des données ontologiques.

4.2.1 Calcul de la couverture minimale

La couverture minimale (C_{min}) d'un ensemble de dépendances fonctionnelles (DF) est un sous ensemble minimal de DF élémentaires [30] permettant la génération de l'ensemble de toutes les DF en appliquant les règles d'Armstrong [Armstrong, 1974]. Le calcul d'un tel ensemble est un élément essentiel du processus de normalisation. Il permet d'éliminer toute forme de redondance de DF. Dans notre approche, nous nous intéressons à la couverture minimale $C_{min}(cc)$ associée à chaque classe canonique (cc). Reprenons l'exemple des dépendances fonctionnelles PFD_U associées à la classe $University$ (U) de la Section 4.1.1. La dépendance PF_3 (U; {$IdUniv$} $\rightarrow$ $region$) n'appartient pas à la couverture minimale de U étant donné qu'elle peut être déduite des dépendances fonctionnelles PF_2 et PF_5 (U; {$IdUniv$} $\rightarrow$ $city$; U; {$city$} $\rightarrow$ $region$). De la même manière, la dépendance PF_4 ne figure pas dans $C_{min}(U)$ car elle peut être déduite des dépendances PF_2, PF_5 et PF_6 (U; {$IdUniv$} $\rightarrow$ $city$; U; {$city$} $\rightarrow$ $region$; U; {$region$} $\rightarrow$ $country$). Ainsi, $C_{min}(U)$ = { PF_1, PF_2, PF_5, PF_6 }. Elle représente l'ensemble minimal des dépendances fonctionnelles entre les propriétés de U permettant ainsi la déduction de l'intégralité des dépendances fonctionnelles définies par le concepteur sur cette classe. Notons que tout ensemble de dépendances fonctionnelles admet au moins une couverture minimale.

30. une DF élémentaire est une DF de la forme $X \rightarrow Y$, où (i) Y est un attribut unique n'appartenant pas à X ($Y \notin X$) et où (ii) il n'existe aucun X' inclus au sens strict dans X (i.e. $X' \subset$ X) tel que $X' \rightarrow$ Y.

4.2.2 Décomposition en relations

La théorie de la normalisation repose sur un principe de décomposition en relations. En effet, la décomposition d'un schéma de relation R(P_1, P_2, ..., P_N) associé à une classe canonique *cc* où P_1, P_2, ..., P_N représentent l'ensemble des propriétés ontologiques applicables de *cc* est la suivante :

$SCHEMA(R) = SCHEMA(R_1) \infty SCHEMA(R_2) \infty ... \infty SCHEMA(R_n)$, où R_1,..., R_n représentent des relations.

En reprenant l'exemple de la classe *University*, son schéma est décomposé tel que suit:

$SCHEMA(U) = SCHEMA(U_1) \infty SCHEMA(U_2) \infty SCHEMA(U_3) \infty SCHEMA(U_4)$, où:

- $SCHEMA(U_1) = (IdUniv, Name, city)$,
- $SCHEMA(U_2) = (city, region)$,
- $SCHEMA(U_3) = (region, country)$,
- $SCHEMA(U_4) = (URI, Univ_{acronym})$.

Afin de préserver une décomposition sans perte, la définition des clés est nécessaire pour la recomposition de la relation initiale. Deux types de clés sont identifiées: (a) les clés primaires et (b) les clés étrangères.

4.2.3 Génération des clés primaires

1. *Définition.*

   Une clé primaire d'une relation ontologique est un ensemble minimum de propriétés ontologiques qui détermine toutes les propriétés de cette relation. Cette clé est déduite à partir de la couverture minimale de la classe en cours de traitement. Soit $R(P_1, P_2,..., P_N)$ un schéma de relation. Soit X un sous-ensemble de P_1, P_2,..., P_N ($X \subseteq \{P_1, P_2,..., P_N\}$). On dit que X est une clé primaire de R si et seulement si:
   - X détermine P_1, P_2,..., P_N ($X \rightarrow P_1, P_2,..., P_N$);
   - Il n'existe pas de sous ensemble $Y \subset X$ tel que Y détermine P_1, P_2,..., P_N ($\nexists Y \parallel Y \rightarrow P_1, P_2,..., P_N$).

   Notons que la clé primaire de la relation R avant décomposition est notée clé primaire principale (C_p).

2. *Exemple.*

   Reprenons l'exemple de la classe *University*. Considérons la relation $U_2(city, region)$. En tenant compte des dépendances fonctionnelles définies sur ses propriétés, la propriété *city* représente la clé primaire de cette relation.

4.2.4 Génération des clés étrangères

Une clé étrangère d'une relation représente une propriété (ou des propriétés) qui est la clé primaire d'une autre relation et ce pour assurer l'intégrité référentielle des données. Dans une ontologie, deux types de classes peuvent être identifiées: (a) les classes simples (C^{Simple}) et (b) les classes complexes ($C^{Complexe}$). Une classe est dite simple si elle ne contient que des propriétés des données. Une classe complexe est une classe ontologique ayant au moins une propriété d'objets. Le co-domaine de la propriété d'objets est appelé classe membre (C^{Membre}). Ainsi, une classe complexe peut avoir une ou plusieurs classes membres. La distinction entre ces catégories de classes nous mène à identifier deux types de clés étrangères: (1) les clés étrangères internes et (2) les clés de classes.

1. **Clés étrangères internes**. Soit $R_i(P_1, P_2, .., P_i)$ une relation associée à une classe canonique CC_i. En appliquant le processus de normalisation, R_i peut être décomposée en un ensemble de relations $R_1(P_1, P_2,.., P_k)$ et $R_2(P_k,.., P_i)$. P_k est dite clé étrangère interne de R_1 si et seulement si P_k est clé primaire de R_2. Ainsi, une clé étrangère interne d'une relation R_i ne concerne que les classes canoniques subissant une décomposition lors du processus de normalisation. Notons que cette clé n'est définie que sur les propriétés des données.
   - *Exemple.* Reprenons l'exemple de la classe $University$. Considérons les deux relations $U_2(city, region)$ et $U_3(region, country)$. La propriété ontologique *region*, clé primaire de U_3, représente une clé étrangère interne de la relation U_2.
2. **Clés étrangères de classes**.
   - *Définition.* Les clés étrangères de classes sont définies uniquement dans le cas des classes complexes . Elles traduisent la relation de connexion entre deux classes canoniques à travers les propriétés d'objets.
   - *Mise en œuvre.* Afin de définir ces clés, nous proposons d'identifier toutes les propriétés d'objets OP_i^c pour chaque classe canonique complexe. Pour chaque $op_i^c \in OP_i^c$, nous vérifions si op_i^c possède une propriété d'objets inverse $inv(op_i^c)$. Si c'est le cas, nous étudions la cardinalité de la propriété op_i^c (Card(op_i^c)) et de son inverse (Card(inv(op_i^c))). Deux situations se présentent. Dans le cas où nous repérons une relation père-fils (une relation un à plusieurs) comme cela est décrit dans la figure 4.5, nous identifions une clé étrangère associée à la relation fils égale à la clé primaire principale de la relation père. Si la relation fils est décomposée en plusieurs relations, la clé étrangère doit être définie pour toutes ces relations. Dans le cas où nous identifions une relation père-père (une association plusieurs vers plusieurs) tel que représenté dans la figure 4.6, nous identifions une relation d'association ayant pour attributs les clés primaires principales associées à la classe complexe et sa classe membre. Ainsi, ces propriétés représentent la clé primaire de la relation d'association et pointent sur les clés primaires principales de $C^{Complexe}$ et C^{Membre}.
   - *Exemple.* Soient les classes ontologiques $Professor$ et $Course$ décrivant respectivement l'ensemble des professeurs et l'ensemble des cours enseignés. La relation *un*

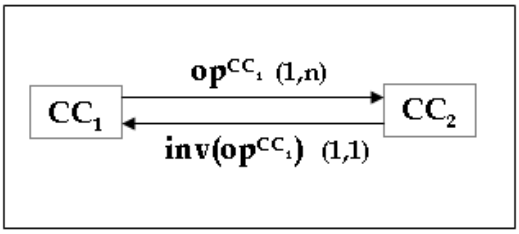

FIGURE 4.5 – Relation hiérarchique père-fils

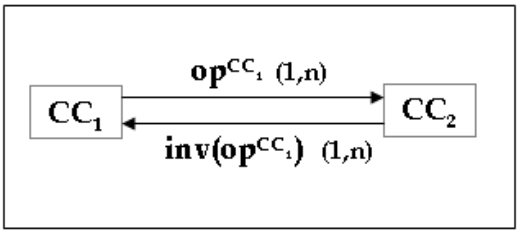

FIGURE 4.6 – Relation maillée père-père

professeur donne un ou plusieurs cours est traduite par la propriété d'objets *Give* ayant pour domaine et co-domaine les deux classes dans l'ordre. Supposons l'existence de la proprié d'objets *GivenBy*, propriété inverse de *Give*, décrivant la relation *un cours est donné par un ou plusieurs professeurs*. En tenant compte des cardinalités des deux propriétés, une relation d'association $Give_{Relation}(IdProf, IdCourse)$ est identifiée entre ces deux classes. Notons que $IdProf$ et $IdCourse$ décrivant respectivement l'identifiant d'un professeur et celui d'un cours représentent les clés primaires associées à ces deux classes.

L'algorithme 1 résume les différentes étapes du processus de gestion des clés étrangères.

4.2.5 Génération des relations

Après le calcul des clés primaires et étrangères, nous générons, pour chaque classe canonique, un ensemble de relations en troisième forme normale. Ces relations englobent les relations issues du processus de décomposition et les relations d'associations. Notons qu'une clé primaire est attribuée à chacune de ces relations.

– *Exemple.*
Reprenons l'exemple de la classe des professeurs $Professor$ (Pr). Soient les propriétés $IdProf$, $name$, $address$ et $city$ décrivant respectivement l'identifiant d'un professeur, son nom, son adresse et sa ville. Soient les dépendances fonctionnelles :
– PF_1^P=$(Pr\,;\{IdProf\}) \rightarrow name$;
– PF_2^P=$(Pr\,;\{IdProf\}) \rightarrow city$;
– PF_3^P=$(Pr\,;\{IdProf\}) \rightarrow address$;
– PF_4^P=$(Pr\,;\{address\}) \rightarrow city$.
Ainsi, les relations $Pr_1(IdProf, name, address)$, $Pr_2(address, city)$ et $Give_{Relation}$*(IdProf, IdCourse)* sont générées. Notons que la propriété $IdProf$ et $address$ sont des clés primaires des relations Pr_1 et Pr_2.

Algorithm 1: Gestion des clés étrangères

Entrées: O: Ontologie conceptuelle canonique;
C^{Simple}: Liste des classes simples de O;
$C^{Complexe}$: Liste des classes complexes de O;
OP_i^c: Liste des propriétés d'objets associées à une classe complexe $c_i^c \in C^{Complex}$;
$Inverse(op_i^c)$: Liste des propriétés d'objets inverses de $op_i^c \in OP_i^c$

Sortie : Identification des clés étrangères

Traitement des classes simples C^{Simple} of O;
pour $\forall\ c_i^s \in C^{Simple}$ **faire**
  Identifier les clés étrangères internes
fin

Traitement des classes complexes $C^{Complexe}$ de O;
pour $c_i^c \in C^{Complexe}$ **faire**
  Identifier les clés étrangères internes
fin
Identification des clés étrangères de classes
pour $\forall\ op_i^c \in OP_i^c$ **faire**
  si $Inverse(op_i^c) \neq \emptyset$ **alors**
    si $Card\ (op_i^c) \neq \emptyset$ **alors**
      pour $\forall\ inv(op_i^c) \in Inverse(op_i^c)$ **faire**
        si $Card\ inv(op_i^c) \neq \emptyset$ **alors**
          si $\exists$ *une relation père-fils entre* op_i^c *et* $inv(op_i^c)$ **alors**
            Identifier une clé étrangère pour la relation fils équivalente à la clé primaire principale associée à la relation père
          fin
          sinon
            si $\exists$ *une relation père-père entre* op_i^c *and* $inv(op_i^c)$ **alors**
              Identifier une relation d'association
            fin
          fin
        fin
      fin
    fin
  fin
fin

4.2.6 Génération des vues

Soit une classe canonique CC_i (P_1, P_2, .., P_n). Après l'application du processus de normalisation, les relations $R_1(P_1, P_2, P_3)$, $R_2(P_3, P_4)$,.., $R_n(P_{n-1}, P_n)$ sont générées. Afin d'assurer la transparence d'accès aux utilisateurs et de leur offrir une interrogation de la base de données sans se soucier de l'implémentation physique des classes, nous proposons la génération d'une vue relationnelle $V_1(P_1, P_2, .., P_n)$ associée à chaque CC_i et définie sur l'ensemble de ses relations normalisées tel que:

$$SCHEMA(V_1) = SCHEMA(R_1) \infty SCHEMA(R_2) \infty ... \infty SCHEMA(R_n)$$

La figure 4.7 retrace le processus de génération d'une vue relationnelle sur les relations normalisées.

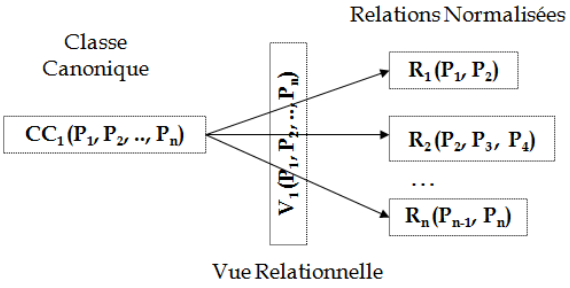

FIGURE 4.7 – Définition d'une vue relationnelle.

– *Exemple.*
Reprenons l'exemple de la classe $Professor$. Une vue relationnelle V_1 est définie sur les relations $Prof_1$, $Prof_2$ et $Give_{Relation}$ telle que :
$SCHEMA(V_1) = SCHEMA(Pr_1) \infty SCHEMA(Pr_2) \infty SCHEMA(Give_{Relation})$.

4.2.7 Algorithme de synthèse adapté aux ontologies

Dans ce paragraphe, nous proposons un algorithme de synthèse adapté aux ontologies et inspiré de l'algorithme de décomposition en troisième forme normale défini pour le processus de conception des bases de données traditionnelles. Cet algorithme prend en entrée une ontologie conceptuelle canonique enrichie par les dépendances fonctionnelles définies sur les propriétés des données et génère pour chaque classe canonique un schéma logique des données en troisième forme normale. Pour chaque classe canonique CC_i, nous identifions l'ensemble des dépendances fonctionnelles élémentaires DEF_{CC_i} défini sur les propriétés applicables de CC_i. A partir de cet ensemble, nous calculons la couverture minimale C_{Min}. Celle-ci est partitionnée en un ensemble de partitions S_i de sorte que les dépendances fonctionnelles de S_i aient la même partie gauche. Chaque partition est transformée en une relation R_i dont la clé est la

partie gauche de S_i. Parmi ces clés, nous identifions la clé primaire principale permettant la détermination de toutes les propriétés impliquées dans la définition des dépendances fonctionnelles entre les propriétés et ce avant la phase de préparation. Ensuite, les clés étrangères sont identifiées et insérées dans toutes les relations R_i. Dans le cas où une dépendance fonctionnelle entre la clé triviale URI et une propriété de donnée a été considérée, une clé étrangère interne est ajoutée dans le schéma de la relation comportant CP_i. Dans le but de préserver la transparence des données, nous définissons le schéma d'une vue relationnelle sur ces relations. Enfin, le schéma logique traduisant les relations normalisées est affiché. L'algorithme 2 résume les différentes étapes décrites ci-dessus.

Algorithm 2: Algorithme de synthèse adapté aux ontologies

Début
  Pour chaque classe CC_i appartenant à l'ontologie canonique faire
    Identifier l'ensemble des dépendances fonctionnelles élémentaires DEF_{CC_i}
    Calculer la couverture minimale C_{Min} de DEF_{CC_i}
    Partitionner C_{Min} en S_1,.., S_n de façon à ce que toutes les dépendances fonctionnelles élémentaire d'une partition aient la même partie gauche
    Construire une relation R_i pour chaque S_i, la partie gauche de S_i étant la clé de R_i
    Identifier la clé primaire principale CP_i
    Insérer chacune des clés étrangères identifiées pour chaque classe CC_i dans toutes les relations R_i
    Si une DF entre l'URI et une propriété de donnée a été explicitement considérée alors
      Ajouter une clé étrangère interne dans le schéma de la relation comportant CP_i
    Fin Si
    Définir une vue relationnelle sur ces relations.
    Afficher le schéma logique des données.
  Fin Pour
Fin

5 Mise en œuvre dans les bases de données à base ontologique

Dans cette section, nous proposons une mise en œuvre de notre approche de conception des bases de données à base ontologique. Dans un premier temps, nous décrivons les efforts réalisés pour une telle mise en œuvre. Dans un deuxième temps, nous présentons une validation de notre approche dans un environnement de $\mathcal{BDBO}$ sous un SGBD industriel (Oracle 11g) et un SGBD académique (OntoDB).

5.1 Efforts réalisés

Le support des dépendances fonctionnelles entre propriétés des données est nécessaire pour l'application de notre approche. Par conséquent, le langage d'ontologies utilisé doit permettre leur représentation et par la suite leur persistance dans la base de données. Étant donné qu'aucun modèle d'ontologies ne permet la représentation de ces dépendances, nous proposons l'extension du méta-modèle du langage d'ontologies manipulé pour le support de ces concepts. Dans la figure 4.8, nous proposons un extrait du méta-modèle générique supportant une telle extension. Soient les méta-classes *Classe*, *Propriété* et *Datatype* décrivant respectivement une classe, une propriété et le co-domaine d'une propriété (type de données dans notre cas). Considérons que ces méta-classes appartiennent au noyau des modèles d'ontologies. Afin de supporter l'ajout des PFD, nous proposons son extension par l'ajout des méta-classes :

- *PFD* définissant une dépendance fonctionnelle élémentaire définie sur les propriétés des données ;
- *PFD.PartieGauche* définissant une partie gauche d'une dépendance *PFD* ;
- *PFD.PartieDroite* définissant une partie droite d'une dépendance *PFD*.

Ces méta-classes sont connectées à travers les propriétés *sa.PFD.PD* et *sa.PFD.PG* qui référencent respectivement la partie droite et la partie gauche de la dépendance en question. Pour chaque méta-classe ajoutée, un ensemble de propriétés sont définies. La propriété *sa_Classe* associée à la méta-classe *PFD* référence la classe à laquelle est associée la dépendance. Les propriétés *PD.Propriété* et *PG.Propriétés*, ayant pour domaines respectifs les méta-classes *PFD. PartieDroite* et *PFD.PartieGauche*, référencent la propriété de la partie droite et les propriétés de la partie gauche de la dépendance.

Une fois le méta-modèle étendu, nous proposons de persister l'ontologie enrichie par les dépendances fonctionnelles entre les propriétés dans la $\mathcal{BDBO}$ cible. Ces dépendances sont par la suite exploitées pour la génération du schéma logique de données et l'amélioration de la qualité de la représentation des données. Enfin, la structure des tables est créée.

5.2 Mise en œuvre dans un SGBD industriel : $\mathcal{BDBO}$ de type$_1$

Afin de valider notre approche, nous présentons, dans cette sous-section, son implémentation sous Oracle 11g. Ce choix se justifie par la place de leader qu'occupe Oracle 11g dans le secteur des bases de données. Dans cette partie, nous proposons la conception de la base de données cible à partir d'une ontologie conceptuelle définie avec le langage OWL et enrichie avec les dépendances fonctionnelles entre les propriétés ontologiques. Durant ce processus de validation, nous présentons un descriptif des différentes étapes suivies ainsi que l'apport de notre approche dans un tel environnement de base de données.

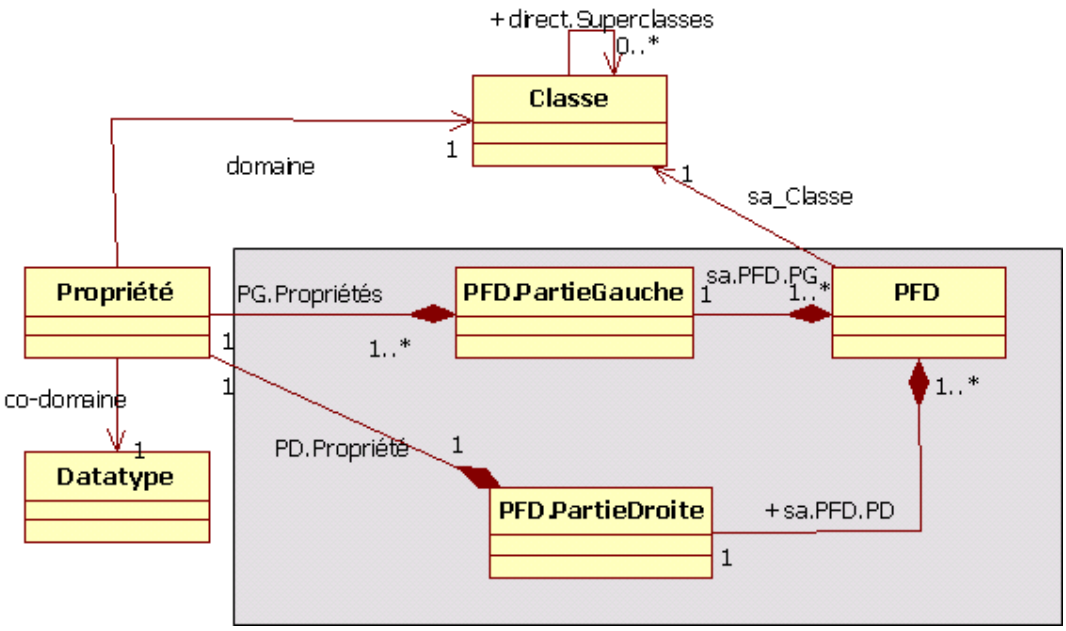

FIGURE 4.8 – Extension générique du méta-modèle

5.2.1 Étapes suivies

La mise en œuvre de notre méthodologie sous Oracle 11g nécessite le suivi d'un ensembles d'étapes : (1) extension du méta-schéma de OWL pour supporter la notion de dépendances fonctionnelles entre les propriétés ontologiques, (2) calcul des clés primaires, (3) chargement et persistance de l'ontologie dans la base de données cible et (4) exploitation des dépendances pour l'amélioration de la qualité de données.

1. **Extension du méta-schéma de OWL**. Étant donné que la conception de la base de données cible est faite à partir d'une ontologie conceptuelle définie avec le modèle d'ontologies OWL et que ce dernier ne permet pas le support des *PFD*, le méta-modèle de celui-ci doit être étendu. Nous proposons d'enrichir le méta-schéma d'OWL par l'ajout des méta-classes *PFD*, *PFD.LeftPart* et *PFD.RightPart*. Celles-ci correspondent respectivement aux méta-classes *PFD*, *PFD.PartieGauche* et *PFD.PartieDroite* du méta-modèle générique. Ces méta-classes sont connectées à travers les propriétés *its.PFD.RP* et *its.PFD.LP* qui correspondent respectivement aux attributs *sa.PFD.PD* et *sa.PFD.PG*. Pour chaque méta-classe ajoutée, un ensemble de propriétés sont définies. La propriété *Its_Class* associée à la méta-classe *PFD* correspond à la propriété *sa_Classe*. Les propriétés *its.RP.Prop* et *its.LP.Prop*, ayant pour domaines respectifs les méta-classes *PFD. RightPart* et *PFD.LeftPart*, correspondent aux propriétés *PD.Propriété* et *PG.Propriétés*. Étant donné que ces dépendances peuvent être exploitées pour le calcul des clés primaires, nous proposons d'enrichir le méta-modèle par l'ajout de la méta-classe *PrimaryKey* dé-

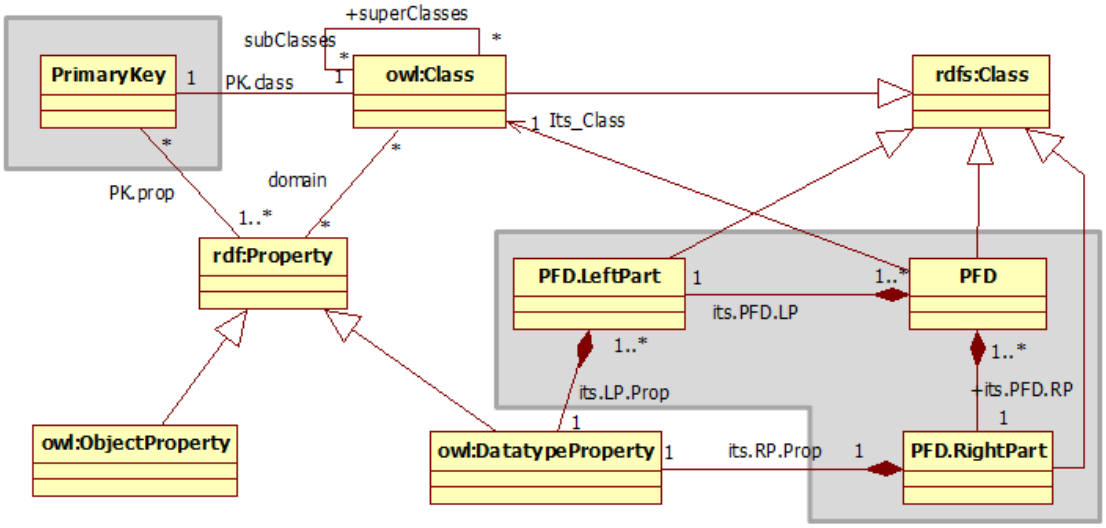

FIGURE 4.9 – Extension du méta-modèle d'OWL.

crivant la clé primaire associée à la relation correspondante à chaque classe canonique. Une clé primaire est unique pour chaque classe ontologique. Cette unicité est assurée par la propriété *PK.class* référençant la classe correspondante à cette clé. De plus, la clé primaire peut être composée d'une ou de plusieurs propriétés. Ceci est défini à travers la propriété *PK.prop*. Notons que la représentation de la clé primaire au niveau méta est établie étant donné que le modèle de stockage de la $\mathcal{BDBO}$ Oracle 11g (table de triplets) ne permet pas sa spécification. La figure 4.9 illustre un fragment du modèle UML décrivant le méta-schéma étendu d'OWL. Une visualisation sous Protégé du méta-modèle enrichi par les dépendances fonctionnelles entre les propriétés ainsi que par la clé primaire par classe est présentée dans la figure 4.10.

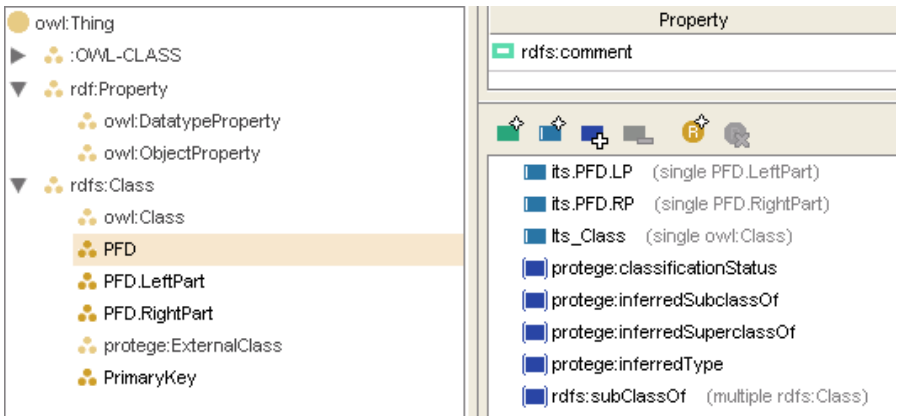

FIGURE 4.10 – Visualisation du méta-modèle enrichi sous Protégé.

2. **Calcul des clés primaires**.
   - *Modélisation.* Les dépendances fonctionnelles entre les propriétés ontologiques sont associées à chaque classe canonique. Elles sont exploitées par la suite pour le calcul des clés primaires pour chacune de ces classes. En effet, pour chaque classe ontologique, une clé primaire est associée. Celle-ci peut être définie par une ou plusieurs propriétés.
   - *Implantation.* Le calcul de la clé primaire est assuré par l'application d'un programme java sur l'ontologie enrichie par les *PFD* et définie avec le langage OWL. Ce programme a pour entrée les *PFD* associées à une classe ontologique et pour sortie la clé générée.
   - *Exemple.* Pour mener notre validation, nous adoptons l'ontologie locale *University* décrivant un fragment de l'ontologie Lehigh University Benchmark comme cela est décrit dans la figure 4.11. Notons que l'ontologie manipulée est canonique. Supposons qu'après l'application du programme de calcul de clé primaire, la propriété *idUniv* est définie comme étant la clé primaire de la classe *University*. Par conséquent, ce calcul déclenche l'instanciation du méta-schéma modélisant la clé primaire par l'ajout des données ontologiques relatives à la définition de cette clé comme cela est décrit dans les balises OWL ci-dessous:

```
<PrimaryKey rdf:ID="PrimaryKey_10">
     <PK.prop rdf:resource="#idUniv"/>
     <PK.class rdf:resource="#University"/>
</PrimaryKey>
```

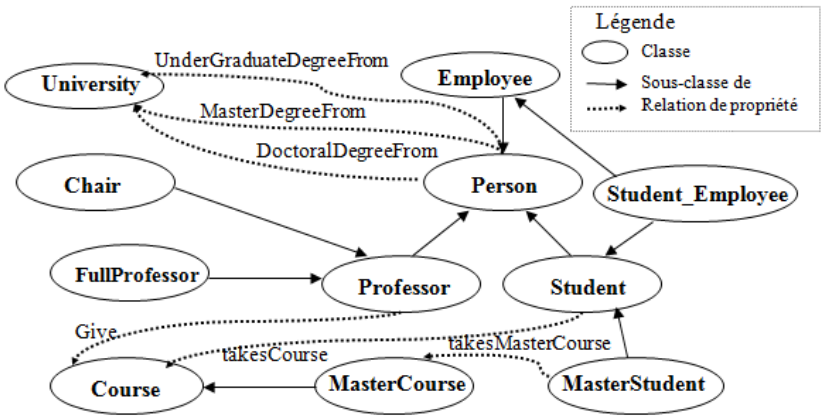

FIGURE 4.11 – Exemple d'ontologie locale canonique: un fragment de LUBM.

3. **Chargement et persistance de l'ontologie**. Dans Oracle, le chargement de l'ontologie nécessite l'enchaînement d'un ensemble d'étapes tel que suit :
   - Conversion de l'ontologie. Les outils proposés par Oracle ne chargent que des données au format N-TRIPLE (.nt). Pour répondre à cette exigence, nous utilisons un convertisseur fourni par l'API Jena 2.6.4 appelé *rdfcat* tel que illustré dans la figure 4.12.

Celui-ci permet la transformation d'un fichier sous format OWL (.owl) en un fichier sous format N-TRIPLE (.nt).
A travers cette conversion, les balises OWL traduisant la définition de la clé principale *idUniv* de la classe *University* (voir paragraphe précédent) sont transformées en un ensemble de triplets de la formet :
(*PrimaryKey*_10, : *type*, *PrimaryKey*),
(*PrimaryKey*_10, *PK.prop*, *idUniv*),
(*PrimaryKey*_10, *PK.class*, *University*).

– Phase de préparation. Une fois l'ontologie décrite sous forme de triplets, une phase de préparation au chargement est nécessaire. Celle-ci nécessite l'enchaînement d'un ensemble d'étapes : (a) activation des modules sémantiques, (b) création du réseau sémantique, (c) création d'une table sémantique et (d) création du modèle sémantique (voir Chapitre 2.5.1.1).
– Chargement. Oracle offre plusieurs techniques de chargement: (i) chargement global, (ii) chargement par lot et (iii) chargement par tuple. Nous avons choisi la première technique pour sa rapidité de chargement. Celle-ci charge les données ontologiques dans une table de relais à l'aide de l'utilitaire *S QLLoader* (sqlldr) avant d'être envoyées dans la base de données au moyen de la procédure.

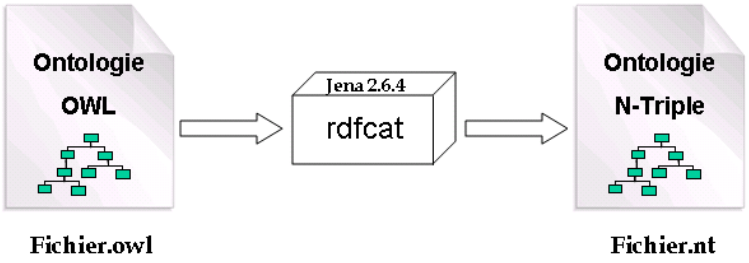

FIGURE 4.12 – Conversion d'une ontologie OWL en une ontologie N-Triple.

4. **Exploitation des dépendances pour l'amélioration de la qualité de données.** Le chargement de l'ontologie implique le stockage des dépendances fonctionnelles entre les propriétés des données ainsi que les clés primaires dites principales calculées pour chaque classe canonique de l'ontologie chargée. Une fois cette étape effectuée, nous proposons la définition d'un ensemble de règles exploitant les clés primaires pour la détection des données inconsistantes. Cette inconsistance concerne la violation de la contrainte d'intégrité traduisant l'unicité de la clé primaire. Par exemple, supposons l'existence des triplets décrivant que University_1 et University_2 sont deux universités ayant pour identifiant la valeur 150 et respectivement pour nom *Université de Poitiers* et *Université de Paris 6*: (University_1, idUniv, 150), (University_1, name, Université de Poitiers), (University_2, idUniv, 150) et (University_2, name, Université de Paris 6) . Étant donné que la propriété *idUniv* est définie comme étant la clé primaire principale de la classe *University*, les triplets chargés présentent une violation de la contrainte d'unicité. En effet, les deux uni-

versités sont différentes mais elles possèdent le même identifiant. Dans le but de détecter de telles données, nous proposons la définition d'une règle permettant de relever les cas de violation d'une clé primaire pour une classe ontologique stockée. Cette identification est effectuée par la génération de triplets sous forme: (s, :comment, 'Inconsistance: violation de la contrainte de clé primaire') où s représente une instance ontologique inconsistante. La définition d'une telle règle se traduit par la création de la base de règle 'testRule' puis par la création d'un index 'rdfs_test_university' prenant en compte la règle définie tel que décrit ci-dessous:

```
EXECUTE SEM_APIS.CREATE_RULEBASE('testRule');
INSERT INTO mdsys.semr_testRule VALUES('inconsistence_rule',
'(?x :PK.prop ?y)(?x :PK.class ?b)(?s ?y ?o) (?c ?y ?o)
 (?s ?p ?w)(?c ?p ?z)(?s :type ?b)(?c :type ?b)
 (?s :differentFrom ?c)', NULL,'(?s :comment
 "inconsistance:violation de la contrainte de clé primaire")',
SEM_ALIASES(SEM_ALIAS('','http://www.lehigh.edu/~zhp2/2004/
0401/univ-bench.owl/')));
COMMIT;
BEGIN
     SEM_APIS.CREATE_RULES_INDEX(
         'rdfs_test_university',
         SEM_Models('UnivModel'),
     SEM_Rulebases('RDFS','testRule'));
END;
/
```

Une application directe de la règle définie ci-dessus sur l'exemple des universités *University*_1 et *University*_2 est la suivante :

(PrimaryKey_10, :PK.prop, idUniv)

(PrimaryKey_10, :PK.class, University)

(University_1, idUniv, 150)

(University_2, idUniv, 150)

(University_1, name, 'Université de Poitiers')

(University_2, name, 'Université de Paris 6')

(University_1, :type, University)

(University_2, :type, University)

(University_1 :differentFrom University_2)

⇒ (University_1 :comment 'inconsistance:violation de la contrainte de clé primaire').

La figure 4.13 décrit le déroulement du processus de détection des données inconsistantes relatives aux universités *University_1* et *University_2*.

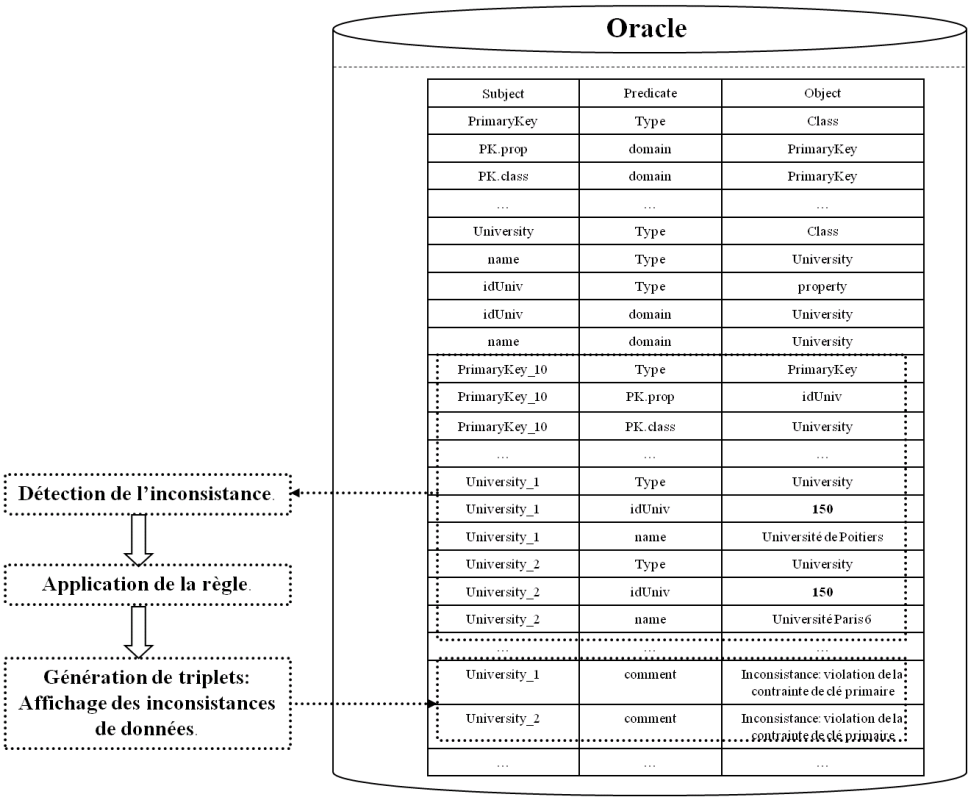

FIGURE 4.13 – Processus de détection des données inconsistantes sous Oracle 11g.

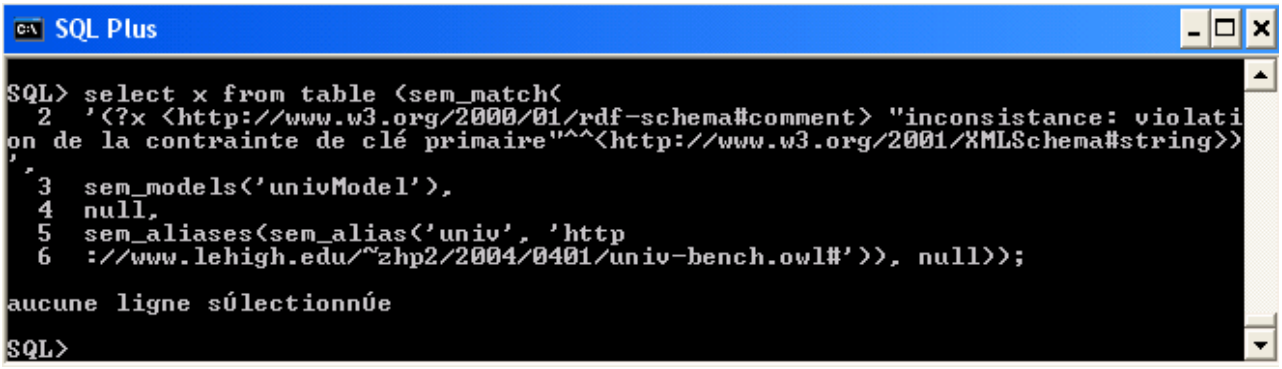

FIGURE 4.14 – Non détection des données inconsistantes.

5.2.2 Apport de notre approche

Dans les bases de données à base ontologique de type 1, les modèles de stockage sont figés. Par exemple, pour la base de données Oracle 11g, la table de triplets est adaptée pour le stockage des concepts ontologiques et des instances les référençant. Par conséquent, notre approche ne peut être appliquée en globalité. Ceci ne limite pas les apports de notre méthodologie à ce type de base de données. En effet, l'intégration des dépendances fonctionnelles entre les propriétés des données dans le modèle d'ontologies permet de calculer les clés primaires et par la suite de les stocker dans la base de données à base de modèles. Ces clés permettent à leur tour de définir des règles de base connues sous le nom de 'rulebases' permettant de poser des contraintes sur les données ontologiques afin que celles-ci soient saisies dans la base conformément aux données attendues. De plus, elles assurent que les données saisies correspondent bien aux limites de l'univers modélisé. Ainsi, ces règles aident à détecter des données inconsistantes. En reprenant l'exemple des universités *University_1* et *University_2*, nous remarquons que lors de l'interrogation de la base Oracle sur les données inconsistantes, aucun résultat n'est retourné (voir figure 4.14). Tandis que lors de l'interrogation de la base sur ces données en se basant sur la règle définie 'testRule' (voir figure 4.15), l'inconsistance est relevée. Notons que cette interrogation est traduite par la requête SPARQL suivante :

```
select x, c from table (sem_match(
'(?x <http://www.w3.org/2000/01/rdf-schema#comment> ?c)',
sem_models('univModel'),
 sem_rulebases('rdfs', 'testRule'),
sem_aliases(sem_alias('univ', 'http
://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#')), null));
```

Une fois détectées, ces données peuvent être supprimées ou modifiées par l'utilisateur final.

5.3 Mise en œuvre dans un SGBD académique : $\mathcal{BDBO}$ de type$_3$

Afin d'avoir une validation complète de notre approche, nous proposons, dans cette sous-section, sa mise en œuvre dans une base de données à base ontologique de type3. Nous avons choisi la base de données OntoDB pour plusieurs raisons.

- Étant donné qu'elle appartient au troisième type d'architecture, il est possible d'assurer l'évolution du modèle d'ontologie utilisé en ajoutant de nouveaux constructeurs. En d'autres termes, elle nous permet d'enrichir le méta-schéma par les dépendances fonctionnelles définies sur les propriétés ontologiques.
- Elle surpasse la plupart des systèmes existants appartenant à l'architecture de type1 et de type2 [Dehainsala et al., 2007].

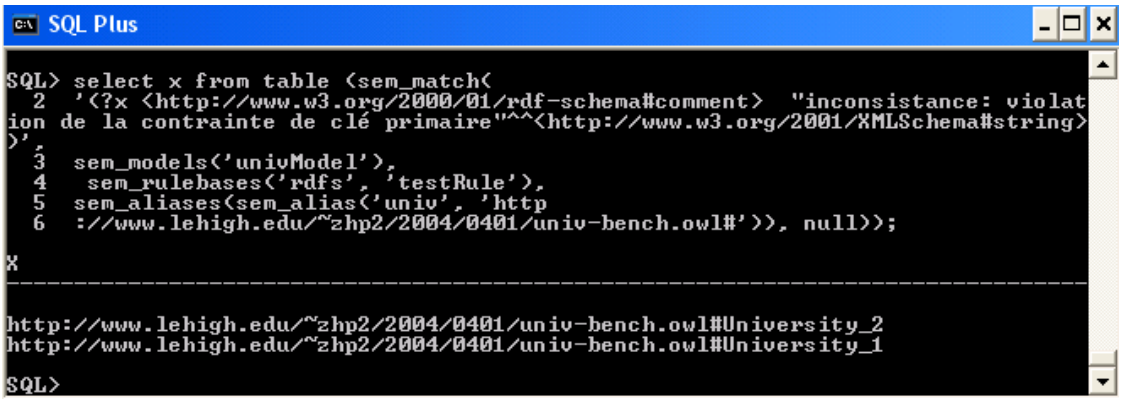

FIGURE 4.15 – Exploitation des 'Rulebases' pour la détection des inconsistances de données sous Oracle 11g.

– Un prototype est disponible, il a été utilisé dans plusieurs projets industriels (sociétés françaises d'automobiles Peugeot et Renault, Institut Français du Pétrole, etc). Celui-ci est associé à un langage de requête appelé OntoQL [Jean et al., 2006a].

Nous proposons un descriptif détaillé des étapes suivies tout au long de la phase de mise en œuvre sous OntoDB avant de dégager les apports de l'adaptation d'une telle approche dans un environnement de base de de données à base de modèles de type$_3$.

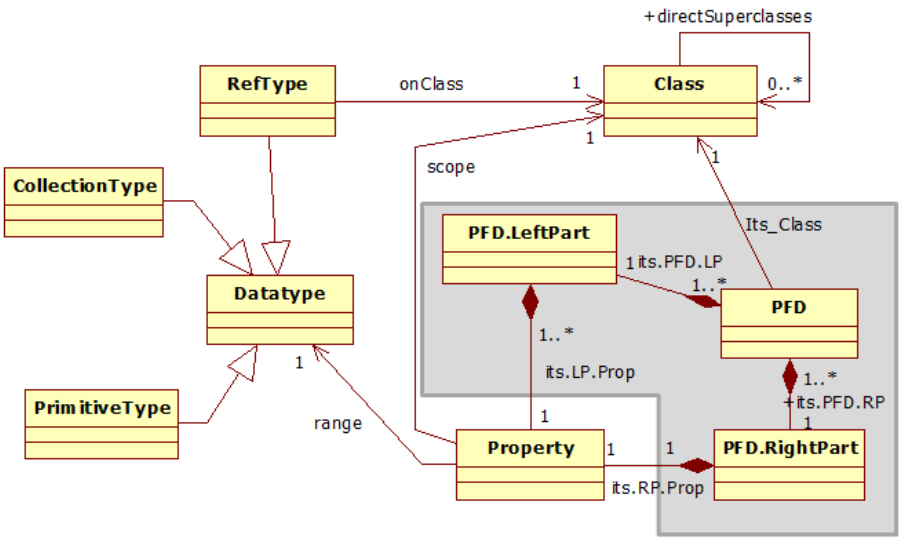

FIGURE 4.16 – Modélisation des dépendances fonctionnelles élémentaires entre les propriétés ontologiques sous OntoDB.

5.3.1 Etapes suivies.

La mise en œuvre de notre approche sous OntoDB nécessite l'enchaînement d'un ensemble d'étapes : (1) extension du méta-schéma par les dépendances fonctionnelles entre les propriétés des données, (2) persistance de l'ontologie canonique enrichie par les dépendances fonctionnelles entre les propriétés des données, (3) génération du schéma logique de données et (4) création de la structure des tables.

1. **Extension du méta-schéma.** Notre approche repose sur le processus de normalisation basé sur les dépendances fonctionnelles définies sur les propriétés des données. Par conséquent, le support des PFD par le noyau initial d'OntoDB s'avère nécessaire. Étant donné qu'OntoDB ne supporte pas la persistance des PFD, nous proposons, durant cette étape l'extension de son méta-modèle par les PFD dans le but de les rendre persistantes dans la base de données en question. Rappelons que le méta-schéma d'OntoDB contient deux principale tables *Entity* et *Attribute* stockant respectivement les entités et les attributs du méta-modèle décrivant la structure de l'ontologie. Pour mener notre validation, nous développons, dans un premier temps, un méta-schéma décrivant la structure initiale du modèle d'ontologies enrichie par les dépendances fonctionnelles entre les propriétés ontologiques. Trois nouvelles entités ont été ajoutées: PFD, $PFD.LeftPart$ et $PFD.RightPart$.

```
CREATE ENTITY #PFD.LeftPart (#its.LP.Prop REF (#property)ARRAY)
CREATE ENTITY #PFD.RightPart (#its.RP.Prop REF (#property))
CREATE ENTITY #PFD (#Its_Class REF (#Class), #its.PFD.RP
REF (#PFD.RightPart), #its.PFD.LP REF (#PFD.LeftPart))
```

   Elles correspondent respectivement aux méta-classes *PFD*, *PFD.PartieGauche* et *PFD. PartieDroite* du méta-modèle générique. Pour chaque entité ajoutée, un ensemble d'attributs est défini. Trois attributs sont associés à l'entité PFD. Le premier attribut Its_Class correspond à la propriété *sa_Classe*. Les attributs $its.PFD.RP$ et $its.PFD.LP$ correspondent respectivement aux propriétés $sa.PFD.PD$ et $sa.PFD.PG$. En ce qui concerne l'entité $PFD.RightPart$, un attribut nommé $its.RP.Prop$ est défini. Ce dernier référence une propriété tandis que l'attribut $its.LP.Prop$ associé à l'entité $PFD.LeftPart$ référence une liste de propriétés. La Figure 4.16 illustre un fragment du modèle UML décrivant le méta-schéma étendu. Dans un deuxième temps, nous instancions les deux tables *Entity* et *Attribute* par l'ajout des nouveaux attributs et entités (*PFD.RightPart*, *PFD.LeftPart*, *its.LP.Prop*, etc.) assurant le support des dépendances fonctionnelles entre les propriétés par la base de données. La Figure 4.17 traduit l'instantiation de ces deux tables. Cette première étape est assuré par l'exécution de l'ensemble de requêtes OntoQL [Jean et al., 2006a] précédentes codant l'instanciation du méta-schéma avec les dépendances fonctionnelles entre les propriétés ontologiques. Plus précisément, elles enrichissent le méta-modèle avec les concepts du modèle UML, assurant la persistance des PFD, tel que décrit dans la figure 4.16.

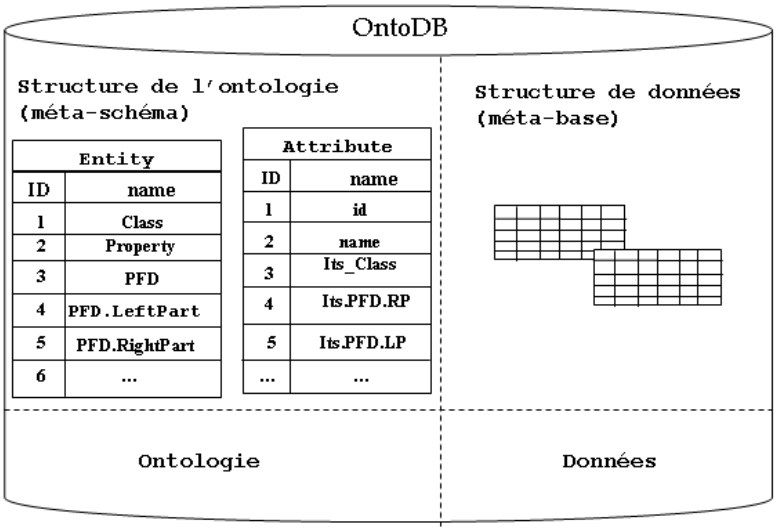

FIGURE 4.17 – Extension du méta-schéma d'OntoDB avec les dépendances fonctionnelles entre les propriétés ontologiques.

2. **Persistance de l'ontologie canonique.** Après l'extension du méta-schéma sous OntoDB, tous les ingrédients pour persister une ontologie enrichie par les *PFD* sont disponibles. Pour illustrer cette étape, nous adoptons un fragment de l'ontologie LUBM tel que défini dans la figure 4.11. Ces concepts ontologiques sont stockés dans la partie ontologie de l'architecture OntoDB. Par exemple, les requêtes OntoQL ci-dessous permettent de persister la classe *University* décrivant le concept d'université. Cette classe est définie comme étant le domaine des propriétés *IdUniv*, *Name*, *Region*, *City* et *Country* décrivant respectivement l'identifiant, le nom, la région et la ville de l'université.

```
CREATE #Class University(DESCRIPTOR (#name[en] = 'University')
        PROPERTIES(IdUniv INT, Name STRING, City STRING,
                   Region STRING, Country STRING))
```

Une fois les classes canoniques créées, les dépendances fonctionnelles élémentaires entre les propriétés sont associées à chacune d'entre elles. Reprenons l'exemple de la classe *University* et supposons l'existence d'une dépendance fontionnelle définie sur les propriétés *IdUniv* et *Name* tel que suit: PFD : $IdUniv \rightarrow Name$. A travers les requêtes OntoQL ci-dessous, cette dépendance est attachée à la classe *University*:

```
Insert Into #PFD (#Its_Class, #its.PFD.RP, #its.PFD.LP)
 Values(
   (Select #oid From #Class c Where c.#name = 'University'),
   (Select #its.RP.Prop.#oid From #PFD.RightPart Where
                              #its.RP.Prop.#name = 'Name'),
    [Select #oid From #Property p Where p.#name = 'IdUniv'])
```

La figure 4.18 traduit l'instanciation de l'exemple décrit ci-dessus.

3. **Génération du schéma logique de données.** La génération du schéma logique de données nécessite l'application de l'algorithme de synthèse appliqué aux ontologies (voir algorithm 2). La première étape de cet algorithme consiste en l'identification de l'ensemble des dépendances fonctionnelles élémentaires définies sur les propriétés des données (DEF_{CC_i}) pour chaque classe canonique. Cette extraction est assurée par l'exécution d'un ensemble de requêtes OntoQL. Par exemple, si nous souhaitons l'extraction des PFD associées à la classe *University*, les requêtes OntoQL ci-dessous sont appliquées:

```
SELECT #PFD.#its.PFD.RP.#its.RP.Prop.#oid,
#PFD.#its.RP.Prop.#its.LP.Prop FROM #PFD
WHERE #PFD.#Its_Class.#name='University';
```

Une fois DEF_{CC_i} extrait (dans notre cas $DEF_{University}$), nous procédons à la normalisation de CC_i à travers l'exécution d'un programme java codant les différentes étapes restantes de l'algorithme 2. Cet algorithme permet de générer le schéma logique de données de l'ontologie canonique étudiée en troisième forme normale. Reprenons l'exemple de la classe *University* où $PFD^U = \{ PF_1, PF_2, PF_3, PF_4, PF_5, PF_5, PF_6, PF_7 \}$ tel que:
– $PF_1 = \{(U; \{IdUniv\}) \rightarrow Name\}$,
– $PF_2 = \{(U; \{IdUniv\}) \rightarrow city\}$,
– $PF_3 = \{(U; \{IdUniv\}) \rightarrow region\}$,
– $PF_4 = \{(U; \{IdUniv\}) \rightarrow country\}$,
– $PF_5 = \{(U; \{city\}) \rightarrow region\}$,
– $PF_6 = \{(U; \{region\}) \rightarrow country\}$,
– $PF_7 = \{(U; \{URI\}) \rightarrow Univ_{acronym}\}$.
En appliquant l'algorithme 2, quatre relations normalisées sont générées :

$\text{University}_1(\underline{IdUniv}, URI, Name, city)$, $\text{University}_2(\underline{city}, region)$,
$\text{University}_3(\underline{region}, country)$, $\text{University}_4(\underline{URI}, Univ_{acronym})$.

Par exemple, la propriété *IdUniv* définit la clé primaire de la relation $University_1$ tandis que les propriétés *URI* et *city* représentent des clés étrangères internes référençant respectivement les clés primaires des relations $University_3$ et $University_4$. Afin de préserver la transparence des données, une vue relationnelle portant le nom de la classe est définie sur ces relations : University($IdUniv$, URI, $Name$, $city$, $region$, $country$, $Univ_{acronym}$). La figure 4.19 traduit le schéma logique de données associé à la classe *University*.

4. **Création de la structure des tables.** Après avoir généré le modèle logique de données de l'ontologie canonique, nous créons, dans la partie *données* sous OntoDB, les tables normalisées ainsi que les vue relationelles définies sur ces tables. En considérant l'exemple de la classe University, quatre tables normalisées ($University_1$, $University_2$, $University_3$, et $University_4$) ainsi qu'une vue relationnelle *University* sont créées. La figure 4.20 illustre la création de la structure de données pour la classe *University* sous OntoDB. Les requêtes OntoQL ci-dessus permet la création de telles structures.

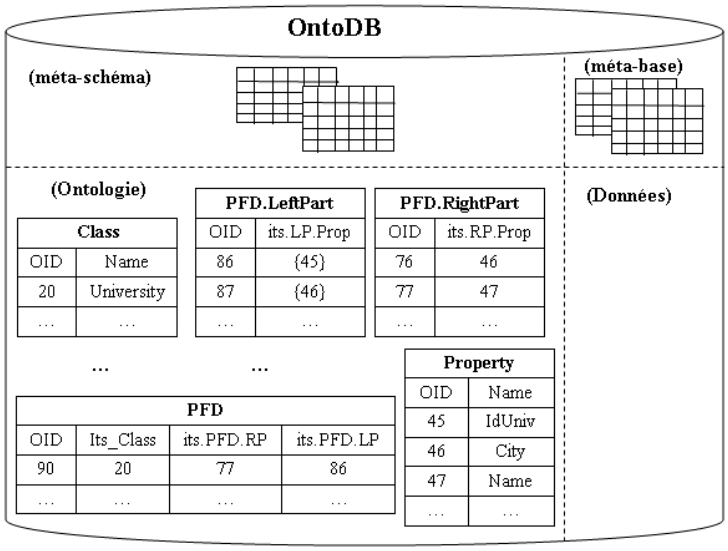

FIGURE 4.18 – Exemple de persistance des concepts ontologiques sous OntoDB.

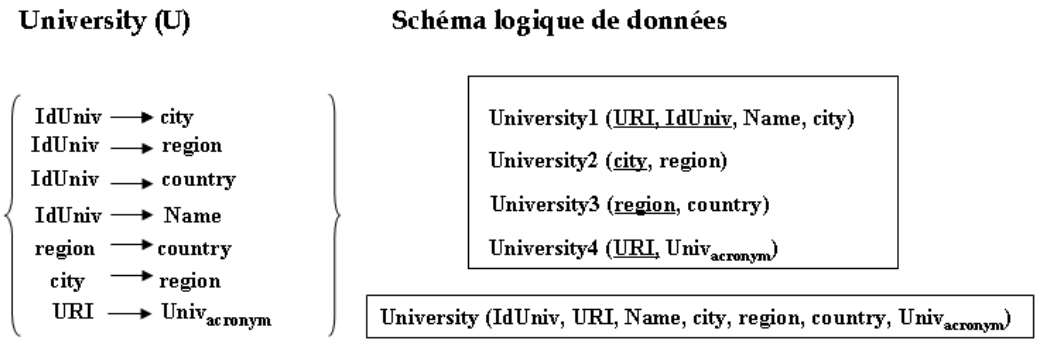

FIGURE 4.19 – Exemple de génération du schéma logique de données en troisième forme normale.

CREATE FDEXTENT OF *University* (*IdUniv*, *name*, *city*, *region*, *country*, $Univ_{acronym}$).

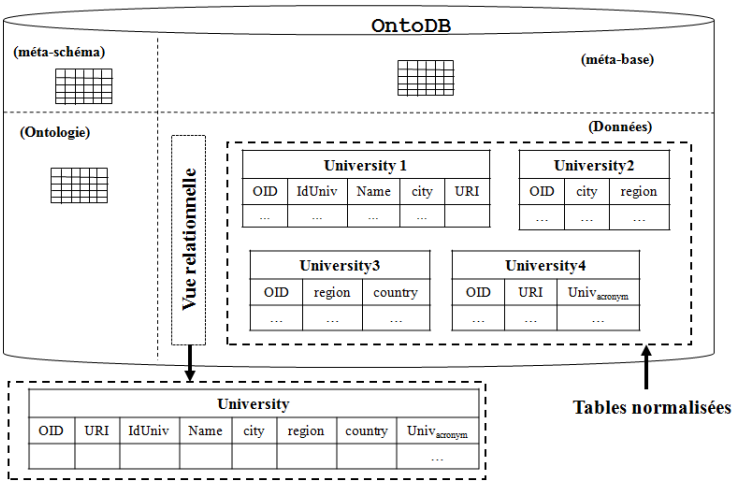

FIGURE 4.20 – Exemple de structure de données normalisée sous OntoDB.

5.3.2 Apport de notre approche

Selon notre approche, la conception des bases de données à base ontologique de type3 à partir d'une ontologie conceptuelle canonique enrichie par les dépendances entre les propriétés présente un ensemble d'avantages. Premièrement, nous proposons la persistance du modèle conceptuel et en particulier des dépendances fonctionnelles élémentaires entre les propriétés ontologiques dans la base de données cible. Celles-ci donnent au concepteur la possibilité d'exprimer des contraintes sur les données ontologiques et de les exploiter au sein de la base de données en question. Deuxièmement, nous proposons une structure de données conforme aux règles de la troisième forme normale. Cette structure offre un stockage de données limitant les formes de redondance des données, et permettant ainsi l'amélioration des performances en réduisant les traitements complexes de mise à jour. De plus, en se basant sur la théorie de la normalisation, nous réduisons les formes d'inconsistance de données. Considérons la structure des données associée à la classe *University* sous OntoDB. Avec l'approche classique de conception sous OntoDB où une table est associée à chaque classe ontologique, la table University(*OID*, *IdUniv*, *Name*, *city*, *region*, *country*, $Univ_{acronym}$) est créée dans la partie *données*. Dans cette table les tuplets (10, 150, Université de Poitiers, Poitiers, Poitou Charentes, France, $U_{Poitiers}$) et (11, 150, Université de Paris 6, Paris, île de France, France, U_{Paris6}) sont insérées. Ces données présentent une inconsistance étant donné que les deux universités (Université de Paris 6 et Université de Poitiers) ont le même identifiant *IdUniv* (150) de caractère unique. Avec l'application du processus de normalisation, ce problème est résolu étant donné que la propriété *IdUniv* est définie comme étant un attribut clé (clé primaire) de la table $University_1$.

University						
OID	IdUniv	Name	city	region	country	$Univ_{acronym}$
10	**150**	Université de Poitiers	Poitiers	Poitou Charentes	France	$U_{Poitiers}$
11	**150**	Université de Paris 6	Paris	Île de France	France	$U_{Paris\ 6}$
...	...	...	...	...	...	...

IdUniv → **city**
IdUniv → **region**
IdUniv → **country**
IdUniv → **Name**

FIGURE 4.21 – Exemple d'inconsistance de données sous OntoDB.

6 Synthèse

L'intégration des dépendances fonctionnelles entre les propriétés des données dans le modèle d'ontologies et son exploitation dans le processus de conception des $\mathcal{BDBO}$ permet l'amélioration de la qualité de la représentation des données dans les bases de données à base ontologique. Dans Oracle, ce type de dépendances permet le calcul de clés primaires et leur exploitation dans la définition de règles posant des contraintes sur les données ontologiques. Ces contraintes ont pour objectif d'améliorer la qualité des données saisies et de réduire les incohérences des données liées aux contraintes d'unicité, etc. Toutefois, le modèle de stockage de données reste toujours non normalisé étant donné que le système Oracle admet une table de triplets figée pour le stockage des instances ontologiques. A l'inverse d'Oracle, un modèle logique de données normalisé est généré pour OntoDB. Ce modèle est défini grâce à l'exploitation des dépendances fonctionnelles définies sur les propriétés ontologiques. Ainsi, les formes de redondance et d'incohérence des données sont réduites et les traitements complexes de mise à jour sont améliorés.

7 Conclusion

Les dépendances fonctionnelles ont largement contribué au succès de la conception des bases de données relationnelles. En effet, elles représentent un atout du processus de la modélisation logique. En parallèle, avec l'émergence des ontologies, les bases de données à base ontologique ($\mathcal{BDBO}$) ont été proposées mais aucune méthodologie prenant en compte leur conception n'a été définie. Par conséquent, dans ce chapitre, nous avons proposé d'exploiter les richesses des dépendances fonctionnelles entre les propriétés ontologiques (PFD) dans le processus de conception des $\mathcal{BDBO}$. Pour ce faire, nous avons d'abord modélisé les dépendances fonctionnelles élémentaires entre les propriétés des données. Puis, nous avons proposé leur persistance en les intégrant dans le modèle d'ontologies. Une fois le support des PFD assuré, nous avons proposé une approche de conception de bases de données à base ontologique exploitant les apports des PFD dans la phase de modélisation logique. Enfin, une mise en œuvre sous

un environnement de bases de données à base de modèles de type1 (Oracle 11g) et de type3 (OntoDB) a été présentée. A travers cette validation, nous avons relevé les apports de notre méthodologie sur chacun de ces deux types de bases de données.

Chapitre 5

Dépendances entre les classes et conception des bases de données à base ontologique

Sommaire

Résumé. Dans le chapitre 3, nous avons montré l'existence des relations de dépendance entre les concepts ontologiques : (1) les dépendances entre les propriétés et (2) les dépendances entre les classes. Le premier type de dépendances a été traité dans le chapitre précédent où une première approche de conception de bases de données à base ontologique ($\mathcal{BDBO}$) a été proposée. Dans notre approche, nous avons proposé une méthodologie de conception de $\mathcal{BDBO}$ à partir d'une ontologie conceptuelle canonique exploitant les dépendances fonctionnelles définies sur les propriétés ontologiques pour la génération d'un modèle logique en troisième forme normale. Dans ce chapitre, nous proposons d'intégrer les dépendances entre les classes dans le modèle d'ontologies et de les exploiter dans le processus de conception des $\mathcal{BDBO}$ à partir d'une ontologie conceptuelle non canonique. Nous proposons, dans un premier temps, la modélisation des dépendances entre les classes ontologiques et leur intégration aux ontologies. Dans un deuxième temps, nous proposons une approche de conception de $\mathcal{BDBO}$ exploitant ces dépendances pour l'identification des classes ontologiques selon leur type (canonique ou non canonique) et l'association du traitement spécifique à chacun de ces deux types de classes. Pour valider notre approche, une mise en œuvre dans un environnement de bases de données à base de modèles est présentée.

1 Introduction

Dans les bases de données relationnelles, les dépendances fonctionnelles occupent une place importante dans le processus de conception. Afin d'exploiter les richesses de ces dépendances dans un contexte ontologique, plusieurs travaux portant sur les dépendances entre les concepts ontologiques ont été proposés [Bellatreche et al., 2010, Calbimonte et al., 2009, Romero et al., 2009]. Dans [Bellatreche et al., 2010, Calbimonte et al., 2009], les auteurs définissent des dépendances sur les propriétés ontologiques tandis que dans [Romero et al., 2009], les dépendances portent sur les classes ontologiques. Dans le chapitre précédent, nous avons intégré le premier type de dépendances dans le processus de conception des bases de données à base ontologique. En effet, dans notre approche, nous avons proposé une méthodologie de conception de $\mathcal{BDBO}$ à partir d'une ontologie conceptuelle canonique (OCC) exploitant les dépendances fonctionnelles définies sur les propriétés ontologiques. Notre hypothèse portant sur le choix du type de l'ontologie est considérée forte étant donnée que la plupart des ontologies manipulées contiennent à la fois des classes primitives et définies. Dans le but de relâcher cette hypothèse, nous proposons, dans ce chapitre, le traitement d'une ontologie conceptuelle non canonique ($OCNC$). En effet, ce type d'ontologie requiert un traitement spécifique des classes non canoniques afin d'éviter toute forme de redondance dans la base de données. Afin de mener à bien ce traitement, la distinction entre les classes selon leur type s'avère nécessaire. Pour ce faire, nous proposons d'exploiter les dépendances définies sur les classes ontologiques pour la classification des classes ontologiques afin de leur attribuer le traitement adéquat. Nous proposons, dans un premier temps, la modélisation des dépendances entre les classes ontologiques et leur persistance dans les ontologies. Dans un deuxième temps, nous proposons une approche de conception de $\mathcal{BDBO}$ exploitant ces dépendances pour l'identification des classes ontologiques selon leur type (canonique ou non canonique) et l'association du traitement spécifique à chacun de ces deux types de classes. Enfin, dans le but de valoriser notre approche, nous présentons sa mise en œuvre dans un environnement de bases de données à base de modèles de Type 3.

2 Modélisation des dépendances entre les classes ontologiques

Dans cette section, nous introduisons le concept des dépendances définies sur les classes ontologiques. Nous commençons par l'identification des constructeurs de non canonicité avant de proposer une classification des dépendances entre les classes. Ensuite, nous proposons leur modélisation afin de faciliter leur compréhension et leur manipulation.

2.1 Identification des constructeurs de non canonicité

Dans cette section, nous nous intéressons aux constructeurs OWL permettant la définition des classes dérivées pouvant exprimer des relations de dépendances entre les classes. En effet,

OWL offre plusieurs constructeurs permettant d'exprimer des équivalences conceptuelles à travers des expressions de classes et des restrictions définies sur les propriétés ontologiques. Ces équivalences permettent par la suite d'identifier des relations de dépendances entre ces classes. Dans notre analyse, nous avons identifié les constructeurs OWL traduisant (1) des opérateurs ensemblistes sur les classes (owl:intersectionOf et owl:unionOf), (2) des restrictions de valeurs sur des propriétés de données (owl:hasValue) et (3) des restrictions de cardinalités sur des propriétés d'objet (owl:cardinality, owl:maxcardinality).

1. **Opérateurs ensemblistes sur les classes ontologiques.** Les langages de modélisation d'ontologies offrent des constructeurs permettant de définir une classe à partir d'opérations ensemblistes telles que l'intersection et l'union. Par exemple, OWL procure les constructeurs owl:unionOf et owl:intersetionOf pour l'expression de telles opérations.
   – *owl:unionOf.* Dans le modèle d'ontologies OWL, le constructeur owl:unionOf traduisant l'opération d'union permet de définir une classe à partir d'une liste de descriptions de classes. Par exemple, considérons les concepts ontologiques *University*, *PublicUniversity* et *PrivateUniversity* définissant respectivement les universités, les universités publiques et les universités privées. En se basant sur la définition des trois classes, le concept *University* englobant à la fois les universités publiques et privées peut être défini comme étant l'union de ces deux types d'universités :
   University ≡ PublicUniversity ∪ PrivateUniversity.
   – *intersectionOf.* Le langage de la modélisation d'ontologies OWL offre un constructeur permettant de traduire l'opérateur d'intersection, noté owl:intersectionOf, définissant une classe à partir d'une liste de descriptions de concepts ontologiques. A titre d'exemple, supposons l'existence des classes ontologiques *Student*, *Employee* et *Student_Employee* décrivant respectivement les étudiants, les employés et les individus à la fois étudiants et salariés. A partir de cette définition, la classe *Student_Employee* peut être définie comme étant l'intersection des descriptions de classes *Student* et *Employee* : Student_Employee ≡ Student ∩ Employee.

2. **Restrictions de valeur sur les propriétés des données.** OWL propose un constructeur noté owl:hasValue permettant de contraindre une propriété P à une valeur V pouvant être soit un individu, soit une valeur de donnée. Ce constructeur offre au concepteur la possibilité de définir une classe dérivée où tous les individus pour lesquels la propriété P concernée a au moins une valeur sémantiquement égale à la valeur V. Prenons l'exemple de la classe *Student* et supposons qu'elle est définie comme étant le domaine de la propriété *level* décrivant le niveau d'études des étudiants. La classe *MasterStudent* décrivant l'ensemble des étudiants inscrits en master peut être ainsi définie comme étant la restriction sur la propriété *level* à la valeur master :

   MasterStudent ≡ ∃ level.{*master*}; Domain (*level*) = *Student*.

3. **Restriction de cardinalités sur des propriétés d'objets.** Dans le langage OWL, les classes ontologiques peuvent être définies à travers des restrictions sur des propriétés

d'objets. Dans cette section, nous nous intéressons aux constructeurs permettant l'expression des restrictions de cardinalités. En effet, OWL offre plusieurs constructeurs permettant la description d'une classe à partir d'une restriction de cardinalités sur une propriété d'objet P. Trois constructeurs de cardinalités sont proposés: owl:cardinality, owl:maxcardinality et owl:mincardinality. Ils traduisent une restriction sur P à travers une contrainte de cardinalité décrivant la classe de tous les individus ayant N valeurs (exactes, au plus ou au moins respectivement selon les constructeurs proposés) sémantiquement distinctes pour la propriété P concernée. Notons que N représente la valeur de la contrainte de cardinalité. Dans le but de générer des dépendances entre les classes ontologiques, nous nous interessons juste aux constructeurs owl:cardinality et owl:maxcardinality suite à leur pouvoir d'expression d'une contrainte de cardinalié où la valeur maximale de N autorisée est égale à 1. Par exemple, considérons la classe *Professor* et Course décrivant respectivement les professeurs et l'ensemble des cours universitaires. Supposons l'existence de la propriété d'objets *Give* décrivant l'action de donner des cours. Notons que cette propriété ait respectivement pour domaine et co-domaine les classes $Professor$ et $Course$ (voir figure 4.11). La classe $VisitingProfessor$ décrivant les professeurs donnant extactement un seul cours à l'université peut être définie comme étant la restriction de la cardinalité sur la propriété *Give* pour une valeur égale à 1 tel que suit : VisitingProfessor ≡ (1) Give. Les requêtes OWL ci-dessus décrivent cette définition.

```
<owl:Class rdf:ID="VisitingProfessor">
   <rdfs:subClassOf> <owl:Restriction>
  <owl:onProperty rdf:resource="#Give"/>
  <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
                                      1</owl:cardinality>
 </owl:Restriction> </rdfs:subClassOf>
<:owl:Class>
```

2.2 Classification des dépendances entre les concepts ontologiques

Notre analyse portant sur les dépendances entre les concepts ontologiques [Bellatreche et al., 2010, Calbimonte et al., 2009, Romero et al., 2009] donne lieu à une classification en : dépendances dirigées par les instances (IDD) et dépendances statiques (SD). La figure 5.1 illustre la classification proposée. Notons que les concepts ontologiques englobent à la fois les propriétés et les classes ontologiques.

– **Dépendances dirigées par les instances.** Les IDD sont similaires aux dépendances fonctionnelles définies dans les bases de données relationnelles. La différence est qu'elles peuvent être définies soit sur les propriétés ontologiques [Bellatreche et al., 2010] (IDD^P où P détermine l'ensemble des propriétés de l'ontologie) soit sur les classes (C) d'une ontologie [Romero et al., 2009] (IDD^C). Prenons le cas des IDD^P et considérons les propriétés des données NSS et $name_{Student}$ désignant respectivement le numéro de sé-

curité sociale et le nom d'un étudiant. Notons que ces propriétés ont pour domaine la classe $Student$ décrivant l'ensemble des étudiants. Supposons que la dépendance fonctionnelle IDD^P: $NSS \rightarrow name_{Student}$ soit définie. Elle signifie que la connaissance d'une valeur de la propriété NSS détermine une seule valeur de la propriété $name_{Student}$. C'est-à-dire, l'existence d'une IDD^P: $P_1 \rightarrow P_2$ ($\{P_1, P_2\} \subset P$) engendre que la connaissance d'une instance de P_1 détermine une instance unique de P_2. D'une manière similaire, si une IDD^C: $C_1 \rightarrow C_2$ ($\{C_1, C_2\} \subset C$) existe, elle signifie que la connaissance d'une instance de C_1 détermine une seule instance de C_2 où C_1 et C_2 représentent deux classes ontologiques. Par exemple, reprenons l'exemple de la classe $VisitingProfessor$ ($\equiv$ (1) *Give*; Domain (*Give*) = $Professor$). Elle signifie qu'un professeur visiteur donne exactement un seul cours universitaire. En se basant sur cette définition, la dépendance IDD^C: $Professor \rightarrow VisitingProfessor$ est définie. Elle signifie que la connaissance d'un professeur détermine au plus un professeur visiteur et ce à partir aux valeurs attribuées à la propriété *Give*.

– **Les dépendances statiques.** Les SD sont identifiées à partir des définitions de classes dérivées. Une dépendance statique SD: $C_1 \rightarrow C_2$ entre deux classes C_1 et C_2 existe si la définition de C_2 peut être dérivée à partir de la définition de C_1. Autrement dit, à partir de la connaissance de la population de C_1, la population de C_2 est déduite. Ces dépendances concernent les classes définies à l'aide d'un ensemble de constructeurs de non canonicité traduisant les relations d'union, d'intersection et de restriction sur la valeur. Si nous considérons le langage de modélisation d'ontologies OWL, les classes définies à l'aide des constructeurs owl:hasValue, owl:unionOf et owl:intersectionOf font l'objet d'une dépendance statique. Pour illustrer, reprenons l'exemple des classes $Student$ et $MasterStudent$ tel que: $MasterStudent \equiv \exists\, level.\{master\}$ et Domain($level$) = $Student$. En se basant sur la définition proposée, la dépendance statique SD: $Student \rightarrow MasterStudent$ est identifiée. Elle signifie que la connaissance de l'ensemble des étudiants permet de déterminer l'ensemble des étudiants inscrits en master.

Après avoir proposé une classification des dépendances entre les concepts ontologiques, nous proposons, dans la section suivante, les règles d'identification des dépendances entre les classes ontologiques. Notons que dans ce chapitre, nous nous intéressons uniquement aux dépendances définies entre les classes et leur impact dans la conception des bases de données à base ontologique.

2.3 Règles de génération des dépendances entre les classes ontologiques

Dans cette section, nous proposons des règles d'identification de dépendances entre les classes C d'une ontologie conceptuelle O. Nous présentons les règles correspondantes à la génération des dépendances statiques ainsi que les dépendances dirigées par les instances portant sur les classes ontologiques.

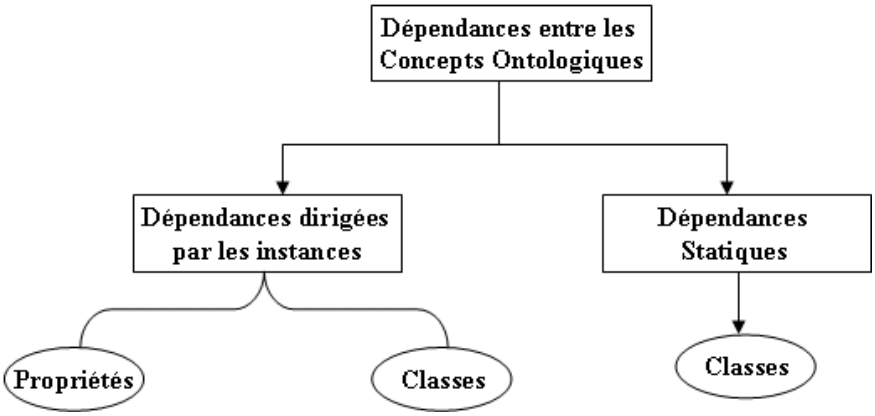

FIGURE 5.1 – Classification des dépendances ontologiques.

2.3.1 Règles de génération des dépendances statiques

Dans ce paragraphe, nous nous intéressons aux classes définies à l'aide des équivalences conceptuelles décrites par le biais des constructeurs traduisant des opérations ensemblistes ou exprimant une contrainte de valeur sur une propriété de données. Pour chaque type de constructeur, nous proposons une règle permettant d'identifier la relation de dépendance entre les classes ontologiques en question.

1. **owl:unionOf.** Soient C_{i1}, C_{i2},.., C_{in} un ensemble de classes appartenant à C. Supposons l'existence d'une classe $C_k \in C$ définie comme étant l'union d'une liste de classes de O telle que: $C_k \equiv C_{i1} \cup C_{i2} \cup .. \cup C_{in}$. Une relation de dépendance entre ces classes est établie : $C_{i1}, C_{i2}, .., C_{in} \rightarrow C_k$. Celle-ci traduit que la connaissance des populations des classes C_{i1}, C_{i2},.., C_{in} permet de déterminer la connaissance de la population de la classe C_k. La règle 1 illustre la définition d'une telle règle. A titre d'exemple, reprenons la définition de la classe *University* (≡ PublicUniversity ∪ PrivateUniversity) du paragraphe 2.1.1. En appliquant la règle proposée, la dépendance PublicUniversity, PrivateUniversity → University est identifiée. Elle signifie que la connaissance des universités privées et publiques permet la détermination de l'ensemble de toutes les universités.

Règle 1 : Règle de génération de SD basée sur l'opérateur d'union

si $C_k \overset{\equiv}{} C_{i1} \cup C_{i2} \cup .. \cup C_{in}$ **alors**
| $C_{i1}, C_{i2}, .., C_{in} \rightarrow C_k$;
fin

2. **owl:intersectionOf.** Soient C_{i1}, C_{i2},.., C_{in} un ensemble de classes appartenant à C. Supposons l'existence d'une classe $C_k \in C$ définie comme étant l'intersection d'une liste de classes de O telle que: $C_k \equiv C_{i1} \cap C_{i2} \cap .. \cap C_{in}$. La relation de dépendance entre ces classes est identifiée : $C_{i1}, C_{i2}, .., C_{in} \rightarrow C_k$. Elle signifie que la connaissance des populations des

classes C_{i1}, C_{i2},.., C_{in} permet de déterminer la population de C_k. La règle 2 illustre la règle de génération de la dépendance entre les classes correspondante à l'opération d'intersection. Par exemple, si nous reprenons l'exemple de la classe *Student_Employee* définie comme étant l'intersection des classes *Student* et *Employee*, la dépendance *Student*, *Employee* → *Student_Employee* est relevée. Elle signifie que la connaissance des étudiants et des employés implique la connaissance des individus à la fois travailleurs et étudiants.

Règle 2 : Règle de génération de SD basée sur l'opérateur d'intersection

si $C_k \equiv C_{i1} \cap C_{i2} \cap .. \cap C_{in}$ **alors**
| $C_{i1}, C_{i2}, .., C_{in} \rightarrow C_k$;
fin

3. **owl:hasValue.** Soit P_i une propriété de donnée ayant pour domaine la classe C_j ($\in C$). Supposons l'existence d'une classe C_k définie comme étant la restriction de la propriété P_i à la valeur x ($C_k \equiv \exists\ P_i.\{x\}$). En se basant sur cette définition, une dépendance entre les classes C_j et C_k est identifiée: $C_j \rightarrow C_k$. Elle signifie que la connaissance de la population de C_j implique la connaissance de la population de C_k. Reprenons l'exemple de la classe *MasterStudent* ($\equiv \exists$ *level*.{*master*}; Domain(*level*) = *Student*). Cette équivalence conceptuelle permet d'identifier la dépendance Student → MasterStudent entre les classes *Student* et *MasterStudent*. Celle-ci traduit que la connaissance des étudiants implique la connaissance des étudiants en master. La règle ci-dessous illustre la règle de génération de la dépendance relative au constructeur *owl:hasValue*.

Règle 3 : Règle de génération de SD basée sur une restriction de valeur

si $C_k \equiv \exists\ P_i.\{x\}$ *and* $Domain(P_i) = C_j$ **alors**
| $C_j \rightarrow C_k$;
fin

2.3.2 Règles de génération des dépendances dirigées par les instances de classes

Après avoir étudié les règles de génération des dépendances statiques, nous proposons dans ce paragraphe, l'étude des règles de génération des dépendances dirigées par les instances de classes. Nous nous intéressons aux classes définies à l'aide des équivalences conceptuelles décrites par le biais des constructeurs exprimant une contrainte de cardinalité sur une propriété d'objets (voir Section 2.1.3). Pour chaque type de constructeur, nous proposons une règle permettant d'identifier la relation de dépendance entre les classes ontologiques en question.

1. **owl:cardinality.** Soit P_i une propriété d'objets ayant pour domaine la classe C_i ($\in C$). Supposons l'existence d'une classe C_k définie comme étant la restriction de la propriété P_i

à une cardinalité égale à 1 ($C_k \equiv (1)\ P_i$). En se basant sur cette définition, une dépendance entre les classes C_i et C_k est identifiée: $C_i \rightarrow C_k$. Elle signifie que la connaissance d'une instance de C_i valuée pour la propriété P_i implique la connaissance d'une seule instance de C_k. La règle 4 illustre la règle de génération de la dépendance relative au constructeur *owl:cardinality*. Reprenons l'exemple de la classe $VisitingProfessor$ ($\equiv$ (1) *Give*; Domain (*Give*) = $Professor$). En se basant sur cette définition, une dépendance entre les classes $VisitingProfessor$ et $Professor$ est identifiée: $Professor \rightarrow VisitingProfessor$. Elle signifie que la connaissance d'une instance de la classe $Professor$ valuée pour la propriété *Give* implique la connaissance d'une seule instance de la classe $VisitingProfessor$.

Règle 4 : Règle de génération de l'IDD relative au constructeur owl:cardinality

si $C_k \equiv (n)\ P_i \mid n = 1$; $Domain(P_i) = C_i$ **alors**
| $C_i \rightarrow C_k$;
fin

2. **owl:maxCardinality.** Soit P_i une propriété d'objet ayant pour domaine la classe C_i ($\in C$). Supposons l'existence d'une classe C_k définie comme étant la restriction de la propriété P_i à une cardinalité maximale égale à 1 ($C_k \equiv (\leq 1)\ P_i$). En se basant sur cette définition, une dépendance entre les classes C_i et C_k est identifiée: $C_i \rightarrow C_k$. Elle signifie que la connaissance d'une instance de C_i valuée pour la propriété P_i implique la connaissance d'une seule instance de C_k. Par exemple, considérons la classe $FullProfessor$ ($\equiv$ ($\leq$1) *Give*; Domain (*Give*) = $Professor$) définissant la liste des professeurs donnant au plus un cours universitaire. En appliquant la règle proposée, une dépendance entre les classes $FullProfessor$ et $Professor$ est établie: $Professor \rightarrow FullProfessor$. Elle signifie que la connaissance d'une instance de la classe $Professor$ valuée pour la propriété *Give* implique la connaissance d'une seule instance de la classe $FullProfessor$. La règle 5 illustre la règle de génération de la dépendance relative au constructeur *owl:maxCardinality*.

Règle 5 : Règle de génération de l'IDD basée sur le constructeur cardinality

si $C_k \equiv (\leq n)\ P_i \mid n = 1$; $Domain(P_i) = C_i$ **alors**
| $C_i \rightarrow C_k$;
fin

2.4 Exemple d'illustration.

Soit C^O l'ensemble des classes d'une ontologie O tel que $C^O = \{C_1, C_2, .., C_{14}\}$. Soient les propriétés ontologiques P_1, P_3, P_6, P_8 et P_{10} ayant respectivement pour domaine les classes C_1, C_3, C_6, C_8 et C_{10}. Supposons que les classes C_1, C_3, C_5, C_6, C_7 et C_{12} sont des classes définies

Définition	Dépendances entre classes
$C_1 \equiv C_2 \cup C_3$	$C_2, C_3 \rightarrow C_1$
$C_3 \equiv \exists P_1.\{x\}$ $Domain(P_1) = C_1$	$C_1 \rightarrow C_3$
$C_6 \equiv (1)\ P_3$ $Domain\ (P_3) = C_3$	$C_3 \rightarrow C_6$
$C_6 \equiv \exists P_1.\{y\}$ $Domain(P_1) = C_1$	$C_1 \rightarrow C_6$
$C_7 \equiv (1)\ P_8$ $Domain\ (P_8) = C_8$	$C_8 \rightarrow C_7$
$C_5 \equiv \exists P_{10}.\{z\}$ $Domain(P_{10}) = C_{10}$	$C_{10} \rightarrow C_5$
$C_{12} \equiv (\leq 1)\ P_6$ $Domain\ (P_6) = C_6$	$C_6 \rightarrow C_{12}$

FIGURE 5.2 – Exemple de dépendances entre les classes ontologiques.

telles que suit: $C_1 \equiv C_2 \cup C_3$; $C_3 \equiv \exists\ P_1.\{x\}$; $C_5 \equiv \exists\ P_{10}.\{z\}$; $C_6 \equiv (1)\ P_3$; $C_6 \equiv \exists\ P_1.\{y\}$; $C_7 \equiv (1)\ P_8$ et $C_{12} \equiv (\leq 1)\ P_6$. En se basant sur ces définitions, un ensemble de dépendances entre ces classes sont identifiées. Par exemple, la dépendance statique $C_2, C_3 \rightarrow C_1$ traduit la définition de la classe C_1. La figure 5.2 résume les différentes dépendances identifiées pour chaque définition de classe.

2.5 Graphe de dépendances entre les classes ontologiques

Afin de faciliter la manipulation et l'exploitation des dépendances entre le classes ontologiques, nous proposons dans cette section, leur modélisation à travers un graphe orienté. Soit C^O l'ensemble de toutes les classes (canoniques et définies) d'une ontologie O. Notons $\mathcal{G} : (P(C^O), A)$ le *graphe de dépendances entre les classes ontologiques* où $P(C^O)$ représente l'ensemble des parties de classes qui représente les nœuds, et un arc $a_k \in A$ entre une paire de parties de classes $p_i(c_i)$ et $p_j(c_j)$ ($\in P(C^O)$) existe si une dépendance entre $p_i(c_i)$ et $p_j(c_j)$ a été définie. Pour illustrer notre modélisation, nous introduisons un exemple d'illustration décrivant un graphe de dépendances entre les classes de l'ontologie décrites dans le paragraphe précédent. Un graphe $\mathcal{G}$ de dépendances entre les classes ontologiques peut être défini pour l'ontologie où $C^O = \{C_1, C_2, .., C_{14}\}$ et A=$\{A_1, A_2, A_3, A_4, A_5, A_6, A_7\}$ où A_i représente la relation de dépendance entre les classes de C^O. Par exemple, l'arc A_2 traduit la relation de dépendance $\{C_1\} \rightarrow C_3$. La figure 5.3 illustre le graphe correspondant à cet exemple.

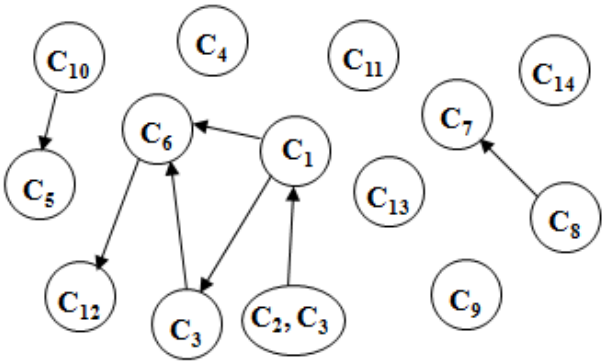

FIGURE 5.3 – Exemple de graphe de dépendances entre les classes ontologiques.

3 Persistance des dépendances entre les classes ontologiques

Dans les sections précédentes, nous avons montré l'existence des relations de dépendances entre les classes ontologiques et nous avons proposé un mécanisme pour leur identification. Afin d'assurer leur intégration et leur modélisation dans le modèle d'ontologies, nous proposons, dans cette section, leur persistance à travers leur intégration dans le modèle d'ontologies.

3.1 Intégration des dépendances entre les classes ontologiques dans le modèle d'ontologies

Afin d'intégrer les dépendances entre les classes ontologiques, nous proposons d'adopter le modèle d'ontologies étendu par les dépendances fonctionnelles entre les propriétés ontologiques décrivant une ontologie O par le quintuplet $< C, \mathcal{P}, Applic, Sub, PFD >$ (voir Chapitre 4 Section 3.1) et l'enrichir avec un nouveau concept noté CD décrivant les dépendances entre le classes ontologiques tel que suit: $O = < C, \mathcal{P}, Applic, Sub, PFD, CD >$ où :

- CD : un mapping d'un ensemble de classes C sur C ($2^C \rightarrow C$) représentant les dépendances définies sur les classes ontologiques. Ces dépendances regroupent à la fois les dépendances statiques (SD) et les dépendances dirigées par les instances (IDD^C). Ainsi, $CD = SD \cup IDD^C$, tel que:
  - IDD^C. Soient I_1, I_2 les populations respectives de C_1 et de C_2. IDD^C: $C_1 \rightarrow C_2$ existe si pour chaque $i_k \in I_1$, il existe un unique $i_j \in I_2$ tel que i_k détermine i_j ($i_k \rightarrow i_j$).
  - SD . Soient I_i, I_j les populations respectives de C_i and C_j. $SD : C_i \rightarrow C_j$ signifie que la connaissance de I_i détermine la connaissance de I_j ($I_k \rightarrow I_j$).

3.2 Exemple d'illustration

Afin d'illustrer la formalisation de l'ontologie tel que décrite dans la section précédente, considérons l'exemple suivant. Soient $C^O = \{C_1, C_2, .., C_{14}\}$ et $P^O = \{P_1, P_2, .., P_{30}\}$ respectivement l'ensemble global des classes et des propriétés d'une ontologie O donnée. Assumons que les classes C_1, C_3, C_5, C_6, C_7 et C_{12} sont définies tel que décrit dans la section 2.4 (voir figure 5.2). En se basant sur ces définitions ainsi que sur les règles de génération de dépendances entre les classes, l'ensemble des dépendances définies sur les classes de O est le suivant: CD^O={ CD_1, CD_2, CD_3, CD_4, CD_5, CD_6, CD_7 } où CD_i décrit une dépendance entre les classes ontologiques tel que suit:

- CD_1={SD_1: C_2, $C_3 \rightarrow C_1$},
- CD_2={SD_2: $C_1 \rightarrow C_3$},
- CD_3={IDD_1^C: $C_3 \rightarrow C_6$},
- CD_4={SD_3: $C_1 \rightarrow C_6$},
- CD_5={IDD_2^C: $C_8 \rightarrow C_7$},
- CD_6={SD_4: $C_{10} \rightarrow C_5$},
- CD_7={IDD_3^C: $C_6 \rightarrow C_{12}$}.

Par exemple, CD_1 traduit la dépendance statique entre les classes C_1, C_2 et C_3 correspondante à la définition de C_1 ($\equiv C_2 \cup C_3$). Tandis que CD_5 décrit la dépendance dirigée par les instances correspondante à la définition de C_7 ($\equiv (1)\ P_8$).

4 Exploitation des dépendances entre les classes ontologiques dans la conception des $\mathcal{BDBO}$

Dans une ontologie conceptuelle, des classes dépendantes les unes des autres peuvent être définies. Ces dépendances, traduisant des contraintes sur la population des classes, permettent d'établir un processus de typage de classes. Ce processus détermine la nature des classes par la distinction entre les classes canoniques et les classes non canoniques. Dans notre démarche de conception de bases de données à base ontologique, nous nous intéressons au typage des classes dans le but de distinguer le traitement spécifique à chacun de ces deux types de classes. En effet, le traitement similaire de la représentation des classes canoniques et non canoniques dans la $\mathcal{BDBO}$ engendre des anomalies de redondances due à la persistance des instances ontologiques appartenant à la fois aux tables associées aux classes canoniques et non canoniques. Afin de faire face à ce problème, nous proposons dans cette section, l'exploitation des relations de dépendance entre les classes ontologiques dans le processus de conception des $\mathcal{BDBO}$. Nous supposons l'existence d'une ontologie non canonique couvrant les besoins de l'utilisateur et enrichie par les dépendances entre les classes ontologiques et les dépendances fonctionnelles entre les propriétés des données pour chaque classe. Cette ontologie est soumise au processus de typage des classes exploitant les dépendances définies sur ses classes. Une fois les classes

canoniques et non canoniques identifiées, un traitement spécifique pour chaque type est appliqué. Il correspond à la génération du modèle logique des données correspondant aux classes de l'ontologie. Celui-ci se traduit par l'application du processus de normalisation sur les classes canoniques et la génération de vues de classes pour les classes non canoniques.

4.1 Processus de typage des classes ontologiques.

Dans cette section, nous proposons l'exploitation des dépendances définies sur les classes ontologiques pour la distinction entre les classes selon leur type. D'abord, nous introduisons la notion de couverture minimale définie sur les dépendances entre les classes. Celle-ci est par la suite exploitée pour la définition d'un algorithme de canonicité permettant l'identification des classes canoniques (primitives) et non canoniques (dérivées).

4.1.1 Couverture minimale des dépendances entre les classes ontologiques.

La couverture minimale (C^+) d'un ensemble de dépendances CD définies sur les classes ontologiques est un sous ensemble minimal de CD permettant la génération de l'ensemble de toutes les dépendances entre les classes. Il permet d'éliminer toute forme de redondance de CD. Le calcul d'un tel ensemble est un élément essentiel pour le déroulement du processus de typage des classes (canonique ou non canonique) tel que décrit dans la section suivante. Notons que tout ensemble de dépendances entre les classes admet au moins une couverture minimale. Reprenons l'exemple des dépendances définies sur les classes ontologiques tel que décrit dans le paragraphe 3.2. La dépendance CD_4 (SD_3: $C_1 \rightarrow C_6$) n'appartient pas à la couverture minimale C^+ étant donnée qu'elle peut être déduite à partir des dépendances CD_2 (SD_2: $C_1 \rightarrow C_3$) et CD_3 (IDD_1^C: $C_3 \rightarrow C_6$). Ainsi, $C^+ = \{ CD_1, CD_2, CD_3, CD_5, CD_6, CD_7 \}$. Cette couverture représente l'ensemble minimale des dépendances entre les classes de l'ontologie O permettant la déduction de l'intégralité des dépendances CD définies sur toutes les classes de O.

4.1.2 Algorithme de canonicité

1. *Description.* Dans cette section, nous proposons un algorithme de canonicité permettant l'identification des types de classes. Il identifie les classes canoniques et non canoniques d'une ontologie à partir de son graphe de dépendances. Notons que (i) les classes canoniques représentent l'ensemble minimal des classes ontologiques permettant par la suite la définition de toutes les classes de l'ontologie et que (ii) les classes non canoniques représentent l'ensemble des classes pouvant être dérivées à partir de la connaissance des classes primitives. L'algorithme proposé a pour entrée un graphe $\mathcal{G} : (P(C^O), A)$ de dépendances entre les classes d'une ontologie O donnée. A partir de ce graphe, l'ensemble des classes isolées $C_{Isolated}$ est calculé. Il représente l'ensemble des classes ontologiques n'intervenant pas dans la définition des dépendances CD. Ces classes sont considérées

comme étant canoniques étant donnée qu'elles ne peuvent pas être dérivées à partir de la connaissance des autres classes ontologiques. Ensuite, les ensembles de couverture minimales S_{C^+} sont calculés. Pour chaque couverture minimale C_i^+ ($\in S_{C^+}$) identifiée, nous calculons l'ensemble des classes canoniques CC_i^O. Une fois CC_i^O identifié, nous calculons l'ensemble des classes non canoniques CNC^O représentant les classes de l'ontologie n'appartenant pas à CC_i^O tel que suit: $CNC^O = C^O$ - CC_i^O. Notons que plusieurs combinaisons (solutions) de typage de classes peuvent être générées. Devant une telle situation, le concepteur doit choisir la solution la plus pertinente. L'algorithme 8 résume les différentes étapes du processus de typage des classes ontologiques.

2. *Exemple.* Reprenons l'exemple d'illustration décrit dans le paragraphe 3.2. La première étape consiste à calculer l'ensemble des classes isolées tel que suit : $C_{Isolated} = \{C_4, C_9, C_{11}, C_{13}, C_{14}\}$. Ces classes sont considérées comme étant des classes canoniques. Ensuite, l'ensemble des couvertures minimales S_{C^+} est calculé. Dans notre cas, une seule couverture C_1^+ est identifiée. Elle englobe toute les dépendances définies sur les classes à l'exception de CD_4 (voir paragraphe précédent). Une fois S_{C^+} défini, l'identification des classes canoniques est effectuée. Deux solutions possibles CC_1^O et CC_2^O sont générées où $CC_1^O = \{C_4, C_9, C_{11}, C_{13}, C_{14}, C_2, C_3, C_8, C_{10}\}$ et $CC_2^O = \{C_4, C_9, C_{11}, C_{13}, C_{14}, C_1, C_2, C_8, C_{10}\}$. Pour chacune des solutions générées, un ensemble de classes non canoniques est identifié : $CNC_1^O = \{C_1, C_5, C_6, C_7, C_{12}\}$ et $CNC_2^O = \{C_3, C_5, C_6, C_7, C_{12}\}$. Ainsi, le concepteur est amené à choisir une solution parmi les deux proposées. Maintenant que le typage des classes est établi, nous proposons dans la section suivante, la génération du schéma logique de données.

4.2 Traitement des classes ontologiques.

L'identification de la canonicité des classes déclenche deux différents traitements quant à la génération de la structure des données ontologiques. Le premier traitement est dédidé aux classes canoniques tandis que le deuxième est dédié aux classes non canoniques.

4.2.1 Traitement des classes canoniques.

Lors de la modélisation logique des données, les classes canoniques subissent le processus de normalisation basé sur les dépendances fonctionnelles définies sur leur propriétés de données. Celui-ci engendre la génération d'un ensemble de tables en troisième forme normale pour chaque classe canonique $cc_i \in CC^O$ tel que décrit dans le Chapitre 4 Section 4.2. Afin de préserver la transparence d'accès aux utilisateurs, nous définissons une vue relationnelle sur ces tables pour chaque classe cc_i. Ainsi, nous offrons aux utilisateurs une interrogation de la base de données sans se soucier de l'implémentation logique des classes canoniques.

Algorithm 8: Algorithme de Canonicité

CD: ensemble de toutes les dépendances entre les classes ontologiques;
LP_i: ensemble des classes appartenant à la partie gauche d'une dépendance $CD_i \in CD$;
RP_i: ensemble des classes appartenant à la partie droite d'une dépendance $CD_i \in CD$;
C^O: ensemble de toutes les classes de l'ontologie O;
$P(C^O)$: ensemble des parties de C^O;
$C_{Isolated}$: ensemble des classes n'appartenant ni à LP_i ni à RP_i pour chaque $CD_i \in CD$;
C_i^+: couverture minimale de CD;
S_{C^+}: ensemble de toutes les couvertures minimales possibles;
CC^O: ensemble de toutes les classes canoniques;
CNC^O: ensemble de toutes les classes non canoniques;
S^{CC}: ensemble des CC^O représentant toutes les solutions possibles de classes canoniques.
S^{CNC}: ensemble des CNC^O représentant toutes les solutions possibles de classes non canoniques.
Entrées: $\mathcal{G} : (P(C^O), A)$
Sortie : S^{CC}, S^{NCC}

Init;
$S^{CC} \leftarrow \Phi$;
$CC_i^O \leftarrow \Phi$;
Processus de typage des classes ontologiques;
Calculer $C_{Isolated}$.
$CC_i^O \leftarrow C_{Isolated}$;
CalculerS_{C^+}.
pour $\forall\ C_i^+ \in S_{C^+}$ **faire**
    RemoveCycle(C_i^+).
    pour $\forall\ CD_i \in C_i^+$ **faire**
        Récupérer la partie gauche LP_i de CD_i
        pour $\forall\ C_i \in LP_i$ **faire**
            si $C_i \notin RP_i$ **alors**
                $CC_i^O = CC_i^O \cup C_i$
            fin
        fin
    fin
    $CNC^O = C^O$ - CC_i^O
    $S^{CC} = S^{CC} \cup CC_i^O$
    $S^{CNC} = S^{CNC} \cup CNC^O$
fin

Procedure RemoveCycle(C_i^+);
pour $\forall\ CD_i \in C_i^+$ **faire**
    si $\exists CD_j \in C_i^+$ *tel que* $(RP_j \subset LP_i \& RP_i = LP_j)$ **alors**
        Supprimer CD_j.
    fin
fin

4.2.2 Traitement des classes non canoniques.

La représentation de certaines données ontologiques telles que les instances de classes non canoniques nécessite un traitement spécifique pour éviter leur redondance dans la bases de données en question. En effet, si nous proposons la génération de tables pour les classes non canoniques, nous favorisons la duplication de certaines données ontologiques dans la base de données étant donnée qu'elles seront stockées à la fois dans les tables associées aux classes non canoniques et celles correspondantes aux classes canoniques permettant la dérivation de ces CNC. Pour faire face à ce problème, nous optons pour une structure de données non redondante. Nous proposons le calcul de ces instances à travers des vues, notées vues de classes, calculées à partir des classes de définition des CNC. Pour chaque opérateur de non canonicité, nous proposons une vue de classe comme cela est décrit ci-dessous. Notons que la vue porte le même nom que la classe non canonique.

– *owl:unionOf.* Pour chaque classe non canonique C_k^O définie à partir d'une expression basée sur l'opérateur d'union, nous proposons le calcul d'une vue de classe tel que suit:

  Si $C_k^O \equiv C_i^O \cup C_j^O$ et $\{P_a, .., P_b\} = Applic(C_i^O) \cap Applic(C_j^O)$ alors

  Create view C_k^O as ((select P_a, .., P_b from C_i^O) union (select P_a, .., P_b from C_j^O)).

  Notons que pour la création de la vue de classe, deux choix de définitions sont possibles. Le premier porte sur la définition d'une vue sur les propriétés communes de C_i^O et C_j^O tandis que dans le deuxième, une jointure externe entre les propriétés des deux classes est considérée. Dans notre démarche, seul le premier choix a été traité. Reprenons l'exemple de la classe $University \equiv PublicUniversity \cup PrivateUniversity$. Afin de représenter l'ensemble des universités, nous proposons de créer une vue de classe *University* calculant toutes ses instances tel que suit:

  Create view *University* as (

  (select *idUniv*, *Name*, *Region*, *City*, *Country* from *PublicUniversity*)

  union (select *idUniv*, *Name*, *Region*, *City*, *Country* from *PrivateUniversity*)).

– *owl:intersectionOf.* Pour chaque classe non canonique C_k^O définie à partir d'une expression basée sur l'opérateur d'intersection, nous proposons le calcul d'une vue de classe tel que suit:

  Si $C_k^O \equiv C_i^O \cap C_j^O$ et $\{P_a, .., P_b\} = Applic(C_i^O) \cap Applic(C_j^O)$ alors

  Create view C_k^O as ((select P_a, .., P_b from C_i^O) intersect (select P_a, .., P_b from C_j^O)).

  Notons que deux choix possibles se présentent pour la création de la vue de classe. Dans le premier, seules les propriétés communes des deux classes sont considérées pour la définition de la vue. Dans le deuxième cas, une jointure externe entre les propriétés de C_i^O et C_j^O est considérée. Dans notre démarche, seul le premier choix a été traité. Par exemple, considérons la classe non canonique des étudiants travailleurs *Student_Employee* définie comme étant l'intersection des classes étudiants et travailleurs ($\equiv Employee \cup Student$). Pour représenter l'ensemble des individues étudiants travailleurs, nous proposons de calculer une vue de classe *Student_Employee* tel que suit:

Create view $Student\_Employee$ as ((select NSS, age, $address$ from $Student$) intersect (select NSS, age, $address$ from $Employee$))

– *owl:hasValue.* Pour chaque classe non canonique C_k^O définie à partir d'une expression basée sur une restriction de valeur, nous proposons le calcul d'une vue de classe tel que suit:

Si $C_k^O \equiv \exists\ P_i^O.\{x\}$ and $Domain(P_i^O) = C_j^O$ alors
Create View C_k^O as (select * from C_j^O where $C_j^O.P_i^O$ = x).

Par exemple, considérons la classe non canonique des étudiants en master définie tel que suit: $MasterStudent \equiv \exists\ level.\{master\}$; Domain($level$) = $Student$. Afin de représenter l'ensemble de ces étudiant, nous proposons de créer une vue de classe portant la même notation que la classe en question tel que suit:

Create View $MasterStudent$ as
(select * from $Student$ where $Student.level$ = 'master').

– *owl:cardinality.* Pour chaque classe non canonique C_k^O définie à partir d'une expression basée sur une restriction de cardinalité exacte égale à 1, nous proposons le calcul d'une vue de classe tel que suit:

Si $C_k^O \equiv (1)\ P_i^O$ tel que $Domain(P_i^O) = C_i^O$ et $Range(P_i^O) = C_j^O$ alors
Create view C_k^O as (select * from C_i^O where $C_i^O.P_i^O$ in (select URI from C_j^O)).

Reprenons l'exemple de la classe des professeurs visiteurs $VisitingProfessor$ ($\equiv$ (1) $Give$) où le domaine de $Give$ est la classe des professeurs ($Professor$). Supposons que cette classe est classifiée comme étant une CNC. Une vue de classe associée à la classe $VisitingProfessor$ est calculée tel que suit:

Create view $VisitingProfessor$ as (select * from $Professor$ where $Professor.Give$ in (select URI from $Professor$)).

– *owl:maxcardinality.* Pour chaque classe non canonique C_k^O définie à partir d'une expression basée sur une restriction de cardinalité maximale égale à 1, nous proposons le calcul d'une vue de classe tel que suit:

Si $C_k^O \equiv (\leq 1)\ P_i^O$ tel que $Domain(P_i^O) = C_i^O$ et $Range(P_i^O) = C_j^O$ alors
Create view C_k^O as (select * from C_i^O where $C_i^O.P_i^O$ in (select URI from C_j^O)).

Reprenons l'exemple de la classe $FullProfessor$ ($\equiv (\leq 1)\ Give$) et supposons qu'elle est une classe non canonique. Une vue de classe associée à cette classe est la suivante:

Create view $FullProfessor$ as (select * from $Professor$ where $Professor.Give$ in (select URI from $Professor$)).

Maintenant que nous avons présenté notre approche, nous proposons dans la section suivante, un exemple de validation traçant les différentes étapes de notre approche dans un environnement de bases de données à base ontologique.

5 Mise en œuvre dans les bases de données à base ontologique

Dans cette section, nous proposons une mise en œuvre de notre approche de conception des bases de données à base ontologique. Rappelons que celle-ci exploite à la fois les dépendances définies sur les classes ontologiques et sur les propriétés des données pour chacune de ces classes. Notons que cette validation est effectuée dans un environnement de base de de données à base de modèles de type$_3$ sous OntoDB.

5.1 Efforts réalisés

Notre approche repose sur le processus de typage des classes ontologiques. Par conséquent, le support des dépendances entre les classes ontologiques par le modèle d'ontologies s'avère nécessaire. Étant donné qu'aucun modèle d'ontologies ne supporte la persistance des CD, nous proposons l'extension de son méta-modèle par ces dépendances et ce dans le but de les rendre persistantes dans la base de données en question.

Dans la figure 5.4, nous présentons un extrait du méta-modèle générique supportant une telle extension. En se basant sur le méta-modèle générique supportant les PFD (figure 4.8), nous proposons dans cette partie son extension par l'ajout des méta-classes :

- *CD* définissant une dépendance définie sur les classes ;
- *CD.PartieGauche* définissant une partie gauche d'une dépendance CD ;
- *CD.PartieDroite* définissant une partie droite d'une dépendance CD.

Pour chaque méta-classe ajoutée, un ensemble d'attributs est défini. Deux attributs sont associés à la méta-classe CD : $sa.CD.PD$ et $sa.CD.PG$. Ils référencent respectivement la partie droite et la partie gauche de la dépendance en question. Pour chacune de ces deux parties, un attribut est défini. En effet, l'attribut $PD.Class$ associé à l'entité $CD.PartieDroite$ référence une classe tandis que l'attribut $PD.Classes$ associé à la méta-classe $CD.PartieGauche$ référence une liste de classes. Afin de persister le type des classes ontologiques, nous définissons les deux méta-classes *Canonique* et *NonCanonique* décrivant respectivement les classes canoniques et non canoniques. Une fois le méta-modèle étendu, nous proposons de persister l'ontologie non canonique enrichie par les dépendances entre les concepts ontologiques (classes et propriétés de données) dans la $\mathcal{BDBO}$ cible. Les dépendances entre classes sont par la suite exploitées pour la distinction entre les classes canoniques et non canoniques. Enfin, la génération du schéma logique de données est établie à l'aide des dépendances entre les propriétés et le schéma de la base est créé.

5.2 Mise en œuvre dans les bases de données à base de modèles de type$_3$

Afin d'avoir une validation complète de notre approche, nous proposons dans cette section, sa mise en œuvre dans une base de données à base ontologique de type3. Pour mener à bien cette

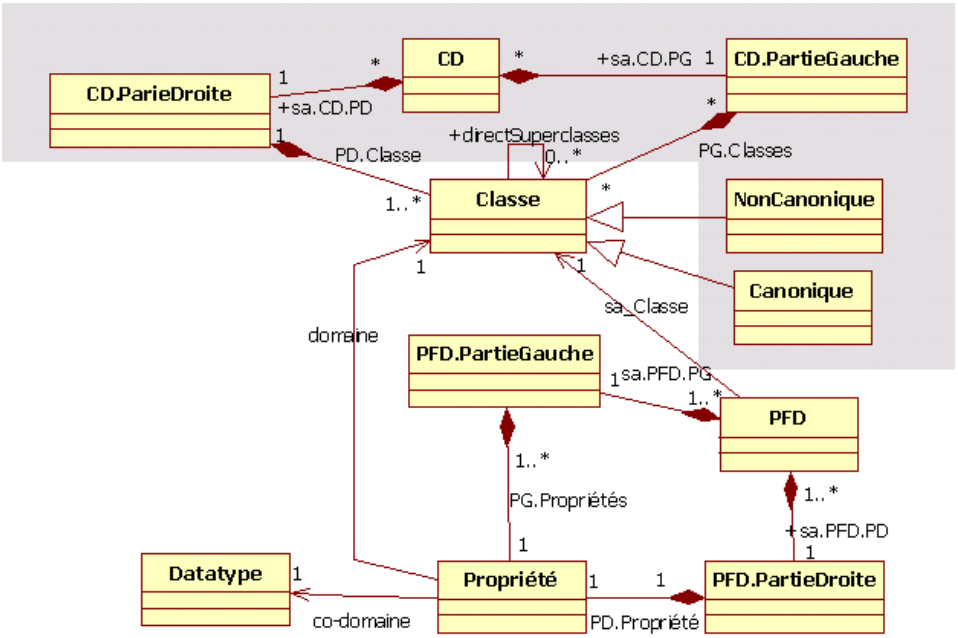

FIGURE 5.4 – Méta-schéma générique : support des dépendances ontologiques

validation, nous avons opté pour une étude de cas dans la base de données OntoDB et ce pour plusieurs raisons (voir Chapitre 4 Section 5.3). Durant cette validation, nous proposons d'étaler les différentes étapes suivies tout au long du processus de conception sous OntoDB. Ensuite, nous présentons les apports de l'application de notre méthodologie pour cette architecture de bases de données.

5.2.1 Étapes suivies

La mise en œuvre de notre approche sous OntoDB nécessite l'enchaînement d'un ensemble d'étapes comme suit.

1. **Extension du méta-schéma.** Notre approche repose sur le processus de typage des classes ontologiques. Par conséquent, le support des dépendances entre les classes ontologiques par le noyau initial d'OntoDB s'avère nécessaire. Étant donné qu'OntoDB ne supporte pas la persistance des CD, nous proposons durant cette étape l'extension de son méta-modèle par les dépendances entre les classes ontologiques dans le but de les rendre persistantes dans la base de données en question.
   – *Description.* Rappelons que le méta-schéma d'OntoDB contient deux principales tables *Entity* et *Attribute* stockant respectivement les entités et les attributs du méta-modèle décrivant la structure de l'ontologie. Pour mener notre validation, nous développons, un méta-schéma décrivant la structure déjà étendue du modèle d'ontologies par les PFD et par la suite enrichie par les dépendances définies sur les classes. Trois nouvelles entités

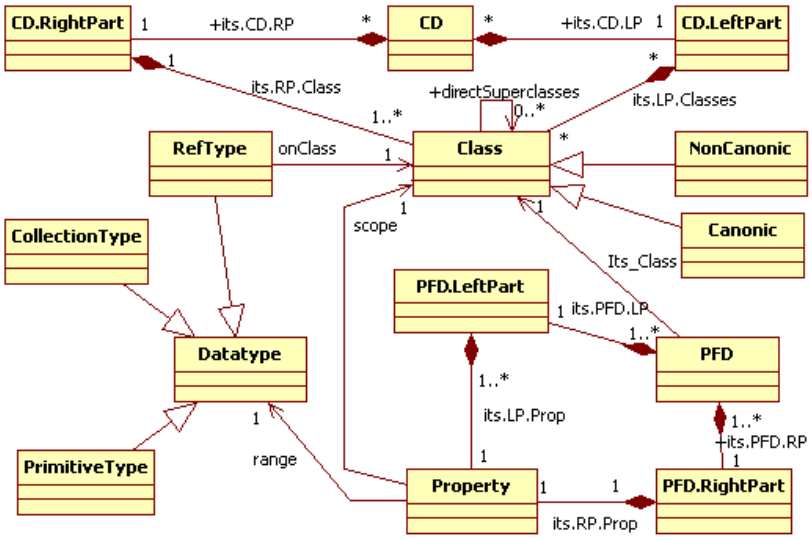

FIGURE 5.5 – Modélisation des dépendances entre les classes sous OntoDB.

ont été ajoutées : *CD*, *CD.LeftPart* et *CD.RightPart*. Elles correspondent respectivement aux méta-classes *CD*, *CD.PartieGauche* et *CD.PartieDroite* du méta-modèle générique. Pour chaque entité ajoutée, un ensemble d'attributs est défini. Deux attributs sont associés à l'entité *CD* : *its.CD.RP* et *its.CD.LP*. Ils correspondent respectivement aux attributs *sa.CD.PD* et *sa.CD.PG*. L'attribut *its.RP.Class* associé à l'entité *CD.RightPart* référence une classe tandis que l'attribut *its.LP.Classes* associé à l'entité *CD.LeftPart* référence une liste de classes. Afin de persister le type des classes ontologiques, nous proposons l'extension du méta-schéma par les deux entités *Canonic* et *NonCanonic* correspondant respectivement aux méta-classes *Canonique* et *NonCanonique*. La Figure 5.5 illustre un fragment du modèle UML décrivant le méta-schéma enrichi.

– *Mise en œuvre.* Cette première étape est assurée par l'exécution d'un ensemble de requêtes OntoQL codant l'instanciation du méta-schéma avec (i) les dépendances entre les classes et (ii) le typage de l'ensemble des classes de l'ontologie tel que décrit dans la figure 5.6. Par exemple, les requêtes permettant l'extension du méta-modèle par les dépendances entre les classes sont les suivantes :

```
CREATE ENTITY #CD.LeftPart (#its.LP.Prop REF (#class)ARRAY)
CREATE ENTITY #CD.RightPart (#its.RP.Prop REF(#class))
CREATE ENTITY #CD (#its.PFD.RP REF (#CD.RightPart),
                   #its.CD.LP REF (#CD.LeftPart))
```

2. **Persistance de l'ontologie non canonique.** Après avoir étendu le méta-schéma sous OntoDB, la base de données à base ontologique est prête pour persister l'ontologie enrichie

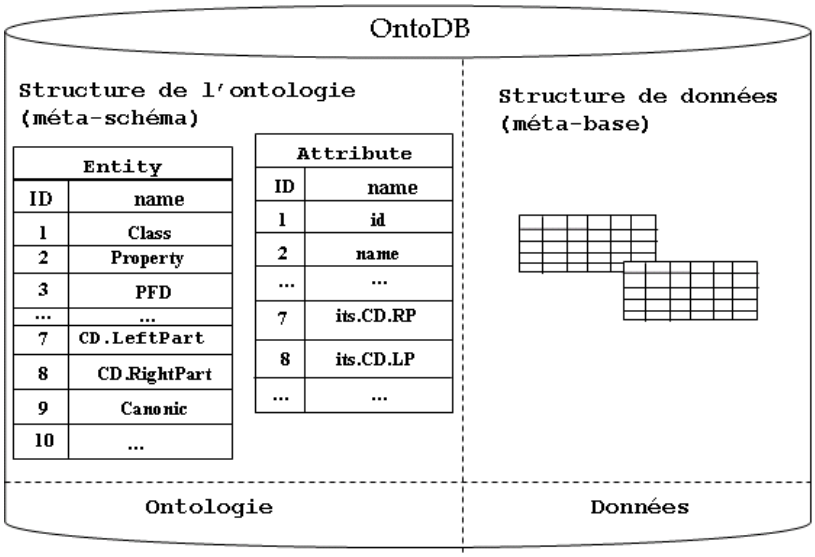

FIGURE 5.6 – Extension du méta-schéma d'OntoDB avec les dépendances entre les classes.

par les dépendances entre les classes. Pour illustrer cette étape, nous utilisons un fragment non canonique de l'ontologie LUBM telle que définie dans la figure 5.7. Ses concepts ontologiques sont ainsi stockés dans la partie ontologie de l'architecture OntoDB. Une fois les classes créées, les dépendances entre elles sont définies. Reprenons l'exemple de la classe $MasterStudent$ définie comme étant la restriction sur la propriété *level* à la valeur master tel que suit: $MasterStudent \equiv \exists\, level.\{master\}$ où le domaine de *level* est la classe des étudiants $Student$. En appliquant les règles de génération des dépendances entre les classes ontologiques, la dépendance $Student \rightarrow MasterStudent$ est définie puis persistée à travers les requêtes OntoQL ci-dessous:

```
INSERT INTO #CD (#its.CD.RP, #its.CD.LP) VALUES
((SELECT #oid from #Class c WHERE c.#name='Student'),
(SELECT #oid from #Class c WHERE c.#name='MasterStudent'))
```

D'une manière similaire, l'instanciation des dépendances entre les propriétés des données pour chaque classe ontologique est persistée (voir Chapitre 4. Section 5.3.1.2). La figure 5.8 traduit l'instanciation de l'exemple décrit ci-dessus.

3. **Typage des classes ontologiques.** Le processus de typage des classes nécessite l'application de l'algorithme de canonicité décrit dans la Section 4.1.2. Reprenons l'exemple de l'ontologie non canonique LUBM et assumons l'existence de l'ensemble des dépendances CD^{LUBM}= { CD_1, CD_2, CD_3, CD_4, CD_5, CD_6 } tel que:
   - CD_1=$\{SD_1$: $PublicUniversity$, $PrivateUniversity \rightarrow University\}$,
   - CD_2=$\{SD_2$: $Employee$, $Student \rightarrow Student\_Employee\}$,
   - CD_3=$\{IDD_1^C$: $Professor \rightarrow FullProfessor\}$,
   - CD_4=$\{SD_3$: $Course \rightarrow MasterCourse\}$,
   - CD_5=$\{SD_4$: $Course \rightarrow DoctoralCourse\}$,

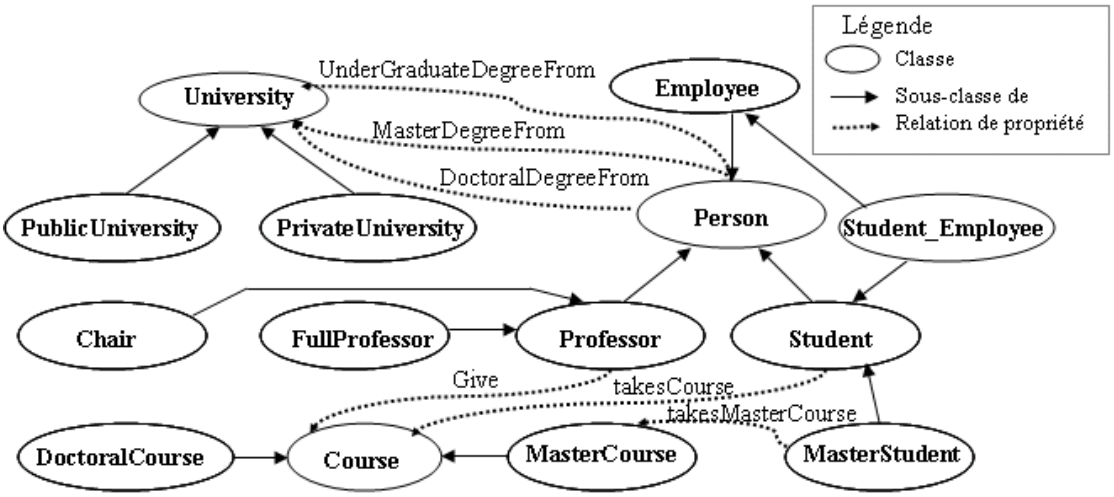

FIGURE 5.7 – Exemple d'ontologie non canonique: un fragment de LUBM.

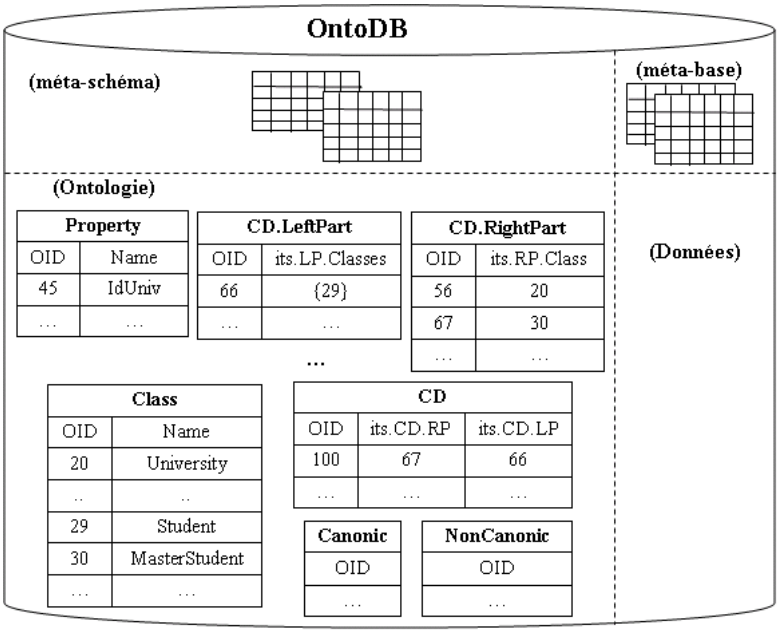

FIGURE 5.8 – Exemple de persistance des dépendances entre les classes sous OntoDB.

– CD_6={SD_5: $Student \rightarrow MasterStudent$}.

La première étape de notre algorithme consiste à calculer l'ensemble des classes isolées de nature canonique : $C_{Isolated}$= {*Chair*, *Person* }. Ensuite, les différentes combinaisons possibles de classification sont calculées : CC^1= { *Chair*, *Person*, *Student*, *Course*, *Professor*, *Employee*, *PrivateUniversity*, *PublicUniversity* } et CNC^1= {*University*, *MasterStudent*, *MasterCourse*, *DoctoralCourse*, *FullProfessor*, *Student_Employee* }. Ce calcul déclenche l'instantiation du méta-schéma modélisant la canonicité des classes par l'ajout des données ontologiques relatives à la définition de la nature des classes. Par exemple, les requêtes OntoQL ci-dessus permettent de persister les identifiants (oid) des classes *Student* et *MasterStudent* dans les deux tables *Canonic* et *NonCanonic* comme cela est décrit dans la figure 5.9.

```
INSERT INTO #Canonic (#oid) VALUES
(SELECT #oid from #Class c WHERE c.#name='Student')
INSERT INTO #NonCanonic (#oid) VALUES
(SELECT #oid from #Class c WHERE c.#name='MasterStudent')
```

Pour chaque type de classes, un traitement spécifique est effectué tel que décrit dans la section suivante.

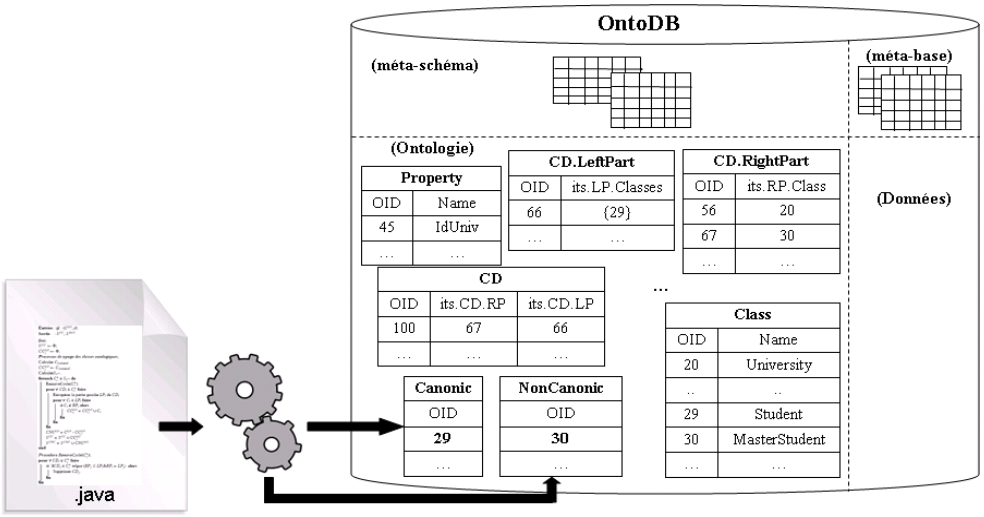

FIGURE 5.9 – Classification des classes ontologiques.

4. **Génération du schéma logique de données.** Durant cette étape, nous proposons un schéma logique de données correspondant à chaque catégorie de classes.
   – *Génération des structures de données relatives aux classes canoniques.* Pour chaque classe canonique, nous proposons d'appliquer le processus de normalisation générant un ensemble de tables normalisées. Reprenons l'exemple la classe des étudiant *Student* définie comme étant le domaine des propriétés *NSS*, *name_Student*, *level* et

takesCourse décrivant respectivement le numéro de sécurité sociale, le nom de l'étudiant, son niveau d'études et l'ensemble de ses cours. Supposons (1) que la propriété d'objets *takesCourse* fait l'objet d'une relation père-père entre les classes des étudiants et des cours (*Course*) et (2) que l'ensemble des dépendances entre les propriétés $PFD^{Student}$ associé à la classe *Student* sont :

$PFD^{Student}$= { PFD_1, PFD_2 } où:
- PFD_1={*Student*: $NSS \rightarrow name\_Student$},
- PFD_2={*Student*: $NSS \rightarrow level$}.

Ainsi, l'exécution de la requête OntoQL ci-dessous permet la création (1) des tables $Student_1(NSS, name\_Student, level)$ et $Student_2(NSS, Id_{Course})$ (la table $Student_2$ est une table d'association) et (2) de la vue relationnelle View *Student* (*NSS*, *name_Student*, *level*, *takesCourse*).

```
CREATE FDEXTENT OF Student (NSS,name_Student, level, Id_Course).
```

– *Génération des structures de données relatives aux classes non canoniques.* Pour chaque classe non canonique, nous proposons la génération d'une vue calculée sur les classes permettant la définition de la *CNC*. Par exemple, reprenons l'exemple de la classe non canonique *MasterStudent*. Étant donnée que l'expression de sa définition est basée sur la description de la classe des étudiants, sa vue aura la même structure de donnée que la vue relationnelle correspondante à *Student* tel que suit: View *MasterStudent* (*NSS*, *name_Student*, *level*, *takesCourse*). Notons que la valeur du champ *level* est toujours égale à la valeur *master* et que le calcul d'une telle vue nécessite l'application du script ci-dessous:

```
Create View MasterStudent as
  (select * from Student where Student.level = 'master').
```

La figure 5.10 illustre l'implémentation de cet exemple dans la partie données sous OntoDB.

5.2.2 Apport de notre approche

Selon notre approche, la conception des bases de données à base ontologique de type3 à partir d'une ontologie conceptuelle non canonique enrichie par les dépendances entre les concepts ontologiques (propriétés et classes) présente un ensemble d'avantages. Dans le chapitre précédent, nous avons montré les avantages qu'apportent les dépendances fonctionnelles entre les propriétés ontologiques quant au processus de conception (voir Chapitre 4 Section 5.3.2). Dans ce chapitre, nous relevons les apports de l'exploitation des dépendances définies sur les classes ontologiques dans un tel processus. Premièrement, nous proposons la persistance de l'ontologie enrichie dans la base de données cible et en particulier des dépendances définies sur les classes. Celles-ci donnent la possibilité au concepteur d'exprimer des contraintes sur les classes ontologiques et de les exploiter pour la définition de la structure de données. Deuxièmement, nous proposons d'exploiter ces dépendances pour typer les classes en les différenciant en canoniques

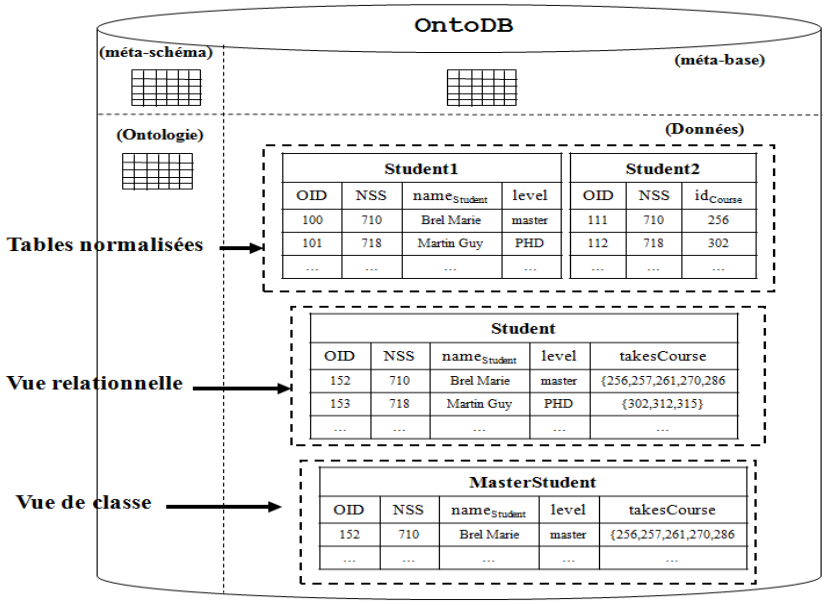

FIGURE 5.10 – Classification des dépendances ontologiques.

et non canoniques tout en persistant leur classification dans la base de données en question. Cette classification va permettre de procéder à un traitement spécifique pour chaque type de classes afin de réduire la redondance des données à la fois des instances de classes canoniques et non canoniques.

6 Conclusion

Dans le chapitre 4, nous avons traité les dépendances définies sur les propriétés ontologiques et leur impact sur la conception des bases de données à base ontologique. Dans le but d'exploiter les richesses des dépendances entre les concepts ontologiques, à la fois définies sur les classes et les propriétés, nous avons consacré ce chapitre à l'intégration de ces deux différents types de dépendances afin de mener à bien le processus de conception des $\mathcal{BDBO}$. Pour ce faire, nous avons d'abord modélisé les dépendances conceptuelles et en particulier les dépendances définies sur les classes ontologiques. Puis, nous avons proposé leur persistance en les intégrant dans le modèle d'ontologies. Une fois le support des dépendances entre les propriétés et les classes assuré, nous avons proposé une approche de conception de bases de données à base ontologique exploitant les apports de ces deux types de dépendances dans la phase de la modélisation logique des données. Enfin, une mise en œuvre sous un environnement de bases de données à base de modèles de type3 (sous OntoDB) a été présentée. A travers cette validation, nous avons relevé les apports de notre méthodologie sur le processus de conception d'une telle base.

Chapitre 6

Approche de conception et de déploiement de bases de données à base ontologique

Sommaire

Résumé. Dans le but de définir une approche dédiée à la conception des $\mathcal{BDBO}$, nous avons proposé dans les chapitres précédents l'exploitation des relations de dépendance entre les concepts ontologiques durant le processus de conception. Nous avons proposé une approche intégrant à la fois les dépendances définies sur les classes ontologiques et celles définies sur les propriétés des données pour la définition d'un modèle logique de données en troisième forme normale. Dans cette approche, la phase de modélisation conceptuelle a été brièvement présentée et aucun mécanisme spécifique du traitement de la subsomption des classes ontologiques n'a été proposé. De plus, aucune approche de déploiement de notre méthodologie dans un environnement de $\mathcal{BDBO}$ n'a été définie. Afin de réaliser ces améliorations, nous proposons dans un premier temps de détailler davantage notre méthodologie dédiée à la conception des bases de données à base ontologique en mettant l'accent sur la phase de modélisation conceptuelle et le processus de placement des classes ontologiques dans la hiérarchie de subsomption. Dans un deuxième temps, nous proposons une approche de déploiement de notre méthodologie dans un environnement de $\mathcal{BDBO}$. Finalement, une étude de cas sous OntoDB est présentée.

1 Introduction

En examinant la littérature, plusieurs types de bases de données ont été proposées : les bases de données relationnelles (BDR), les bases de données objet (BDO), les bases de données relationnelles objet (BDRO), les bases de données à base ontologique ($\mathcal{BDBO}$), etc. A l'exception des $\mathcal{BDBO}$ où aucune approche de conception n'a été présentée, chaque type de base de données a été accompagné par la proposition d'une méthodologie de conception inspirée dans la plupart du temps du processus de conception des BDR. Ces méthodologies ont permis de formaliser les étapes préliminaires du développement de ces systèmes afin de répondre aux besoins applicatifs de l'utilisateur et de limiter les anomalies de conception dans la base de données cible. Dans le but de proposer une approche dédiée à la conception des $\mathcal{BDBO}$, nous avons proposé dans les chapitres précédents l'exploitation des relations de dépendance entre les concepts ontologiques durant le processus de conception. Nous avons proposé une approche intégrant à la fois les dépendances définies sur les classes ontologiques et celles définies sur les propriétés des données pour la définition d'un modèle logique de données en troisième forme normale. Dans cette approche, contrairement à la phase de modélisation logique, la phase de modélisation conceptuelle a été brièvement présentée et aucun mécanisme spécifique au traitement de la subsomption des classes ontologiques n'a été proposé. En effet, en se basant sur la définition des classes ontologiques, un mécanisme de placement de classes dans la hiérarchie de subsomption peut être déduit. Celui-ci va permettre d'améliorer davantage la description du modèle conceptuel de données et par la suite de concevoir une base de données plus performante. De plus, vue la diversité des architectures des $\mathcal{BDBO}$ et la variété de leurs modèles de stockage, plusieurs options de déploiement se présentent. La variété de ces options rend la phase de déploiement complexe. Dans le but de maîtriser le processus de déploiement de $\mathcal{BDBO}$, nous proposons d'abord à le formaliser, ensuite étudier sa complexité, finalement proposer des solutions de déploiement. Afin de mener ces améliorations, nous proposons dans un premier temps de détailler davantage notre méthodologie dédiée à la conception des bases de données à base ontologique en mettant l'accent sur la phase de modélisation conceptuelle et le processus de placement des classes ontologiques dans la hiérarchie de subsomption. Dans un deuxième temps, nous étudions la complexité de la phase de déploiement et nous proposons une approche de déploiement de notre méthodologie dans les $\mathcal{BDBO}$. Nous présentons une étude de cas dans un environnement de bases de données à base de modèles de Type 3 sous OntoDB. Finalement, une étude comparative entre notre approche et deux approches étudiées dans l'état de l'art est proposée.

2 Méthodologie de conception de $\mathcal{BDBO}$

Dans cette section, nous présentons d'abord une description générale de notre méthodologie de conception de bases de données à base ontologique. Ensuite, nous décrivons en détail les

différentes étapes de notre approche et en particulier la phase de modélisation conceptuelle de données et le processus de placement des classes ontologiques.

2.1 Description de notre approche

Notre méthodologie consiste a générer une $\mathcal{BDBO}$ en troisième forme normale (3FN) à partir d'une ontologie conceptuelle partagée et enrichie par les dépendances entre les classes (CD) et les dépendances fonctionnelles définies sur les propriétés de chaque classe (PFD). Dans notre étude, nous nous sommes intéressés aux ontologies définies avec le modèle d'ontologie OWL. La première étape de notre approche consiste à choisir l'ontologie globale décrivant le domaine de l'application traitée (étape 1). Afin de répondre aux besoins du cahier de charges, le concepteur procède à la construction de l'ontologie locale à partir de l'ontologie de domaine choisie (étape 2). Ensuite, nous procédons à l'identification du type des classes: canonique ou non canonique (étape 3). Une fois le typage de classes est établi, deux nouvelles étapes sont lancées en parallèle. L'étape 5 consiste à placer les classes non canoniques dans leur hiérarchie de subsomption tandis que l'étape 4 permet la génération du modèle logique normalisé de la base de données. Enfin, la structure de données est physiquement créée dans la $\mathcal{BDBO}$ cible (étape 6). La figure 6.1 résume ces différentes étapes.

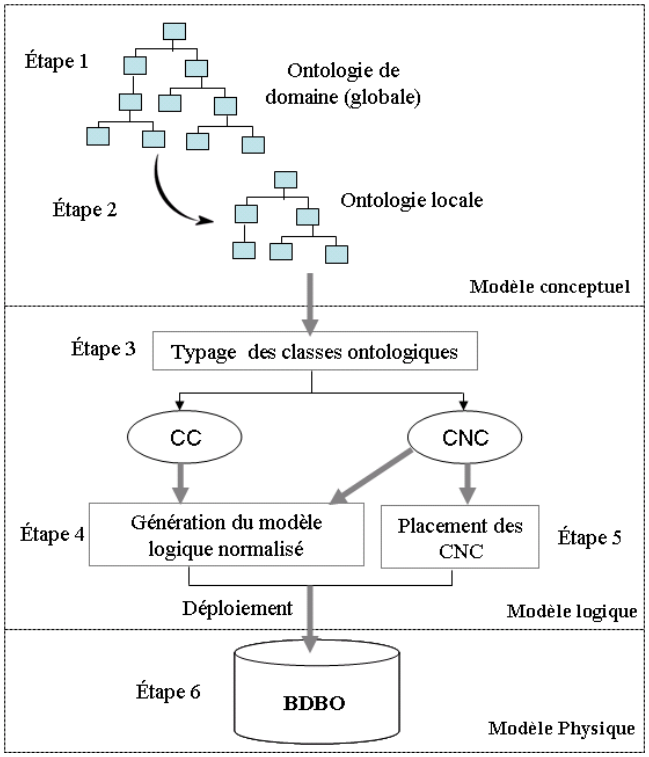

FIGURE 6.1 – Approche de conception des bases de données à base ontologique.

2.2 Étapes de notre approche

Notre approche de conception de $\mathcal{BDBO}$ repose sur six principales étapes: (1) choix de l'ontologie de domaine, (2) construction de l'ontologie locale, (3) identification des classes canoniques et non canoniques, (4) organisation des classes dans la hiérarchie de subsomption, (5) génération du modèle logique normalisé et (6) modélisation physique.

2.2.1 Étape 1 : Choix de l'ontologie globale.

Les méthodologies de conception de bases de données exigent du concepteur à la fois, (1) une connaissance du domaine modélisé et (2) une maîtrise des techniques de modélisation. Le premier aspect nécessite un grand effort du concepteur quant à la collecte des données relatives au domaine de l'application. Celui-ci doit maîtriser le domaine étudié en rassemblant toutes les données le décrivant. Afin de lui faciliter la tâche, nous proposons la réutilisation de la connaissance du domaine formalisée à travers l'ontologie partagée OG. Celle-ci est identifiée à partir du domaine de l'application déduit à partir des besoins du cahier des charges. Notons que la conception de la base de données à partir de l'ontologie de domaine (a) permet au concepteur de réutiliser la connaissance du domaine formalisée dans l'ontologie, (b) facilite l'échange de données, (c) permet l'intégration automatique et (d) offre un accès aux données au niveau connaissance (niveau ontologique) [Fankam et al., 2009].

2.2.2 Étape 2 : Construction de l'ontologie locale.

Après avoir identifié l'ontologie de domaine, le concepteur a pour tâche de définir une ontologie dite locale définie par (i) spécialisation d'une ontologie partagée de domaine et par (ii) importation sélective de propriétés et (iii) extension éventuelle de celle-ci par les concepts ontologiques nécessaires qui n'y figuraient pas. Ainsi, en se basant sur les exigences du cahier des charges, trois scénarii d'extraction sont possibles.

1. $LO = OG$ signifie que OG couvre tous les besoins du concepteur. L'expression des besoins en termes de classes, de propriétés ainsi que des relations de subsomption coïncide exactement avec la description de l'ontologie globale. Autrement dit, la description de l'ontologie locale correspond exactement à la description de l'ontologie globale. Dans ce cas, les concepts ontologiques locaux sont égaux aux concepts ontologiques globaux. Ainsi, en se basant sur le modèle formel de l'ontologie $OG = < C, \mathcal{P}, Applic, Sub, PFD, CD >$ (voir Chapitre 53.1, $LO = OG$ signifie que:
   – $C^{LO} = C^{OG}$,
   – $P^{LO} = P^{OG}$,
   – $\forall\ c_i \in C^{LO}, C^{OG}$; Applic$(c_i^{LO})$ = Applic(c_i^{OG}),
   – $\forall\ c_i \in C^{LO}, C^{OG}$; Sub$(c_i^{LO})$ = Sub(c_i^{OG}),
   – $\forall\ c_i \in C^{LO}, C^{OG}$; $PFD^{c_i^{LO}} = PFD^{c_i^{OG}}$,

– $CD^{LO} = CD^{OG}$.

2. $LO \subset OG$ signifie que OG est largement suffisante pour couvrir les besoins de l'utilisateur. En effet, la description des besoins en termes de concepts ontologiques décrit un fragment de l'ontologie globale. La construction de LO est réalisée à l'aide de l'opérateur $OntoSub$. Cette relation est un opérateur d'articulation permettant de connecter LO à OG tout en offrant une grande indépendance de l'ontologie locale. Grâce à cette relation, une classe locale peut importer une partie ou la totalité des propriétés définies dans les classes référencées. Ainsi, $LO \subset OG$ signifie que:
   – $C^{LO} \subseteq C^{OG}$,
   – $P^{LO} \subseteq P^{OG}$,
   – $\forall\ c_i \in C^{LO}$,C^{OG}, $\text{Applic}(c_i^{LO}) \subseteq \text{Applic}(c_i^{OG})$,
   – $\forall\ c_i \in C^{LO}$,C^{OG}, $\text{Sub}(c_i^{LO}) \subseteq \text{Sub}(c_i^{OG})$,
   – $\forall\ c_i \in C^{LO}$,C^{OG}, $PFD^{c_i^{LO}} \subseteq PFD^{c_i^{OG}}$,
   – $CD^{LO} \subseteq CD^{OG}$.
   – au moins une des six inclusions précédentes est une inclusion stricte ($\subset$).
3. $LO \supset OG$ signifie que l'ontologie OG ne remplit pas toutes les exigences du concepteur. Dans ce cas, celui-ci extrait de l'ontologie OG, un fragment correspondant à ses exigences et l'enrichit localement en y ajoutant de nouveaux concepts ontologiques. Notons que les concepts ajoutés peuvent englober :
   – les classes : $\exists\ c_i \in C^{LO}$ tel que $c_i \notin C^{OG}$;
   – les propriétés : $\exists\ p_i \in P^{LO}$ tel que $p_i \notin P^{OG}$;
   – la relation de subsomption ;
   – les dépendances entre les propriétés: $\exists\ PFD_J^{c_i^{LO}} \in PFD^{c_i^{LO}}$ tel que $PFD_J^{c_i^{LO}} \notin PFD^{c_i^{OG}}$;
   – les dépendances entre les classes ontologiques: $\exists\ CD_i \in CD^{LO}$ tel que $CD_i \notin CD^{OG}$.
   Pour le cas des dépendances entre les propriétés des données, elles peuvent être définies à la fois sur les propriétés importées et celles ajoutées. De la même manière, les dépendances entre les classes peuvent inclurent les classes importées et celles rajoutées.

2.2.3 Étape 3 : Identification des classes canoniques et non canoniques

Une fois l'ontologie locale définie, le concepteur exploite les dépendances définies sur les classes ontologiques, invoquées par l'algorithme de canonicité (voir Chapitre 5 algorithme 8) afin d'identifier l'ensemble des classes canoniques et non canoniques. Notons que cette étape est détaillée dans le Chapitre5.4.1.

2.2.4 Étape 4 : Organisation des classes dans la hiérarchie de subsomption

Les classes de l'ontologie peuvent être définies sans spécifier la relation de subsomption. Dans ce cas, la hiérarchie des classes canoniques et non canoniques n'est pas correctement définie. Dans le but de compléter la description de cette hiérarchie, nous proposons durant cette

étape d'exploiter les raisonneurs existants (Pellet, Racer, etc.) pour établir le placement des classes ontologiques. En effet, le mécanisme de placement se traduit essentiellement à partir des définitions de classes. Par conséquent, nous nous intéressons dans notre approche aux classes dérivées décrites à travers les opérateurs ensemblistes, les restrictions de valeurs et celles de cardinalité. Nous proposons d'étudier le mécanisme de placement pour les différents types de classes définies et ce à travers la définition de règles de placement correspondantes à chacun de ces types. Dans ce qui suit, notons que C_i^{LO}, C_j^{LO}, C_k^{LO} et C_n^{LO} sont des classes de l'ontologie locale *LO*.

Règle 6 : Règle de placement pour le constructeur Union

si $C_k^{LO} \equiv C_i^{LO} \cup C_j^{LO} \cup .. \cup C_n^{LO}$ **alors**
C_k^{LO} is superclassOf $C_i^{LO}, C_j^{LO}, .., C_n^{LO}$;
fin

1. **Opérateur d'union.** Supposons que C_k^{LO} est définie comme étant l'union de C_i^{LO}, C_j^{LO}.. C_n^{LO} ($C_k^{LO} \equiv C_i^{LO} \cup C_j^{LO} \cup .. \cup C_n^{LO}$). En se basant sur cette définition, la hiérarchie de ces classes peut être déduite tel que C_k^{LO} est une super-classe de C_i^{LO}, C_j^{LO},.., C_n^{LO}. La règle 6 résume la règle de placement correspondante à l'opérateur d'union.

   Prenons l'exemple de la classe décrivant des universités (*University*) définie comme étant l'union des universités privées (*PrivateUniversity*) et des universités publiques (*PublicUniversity*). En appliquant la règle proposée, nous constatons que la classe *University* est une superclasse de *PrivateUniversity* et *PublicUniversity*.

2. **Opérateur d'intersection.** Supposons que la classe C_k^{LO} est définie tel que l'intersection de C_i^{LO}, C_j^{LO}.. C_n^{LO} ($C_k^{LO} \equiv C_i^{LO} \cap C_j^{LO} \cap .. \cap C_n^{LO}$). En se basant sur cette définition, la hiérarchie de ces classes peut être déduite. C_k^{LO} est une sous-classe de C_i^{LO}, C_j^{LO},.., C_n^{LO}. La règle 7 résume la règle de placement correspondante à l'opérateur ensembliste traduisant l'opérateur d'intersection. Considérons l'exemple de la classe des étudiant travailleurs (*Student_Employee*) définie comme étant l'intersection des classes *Student* et *Employee*. En se basant sur la règle définie, la hiérarchie de ces trois classes est déduite : *Student_Employee* est sous-classe de *Student* et *Employee*.

Règle 7 : Règle de placement pour le constructeur Intersection

si $C_k^{LO} \equiv C_i^{LO} \cap C_j^{LO} \cap .. \cap C_n^{LO}$ **alors**
C_k^{LO} is subclassOf $C_i^{LO}, C_j^{LO}, .., C_n^{LO}$;
fin

3. **Restriction de valeur.** Considérons que la classe C_k^{LO} est décrite à travers une restriction de valeur définie sur la propriété P_i^{LO}. Supposons que cette propriété a pour domaine la classe C_j^{LO}. En se basant sur cette définition, une relation de subsomption relative aux

classes C_k^{LO} et C_j^{LO} est établi. Celle-ci définit que C_k^{LO} est sous-classe de C_j^{LO}. La règle 8 décrit le placement relatif à une classe dérivée à travers une restriction de valeur.

Considérons l'exemple de la classe *Student* et supposons qu'elle est définie comme étant le domaine de la propriété *level* décrivant le niveau d'études des étudiants. Admettons que la classe *MasterStudent* décrivant l'ensemble des étudiants inscrits en master est définie comme étant la restriction sur la propriété *level* à la valeur master. En se basant sur la règle proposée, nous constatons que la classe *MasterStudent* est une sous-classe de *Student*

Règle 8 : Règle de placement pour le constructeur hasValue

si $C_k^{LO} \equiv \exists\ P_i^{LO}.\{x\}$ *and* $Domain(P_i^{LO}) = C_j^{LO}$ **alors**
  C_k^{LO} is subclassOf C_j^{LO} ;
fin

4. **Restriction de cardinalité.** Considérons que la classe C_k^{LO} est décrite à travers une restriction de cardinalité définie sur la propriété d'objet P_i^{LO}. Supposons que cette propriété a pour domaine la classe C_i^{LO} et que la cardinalité traitée est une cardinalité exacte ou maximale de valeur 1. En se basant sur cette définition, une relation de subsomption relative aux classes C_i^{LO} et C_k^{LO} est établie. Celle-ci définit que C_k^{LO} est sous-classe de C_i^{LO}. Les règles 9 et 10 résument les règles de placement relatives à la définition d'une classe définie à travers une restriction de cardinalité.

   Prenons l'exemple de la classe $VisitingProfessor$ décrivant les professeurs donnant exactement un seul cours à l'université et définie comme étant la restriction de la cardinalité sur la propriété *Give* pour une valeur égale à 1. Notons que cette propriété décrit l'action de donner des cours et qu'elle a pour domaine la classe des professeurs $Professor$. En appliquant la règle 9, nous constatons que $VisitingProfessor$ est une sous-classe de $Professor$.

Règle 9 : Règle de placement traduisant une restriction de cardinalité exacte

si $C_k^{LO} \equiv (n)\ P_i^{LO} \mid n = 1$; $Domain(P_i^{LO}) = C_i^{LO}$ **alors**
  C_k^{LO} is subclassOf C_i^{LO} ;
fin

Règle 10 : Règle de placement traduisant une restriction de cardinalité maximale

si $C_k^{LO} \equiv (\leq n)\ P_i^{LO} \mid n = 1$; $Domain(P_i^{LO}) = C_i^{LO}$ **alors**
  C_i^{LO} is superclassOf C_k^{LO} ;
fin

2.2.5 Étape 5 : Génération du modèle logique de données.

La génération du modèle logique de données normalisé comporte à la fois (i) la génération des schémas de tables normalisées définies sur les classes canoniques, (ii) le schéma de vues relationnelles définies sur les classes canoniques et (iii) le schéma de vues de classes correspondantes aux classes non canoniques. Cette étape est détaillée dans le Chapitre5.4.2.

2.2.6 Étape 6: Modélisation physique.

Après avoir défini le schéma logique de données, une nouvelle étape est établie. Elle définit le déploiement du schéma logique de données dans la base de données cible. Autrement dit, cette étape consiste en la création des structure de données à travers la création physique des tables normalisées, des vues relationnelles et des vues de classes. Dans la section suivante, une approche de déploiement dans les bases de données à base de modèles est proposée.

3 Approche de déploiement dans les bases de données à base de modèles

Le déploiement d'une base de données peut être défini comme le processus allant de l'acquisition du SGBD à son exécution. Cette phase constitue une étape importante du cycle de vie de conception d'une base de données. Dans les bases de données traditionnelles, le déploiement concerne en général le choix du SGBD stockant la base de données (Oracle, SQL Server, MySQL, etc.). Souvent, ces systèmes se caractérisent par une même architecture où deux composantes sont définies: (i) une partie méta-base contenant le catalogue et (ii) une partie données stockant les instances. Une deuxième caractéristique de ces derniers est qu'ils utilisent le même modèle de stockage, où la structure tabulaire est utilisée pour stocker toute instance. Avec l'arrivée des bases de données orientées objet et XML, un nombre important de travaux a été proposé pour définir de nouveaux modèles de stockage adéquats [Tian et al., 2002].

D'autres modèles ont vu le jour avec l'apparition des $\mathcal{BDBO}$ (table de triplet, représentation horizontale, représentation hybride). Ces dernières ont également contribué à la naissance de nouvelles architectures de SGBD (type 1, type 2 et type 3). La combinaison des deux dimensions liées aux modèles de stockage et aux architectures entraîne une grande diversité de systèmes proposés ce qui rend le processus de déploiement de $\mathcal{BDBO}$ plus complexe.

Pour résumer, nous pouvons dire que le déploiement dans le cycle de vie de conception de bases de données traditionnelles se fait au niveau de SGBD global tout en encapsulant ses composantes, tandis que le déploiement dans les $\mathcal{BDBO}$ se déroule de manière détaillée, où toutes les composantes du SGBD (architecture, modèles de stockage) sont examinées. La figure 6.2 décrit le processus de déploiement dans les deux types de bases de données.

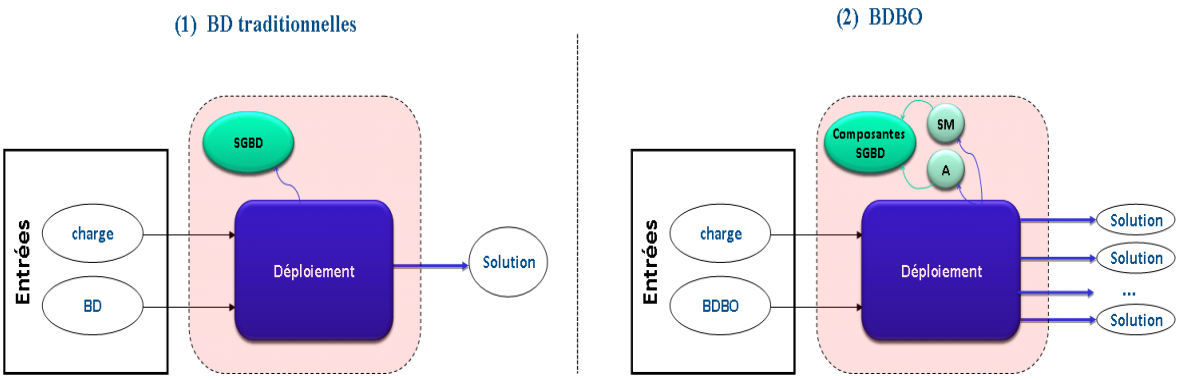

FIGURE 6.2 – Processus de déploiement dans les bases de données

En examinant les travaux sur la conception des $\mathcal{BDBO}$, nous remarquons que le processus de déploiement est effectué d'une manière isolée en ignorant les différentes phases de conception. De nos jours, le déploiement des ontologies dans les $\mathcal{BDBO}$ existantes telles que Rdfsuite [Alexaki et al., 2001b], Jena [Carroll et al., 2004], OntoDB [Jean et al., 2007], Sesame [Broekstra et al., 2002], Owlgres [Stocker and Smith, 2008], SOR [Lu et al., 2007], Oracle [Das et al., 2004], etc. se fait d'une manière statique. Autrement dit, le schéma logique des $\mathcal{BDBO}$ est figé en dépit de la modélisation conceptuelle et logique de données. La figure 6.3.(1) traduit le processus de déploiement des ontologies et de leurs instances dans les $\mathcal{BDBO}$ existantes.

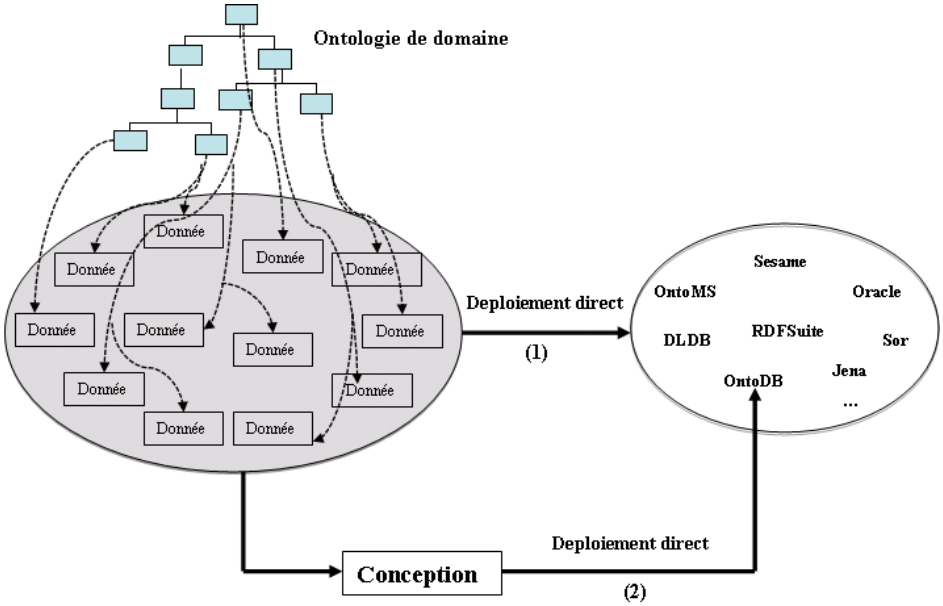

FIGURE 6.3 – Déploiement statique d'une ontologie.

Dans la section précédente, nous avons proposé une méthodologie de conception de $\mathcal{BDBO}$.

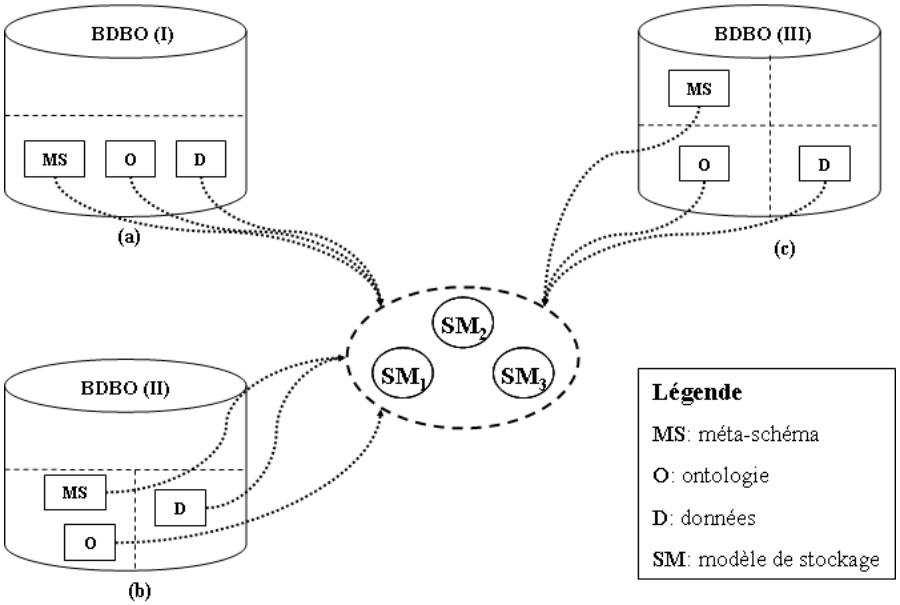

FIGURE 6.4 – Combinaisons de déploiement de notre approche dans les $\mathcal{BDBO}$

Cependant, aucun processus de déploiement de la méthodologie proposée n'a été présenté. Dans le but de répondre à ce besoin et de compléter le cycle de vie des $\mathcal{BDBO}$, nous proposons dans cette section d'étudier le problème de déploiement des $\mathcal{BDBO}$. Cette approche a pour objectif d'offrir à la personne en charge de cette tâche un ensemble de modèles de déploiement lui facilitant le choix de la solution la plus pertinente pour lui. Dans un premier temps, nous proposons l'étude de la complexité de la phase de déploiement. Dans un deuxième temps, nous étudions le problème de déploiement dans les $\mathcal{BDBO}$ en proposant une approche de déploiement supposant l'existence des phases conceptuelle et logique.

3.1 Étude de la complexité

Vu la diversité des architectures de $\mathcal{BDBO}$ et la variété des modèles de stockage [Pan and Heflin, 2003] utilisés pour les différentes composantes des base de données, nous relevons la difficulté du processus de déploiement dans les $\mathcal{BDBO}$. En effet, la variété des architectures de bases de données et des modèles de stockage rendent le processus de déploiement plus complexe. Pour illustrer cette difficulté, nous proposons d'étudier la complexité des options de déploiement. Soient :

- $\mathcal{A} = \{A_1, A_2, .., A_n\}$ l'ensemble de toutes les architectures possibles ;
- $\mathcal{SM} = \{SM_1, SM_2, .., SM_m\}$ l'ensemble des modèles de stockage existants ;
- $\mathcal{T} = \{Type_1, Type_2, .., Type_p\}$ l'ensemble des types de bases de données existantes. Étant donné que les architectures de bases de données peuvent être implémentées en utilisant différents modèles de stockage, nous définissons un type de base de données $Type_i$

comme étant un couple composé d'une architecture A_i choisie et un ensemble de modèles de stockage $E^{SM_j} \subseteq \mathcal{SM}$ tel que suit : $Type_i =< A_i, E^{SM_j} >$;

– $Part = \{Part_1, Part_2, .., Part_n\}$ l'ensemble des parties nécessaires pour le déploiement de notre approche.

Ainsi, le nombre de possibilités de déploiement $\mathcal{D}$ dont le concepteur doit tenir compte est donné par l'équation suivante :

$$D = card(\mathcal{A}) \times (card(\mathcal{SM}))^{card(Part)} \quad (1)$$

Dans le chapitre 2.3.3, nous avons identifié trois principaux modèles de stockage (table de triplet (SM_1), représentation horizontale (SM_2) et représentation hybride (SM_3)) et trois architectures majeures de bases de données (bases de données à deux composantes, bases de données à trois composantes et bases de données à quatre composantes). Rappelons que le déploiement de notre approche nécessite la prise en compte de trois principales parties (card(Part)=3) : méta-schéma ($Part_1$), ontologie ($Part_2$) et données ($Part_3$). En se basant sur les architectures et les modèles de stockage identifiés ainsi que les parties nécessaires pour le déploiement de notre approche, la complexité du processus de déploiement est évaluée à $(3 \times \left(3^3\right))$. La figure 6.4 illustre les différentes combinaisons de déploiement de notre approche dans les bases de données à base de modèles. En effet, pour chaque architecture de $\mathcal{BDBO}$, trois modèles de stockage peuvent être choisis pour le déploiement sur chacune des trois parties méta-schéma, ontologie et données. Par conséquent, l'étude de 81 cas devient une tâche difficile. Dans le but de simplifier une telle tâche, nous proposons d'étudier le problème de déploiement dans la section suivante.

3.2 Problème de déploiement d'une $\mathcal{BDBO}$

La complexité de déploiement des $\mathcal{BDBO}$ nous a permis d'identifier un nouveau problème dans le cycle de base de données, à savoir le problème de déploiement de $\mathcal{BDBO}$. Ce problème peut être formalisé comme suit. Étant donné :

1. une $\mathcal{BDBO}$ stockant à la fois les instances et leur ontologie et,
2. une charge de requêtes Q (identifiée lors du processus d'identification de besoins).

Le problème de déploiement consiste à trouver un SGBD avec ses composantes permettant d'optimiser Q. Pour résoudre ce problème, nous proposons deux solutions : (1) une solution exhaustive et (2) une solution dirigée par le déployeur.

3.2.1 Solution exhaustive

Comme cela est décrit dans la figure 6.6.(1), cette approche consiste à définir toutes les options de déploiement en considérant les différentes combinaisons possibles des deux composantes : architecture et modèles de stockage. A l'aide des modèles de coût estimant le coût

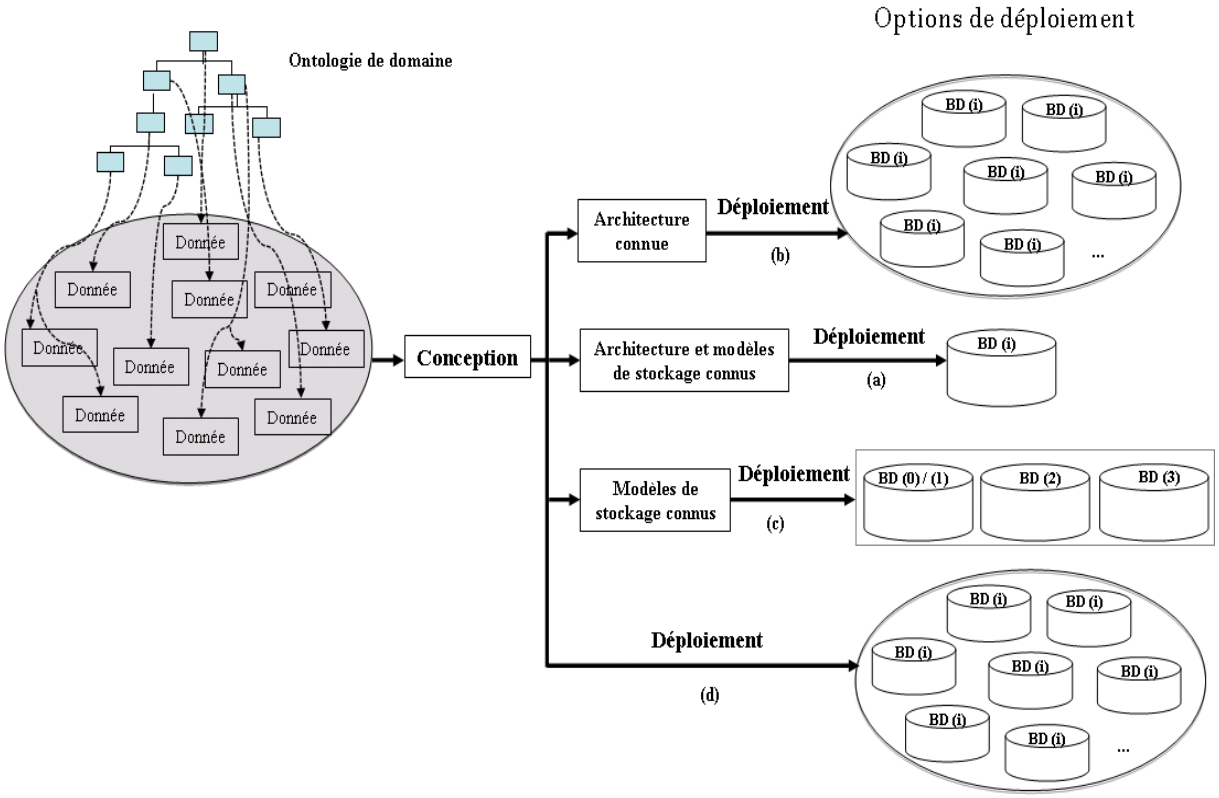

FIGURE 6.5 – Déploiement dans les bases de données à base de modèles.

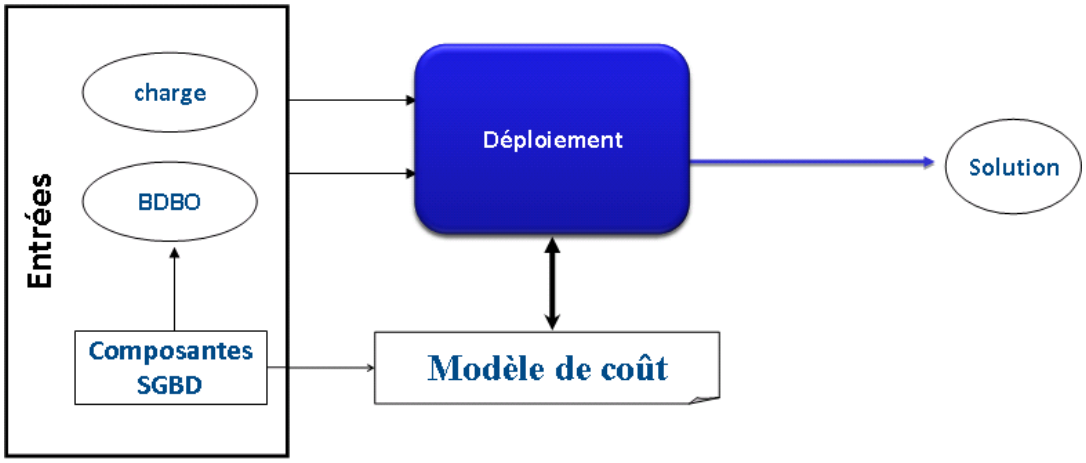

FIGURE 6.6 – Solution exhaustive

d'exécution de requêtes selon chaque option [Mbaiossoum et al., 2013], nous pouvons facilement déterminer le schéma de déploiement ayant le coût minimal. Cette solution reste coûteuse. Afin de réduire sa complexité, nous proposons dans la section suivante une solution dirigée par le déployeur. L'idée dernière cette solution est de faire participer ce dernier dans le processus de sélection du schéma de déploiement.

3.2.2 Approche dirigée par le déployeur

Dans cette approche, le déployeur choisit dans un premier temps une ou plusieurs composantes du SGBD selon ses compétences. Il peut figer l'architecture et/ou les modèles de stockage. Par conséquent, un nombre réduit d'options de déploiement peut être obtenu. Dans un deuxième temps, à l'aide des modèles de coût [Mbaiossoum et al., 2013], le schéma de déploiement ayant un coût minimal est obtenu. La figure 6.7 illustre l'approche proposée.

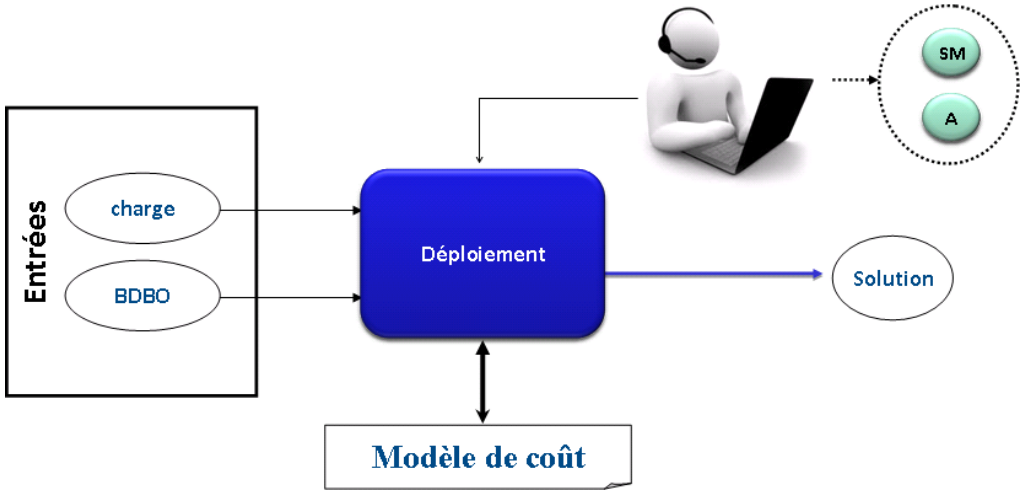

FIGURE 6.7 – Approche de déploiement dirigée par le concepteur

Pour illustrer la solution dirigée par le déployeur, considérons le scénario suivant. Supposons que le concepteur exprime ses préférences quant aux modèles de stockage utilisés durant le processus de déploiement. Ainsi, les structures de données dédiées au stockage des différentes données correspondantes aux trois niveaux (méta-schéma, ontologie et données) sont choisies au préalable par le concepteur. Par contre, aucun détail n'est donné concernant l'architecture de la base de données cible. Dans ce cas, trois solutions peuvent être établies : l'architecture de la base choisie repose sur (a) deux ou (b) trois ou (c) quatre composantes. Par conséquent, la complexité d'une telle approche est égale à $(3 \times (1^3))$. La figure 6.5.(c) résume ce scénario où le concepteur choisit les modèles de stockage utilisés dans la base de données cible. Afin d'aider le déployeur dans son choix final, un modèle de coût est appliqué pour chaque type d'architecture [Mbaiossoum et al., 2013] comme suit :

– architecture Type 1 : coût(Q, $\mathcal{BDBO}$) = coût_accès(méta-base) + coût_accès(données) ;

- architecture Type 2 : coût(Q, $\mathcal{BDBO}$) = coût_accès(méta-base) + coût_accès(schéma de l'ontologie) + coût_accès(données) ;
- architecture Type 3 : coût(Q, $\mathcal{BDBO}$) = coût_accès(méta-base) + coût_accès(méta-schéma de l'ontologie) coût_accès(schéma de l'ontologie) + coût_accès(données).

Ainsi, une solution est favorisée parmi seulement trois options de déploiement.

Un deuxième scénario serait de fixer l'architecture de la base ainsi que les modèles de stockage dédiés à chacune de ses parties. Dans ce cas, en se basant sur ses connaissances portant sur les $\mathcal{BDBO}$ existantes, le concepteur peut choisir une plate-forme de base de données existante, selon ses préférences, pour le déploiement de la méthodologie proposée. Par conséquent, la complexité de cette approche est évaluée à $(1 \times \left(1^3\right))$. Un tel scénario est étudié en détail dans la section 4.

4 Étude de cas

Dans cette section, nous proposons une étude de cas détaillée traduisant l'application de notre approche sous une base de données à base ontologique de type 3 nommée OntoDB. Afin de mener notre étude, nous proposons la séquence d'étapes suivantes : (1) extension du méta-schéma pour le support des dépendances entre les concepts ontologiques, (2) choix de l'ontologie de domaine, (3) définition de l'ontologie locale, (4) typage de classes, (5) placement des classes, (6) génération du schéma logique de données et (7) génération du modèle physique de données.

4.1 Extension du méta-schéma.

Notre approche a pour entrée une ontologie enrichie à la fois par les dépendances entre les propriétés des données et les classes ontologiques. Afin de supporter une telle structure, le modèle d'ontologies d'OntoDB doit être étendu tel que décrit dans le Chapitre 5.section5.2. Notons que cette extension concerne les dépendances ontologiques et la canonicité des classes.

4.2 Choix de l'ontologie de domaine.

Supposons que l'ontologie de domaine choisie est celle des universités . Supposons l'existence du besoin de concevoir une base de données à base ontologique gérant le domaine universitaire. Pour couvrir le domaine de l'application, nous proposons d'adopter l'ontologie Lehigh University Benchmark (LUBM) comme ontologie globale. Les requêtes OntoQL ci-dessus décrivent la création d'un fragment d'une telle ontologie dans la $\mathcal{BDBO}$ OntoDB.

```
CREATE #Class Person(DESCRIPTOR (#name[en] = 'Person')
```

```
       PROPERTIES(NSS INT, Name STRING, Age INT))
CREATE #Class Student under Person (DESCRIPTOR (#name[en] = 'Student')
       PROPERTIES(NSS INT, Name STRING, Age INT, level STRING))
CREATE #Class University(DESCRIPTOR (#name[en] = 'University')
       PROPERTIES(IdUniv INT, NameUniv STRING, City STRING,
                    Region STRING, Country STRING))
CREATE #Class Course(DESCRIPTOR (#name[en] = 'Course')
               PROPERTIES(IdCourse INT, Course_Name STRING,
                    Duration STRING,Course_level STRING))
...
```

4.3 Définition de l'ontologie locale.

Après avoir identifié l'ontologie de domaine, nous procédons à la construction de l'ontologie locale. En se référant aux besoins du cahier de charges, nous remarquons que l'ontologie de domaine ne remplit pas toutes les exigences du concepteur ($LO \supset O$). Dans un premier temps, nous définissons un fragment initial (F_i) de l'ontologie par importation d'un ensemble de classes couvrant une partie des exigences de l'utilisateur et une partie ou la totalité des propriétés définies dans les classes importées. Dans un deuxième temps, les besoins non couverts sont traités par l'ajout de nouveaux concepts. Dans le but de répondre aux exigences du cahier de charges, F_i est ainsi enrichi par les classes, les propriétés et les dépendances manquantes. Par exemple, le concepts décrivant les étudiants inscrits en master ($MasterStudent$) n'est pas défini à travers l'ontologie globale ($MasterStudent \notin C^{LUBM^G}$ où C^{LUBM^G} représente les classes de l'ontologie de domaine). Par conséquent, cette classe est ajoutée au niveau de l'ontologie locale par la requêtes OntoQL suivante :

```
CREATE #Class MasterStudent (DESCRIPTOR (#name[en] = 'MasterStudent')
       PROPERTIES(NSS INT, Name STRING, Age INT, level STRING))
```

De plus, la définition de la classe des universités $University$ est modifiée comme étant l'union des universités privées et publiques ($University \equiv PublicUniversity \cup PrivateUniversity$). La figure 6.8 représente l'ensemble des classes ontologiques locales importées à partir de l'ontologie de domaine et celles ajoutées par le concepteur. La description de l'ontologie locale est illustrée par la figure 5.7.

4.4 Typage des classes ontologiques.

Une fois l'ontologie locale définie, nous procédons à l'application de l'algorithme de canonicité afin d'identifier l'ensemble des classes ontologiques canoniques et non canoniques. Notons que cette étape est détaillée dans le Chapitre 5.3 et que la solution identifiée est la

Classes locales	Classes importées	Classes ajoutées
University	X	
PublicUniversity		X
PrivateUniversity		X
Person	X	
Employee	X	
Student_Employee		X
Student	X	
Professor	X	
Course	X	
DoctoralCourse		X
MasterCourse		X
Chair	X	
MasterStudent		X
MasterCourse		X
FullPtrofessor		X

FIGURE 6.8 – Définition des classes de l'ontologie locale.

suivante: CC^1= { *Chair*, *Person*, *Student*, *Professor*, *Employee*, *PrivateUniversity*, *Course*, *PublicUniversity* } et CNC^1= {*University*, *MasterStudent*, *MasterCourse*, *DoctoralCourse*, *FullProfessor*, *Student_Employee* } où CC^1 et CNC^1 représentent respectivement les classes canoniques et les classes non canoniques de l'ontologie locale. Les requêtes OntoQL ci-dessous décrivent un exemple de typage des classes ontologiques.

```
INSERT INTO #Canonic (#oid) VALUES
(SELECT #oid from #Class c WHERE c.#name='Course')
INSERT INTO #Canonic (#oid) VALUES
(SELECT #oid from #Class c WHERE c.#name='Person')
INSERT INTO #Canonic (#oid) VALUES
(SELECT #oid from #Class c WHERE c.#name='Student')
INSERT INTO #Canonic (#oid) VALUES
(SELECT #oid from #Class c WHERE c.#name='PrivateUniversity')
INSERT INTO #NonCanonic (#oid) VALUES
(SELECT #oid from #Class c WHERE c.#name='MasterStudent')
INSERT INTO #NonCanonic (#oid) VALUES
(SELECT #oid from #Class c WHERE c.#name='University')
INSERT INTO #NonCanonic (#oid) VALUES
(SELECT #oid from #Class c WHERE c.#name='MasterStudent')
INSERT INTO #NonCanonic (#oid) VALUES
```

```
(SELECT #oid from #Class c WHERE c.#name='MasterCourse')
...
```

4.5 Processus de placement.

Afin d'assurer le processus de placement des classes ontologiques, nous nous somme référés à l'utilisation du raisonneur Pellet 1.5.2 [Bijan, 2004]. Celui-ci raisonne sur les définitions des classes ontologiques et en particulier sur les classes dérivées pour leur organisation dans la hiérarchie de subsomption. Reprenons l'exemple de la classe *University* décrite comme étant l'union des classes *PublicUniversity* et *PrivateUniversity*. En s'appuyant sur le raisonnement établi par Pellet, une nouvelle hiérarchie de subsomption est inférée (voir figure 6.9) comme suit: *PublicUniversity*, *PrivateUniversity* $\subset$ *University*. Les requêtes OntoQL ci-dessous permettent de persister la classe des universités publiques *PublicUniversity* dans sa hiérarchie de subsomption inférée ($\subset$ *University*).

```
CREATE #Class PublicUniversity under University
       (DESCRIPTOR (#name[en] = 'PublicUniversity')
        PROPERTIES(IdUniv INT, Name STRING, City STRING,
                           Region STRING, Country STRING))
```

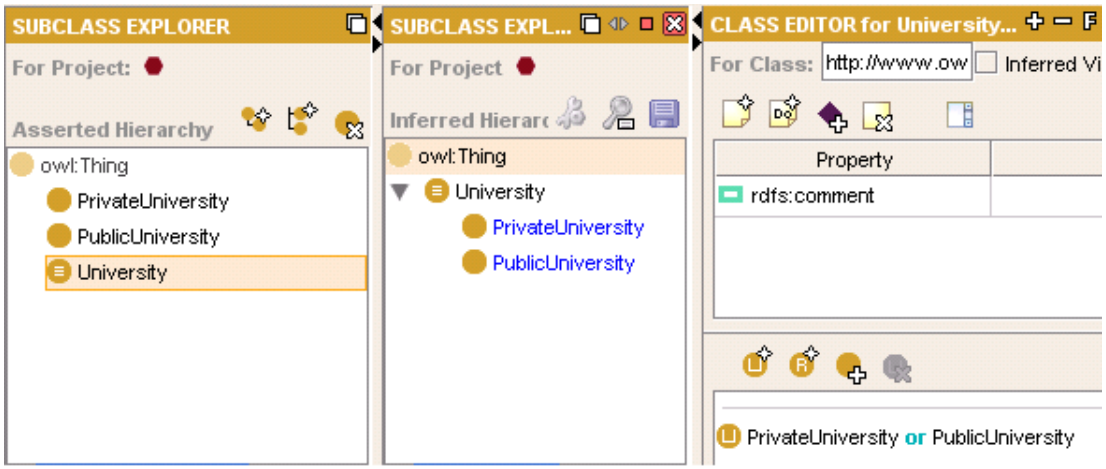

FIGURE 6.9 – Exemple de placement de classes à l'aide du raisonneur Pellet.

La figure 6.10 décrit un exemple de persistance de l'ontologie locale LUBM après l'application du processus de placement. Notons que la table *subclassOf* stocke la hiérarchie des classes ontologiques.

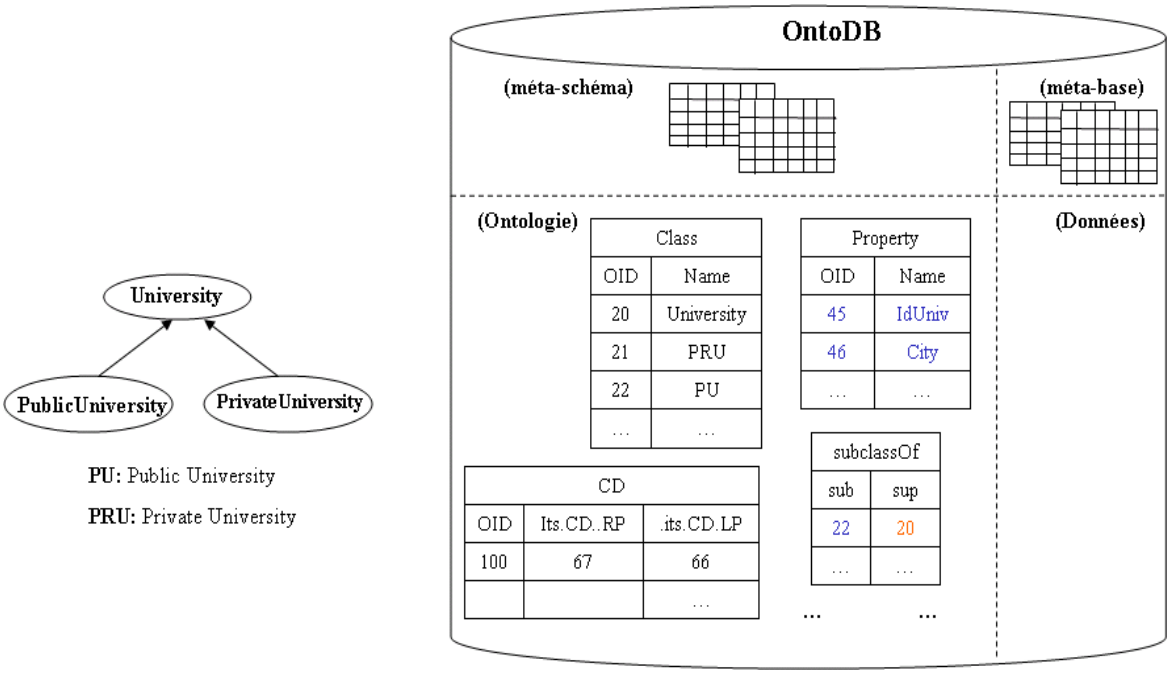

FIGURE 6.10 – Persistance de l'ontologie locale après le processus de placement

4.6 Génération du schéma logique de données.

Durant l'application de notre approche sur l'ontologie LUBM, nous avons classé les classes ontologiques selon leur canonicité. Pour chaque type de classe, nous proposons un traitement spécifique tel que décrit dans la section 2.2.5. Par exemple, en se référant aux CC et CNC identifiées, nous proposons de traiter le cas des classes *Course* et *DoctoralCourse* décrivant respectivement l'ensemble des cours universitaires et la totalité des cours doctoraux.

Étant donnée que *Course* est une classe canonique, nous procédons à sa normalisation en se basant sur les dépendances fonctionnelles définies sur ses propriétés des données Id_{Course}, *Course_Name*, *Duration* et *Course_level* décrivant respectivement l'identifiant d'un cours, son intitulé, sa durée et son niveau d'enseignement. En exécutant la requête OntoQL ci-dessous, les structures de données associées à cette classe sont générées.

```
CREATE ALLEXTENT OF Course (IdCourse,Course_Name,Duration,Course_level)
```

Ainsi, une table normalisée nommée *Course*_0 (Id_{Course}, *Course_Name*, *Duration*, *Course_level*) est générée. De plus, une vue relationnelle portant le nom de la classe en question (View *Course*(Id_{Course}, *Course_Name*, *Duration*, *Course_level*) est définie. La figure 6.11 traduit cet exemple. En ce qui concerne la classe non canonique *DoctoralCourse* décrite comme étant une restriction de la classe *Course* sur la propriété *Course_level* ($\equiv \exists\, Course\_level.\{Phd\}$), une vue de classe est définie comme suit :

```
Create View DoctoralCourse as
   (select * from Course where Course_level = 'Phd')
```

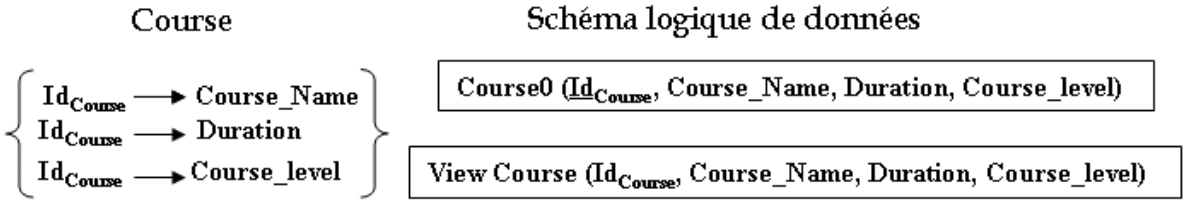

FIGURE 6.11 – Exemple de traitement d'une classe canonique

5 Comparaison des approches de conception

Dans cette section, nous proposons une étude comparative de notre méthodologie de conception des $\mathcal{BDBO}$ et des deux principales approches étudiées dans le chapitre 1.4 : celle de [del Mar Roldán García et al., 2005] et celle de [Fankam et al., 2009]. Dans cette étude, nous nous intéressons à six principaux critères.

1. Réutilisation de la connaissance du domaine. Ceci est assuré par la réutilisation de la sémantique formalisée du domaine telle qu'elle est définie dans les ontologies. Dans [del Mar Roldán García et al., 2005], les auteurs ébauchent une méthodologie pour la conception des bases de données en se basant sur une ontologie de domaine prédéfinie. Dans [Fankam et al., 2009], Fankam et .al proposent une approche de conception de base de données fondée sur les ontologies de domaine. Dans notre approche, nous proposons de choisir une ontologie partagée identifiée à partir du domaine de l'application. Ainsi, la réutilisation de la connaissance du domaine est assurée par les trois approches grâce à l'exploitation de l'ontologie du domaine.
2. Intégration. Ce critère concerne l'intégration future des bases de données conçues à partir de la même ontologie de domaine. Contrairement à l'approche de [del Mar Roldán García et al., 2005] où le lien entre données et ontologie est indéfini, dans notre approche ainsi que dans celle de [Fankam et al., 2009], ce lien est indiqué ce qui facilite et assure une telle intégration.
3. Gestion des dépendances ontologiques. Ce critère concerne la modélisation des dépendances entre les concepts ontologiques et leur intégration dans le modèle d'ontologies. Ces dépendances incluent (i) les dépendances définies sur les propriétés ontologiques et (ii) les dépendances entre les classes ontologiques. Contrairement aux approches de [del Mar Roldán García et al., 2005] et de [Fankam et al., 2009], dans notre approche,

nous proposons la modélisation de telles dépendances et leur intégration dans le modèle d'ontologies en étendant le modèle formel proposé dans [Bellatreche et al., 2003] par les dépendances entre classes (CD) et les dépendances entre propriétés (PFD).

4. Placement. Ce critère décrit l'organisation des classes ontologiques (canoniques et non canoniques) dans la bonne hiérarchie de subsomption. Contrairement aux approches de [del Mar Roldán García et al., 2005] et de [Fankam et al., 2009] où les classes de l'ontologie peuvent être définies sans spécifier leur relation de subsomption, nous proposons dans notre méthodologie, de compléter la description de cette hiérarchie en (a) définissant un ensemble de règles de placement pour un ensemble de constructeurs (union, intersection, etc.) et (b) en exploitant les raisonneurs existants pour établir le placement de ces classes.
5. $\mathcal{BDBO}$ en troisième forme normale. Étant donné que les dépendances ontologiques ne sont pas gérées par les approches proposées par [del Mar Roldán García et al., 2005] et [Fankam et al., 2009], offrir un schéma normalisé reste une tâche très difficile pour le concepteur. Par contre, dans notre approche, nous proposons la génération d'un schéma logique de données normalisé exploitant les dépendances définies entre les concepts ontologiques et réduisant par la suite la redondance et l'inconsistance des données dans la base de données cible.
6. Déploiement. Ce critère concerne la description du processus de déploiement de l'approche proposée dans la base de données cible. Vue la diversité des architectures des $\mathcal{BDBO}$ et la variété des modèles de stockage, nous proposons une étude du processus de déploiement de notre approche dans les bases de données à base ontologique. Notons que ce point n'a pas été traité dans les deux autres approches.

La figure 6.12 illustre l'étude comparative proposée.

6 Conclusion

Dans ce chapitre, deux approches sont présentées. Dans un premier temps, nous avons proposé une méthodologie dédiée à la conception d'une base de données à base ontologique à partir d'une ontologie de domaine. Inspirée des méthodologies de conception des bases de données classiques, cette approche traite les différentes phases traditionnelles du processus de conception: les modélisations conceptuelle, logique et physique. Cette méthode considère les ontologies locales comme des modèles conceptuels et emprunte des techniques formelles issues à la fois de la théorie des graphes pour l'analyse de dépendances, du raisonnement déduit de la logique de description pour le placement des classes et de la théorie des bases de données relationnelles pour la création des vues relationnelles, définies sur des tables normalisées et associées aux classes canoniques. De plus, elle considère différents types de dépendances entre les concepts ontologiques à des niveaux différents de conception. Dans un deuxième temps, étant donné la diversité des architectures des bases de données à base ontologique et la variété

Critères \ Approches	Roldan-Garcia et al. [2009]	Fankam et al. [2009]	Notre méthodologie
Réutilisation de la connaissance du domaine	oui	oui	oui
Intégration	non	oui	oui
Gestion des dépendances ontologiques	non	non	oui
Placement	non	non	oui
BDBO en 3FN	non	non	oui
Déploiement	non	non	oui

FIGURE 6.12 – Comparaison des approches de conception.

de leurs modèles de stockage, nous avons étudié la complexité de la phase de déploiement. En se basant sur cette étude, nous avons proposé une approche présentant un déploiement à la carte ayant pour but de satisfaire les préférences du déployeur quant à ses choix portant sur les architectures et les modèles de stockage. Celle-ci permet d'abord de réduire la combinatoire des solutions possibles avant le calcul de la meilleur solution suivant un modèle de coût donné. Enfin, un scénario de déploiement dans un environnement de base de données à base de modèles OntoDB est présenté. Dans le chapitre suivant, nous présentons l'implémentation de ce scénario une telle base.

Chapitre 7

Prototype de validation

Sommaire

Résumé. Dans les deux derniers chapitres, deux principales approches ont été proposées. La première décrit une méthode de conception d'une base de données à base ontologique à partir d'une ontologie conceptuelle tandis que la seconde présente le mécanisme de déploiement des données ontologiques après la conception d'une telle base. Dans ce chapitre, nous montrons la faisabilité de leur implantation en présentant leur mise en oeuvre sur la $\mathcal{BDBO}$ OntoDB. De plus, nous proposons un outil d'aide à la conception des bases de données à base de modèles à partir d'une ontologie conceptuelle adoptant notre approche de conception. Cet outil a pour objectif d'accompagner le concepteur durant les différentes étapes de modélisation à travers la proposition d'une interface graphique.

1 Introduction

Dans cette thèse, deux approches sont proposées. La première présente une méthode de conception d'une base de données à base ontologique à partir d'une ontologie conceptuelle (canonique ou non-canonique). La deuxième approche traduit le mécanisme de déploiement des données ontologiques après la conception d'une telle base. Dans ce chapitre, nous commençons par la proposition d'un outil d'aide à la conception des bases de données à base de modèles à partir d'une ontologie conceptuelle. Ensuite, nous présentons la validation de notre approche dans un environnement de $\mathcal{BDBO}$ sous OntoDB. Celle-ci nécessite l'extension du langage OntoQL étant donné qu'il n'offre pas les constructeurs nécessaires pour la création (1) des dépendances fonctionnelles entre les propriétés ontologiques, (2) des dépendances définies sur les classes ontologiques et (3) la génération du modèle logique de données en troisième forme normale. Notons que dans cette validation, nous traitons uniquement les ontologies conceptuelles canoniques. Enfin, nous présentons les développements que nous avons menés autour ce processus de validation.

2 OBDBDesignTool : un outil d'aide à la conception des BD à partir d'une ontologie conceptuelle

Dans cette section, nous présentons dans un premier temps une description de l'outil proposée suivie de la description de son interface graphique.

2.1 Description de l'outil

OBDBDesignTool est un outil de conception permettant d'accompagner l'utilisateur durant le processus de conception des bases de données à partir d'une ontologie de domaine. C'est un outil d'aide à la conception offrant aux utilisateurs les moyens pour leur faciliter la modélisation (a) conceptuelle, (b) logique et (c) physique. L'architecture générale de ce système est composée de cinq principaux modules comme cela est décrit dans la figure 7.1. Le premier module est dédié au chargement de l'ontologie globale (OG). Le deuxième offre la possibilité de définir le modèle conceptuel (OL) en raffinant et/ou en enrichissant l'ontologie de domaine. Le troisième module est conçu pour l'implémentation de l'algorithme de canonicité et la classification des classes ontologiques selon leur type. Après avoir effectué cette classification, nous traitons dans le quatrième module la génération du schéma logique de données normalisé. Ainsi, en se basant sur ce modèle, le concepteur va entamer la phase de la modélisation physique en générant à travers le cinquième module le script adéquat à la structure de stockage choisie. La figure 7.2 illustre un diagramme de cas d'utilisation de notre outil.

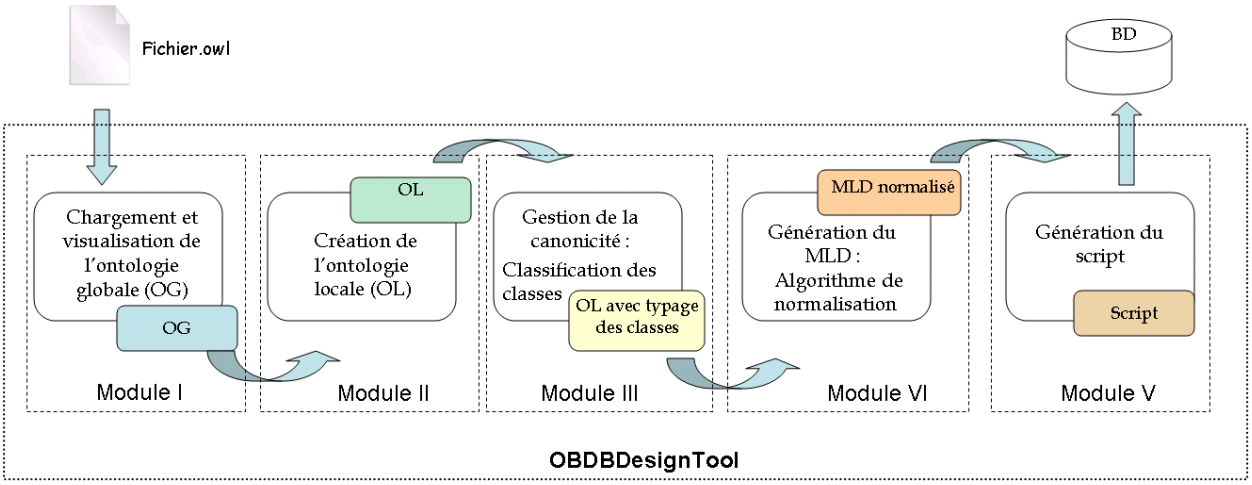

FIGURE 7.1 – Architecture générale du système

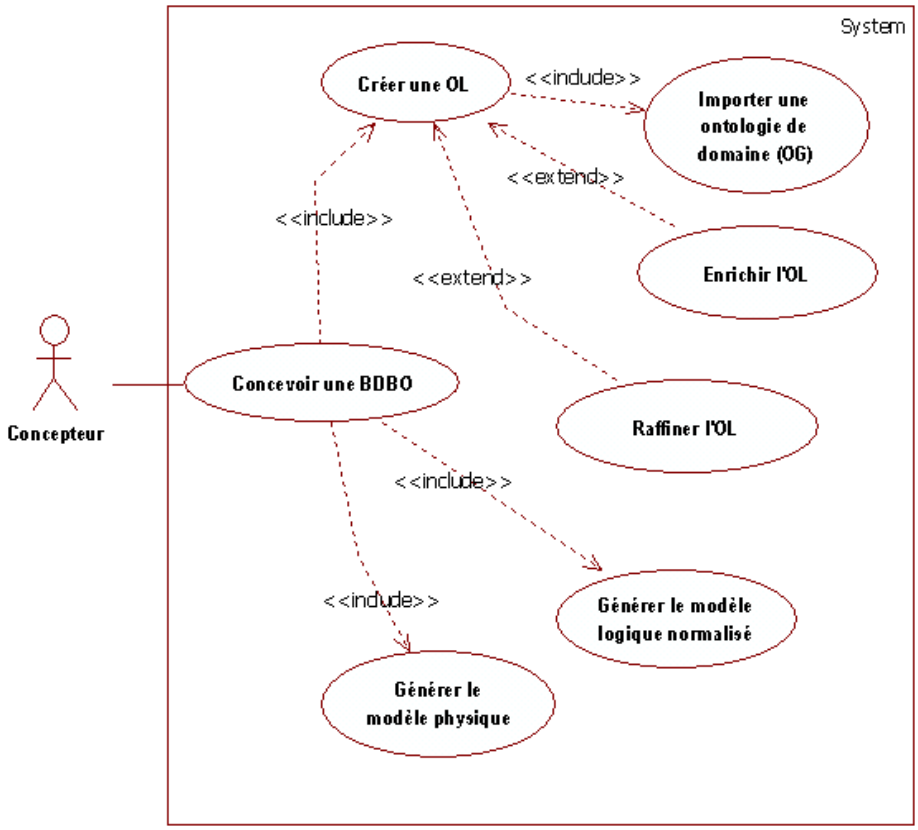

FIGURE 7.2 – Diagramme de cas d'utilisation d'OBDBDesignTool

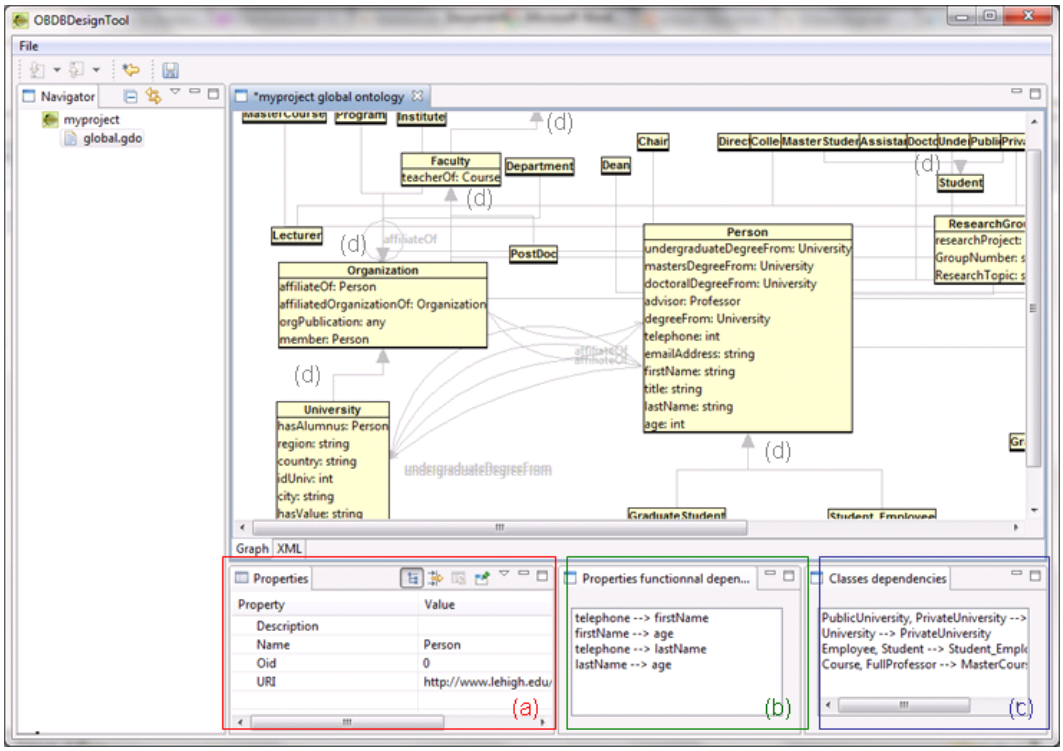

FIGURE 7.3 – Exemple de visualisation de l'ontologie globale

2.2 Description de l'interface graphique

Dans cette section, nous présentons la description des différents modules d'OBDBDesign-Tool.

2.2.1 Module I : Chargement et visualisation de l'ontologie globale

Le premier module permet de charger et de visualiser l'ontologie globale. Il reçoit en entrée une ontologie dont le format est OWL (fichier d'extension .owl) et renvoie en sortie l'affichage de l'ontologie manipulée sous forme graphique. La figure 7.3 illustre un exemple d'affichage de l'ontologie globale 'Lehigh University' décrivant le domaine universitaire. L'affichage permet de visualiser les classes de l'ontologie, les propriétés (figure 7.3.(a)), la hiérarchie des concepts (figure 7.3.(d)) ainsi que les dépendances entre concepts ontologiques : entre propriétés (figure 7.3.(d)) et entre classes (figure 7.3.(c)).

2.2.2 Module II : Définition de l'ontologie locale

Ce module offre aux concepteurs le moyen de définir l'ontologie locale à partir de l'ontologie globale en :

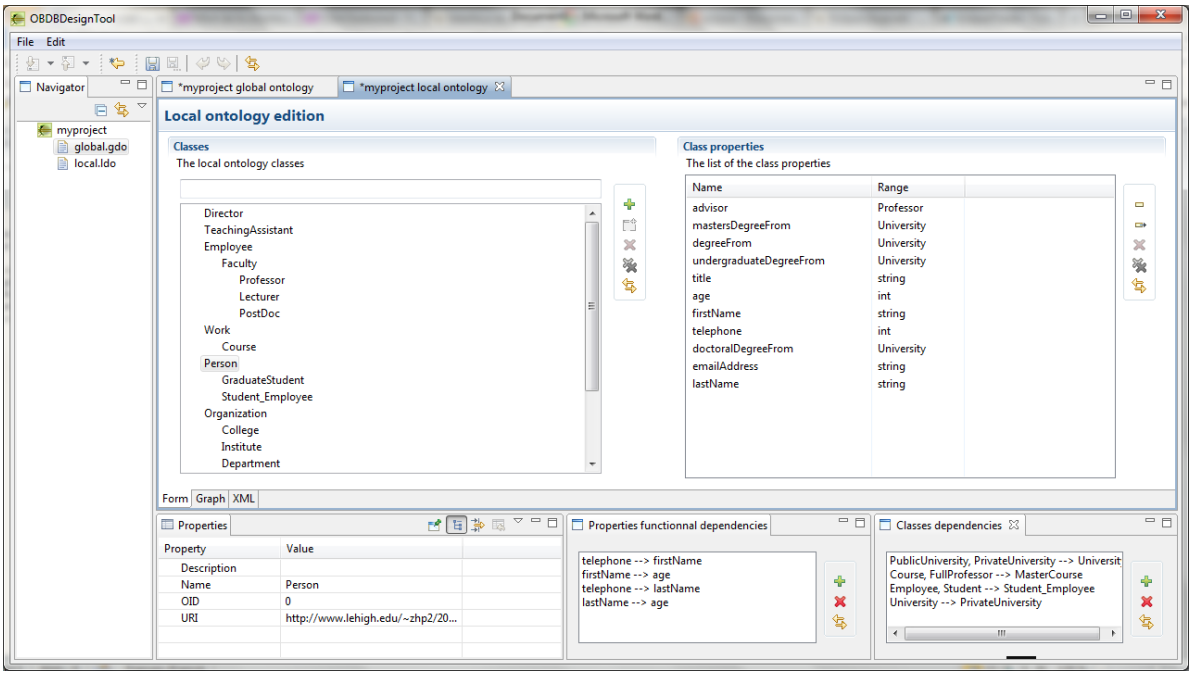

FIGURE 7.4 – Exemple de définition d'une ontologie locale

- ajoutant et/ou supprimant des classes ;
- ajoutant et/ou supprimant des propriétés ;
- ajoutant et/ou supprimant des dépendances fonctionnelles entre propriétés ;
- ajoutant et/ou supprimant des dépendances entre classes.

La figure 7.4 illustre le formulaire d'édition de l'ontologie locale. Les signes + et - désignent respectivement les actions d'ajout et de suppression d'un concept ontologique. Pour un meilleur affichage, l'OL est affichée en lecture seule sous forme d'un diagramme de classes (voir la figure 7.5.(a)).

2.2.3 Module III : Gestion de la canonicité

Avant la génération du modèle logique de données, l'outil offre au concepteur la possibilité de calculer les différentes solutions quant à la canonicité des classes. Pour chaque solution, les classes sont affichées selon leur type (canonique ou non canonique). Les figures 7.6 et 7.7 illustrent un exemple de classification des classes ontologiques où deux solutions de classification sont proposés.

2.2.4 Module VI : Génération du schéma relationnel

A travers ce module, le concepteur peut définir le modèle logique de données pour chaque solution générée lors du calcul de la canonicité des classes. De cette manière, il a la possibilité

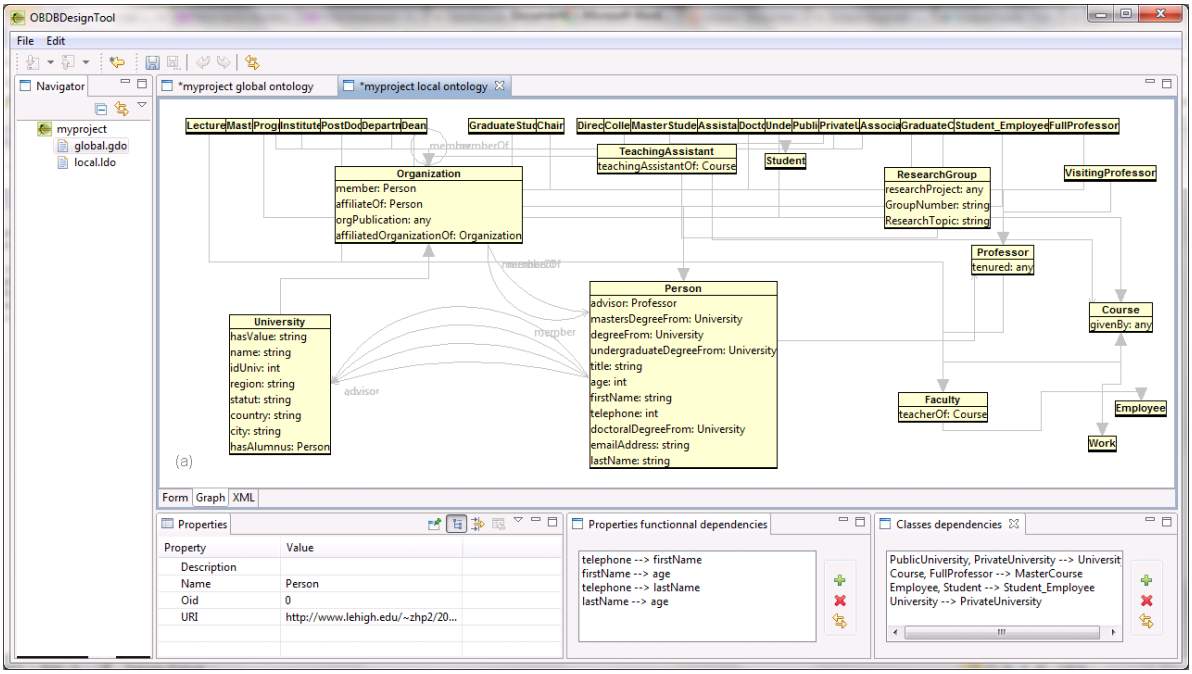

FIGURE 7.5 – Affichage de l'ontologie locale sous forme de diagramme de classes.

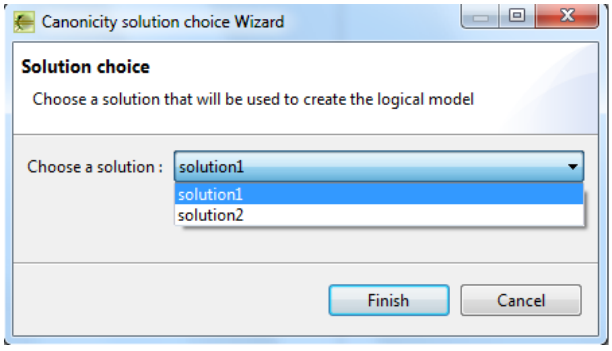

FIGURE 7.6 – Génération des différentes solutions de classification de classes

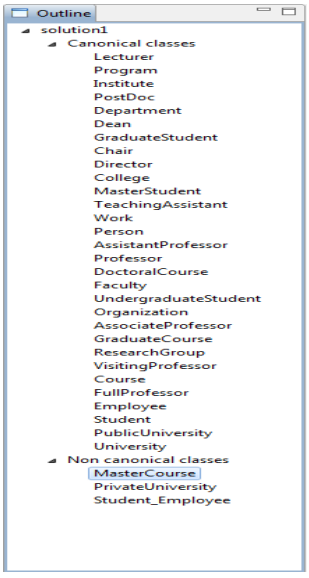

FIGURE 7.7 – Exemple de classification de classes selon la solution 1

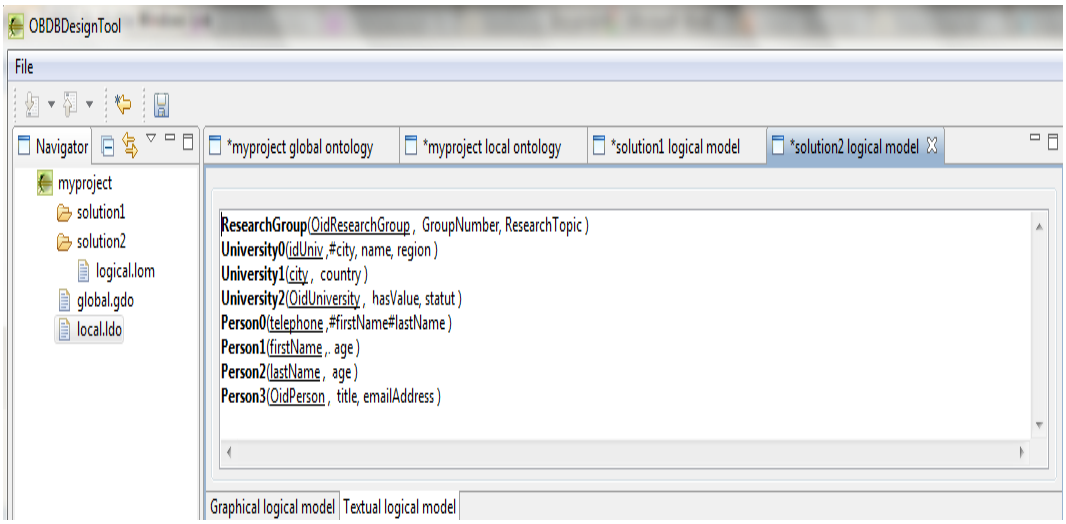

FIGURE 7.8 – Exemple de schéma logique de données

de choisir le schéma logique de données répondant mieux à ses attentes. La figure 7.8 illustre un exemple de schéma logique de données normalisé pour les classes canoniques.

2.2.5 Module V : Génération du script

Après avoir défini le modèle logique de données, l'utilisateur choisit la base de données cible sur laquelle il souhaite déployer son modèle. Autrement dit, il choisit le langage adéquat à la base de données choisie lui permettant la création des modèles et l'interrogation des données. Ainsi, à travers notre outil, nous proposons à l'utilisateur de choisir un langage parmi une liste donnée (OntoQL, SQL, XML, NTRIPLE, N3, etc) (voir figure 7.9) puis de générer automatiquement le script approprié à ses besoins. Ce script est à la fois affiché sur l'interface graphique tel qu'illustré dans la figure 7.10 et exporté dans un fichier ayant le format de sortie choisi.

Après avoir présenté OBDBDesignTool, nous proposons de déployer notre approche dans un environnement de $\mathcal{BDBO}$ sous OntoDB tel que décrit dans la section suivante.

3 Validation sous OntoDB : Extension du langage OntoQL

OntoloQL est un langage d'exploitation des bases de données à base ontologique comportant à la fois un langage de définition de données (LDD) et un langage de manipulation de données (LMD). Le LDD permet de créer les éléments du modèle de données : les classes, les propriétés des ontologies ainsi que les extensions de ces classes. Notons que ces extensions se traduisent par la définition d'une table par classe ontologique. En ce qui concerne le LMD, il offre la possibilité de créer, de mettre à jour et de supprimer les instances des classes dans la

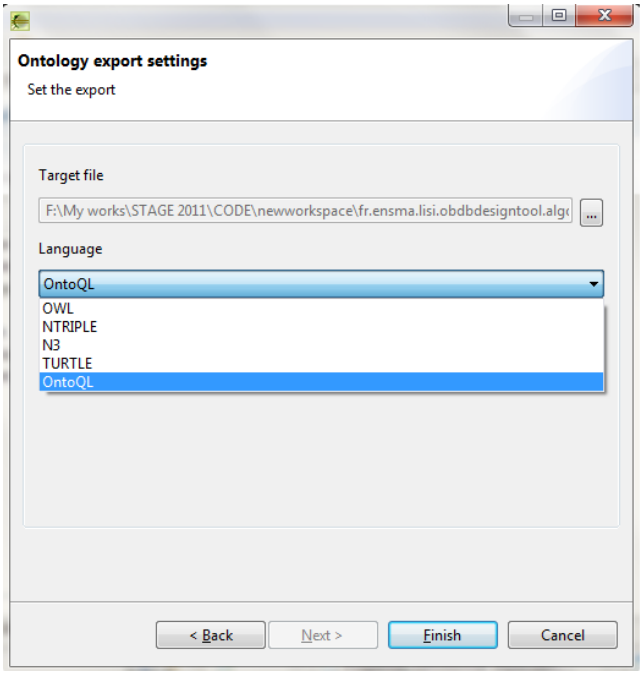

FIGURE 7.9 – Liste des langages d'exportation des modèles

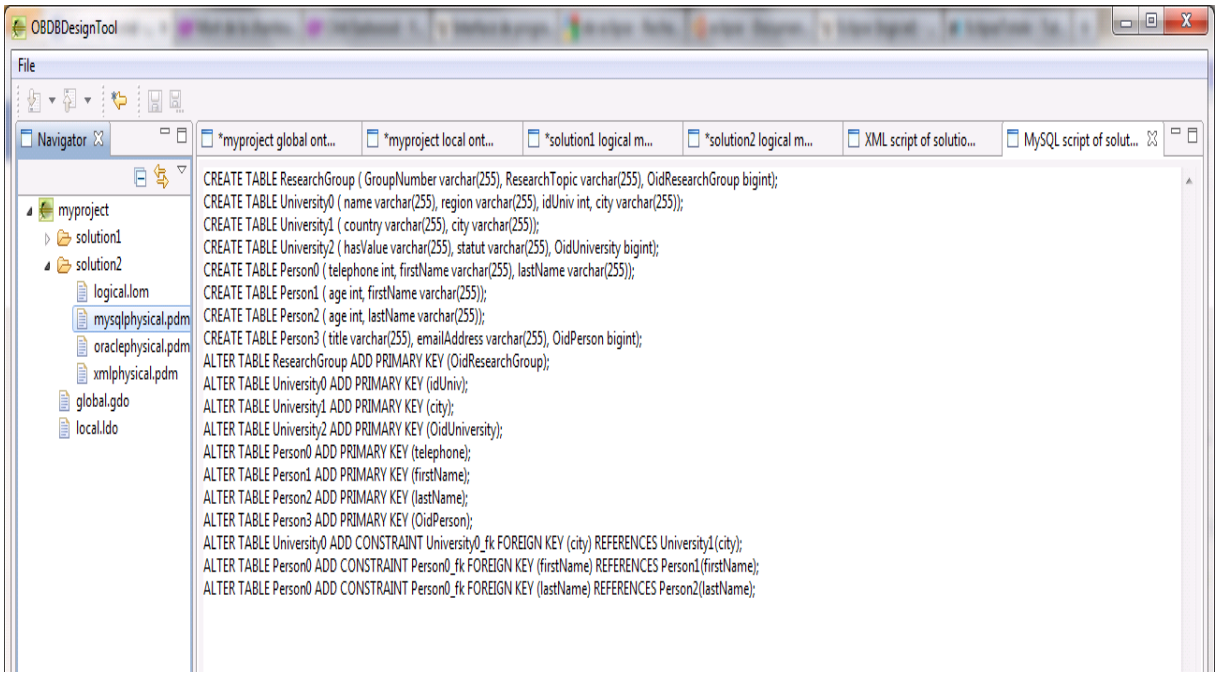

FIGURE 7.10 – Exemple de script généré

$\mathcal{BDBO}$. Étant donné qu'OntoQL ne permet ni la création des dépendances ontologiques entre les propriétés ni entre les classes et qu'il n'offre aucune extension de classes prenant en considération le modèle logique de données normalisé, nous proposons dans cette section, d'enrichir le LDD d'OntoQL par :

- l'ajout des dépendances fonctionnelles entre les propriétés de données (PFD) ;
- l'ajout des dépendances définies entre les classes ontologiques (CD) ;
- l'extension des classes en se référant au modèle logique de données normalisé.

3.1 Création des dépendances ontologiques

Tel que proposée dans la partie contribution, notre méthodologie de conception des bases de données à base ontologique est basée sur deux différents types de dépendances : les dépendances fonctionnelles entre les propriétés ontologiques et les dépendances définies sur les classes. Dans cette section, nous proposons l'extension du langage OntoQL pour permettre la création de telles relations.

3.1.1 Création d'une dépendance entre les classes ontologiques

La syntaxe de création d'une dépendance définie sur les classes ontologiques est la suivante :

⟨*class dependency definition*⟩::= **CREATE CD OF** (⟨*class id list*⟩ ⟨*class id*⟩)

L'en-tête de cette clause permet de créer une dépendance élémentaire entre les classes ontologiques (CREATE CD OF). Une CD est définie par une partie gauche constituée d'une liste de classes et par une partie droite contenant une seule classe ontologique. Ainsi, le corps de cette clause est constitué de l'élément (⟨*class id list*⟩) permettant l'indication de la liste des classes incluses dans la partie gauche et de l'élément (⟨*class id*⟩) indiquant la classe incluse dans la partie droite.

Exemple. Créer la dépendance élémentaire $Student \rightarrow MasterStudent$ décrivant que la connaissance de la population de la classe des étudiants (Student) implique la connaissance de la population de la classe des étudiants inscrits en master (MasterStudent) :

$CREATE\ CD\ OF\ (Student,\ MasterStudent)$

3.1.2 Création d'une dépendance fonctionnelle entre les propriétés ontologiques

Dans cette partie, nous traitons la définition des dépendances fonctionnelles entre les propriétés ontologiques au sein de la même classe. La syntaxe de création d'une telle dépendance est la suivante :

⟨*functional property dependency definition*⟩::= **CREATE FD OF** *classid* (⟨*property id*⟩ ⟨*property id list*⟩)

L'en-tête de cette clause permet de créer une dépendance fonctionnelle élémentaire définie sur les propriétés ontologiques (CREATE FD) en indiquant le nom de la classe (OF ⟨*classid*⟩) à laquelle elle est associée. Une PFD est définie par une partie droite contenant une seule propriété ontologique et une partie gauche constituée d'une liste de propriétés. Ainsi, le corps de cette clause est constitué de l'élément (⟨*property id*⟩) indiquant la propriété incluse dans la partie droite et de l'élément (⟨*property id list*⟩) permettant l'indication de la liste des propriétés incluses dans la partie gauche.

Exemple. Créer la dépendance fonctionnelle élémentaire $idUniv \rightarrow Univ\_name$ associée à la classe *Private_University* décrivant que la connaissance d'une valeur de l'identifiant de l'université (idUniv), quelle qu'elle soit, implique la connaissance d'un seul nom d'université (Univ_name) :

$CREATE\ FD\ OF\ Private\_University(Univ\_name, idUniv)$

3.2 Création de l'extension d'une classe

Le LDD nous offre la possibilité de créer des extensions permettant le stockage des instances de chaque classe et ce à partir des concepts ontologiques créés (classes et propriétés). Cette extension doit respecter un ensemble de règles conformes au langage OntoQL telles que suit :

1. une extension peut ne contenir qu'un sous-ensemble de propriétés applicables sur une classe ;
2. à chaque classe est associée une seule extension ;
3. aucune relation d'héritage n'existe entre les extensions ;
4. une extension est implantée au niveau logique.

Rappelons que dans l'ancienne version d'OntoQL sur OntoDB, l'extension d'une classe se traduit par l'extension créée au niveau logique selon la représentation horizontale, c'est-à-dire par une table non normalisée comprenant une colonne pour chaque propriété de l'extension. Dans cette section, nous proposons deux fonctionnalités permettant l'extension d'une classe en troisième forme normale. La première est basée sur la connaissance préalable de l'existence des dépendances fonctionnelles définies sur les propriétés de la classe manipulée tandis que la deuxième traite l'extension générique de la classe.

3.2.1 Dépendances fonctionnelles entre propriétés et création de l'extension d'une classe

Dans cette section, nous proposons la syntaxe correspondante à l'extension normalisée d'une classe ontologique canonique ayant des dépendances fonctionnelles définies sur ses propriétés telles que suit :

⟨*FD extension definition*⟩::= **CREATE FDEXTENT OF** *class id* (⟨*property id list*⟩)

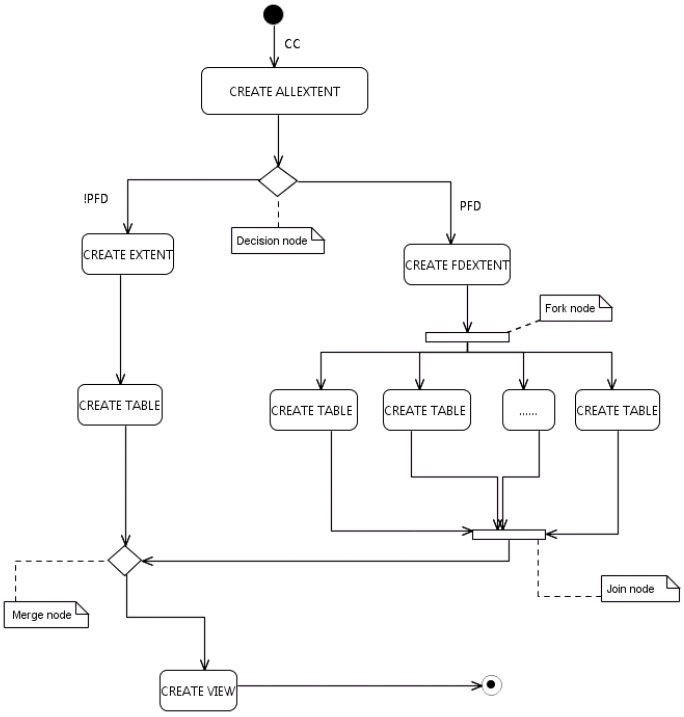

FIGURE 7.11 – Diagramme d'activités : modélisation du processus d'extension d'une CC

Cette extension permet de créer l'ensemble des tables normalisées (troisième forme normale) associées à la classe en question. L'en-tête de cette clause permet de définir une extension en se basant sur les dépendances fonctionnelles définies sur les propriétés en indiquant seulement le nom de la classe (OF ⟨*classid*⟩) dont elle est l'extension. Etant donné qu'il n'y a pas de relation d'héritage entre les extensions, l'en-tête ne comprend pas le mot clé UNDER gérant ce processus. Le corps de cette clause est constitué de l'élément ⟨*property id list*⟩ permettant l'indication de la liste des propriétés incluses dans cette extension. Les noms de cette (ces) table(s) et de ses (leurs) colonnes sont générés automatiquement (nom de la classe + i ; i ∈ ℕ). *Exemple*. Créer l'extension de la classe *Private_University* (PrU).

CREATE FDEXTENT OF *Private_University* (IdUniv,name,city,region,country)

Notons qu'un ensemble de dépendances fonctionnelles entre les propriétés de cette classe est défini (PFD^{PrU} = { $IdUniv \rightarrow name$, $IdUniv \rightarrow city$, $IdUniv \rightarrow region$, $IdUniv \rightarrow country$, $city \rightarrow region$, $region \rightarrow country$ }). Ainsi, l'exécution de la balise décrite ci-dessus déclenche la création de trois tables normalisées telles que suit :

1. $Private\_University_0$(oid, IdUniv, name, city) où l'oid présente l'identifiant unique généré automatiquement par la base de données OntoDB ;
2. $Private\_University_1$(oid, city, region) ;
3. $Private\_University_2$(oid, region, country).

3.2.2 Création générique de l'extension d'une classe

En proposant les instructions liées à l'extension d'une classe (*CREATE EXTENT* (déjà fourni dans la première version d'OntoQL) et *CREATE FDEXTENT* (décrit dans la section précédente)), l'utilisateur doit impérativement connaître si un ensemble de dépendances fonctionnelles est défini sur les propriétés de cette classe ou pas afin de choisir l'instruction appropriée. Rappelons que l'instruction *CREATE EXTENT* traite l'extension des classes sans considérer leurs PFD tandis que l'instruction *CREATE FDEXTENT* crée uniquement l'extension des classes ayant des PFD. Afin de faciliter la tâche de l'extension d'une classe ontologique, nous proposons une nouvelle clause permettant de traiter l'extension d'une classe canonique (*CC*) en dépit de la définition des PFD. La syntaxe de la clause proposée est la suivante :

⟨*Allextension definition*⟩::= **CREATE ALLEXTENT OF** *classid* (⟨*property id list*⟩) [⟨*logical clause*⟩]

⟨*logical clause*⟩ ::= **TABLE** [⟨*table and column name*⟩]

⟨*table and column name*⟩ ::= ⟨*table name*⟩ [(⟨*column name list*⟩)]

Cette instruction crée l'extension adéquate pour la classe manipulée. En d'autres termes, si la classe (*CC*) ne possède pas de PFD (!PFD), le processus associé à la clause CREATE EXTENT est déclenché. Celui-ci évoque la clause *CREATE TABLE* permettant la création d'une seule table déclenchant par la suite la création d'une vue relationnelle (*CREATE VIEW*). Dans le cas contraire, la clause *CREATE FDEXTENT* est évoquée. Celle-ci déclenche dans un premier temps la création d'un ensemble de tables normalisées correspondantes à *CC* (*CREATE TABLE*). Dans un deuxième temps, une vue définie sur ces tables est créée à l'aide de la fonction *CREATE VIEW*. La figure 7.11 retrace le diagramme d'activités modélisant le processus d'extension d'une classe canonique.

4 Implémentation

Dans cette section, nous présentons les efforts de développement réalisés pour la mise en œuvre de notre prototype de validation. Dans un premier temps, nous traitons la mise en œuvre de l'outil d'aide à la conception OBDBDesignTool. Dans un deuxième temps, nous étudions l'implantation de la méthodologie définie sur OntoDB.

4.1 Mise en œuvre d'OBDBDesignTool

Dans cette partie, nous présentons la mise en œuvre de notre outil d'aide à la conception. Nous commençons par introduire l'environnement de travail utilisé. Ensuite, nous présentons l'architecture générale de l'outil suivie d'une analyse conceptuelle des principaux modèles utilisés lors de la réalisation d'OBDBDesignTool. Enfin, nous décrivons la phase de génération du

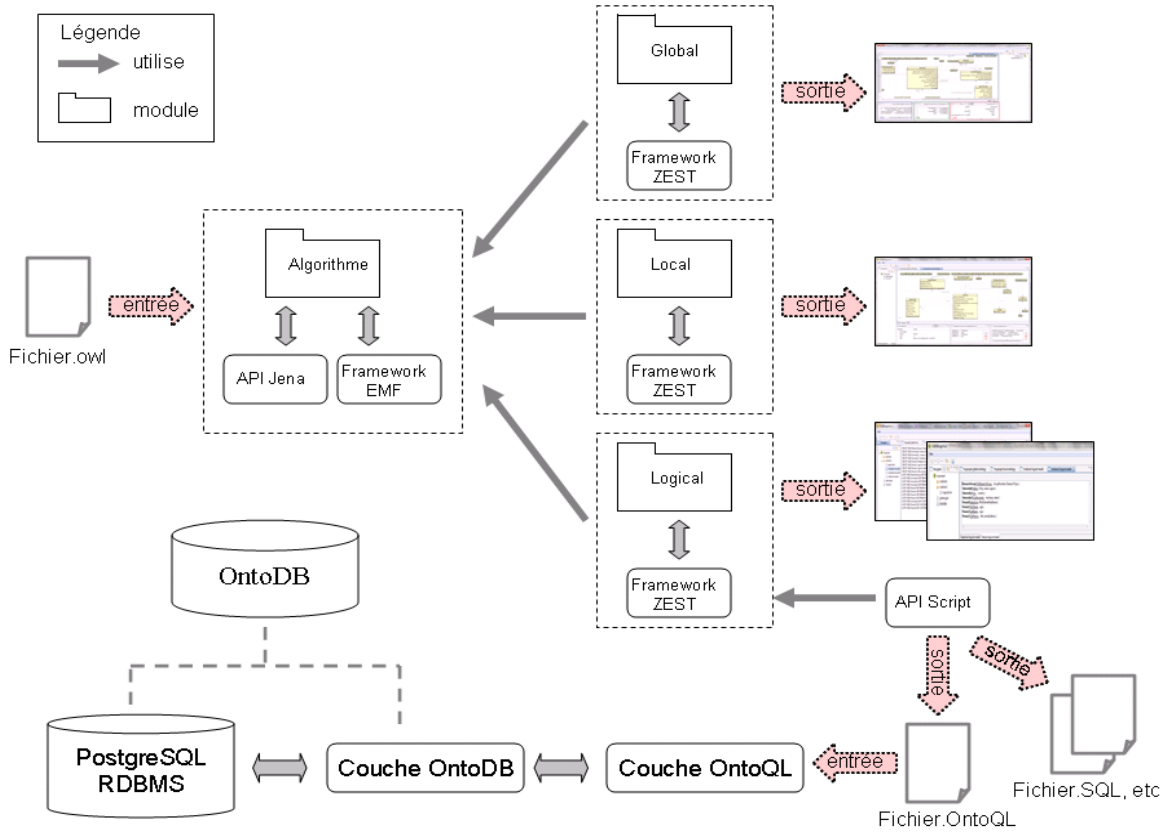

FIGURE 7.12 – Architecture générale d'OBDBDesignTool

code à partir des modèles définis.

4.1.1 Environnement de travail

Pour la mise en œuvre de notre projet, nous avons utilisé :

1. le langage de développement Java dans un environnement de développement Eclipse. Ce choix a été fait pour la simplicité, la portabilité et la robustesse du langage ainsi que les bibliothèques qu'il offre pour la construction des interfaces graphiques.
2. le framework de modélisation EMF [31] offrant :
   - les outils nécessaires pour la transformation des modèles décrits en XMI (diagramme de classes UML par exemple) en un code Java ;
   - la génération d'une API pour les modèles tout en mettant à disposition les moyens nécessaires pour l'interrogation de ces derniers ;
   - l'édition transactionnelle des modèles. Cette technique nous permet de mieux contrôler les modifications, permettant ainsi de les défaire et de les refaire (implémentation du

31. http ://www.eclipse.org/modeling/emf/

undo/redo) ;
- le "data binding". JFace data binding est une technique permettant de connecter et synchroniser des objets. Typiquement, cette technique est utilisée pour lier les composants de l'interface graphique aux attributs du modèle de domaine (voir section 4.1.3).

3. Zest [32], la boite à outils de visualisation de graphes. Basée sur SWT [33] et Draw2d [34], elle est utilisée pour la visualisation de certains de nos modèles sous forme de diagrammes.
4. le framework de test unitaire JUnit [35] qui permet la réalisation des tests unitaires pour le langage Java dans les environnements de développement associés tels que Eclispe et Netbeans.
5. l'API Jena [36] qui fournit une collection d'outils et de bibliothèques Java pour le développement d'applications du Web Sémantique. Nous avons utilisé cette API essentiellement pour l'extraction de l'ontologie globale et l'exportation de l'ontologie locale.

4.1.2 Architecture de l'outil

Dans la figure 7.12, nous présentons l'architecture générale d'OBDBDesignTool. En effet, notre outil a pour entrée un fichier owl décrivant l'ontologie manipulée extraite au moyen de l'API Jena et modélisée en utilisant le framework EMF. Un ensemble de traitements est effectué sur l'ontologie à l'aide du module Algorithme codant les différents algorithmes développés pour l'implémentation de notre méthodologie. Ces algorithmes sont initiés par différents modules : (a) le module Global permettant la visualisation de l'ontologie globale, (b) le module Local définissant et visualisant l'ontologie locale, (c) le module Logical générant et visualisant le modèle logique. Précisons que la visualisation des interfaces homme-machine est assurée grâce au framework ZEST. Enfin, une API Script a été définie pour la génération du script dans un langage de manipulation de $\mathcal{BDBO}$ permettant ainsi la création du modèle physique dans la base de données cible.

4.1.3 Analyse conceptuelle

Pour la réalisation de notre projet, nous avons adopté une approche dirigée par les modèles. Pour ce faire, nous proposons dans un premier temps une analyse conceptuelle des principaux modèles utilisés lors de la réalisation de notre outil. Ces modèles, incluant les modèles du domaine ainsi que les modèles de données pour l'interface homme-machine, sont transformés par la suite en code java à l'aide de la technologie EMF. Dans un deuxième temps, nous décrivons la conception des plug-ins utilisés.

32. http ://www.eclipse.org/gef/zest/
33. http ://www.eclipse.org/swt/
34. http ://www.eclipse.org/gef/draw2d/index.php
35. http ://www.junit.org/
36. http ://jena.sourceforge.net

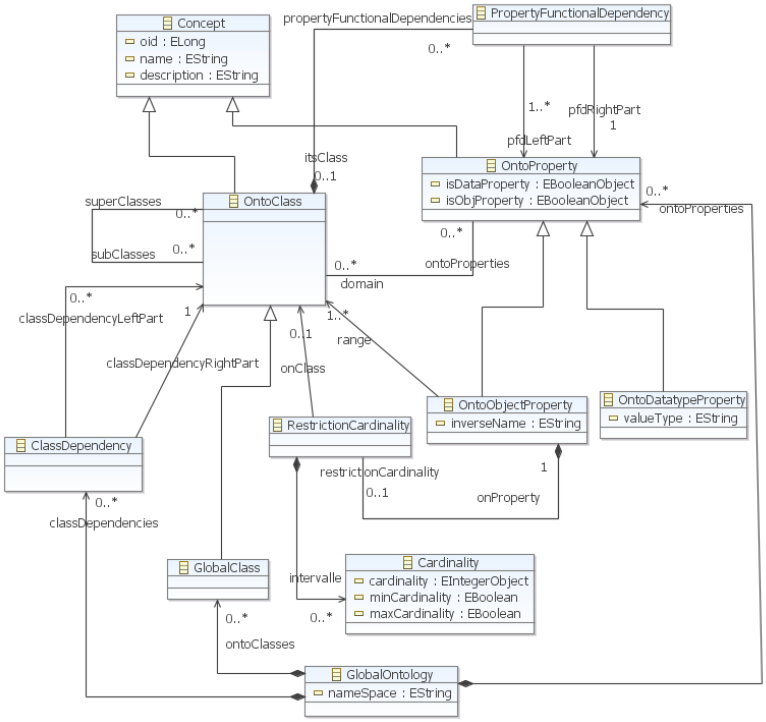

FIGURE 7.13 – Modèle statique de l'ontologie globale

– Conception des modèles du domaine. Dans ce paragraphe, nous traitons les quatre modèles manipulés : (1) le modèle de l'ontologie globale, (2) le modèle de l'ontologie locale, (3) le modèle de graphe et (4) le modèle logique de données.

1. *Modèle de l'ontologie globale.* Le modèle statique de l'ontologie globale est inspiré de la structure des ontologies OWL comme cela est décrit sur le méta-modèle de la figure 7.13. Dans ce paragraphe, nous présentons une brève description de ses éléments.
   – *GlobalOntology* décrit une ontologie globale ayant un domaine d'unicité des noms (*namespace*) et composée d'un ensemble de concepts : classes et propriétés.
   – *OntoClass* représente une classe ontologique qui possède un identifiant (*oid*), un nom (*name*) et une description (*description*). Une classe peut avoir plusieurs sous-classes (*subClasses*) et defférentes super-classes (*superClasses*).
   – *OntoProperty* est une propriété qui possède un identifiant (*oid*), un nom (*name*), une description (*description*), un domaine (*domain*) et un co-domaine (*range*). Une propriété peut être de type simple (*OntoDatatypeProperty*). Dans ce cas, son domaine de valeurs (*valueType*) est l'ensemble des types primitifs (int, string, boolean, etc.). Elle peut également prendre ses valeurs dans une classe. Dans ce cas, on parle de propriété d'objets (*OntoObjectProperty*) .
   – Des restrictions peuvent être définies sur les propriétés. Dans le cas d'une restriction de cardinalités (*RestrictionCardinality*), la cardinalité peut être minimale

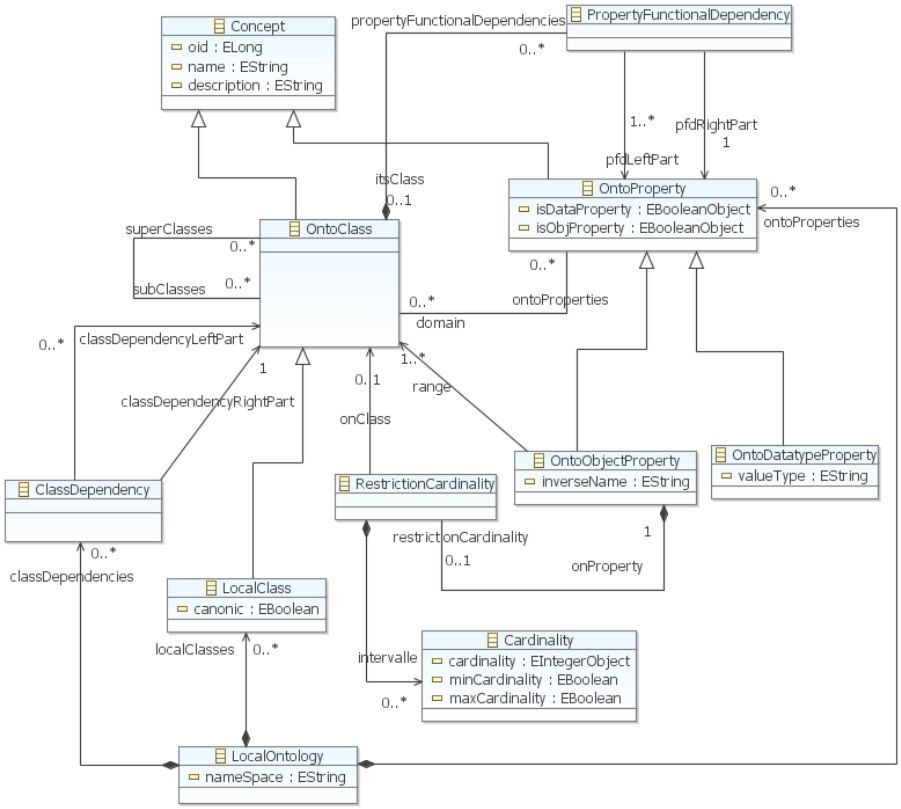

FIGURE 7.14 – Modèle statique de l'ontologie locale

(*minCardinality*), maximale (*maxCardinality*) ou exacte (*Cardinality*).

- Les dépendances entre classes sont représentées par la classe *ClassDependency*. Les instances de cette classe constituent l'ensemble des dépendances entre les classes de l'ontologie. Chaque dépendance est composée d'une partie gauche (*classDependencyLeftPart*) et une partie droite (*classDependencyRightPart*).
- Les dépendances fonctionnelles entre propriétés sont représentées par la classe *PropertyFunctionalDependency*. Une telle dépendance est définie sur une classe donnée (*itsClass*) et est composée d'une partie gauche (*pfdLeftPart*) et d'une partie droite (*pfdRightPart*).

2. *Modèle de l'ontologie locale.* L'ontologie locale est similaire à l'ontologie globale. Toutefois, l'ontologie locale doit tenir compte de la classification des classes ontologiques selon leur canonicité. Son modèle statique représenté par la figure 7.14 se différencie de celui de l'ontologie globale par l'ajout de la classe *LocalClass* ayant pour propriété *canonic* qui permet de distinguer les classes canoniques de celles non canoniques.

3. *Modèle de graphe.* Dans ce paragraphe, nous traitons le modèle de graphe de dépendances. Les classes de ce diagramme, illustré dans la figure 7.15, sont décrites

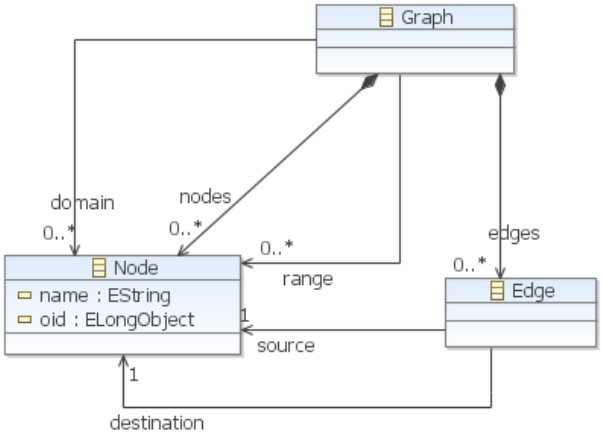

FIGURE 7.15 – Modèle statique d'un graphe de dépendances entre concepts ontologiques

brièvement ci-dessous.

- La classe *Graph* décrit le graphe constitué principalement de nœuds (*Node*) et des arcs (*Edge*).
- La classe *Node* définit un nœud du graphe décrivant (i) une ou plusieurs classe(s) de l'ontologie dans le cas d'un graphe de dépendances entre classes et (ii) une ou plusieurs propriété(s) dans le cas d'un graphe de dépendance entre propriétés.
- La classe *Edge* définit un arc du graphe qui décrit une relation de dépendances entre les concepts ontologiques (entre classes ou entre propriétés). Un arc possède une source (*source*) et une destination (*destination*) de type *Node*.

4. *Modèle logique de données.* Dans ce paragraphe, nous traitons le modèle logique de données en troisième forme normale. Les classes de ce diagramme, illustré dans la figure 7.16 sont décrites brièvement ci-dessous.
   - La classe *LogicalModel* représente le modèle logique de données composé d'un ensemble de relations (*relations*).
   - La classe *Relation* modélise une relation. Elle est décrite par une clé primaire (*primaryKey*), un ensemble de clés étrangères (*foreignKeys*) et un ensemble de colonnes (*columns*).
   - La classe *Column* représente une colonne d'une table. Une ou plusieurs colonnes peuvent constituer une clé primaire ou étrangère.

– Conception des modèles de données pour l'interface homme-machine La visualisation des données manipulées requiert la conception de deux principales structures. La première consiste en un diagramme pour l'affichage de l'ontologie globale, de l'ontologie locale et du modèle relationnel. Tandis que la deuxième est dédiée à l'affichage des solutions de classification sous forme arborescente.
  – *Modèle de représentation de diagramme.*
    Ce modèle permet d'organiser les classes et les propriétés des ontologies globale et

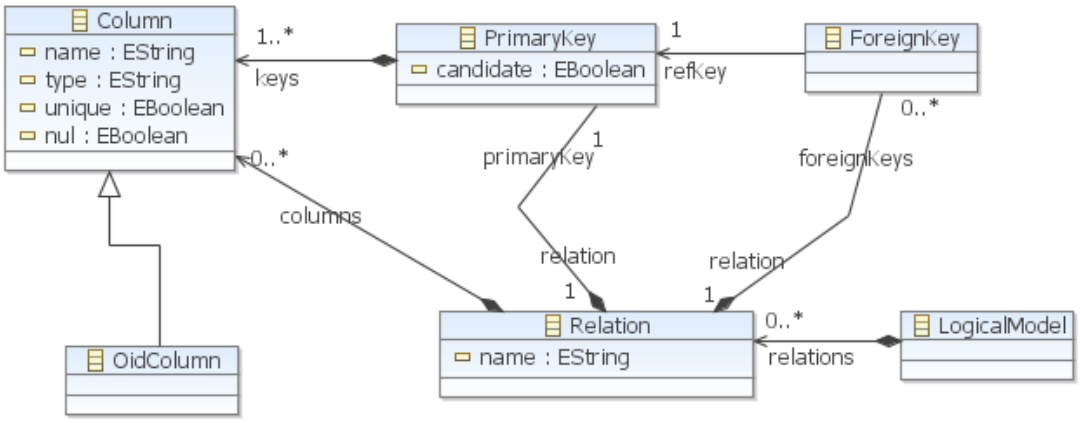

FIGURE 7.16 – Modèle statique d'un modèle relationnel

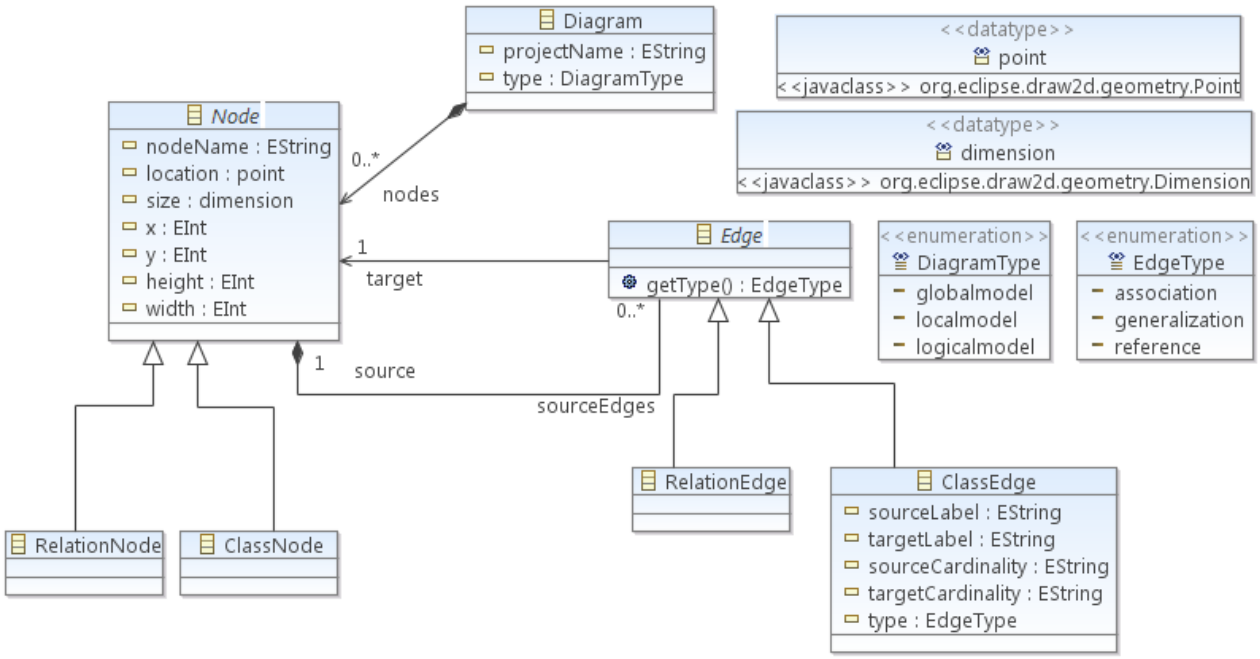

FIGURE 7.17 – Modèle statique de diagramme

locale sous forme d'un diagramme de classes. De plus, ce modèle est organisé de sorte à pouvoir utiliser la représentation des classes pour la description graphique des relations d'un modèle logique normalisé. Illustré dans la figure 7.17, ce modèle est composé des principaux éléments décrits ci-dessous.

- Le conteneur de haut niveau est un diagramme (classe *Diagram*). Il appartient à un projet de conception de $\mathcal{BDBO}$ (*projectName*) et a un type (*type*). Tois types de diagrammes sont distingués : le diagramme d'ontologie globale, le diagramme d'ontologie locale et le diagramme de modèle logique. Chaque diagramme est composé de nœuds (*nodes*).
- Un nœud (classe abstraite *Node*) représente soit une classe (*ClassNode*), soit une relation (*RelationNode*). Un nœud de classe référence une classe d'un modèle d'ontologies (globale ou locale) tandis qu'un nœud de relation référence une relation d'un modèle logique normalisé. Chaque nœud est caractérisé par son nom (*nodeName*), sa position (*location*) et sa taille (*size*).
- Un arc (classe abstraite *Edge*) représente une association entre classes, une relation de généralisation entre classes ou une référence (clé étrangère) entre relations. Chaque arc possède un nœud source (propriété *source*) et un nœud cible (propriété *destination*). Un arc entre classes (ClassEdge) peut avoir une étiquette et une cardinalité pour le nœud source (*sourceLabel*, *sourceCardinality*) ainsi que pour le nœud cible (*targetLabel*, *targetCardinality*). Dans ce cas il s'agit d'une association entre classes.

– *Modèle de typage des classes ontologiques.* Le typage des classes d'une ontologie locale se fait selon leur canonicité : (1) classes canoniques et (2) classes non canoniques. Par conséquent, nous avons conçu un modèle de données servant à l'affichage et à la sauvegarde des solutions de classification.

Illustré dans la figure 7.18, ce modèle comporte trois classes principales :

- *Classiffication*. Une solution de classification est rattachée à un projet de conception de $\mathcal{BDBO}$ défini par la propriété *projectName*. Une classification peut engendrer plusieurs solutions. Chaque solution a un nom décrit par la propriété *solution*. Une solution peut se faire selon deux catégories de classes : les classes canoniques et les classes non canoniques.
- *ClassificationCategory*. Une catégorie de classe est représentée par la classe *ClassificationCategory*. Celle-ci est définie par la propriété *category* qui est une énumération de la classe *ClassCategory* (canonical pour une classe canonique et noncanonical pour une classe non canonique).
- *SolutionClass*. Cette classe représente une classe ontologique définie uniquement par son nom (*name*).

– Conception des plug-ins Différents plug-ins ont été conçus pour l'implémentation de notre outil comme suit :

- *model*. Ce plug-in contient les modèles de données (modèle de l'ontologie globale,

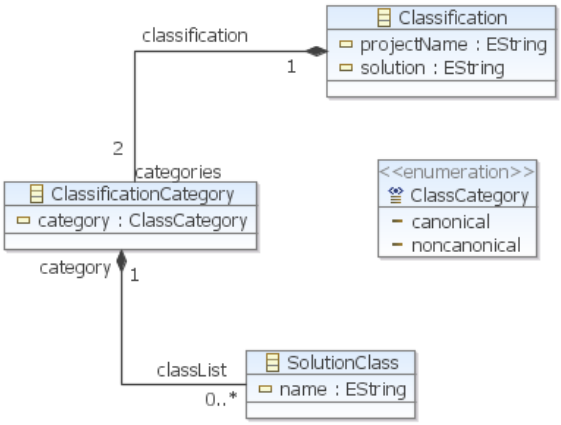

FIGURE 7.18 – Modèle statique de classification

modèle de l'ontologie locale, etc.). Permettant la création d'objets métier, il est utilisé par tous les autres plug-ins qui manipulent ce type d'objets.

– *api*. Pour des raisons d'ouverture et d'évolution, deux API ont été développées. La première est dédiée à l'exportation de l'ontologie locale vers un autre modèle d'ontologie et la génération de script dans un langage de manipulation de $\mathcal{BDBO}$. La deuxième api est plutôt dédiée à la génération du script SQL. Le plug-in *api* utilise seulement les modèles du plug-in *model*.
– *algorithm*. Ce plug-in contient l'implémentation des différents algorithmes de la méthodologie de conception de $\mathcal{BDBO}$ (extraction de l'ontologie globale, création de l'ontologie locale, algorithme de canonicité, algorithme de normalisation, algorithmes de génération de scripts, etc.). Il utilise les modèles définis dans le plug-in *model* et les API du plug-in *api*.
– *common*. Il contient les éléments communs aux plug-ins *global*, *local* et *logical*. Il s'agit par exemple des modèles d'interface homme-machine (modèle statique de diagramme, etc.). De plus, il gère la création et la suppression de projets de conception de $\mathcal{BDBO}$. Notons que *common* dépend des plug-ins *model* et *algorithm*.
– *global*. Ce plug-in gère l'ontologie globale. Il sert d'interface pour l'interaction homme-machine lors de l'importation de l'ontologie globale. Ainsi il gère l'importation, l'affichage et la sauvegarde de l'ontologie globale.
– *local*. Ce plug-in gère l'ontologie locale. Il sert d'interface pour l'interaction homme-machine lors de la création, l'édition et la sauvegarde de l'ontologie locale. Il reçoit une ontologie globale via le plug-in *global* servant de base pour la création de l'ontologie locale.
– *logical*. Ce plug-in gère la génération, l'affichage et la sauvegarde des modèles logique et physique. Il reçoit une ontologie locale via le plug-in *local* pour générer les modèles logique et physique.

La figure 7.19 met en évidence les relations de dépendance entre ces différents plug-ins.

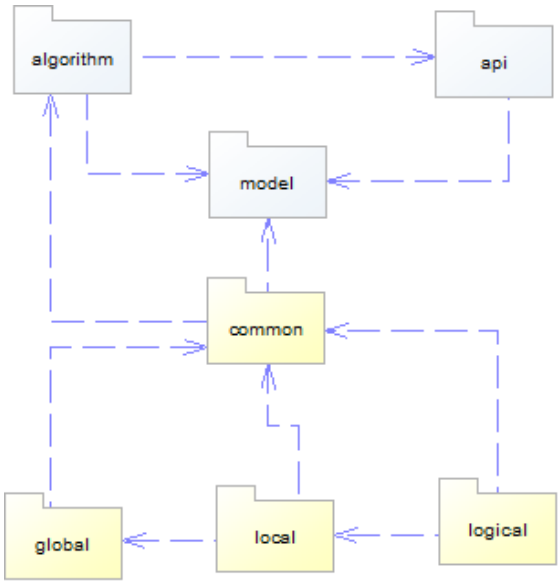

FIGURE 7.19 – Relations de dépendance entre les plug-ins

4.1.4 Génération du code

Nous générons le code java associé aux modèles définis lors de la phase d'analyse conceptuelle en utilisant l'outil de modélisation EMF. Pour chaque modèle défini, nous définissons un modèle de génération permettant de définir l'emplacement du code généré correspondant au modèle manipulé. A partir de ce modèle, nous générons par la suite les fichiers java structurés en trois différents packages. Le premier, portant le même nom que le modèle (par exemple modele), définit les interfaces et une fabrique (Factory) pour la création des instances des classes du modèle. Le deuxième (modele.impl) contient les implémentations des interfaces du package précédent. Le troisième package (modele.util) contient des utilitaires tels que *AdapterFactory*. La figure 7.21 résume le processus de génération de code avec EMF.

4.2 Implantation de la méthodologie sous OntoDB

Afin de valider notre approche de conception des $\mathcal{BDBO}$, nous proposons dans cette partie son implantation sous OntoDB. Pour ce faire, nous proposons d'introduire l'environnement de travail utilisé suivi de l'architecture générale du processus de validation sous OntoDB. Enfin, nous décrivons les principaux développements réalisés.

4.2.1 Environnement de travail

Afin de mener notre validation, nous avons utilisé :

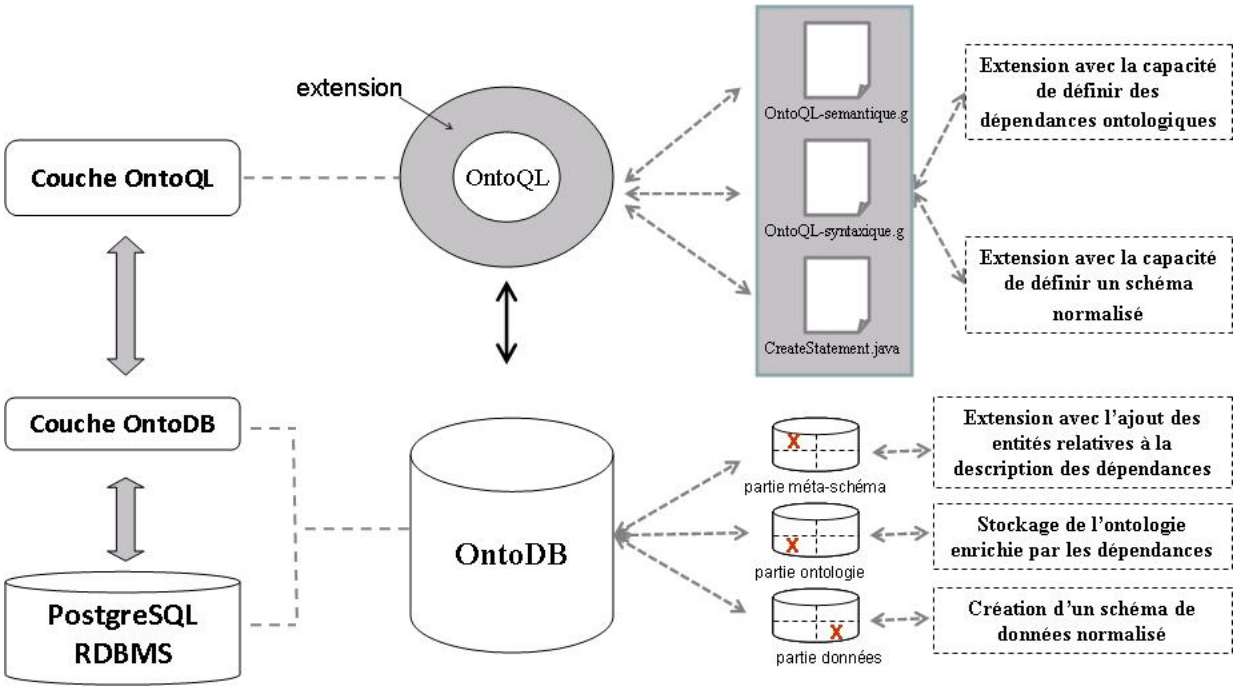

FIGURE 7.20 – Architecture générale du processus de validation sous OntoDB.

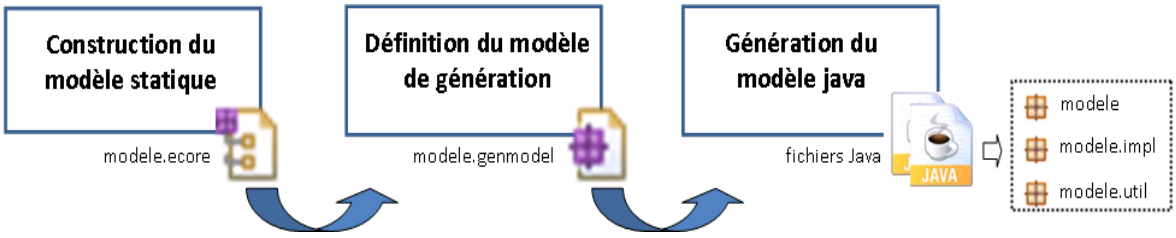

FIGURE 7.21 – Processus de génération de code

– la *BDBO* OntoDB (voir chapitre 2.5.2.5). Celle-ci est utilisée pour le déploiement de notre méthodologie de conception. Rappelons qu'OntoDB utilise le SGBD PostgreSQL et qu'elle se compose de quatre parties : (1) la partie méta-base, (2) la partie méta-schéma, (3) la partie ontologie et (4) la partie données.
– le langage OntoQL (voir chapitre 2.3.4.3). Celui-ci est utilisé pour l'extension du méta-schéma d'OntoDB, le stockage de l'ontologie dans sa partie ontologie et la définition du schéma relationnel en troisième forme normale.

4.2.2 Architecture générale du processus de validation sous OntoDB

Telle qu'illustrée dans la figure 7.20, l'architecture générale du processus de validation sous OntoDB repose sur deux principales couches : la couche OntoQL et la couche OntoDB. En effet, nous avons étendu la couche OntoQL par l'extension de sa grammaire (OntoQL-syntaxique.g) ainsi que de sa sémantique (OntoQL-semantique.g) afin de permettre la définition des dépendances ontologiques et de permettre la définition du schéma logique normalisé. Exécutée sur OntoDB, cette couche va permettre l'extension de la *BDBO* avec :

– l'ajout des entités relatives à la description des dépendances dans la partie méta-schéma ;
– le stockage des dépendances dans la partie ontologie ;
– la création d'un schéma de données normalisé dans la partie données.

4.2.3 Développements réalisés

Dans ce paragraphe, nous présentons les principaux développements réalisés pour l'implantation de la méthodologie sous OntoDB

1. **Définition des clauses.** Afin de créer les clauses définies dans la section 3, nous avons étendu la grammaire ainsi que la sémantique d'OntoQL (voir la figure 7.22) comme suit.

   Dépendance entre les classes ontologiques.

```
-- syntaxe
ddlCD
: CD^ OF! OPEN! identifier(COMMA! identifier)
* CLOSE!;

-- Sémantique
createStatement
: #(CREATE { beforeStatement( "create", CREATE ); }
(ddlCD) )
 {beforeStatementCompletion( "create" );
  postProcessCreate( #createStatement );
afterStatementCompletion( "create" );
```

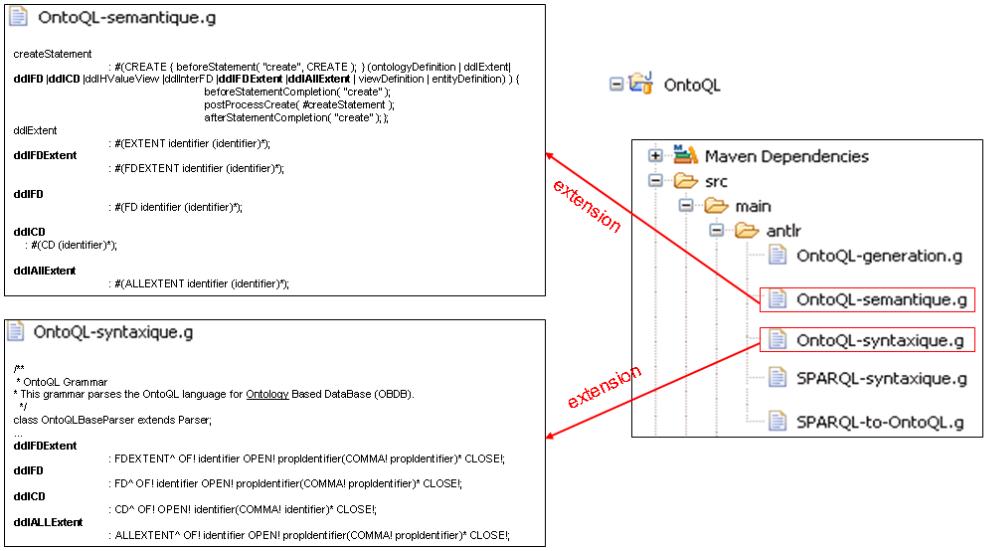

FIGURE 7.22 – Extension du langage OntoQL

```
};
ddlCD
    : #(CD (identifier)*);
```

Dépendance entre les propriétés ontologiques

```
-- syntaxe
ddlFD
: FD^ OF! identifier OPEN! propIdentifier(COMMA! propIdentifier)
* CLOSE!;

-- Sémantique
createStatement
: #(CREATE { beforeStatement( "create", CREATE ); }
(ddlFD) )
 {beforeStatementCompletion( "create" );
  postProcessCreate( #createStatement );
afterStatementCompletion( "create" );
};
ddlFD
: #(FD identifier (identifier)*);
```

Extension d'une classe ayant des dépendances fonctionnelles.

```
-- syntaxe
ddlFDExtent
: FDEXTENT^ OF! identifier OPEN! propIdentifier(COMMA! propIdentifier)
```

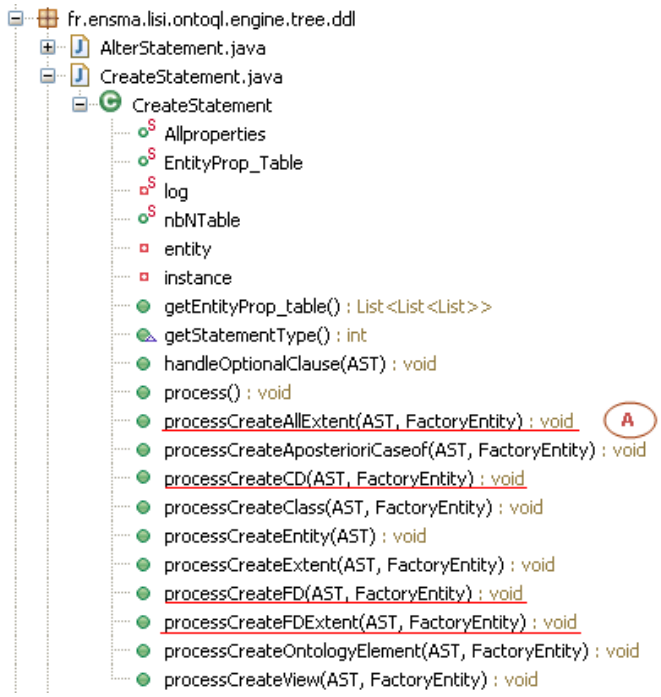

FIGURE 7.23 – Implémentation des clauses dans OntoQL

```
* CLOSE!;

-- Sémantique
createStatement
: #(CREATE { beforeStatement( "create", CREATE ); }
(ddlFDExtent) )
 {beforeStatementCompletion( "create" );
  postProcessCreate( #createStatement );
afterStatementCompletion( "create" );
};

ddlFDExtent
: #(FDEXTENT identifier (identifier)*);
```

Extension générique d'une classe.

```
-- syntaxe
ddlALLExtent
: ALLEXTENT^ OF! identifier OPEN! propIdentifier(COMMA! propIdentifier)
* CLOSE!;

-- Sémantique
```

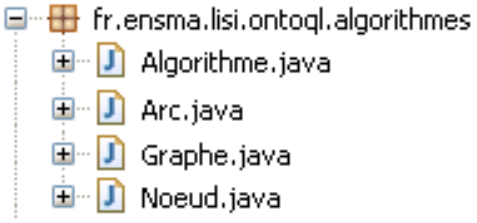

FIGURE 7.24 – Structure du package contenant les algorithmes implémentés

```
createStatement
: #(CREATE { beforeStatement( "create", CREATE ); }
(ddlAllExtent) )
 {beforeStatementCompletion( "create" );
  postProcessCreate( #createStatement );
afterStatementCompletion( "create" );
};
ddlAllExtent
: #(ALLEXTENT identifier (identifier)*);
```

Une fois la grammaire et la sémantique définies, l'implémentation de telles clauses est effectuée par la définition d'un ensemble de méthodes décrites avec le langage java. La figure 7.23 décrit les modifications adéquates effectuées sur OntoQL où chaque méthode soulignée présente un processus de création implémenté associé à une clause définie. Par exemple, la méthode *ProcessCreateCD* implémente la clause permettant la création d'une dépendance entre les classes ontologiques.

2. **Implémentation des algorithmes.** Nous avons défini un nouveau package noté *fr.ensma.lisi.ontoql.algorithmes* (voir figure 7.24) qui regroupe toutes les classes des algorithmes implémentés au cours de notre travail. Rappelons que deux algorithmes ont été traité : l'algorithme de la canonicité et l'algorithme de synthèse adapté aux ontologies. Ces derniers présentent le corps de notre application. Ils illustrent deux principales étapes de notre méthodologie de conception : (1) le typage des classes ainsi que (2) la génération du modèle logique de données en troisième forme normale.
3. **Construction du schéma de la base OntoDB.** Le déploiement de notre méthodologie sur OntoDB nécessite : (1) l'extension de son méta-schéma, (2) le stockage de l'ontologie manipulée et (3) la construction du schéma relationnel en troisième forme normale. Pour ce faire, un ensemble de développements est réalisé sur OntoQL. Notons que ces derniers sont exécutés sur OntoDB. En ce qui concerne la première étape, nous avons défini deux méthodes permettant l'extension du méta-modèle avec les dépendances entre classes (MMCFDCreate(statement)) et les dépendances définies entre les propriétés (MMPFDCreate(statement)). La figure 7.25 illustre la description de la deuxième méthode.

```
// Extend the Meta-model with property dependencies
public void MMPFDCreate(OntoQLStatement statement) throws JOBDBCException, ClassNotFoundException{
    // TODO Auto-generated method stub
    queryOntoQL = new StringBuffer();
    queryOntoQL.append("CREATE ENTITY #Right_Part(#oid int, #ItsRightProperty REF (#Property))");
    statement.executeUpdate(queryOntoQL.toString());
    queryOntoQL = new StringBuffer();
    queryOntoQL.append("CREATE ENTITY #Left_Part(#oid int, #ItsLeftProperties REF(#Property)ARRAY)");
    statement.executeUpdate(queryOntoQL.toString());
    queryOntoQL = new StringBuffer();
    queryOntoQL.append("CREATE ENTITY #Functional_Dependency (#oid int, #ItsClass REF (#Class)," +
            " #ItsRightPart REF (#Right_Part), #ItsLeftPart REF (#Left_Part))");
    statement.executeUpdate(queryOntoQL.toString());
    queryOntoQL = new StringBuffer();
}
```

FIGURE 7.25 – Description de la méthode MMPFDCreate

Afin de mener notre validation, nous avons traité le cas de l'ontologie des universités LUBM (voir chapitre4.4.11). Le stockage de l'ontologie implique la création des classes, des propriétés, des dépendances et la spécification des relations de subsomption. Par exemple, l'exécution du code suivant permet la création et le stockage de la dépendance $FD = \{Student; \{NSS\} \rightarrow name\_Student\}$ dans OntoDB.

```
...
queryOntoQL = new StringBuffer();
queryOntoQL.append("CREATE FD OF
Student (name_Student,NSS)");
statement.executeUpdate(queryOntoQL.toString());
```

En ce qui concerne le schéma des données, pour chaque classe ontologique nous exécutons la clause d'extension de classe générant ainsi le schéma de données normalisé dans la partie *données* d'OntoDB. Cette clause invoque la fonction *processCreateAllExtent(AST,FactoryEntity)* (figure 7.23.A) implémentant le processus de génération du schéma normalisé pour une classe ontologique donnée. Par exemple, le code suivant permet l'extension de la classe des étudiants *Student* en générant trois tables normalisées (selon l'algorithme de normalisation) et une vue sur ces tables.

```
...
queryOntoQL = new StringBuffer();
queryOntoQL.append("CREATE AllEXTENT OF
Student(NSS,name_Student,S_region,level,
     S_city,departement,pays,takes_DC,authorOf)");
```

5 Conclusion

Afin de faciliter le processus de conception, un outil d'aide à la conception des bases de données à base de modèles à partir d'une ontologie conceptuelle est proposé. Nommé OBDB-DesignTool, cet outil permet d'accompagner le concepteur durant les différentes étapes de modélisation : conceptuelle, logique et physique. Basé sur les algorithmes de canonicité et de normalisation, il offre au concepteur une interface graphique lui permettant de concevoir une base de données normalisée à partir d'une ontologie conceptuelle afin d'améliorer la qualité (a) du schéma de la base de données et (b) des données stockées (redondance, inconsistance, etc.). Dans un deuxième temps, une validation de notre approche de conception dans un environnement de bases de données à base ontologique est présentée. Une implémentation d'une telle approche sous OntoDB a été réalisé. Elle étend le langage OntoQL afin de permettre la génération d'un schéma de données normalisé (en 3FN) en tenant compte de la nature des classes (canonique, non canonique), des dépendances définies sur les classes ontologiques et des dépendances fonctionnelles définies sur les propriétés de chaque classe. Cette extension concerne essentiellement :

- la création d'une dépendance définie sur les classes ontologiques ;
- la création d'une dépendance fonctionnelle entre les propriétés ontologiques ;
- la création générique de l'extension d'une classe.

Dans la seconde partie de ce chapitre, nous avons présenté les principaux développements que nous avons réalisés pour la mise en œuvre de l'outil OBDBDesignTool. Ces derniers incluent (a) la définition des clauses, (b) l'implémentation des algorithmes et (c) la construction du schéma de la base OntoDB.

Conclusion et perspectives

Conclusion

Depuis les années 2000, les ontologies ont connu un réel succès dans différents domaines (Web sémantique, ingénierie, e-commerce, etc.). La quantité de données à base ontologique a considérablement augmenté ce qui a donné naissance à une nouvelle structure de stockage, appelée base de données à bases ontologiques ($\mathcal{BDBO}$). Ce type de base de données permet d'assurer la gestion simultanée des données et des ontologies dans le même référentiel. Ces $\mathcal{BDBO}$ se différencient principalement selon leurs architectures, leurs modèles de stockage, les langages d'interrogation (RQL, SPARQL, OntoQL, etc.) et les mécanismes utilisés pour définir le lien entre les instances et les concepts ontologiques. En revanche, elles figent toutes le modèle logique sans tenir compte de la phase de la modélisation conceptuelle. Par conséquent, la qualité des données représentées est médiocre. Des données redondantes ainsi que des formes d'inconsistances peuvent être identifiées.

Dans le but d'améliorer la qualité du schéma de la base de données et des données stockées, nous avons proposé dans nos travaux, une approche de conception et de déploiement des bases de données à base ontologique prenant en compte les bases fondamentales d'un cycle de vie de bases de données. Cette approche a été accompagnée par la proposition d'un outil d'aide à la conception aidant l'utilisateur durant le processus de conception.

Ainsi, les différentes contributions de ce travail sont les suivantes.

Analyse du concept d'ontologie et de ses structures de stockage

Les ontologies sont des modèles décrivant à la fois les données et leur sémantique dans un domaine donné. Ces données peuvent être de nature primitive (canonique) ou définie (non canonique). Des structures de données, appelées bases de données à base ontologique, ont été proposées pour la gestion et la persistance des ontologies et des données la référençant dans un même référentiel. Dans notre étude, nous avons présenté l'architecture globale d'une $\mathcal{BDBO}$

ainsi que les principaux modèles de stockage qu'elle utilise pour le stockage des ontologies et des données à base ontologique. Une analyse détaillée d'une panoplie de $\mathcal{BDBO}$ a été ensuite proposée. Celle-ci nous a permis de proposer une classification des $\mathcal{BDBO}$ selon leurs architectures d'une part et de leurs structures d'autre part. De plus, nous avons identifié les limites de ces bases de données quant au processus de persistance des données. Chaque $\mathcal{BDBO}$ admet une implémentation adoptant une représentation figée pour (a) la représentation des concepts ontologiques et (b) pour la représentation des instances. L'ontologie et ses données sont chargées directement dans la base de données cible. L'ontologie manipulée subit un chargement direct selon des règles d'insertion figées par la base de données cible. Aucun processus de conception n'est défini. L'absence d'une telle phase dans le cycle de vie des $\mathcal{BDBO}$ peut engendrer des lacunes dans le système. La redondance des données est tolérée et des formes d'inconsistances peuvent être identifiées. De plus, aucune méthodologie de déploiement ni d'optimisation de données n'est présentée. Seuls un chargement direct et une interrogation de données sont offerts.

Analyse des relations de dépendance et proposition de leur classification

Plusieurs modèles d'ontologies ont été développés dans différents domaines. Ces modèles offrent aux concepteurs des constructeurs, des fonctions et des procédures permettant de définir des concepts d'un domaine en termes de classes, de propriétés et de relations les reliant. En examinant les concepts ontologiques quant à leur définition, nous avons identifié des relations de dépendances les reliant. Ces relations sont exprimées essentiellement à travers des constructeurs que nous identifions et que nous qualifions de constructeurs de non canonicité. En se basant sur ces derniers, nous avons proposé des règles de génération de dépendances définies sur les classes ontologiques. De plus, notre analyse a donné lieu à une nouvelle classification des dépendances entre les concepts ontologiques.

Extension du modèle d'ontologies par les dépendances conceptuelles

Les ontologies conceptuelles sont riches en terme d'expressivité. Elles permettent de définir des concepts dérivés à partir d'expressions d'équivalence, d'opérateurs ensemblistes, de fonctions, etc. Ces définitions nous ont permis d'identifier des relations de dépendances entre les différents concepts ontologiques. Ainsi, nous avons proposé leur modélisation puis, nous avons enrichi le modèle d'ontologies par l'ajout des relations de dépendances entre les classes et entre les propriétés.

Exploitation des dépendances entre classes dans le processus de conception des $\mathcal{BDBO}$

Le traitement d'une ontologie conceptuelle non canonique requiert un traitement spécifique pour les classes non canoniques afin d'éviter la redondance dans la base de données. Afin de mener ce traitement, nous avons exploité les dépendances définies sur les classes ontologiques pour typer les classes en les différenciant en canoniques et non canoniques. Cette classification nous a permis de proposer une approche de conception de $\mathcal{BDBO}$ exploitant ces dépendances pour l'identification des classes ontologiques selon leur type (canonique ou non canonique) et l'association d'un traitement spécifique à chacun de ces deux types de classes.

Exploitation des dépendances fonctionnelles entre les propriétés dans le processus de conception des $\mathcal{BDBO}$

Dans une ontologie, un ensemble de propriétés dépendantes les unes des autres peut être associé à chaque classe ontologique. Afin d'exploiter ces relations de dépendance sur les propriétés ontologiques et leur impact dans le processus de normalisation, nous avons proposé une approche de conception des bases de données à base ontologique intégrant les dépendances fonctionnelles entre les propriétés dans la phase de la modélisation logique.

Définition d'une méthodologie de conception de $\mathcal{BDBO}$

Dans les bases de données sémantiques existantes, les ontologies sont rendues persistantes dans des structures dont les modèles de stockage de données sont figés. L'ontologie manipulée subit un chargement direct selon des règles d'insertion figées par la base de données cible. Aucun processus de conception n'est défini. L'absence d'un tel processus dans le cycle de vie des $\mathcal{BDBO}$ écarte le concepteur de la phase de conception et engendre des lacunes dans le système. Dans le but de répondre à ces insuffisances, nous avons proposé une méthodologie de conception des bases de données à base ontologique à partir d'une ontologie de domaine assurant l'intégration du concepteur dans les différentes phases du processus de conception. Dans les bases de données sémantiques existantes, les ontologies sont rendues persistantes dans des structures dont les modèles de stockage de données sont figés. L'ontologie manipulée subit un chargement direct selon des règles d'insertion figées par la base de données cible. Aucun processus de conception n'est défini. L'absence d'une telle phase dans le cycle de vie des $\mathcal{BDBO}$ écarte le concepteur de la phase de conception et engendre des lacunes dans le système (redondance, inconsistance des données, etc.). Dans le but de répondre à ces insuffisances, nous avons proposé une méthodologie de conception des bases de données à base ontologique à partir d'une ontologie de domaine assurant l'intégration du concepteur dans les différentes phases du processus de conception. Inspirée des méthodologies de conception des bases de données

classiques, cette approche traite les différentes phases traditionnelles du processus de conception: la modélisation conceptuelle, logique et physique. Cette méthode considère les ontologies locales comme des modèles conceptuels et emprunte des techniques formelles issues à la fois de la théorie des graphes pour l'analyse de dépendance, du raisonnement déduit de la logique de description pour le placement des classes et de la théorie des bases de données relationnelles pour la création des vues relationnelles, définies sur des tables normalisées et associées aux classes canoniques. De plus, elle considère différents types de dépendances entre les concepts ontologiques à des niveaux différents de conception.

Définition d'une approche de déploiement de notre méthodologie de conception

Vues la diversité des architectures des $\mathcal{BDBO}$ et la variété de leurs modèles de stockage, nous avons proposé une méthodologie de déploiement de notre approche de conception prenant en compte ces deux principaux facteurs. Pour ce faire, nous avons étudié la complexité de la phase de déploiement et nous avons proposé une approche de déploiement dirigée par le concepteur. Celle-ci a pour but (i) de satisfaire les préférences du concepteur quant à ses choix portant sur les architectures et les modèles de stockages étudiés, (ii) de réduire la combinatoire des solutions possibles et (iii) de calculer la meilleur solution suivant un modèle de coût donné.

Extension du langage OntoQL

L'implémentation de notre méthodologie dans un environnement de $\mathcal{BDBO}$ sous OntoDB a nécessité l'extension du langage OntoQL. Ce langage d'exploitation des bases de données à base ontologique comporte à la fois un langage de définition de données (LDD) et un langage de manipulation de données (LMD). Nous avons enrichi son LDD par l'ajout des constructeurs nécessaires pour (1) la création des dépendances fonctionnelles entre les propriétés ontologiques, (2) la création des dépendances définies sur les classes ontologiques et (3) la génération du modèle logique de données en troisième forme normale.

Proposition d'un outil d'aide à la conception des bases de données à partir d'une ontologie conceptuelle

Dans le but d'accompagner le concepteur durant le processus de conception des bases de données à partir d'une ontologie de domaine, nous avons proposé un outil d'aide à la conception, appelé OBDBDesignTool, facilitant aux utilisateurs la définition des modèles conceptuel, logique et physique.

Perspectives

Les travaux présentés dans ce mémoire laissent envisager de nombreuses perspectives tant à caractère théorique que pratique. Dans cette section, nous présentons brièvement celles qui nous paraissent être les plus intéressantes.

Intégration de la complétude calculatoire dans les dépendances ontologiques

Dans notre approche, la hiérarchie des concepts non canoniques est calculée à l'aide d'un raisonneur avant la phase de déploiement dans la $\mathcal{BDBO}$. Une fois l'ontologie rendue persistante, il n'est plus possible de raisonner sur la hiérarchie des concepts ontologiques. Une modification ou une mise à jour affectant la structure de l'ontologie nécessite la reproduction entière du processus de construction de la $\mathcal{BDBO}$. Afin d'éviter cela, des mécanismes doivent être mis en place. Par exemple, il serait utile de disposer d'opérateurs permettant de faire des raisonnements sur l'ontologie dans la $\mathcal{BDBO}$. Dans le but d'offrir la possibilité de raisonner sur l'ontologie stockée dans la $\mathcal{BDBO}$, une extension des $\mathcal{BDBO}$ avec la sémantique comportementale (complétude calculatoire) a été initiée dans le cadre des travaux de thèse de Youness Bazhar. Cette extension offre la possibilité de définir des opérateurs qui peuvent être implémentés par des programmes externes, des Services Web ou bien peuvent invoquer des raisonneurs sémantiques externes. En particulier, des opérateurs tels que l'union, l'intersection, etc. peuvent être définis permettant ainsi de calculer les concepts ontologiques non canoniques sans avoir à utiliser directement un raisonneur sémantique. Par conséquent, la contrainte de précédance entre les étapes de notre approche (en particulier le placement et le déploiement) est relâchée. De plus, les concepts ontologiques peuvent être à la fois calculés par des mécanismes internes de bases de données ainsi que par des mécanismes externes tels que des services Web.

Utilisation des dépendances ontologiques pour la réconciliation et la fusion de données

Le développement spectaculaire des bases de données sémantiques peut engendrer le besoin de leur intégration. Il est à noter que les $\mathcal{BDBO}$ sont intégrées en relâchant la contrainte d'existence d'un identifiant commun ce qui peut entraîner de la redondance dans la source cible. Au laboratoire LIAS et dans le cadre des travaux de thèse de Abdelghani Bakhtouchi, l'extension des modèles d'ontologies par les dépendances conceptuelles a été exploitée pour proposer une méthode complète de réconciliation et de fusion de données. A l'aide des relations de dépendances fonctionnelles définies sur les propriétés ontologiques, le concepteur génère toutes les clés candidates pour chaque classe de l'ontologie. En conséquence, un concepteur de source peut choisir sa ou ses clés primaires à partir des clés candidates communes entre les sources. Cette clé est utilisée par la suite pour détecter les instances se référant à la même entité du monde

réel (doublons) ce qui permettra une réconciliation exacte des données venant des sources n'utilisant pas un identifiant commun.

Étude de l'héritage des dépendances dans une ontologie locale

Dans notre approche, nous supposons l'existence d'une ontologie de domaine intégrant les dépendances conceptuelles et en particulier les dépendances fonctionnelles définies sur les propriétés ontologiques. Lors de la définition de l'ontologie locale, le concepteur peut choisir un ensemble restreint de propriétés ontologiques. En ce qui concerne les dépendances fonctionnelles, l'importation est plus délicate. Dans certains cas, le processus d'importation présente des difficultés de choix. Par exemple, si on a une dépendance a,b,d $\rightarrow$ c, où a,b, c et d sont des propriétés ontologiques globales et si le concepteur a choisi de n'importer que les propriétés a,b et c, nous nous posons la question si la représentation totale ou partielle (a,b $\rightarrow$ c) de la dépendance dans l'ontologie locale aurait un sens. Ainsi, dans le cadre d'un sujet de Master de recherche, des travaux portant sur l'étude de l'héritage des dépendances dans une ontologie locale sont en cours.

Optimisation des bases de données à base ontologique

Le cycle de vie des bases de données sémantiques peut être enrichi davantage par une nouvelle étape: l'optimisation. Cette phase a pour objectif de réduire le temps d'exécution de la charge des requêtes. Ceci peut être réalisé par l'utilisation d'index, des vues matérialisées et/ou du partitionnement (fragmentation horizontale, fragmentation verticale, fragmentation mixte). Une autre thèse a été initiée au LIAS pour intégrer au cycle de vie proposé la phase d'optimisation des bases de données à base ontologique à l'aide des vues matérialisées et son impact sur la représentation et l'interrogation des données ontologiques.

Bibliographie

[Agrawal et al., 2001] Agrawal, R., Somani, A., and Xu, Y. (2001). Storage and querying of e-commerce data. In *VLDB*, pages 149–158.

[Alexaki et al., 2001a] Alexaki, S., Christophides, V., Karvounarakis, G., and Plexousakis, D. (2001a). On storing voluminous rdf descriptions: The case of web portal catalogs. In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB'01)*, pages 43–48.

[Alexaki et al., 2001b] Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., and Tolle, K. (2001b). The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In *Proceedings of the Second International Workshop on the Semantic Web (SemWeb)*.

[Armstrong, 1974] Armstrong, W. W. (1974). Dependency structures of data base relationships. pages 580–583. In IFIP Congress.

[Bancilhon et al., 1992] Bancilhon, F., Delobel, C., and Kanellakis, P. C., editors (1992). *Building an Object-Oriented Database System, The Story of O2*. Morgan Kaufmann.

[Belaid et al., 2009] Belaid, N., Aït Ameur, Y., and Rainaud, J. F. (2009). A semantic handling of geological modeling workflows. In *International ACM Conference on Management of Emergent Digital EcoSystems*, pages 83–90.

[Bellatreche et al., 2010] Bellatreche, L., Aït Ameur, Y., and Chakroun, C. (2010). A design methodology of ontology based database applications. In *Logic Journal of the IGPL*.

[Bellatreche et al., 2003] Bellatreche, L., Pierra, G., Xuan, D. N., and Dehainsala, H. (2003). An automated information integration technique using an ontoly-based database approach. In *ISPE CE*, pages 217–223.

[Bijan, 2004] Bijan, S. E. P. (2004). Pellet: An owl dl reasoner. In *International Workshop on Description Logics (DL2004)*, pages 6–8.

[Borgida and Weddell, 1997] Borgida, A. and Weddell, G. E. (1997). Adding uniqueness constraints to description logics (preliminary report). In *Deductive and Object-Oriented Databases, 5th International Conference (DOOD'97)*, pages 85–102.

[Broekstra and Kampman, 2004] Broekstra, J. and Kampman, A. (2004). Serql: An rdf query and transformation language.

[Broekstra et al., 2002] Broekstra, J., Kampman, A., and Harmelen, F. V. (2002). Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International Semantic Web Conference*, pages 54–68.

[Calbimonte et al., 2009] Calbimonte, J. P., Porto, F., and Keet, C. M. (2009). Functional dependencies in owl abox. In *Brazilian Symposium on Databases (SBBD)*, pages 16–30.

[Calì et al., 2002] Calì, A., Calvanese, D., Giacomo, G. D., and Lenzerini, M. (2002). On the expressive power of data integration systems. In *21st International Conference on Conceptual Modeling (ER)*, pages 338–350.

[Calvanese et al., 2008] Calvanese, D., Giacomo, G. D., Lembo, D., Lenzerini, M., and Rosati, R. (2008). Path-based identification constraints in description logics. pages 231–241.

[Calvanese et al., 2001] Calvanese, D., Giacomo, G. D., and Lenzerini, M. (2001). Identification constraints and functional dependencies in description logics. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'2001)*, pages 155–160.

[Carroll et al., 2004] Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: implementing the semantic web recommendations. In *13th international conference on World Wide Web (WWW'2004)*, pages 74–83.

[Chen, 1975] Chen, P. P. (1975). The enity-relationship model: Toward a unified view of data. In *VLDB*, page 173.

[Chong et al., 2005] Chong, E. I., Das, S., Eadon, G., and Srinivasan, J. (2005). An efficient sql-based rdf querying scheme. In *VLDB*, pages 1216–1227.

[Codd, 1970] Codd, E. F. (1970). A relational model of data for large shared data banks. volume 13, pages 377–387.

[Das et al., 2004] Das, S., Chong, E. I., Eadon, G., and Srinivasan, J. (2004). Supporting ontology-based semantic matching in rdbms. In *VLDB*, pages 1054–1065.

[Dehainsala et al., 2007] Dehainsala, H., Pierra, G., and Bellatreche, L. (2007). Ontodb: An ontology-based database for data intensive applications. In *DASFAA*, pages 497–508.

[del Mar Roldán García et al., 2005] del Mar Roldán García, M., Delgado, I. N., and Montes, J. F. A. (2005). A design methodology for semantic web database-based systems. In *Third International Conference on Information Technology and Applications (ICITA)*, pages 233–237.

[Fankam et al., 2009] Fankam, C., Bellatreche, L., Hondjack, D., Ameur, Y. A., and Pierra, G. (2009). Sisro, conception de bases de données à partir d'ontologies de domaine. *Technique et Science Informatiques*, 28(10).

[Fensel et al., 2001] Fensel, D., v. Harmelen, F., Horrocks, I., McGuinness, D. L., and Patel-Schneider, P. F. (2001). Oil: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45.

[Gruber, 1993a] Gruber, T. R. (1993a). A translation approach to portable ontology specifications. In *Knowledge Acquisition*, volume 5, pages 199–220.

[Gruber, 1993b] Gruber, T. R. (1993b). A translation approach to portable ontology specifications. In *Chapter: Towards principles for the design of ontologies used for knowledge sharing*. Kluwer Academic Publisher.

[Guo et al., 2005] Guo, Y., Pan, Z., and Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *Web Semantics*, 3(2-3):158–182.

[Harris and Gibbins, 2003] Harris, S. and Gibbins, N. (2003). 3store: Efficient bulk rdf storage. In *PSSS*.

[Hondjack, 2007] Hondjack, D. (2007). Explicitation de la sémantique dans les bases de données: Le modèle ontodb de bases de données à base ontologique. In *PhD thesis,LISI/ENSMA et Université de Poitiers*.

[Horrocks and Patel-Schneider, 2003] Horrocks, I. and Patel-Schneider, P. F. (2003). Reducing owl entailment to description logic satisfability. In *Description Logics*.

[Horrocks et al., 2003] Horrocks, I., Patel-Schneider, P. F., and van Harmelen, F. (2003). From shiq and rdf to owl: the making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26.

[IEC61360-4, 1999] 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. In *Technical report, International Standards Organization*.

[Jean et al., 2006a] Jean, S., Ameur, Y. A., and Pierra, G. (2006a). Querying ontology based databases - the ontoql proposal. In *Proceedings of the 18th International Conference on Software Engineering & Knowledge Engineering (SEKE'2006)*, pages 166–171.

[Jean et al., 2007] Jean, S., Dehainsala, H., Nguyen Xuan, D., Pierra, G., Bellatreche, L., and Aït-Ameur, Y. (2007). Ontodb: It is time to embed your domain ontology in your database. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07) (Demo Paper)*, pages 1119–1120.

[Jean et al., 2006b] Jean, S., Pierra, G., and Ameur, Y. A. (2006b). Domain ontologies: A database-oriented analysis. In *WEBIST (1)*, pages 341–351.

[Karvounarakis et al., 2002] Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., and Scholl, M. (2002). Rql: a declarative query language for rdf. In *WWW*, pages 592–603.

[Lenzerini, 2002] Lenzerini, M. (2002). Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 233–246.

[Lu et al., 2007] Lu, J., Ma, L., Zhang, L., Brunner, J. S., Wang, C., Pan, Y., and Yu, Y. (2007). Sor: A practical system for ontology storage, reasoning and search. In *VLDB*, pages 1402–1405.

[Mbaiossoum et al., 2013] Mbaiossoum, B., Bellatreche, L., and Jean, S. (2013). Towards performance evaluation of semantic databases management systems. In *proceeding of the 29th British National Conference on Databases (BNCOD)*.

[Motik et al., 2009] Motik, B., Horrocks, I., and Sattler, U. (2009). Bridging the gap between owl and relational databases. *Journal of Web Semantics*, 7(2):74–89.

[Pan and Heflin, 2003] Pan, Z. and Heflin, J. (2003). Dldb: Extending relational databases to support semantic web queries. In *The First International Workshop on Practical and Scalable Semantic Systems*.

[Pan et al., 2008] Pan, Z., Zhang, X., and Heflin, J. (2008). Dldb2: A scalable multi-perspective semantic web repository. In *International Conference on Web Intelligence*, pages 489–495.

[Park et al., 2007] Park, M. J., Lee, J. H., Lee, C. H., Lin, J., Serres, O., and Chung, C. W. (2007). An efficient and scalable management of ontology. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, volume 4443 of *Lecture Notes in Computer Science*. Springer.

[Petrini and Risch, 2007] Petrini, J. and Risch, T. (2007). Sward: Semantic web abridged relational databases. In *DEXA Workshops*, pages 455–459.

[Pierra, 2000] Pierra, G. (2000). Représentation et échange de données techniques. In *Mec. Ind.*, volume 1, pages 397–414.

[Pierra, 2003] Pierra, G. (2003). Context-explication in conceptual ontologies: the plib approach. In *Proceedings of the 10th ISPE International Conference on Concurrent Engineering (ISPE CE 2003)*, pages 243–253.

[Pierra, 2008] Pierra, G. (2008). Context representation in domain ontologies and its use for semantic integration of data. *Journal on Data Semantics*, 10:174–211.

[Pierra et al., 2005] Pierra, G., Hondjack, D., Ameur, Y. A., and Bellatreche, L. (2005). Bases de données à base ontologique. principe et mise en oeuvre. *Ingénierie des Systèmes d'Information*, 10(2):91–115.

[Prud'hommeaux and Seaborne,] Prud'hommeaux, E. and Seaborne, A. Sparql query language for rdf. *Available at* `http://www.w3.org/TR/rdf-sparql-query/`.

[Rochfeld, 1986] Rochfeld, A. (1986). Merise, an information system design and development methodology, tutorial. In *ER*, pages 489–528.

[Romero et al., 2009] Romero, O., Calvanese, D., Abelló, A., and Rodriguez-Muro, M. (2009). Discovering functional dependencies for multidimensional design. In *DOLAP*, pages 1–8.

[Soutou and Brouard, 2012] Soutou, C. and Brouard, F., editors (2012). *ULM 2 pour les bases de données*. Eyrolles.

[Spyns et al., 2002] Spyns, P., Meersman, R., and Jarrar, M. (2002). Data modelling versus ontology engineering. *SIGMOD Record*, 31(4):12–17.

[Stocker and Smith, 2008] Stocker, M. and Smith, M. (2008). Owlgres: A scalable owl reasoner. In *The Sixth International Workshop on OWL: Experiences and Directions*.

[Stonebraker and Moore, 1996] Stonebraker, M. and Moore, D. (1996). *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann.

[Sure et al., 2002] Sure, Y., Angele, J., and Staab, S. (2002). Ontoedit: Guiding ontology development by methodology and inferencing. In *Confederated International Conferences DOA, CoopIS and ODBASE*, pages 1205–1222.

[Tapucu et al., 2009] Tapucu, D., Diallo, G., Jean, S., Aït-Ameur, Y., Ünalir, M. O., and Belaidi, N. (2009). Définition et exploitation des préférences au niveau sémantique. In *Journées Francophones sur les Ontologies (JFO 2009)*, pages 29–36.

[Tian et al., 2002] Tian, F., DeWitt, D. J., Chen, J., and Zhang, C. (2002). The design and performance evaluation of alternative xml storage strategies. *SIGMOD Record*, 31(1):5–10.

[Wilkinson et al., 2003] Wilkinson, K., Sayers, C., Kuno, H. A., and Reynolds, D. (2003). Efficient rdf storage and retrieval in jena2. In *SWDB*, pages 131–150.

[Wu et al., 2008] Wu, Z., Eadon, G., Das, S., Chong, E. I., Kolovski, V., Annamalai, M., and Srinivasan, J. (2008). Implementing an inference engine for rdfs/owl constructs and user-defined rules in oracle. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008*, pages 1239–1248.

Publications

1. Chedlia Chakroun, Ladjel Bellatreche, Yamine Aït-Ameur, Nabila Berkani, Stéphane Jean, Be Careful When Designing Semantic Databases: Data and Concepts Redundancy, in proceeding of the 7th IEEE International Conference on Research Challenges in Information Science (RCIS), Paris, France, May, 2013.

2. Chedlia Chakroun, Ladjel Bellatreche, Yamine Aït-Ameur, It is time to propose a complete methodology for designing semantic databases, in proceedings of the 9th International Conference on Web Information Systems and Technologies (WEBIST), Aachen, Germany, May 2013

3. Youness Bazhar, Chedlia Chakroun, Yamine Aït-Ameur, Ladjel Bellatreche, Stéphane Jean, Extending Ontology-Based Database with Behavioral Semantics, in proceedings of the 11th international conference on Ontologies, DataBases, and Applications of Semantics (ODBASE), pages 879-896, Rome, Italy, September, 2012.

4. Abdelghani Bakhtouchi, Chedlia Chakroun, Ladjel Bellatreche, Yamine Aït-Ameur, Mediated Data Integration Systems using Functional Dependencies Embedded in Ontologies, Recent Trends in Information Reuse and Integration, pages 227-256, 2012.

5. Chedlia Chakroun, Ladjel Bellatreche, Yamine Aït-Ameur, The Role of Class Dependencies in Designing Ontology-based Databases, in proceedings of the 7th International IFIP Workshop on Semantic Web & Web Semantics (SWWS'2011), pages 444-453, Crete, Greece, October, 2011.

6. Ladjel Bellatreche, Yamine Aït-Ameur, Chedlia Chakroun, A design methodology of ontology based database applications, Logic Journal of the IGPL - Oxford University Press, volume 19, number 5, pages 648-665, 2011.

Table des figures

Liste des tableaux

Glossaire

API : Application Programming Interface

BD : Base de données
BDR : Base de données relationnelle
BDBO : Base de données à base ontologique
BDT : Base de données traditionnelle
BDO : Base de données objet
CC : Classe canonique
CNC : Classe non canonique
ED : Entrepôt de données
DC : Dépendance entre classes
PFD : Dépendance fonctionnelle entre propriétés

LIAS : Laboratoire d'Informatique et d'Automatique pour les Systèmes
LO : Ontologie Locale

MC : Modèle Conceptuel

OC : Ontologie Conceptuelle
OCC : Ontologie Conceptuelle Canonique
OCNC : Ontologie Conceptuelle Non Canonique
OL : Ontologie Linguistique
OWL : Web Ontology Language

PLIB : Parts Library - Norme ISO 13584
RDF : Ressource Description Framework

UML : Unified Modeling Language
URI : Uniform Resource Identifier

URL : Uniform Resource Locator

W3C : WorldWideWeb Consortium

Contribution à la définition d'une méthode de conception de bases de données à base ontologique

Présentée par :

Chedlia CHAKROUN

Directeurs de thèse :

Yamine AIT AMEUR et Ladjel BELLATRECHE

Résumé. Récemment, les ontologies ont été largement adoptées par différentes entreprises dans divers domaines. Elles sont devenues des composantes centrales dans bon nombre d'applications. Ces modèles conceptualisent l'univers du discours aux moyens de concepts primitifs et parfois redondants (calculés à partir de concepts primitifs). Au début, la relation entre ontologies et base de données a été faiblement couplée. Avec l'explosion des données sémantiques, des solutions de persistance assurant une haute performance des applications ont été proposées. En conséquence, un nouveau type de base de données, appelée base de données à base ontologique (BDBO) a vu le jour. Plusieurs types de BDBO ont été proposés, ils utilisent différents SGBD. Chaque BDBO possède sa propre architecture et ses modèles de stockage dédiés à la persistance des ontologies et de ses instances. A ce stade, la relation entre les bases de données et les ontologies devient fortement couplée. En conséquence, plusieurs études de recherche ont été proposées sur la phase de conception physique des BDBO. Les phases conceptuelle et logique n'ont été que partiellement traitées. Afin de garantir un succès similaire au celui connu par les bases de données relationnelles, les BDBO doivent être accompagnées par des méthodologies de conception et des outils traitant les différentes étapes du cycle de vie d'une base de données. Une telle méthodologie devrait identifier la redondance intégrée dans l'ontologie. Nos travaux proposent une méthodologie de conception dédiée aux bases de données à base ontologique incluant les principales phases du cycle de vie du développement d'une base de données : conceptuel, logique, physique ainsi que la phase de déploiement. La phase de conception logique est réalisée grâce à l'incorporation des dépendances entre les concepts ontologiques. Ces dépendances sont semblables au principe des dépendances fonctionnelles définies pour les bases de données relationnelles. En raison de la diversité des architectures des BDBO et la variété des modèles de stockage utilisés pour stocker et gérer les données ontologiques, nous proposons une approche de déploiement à la carte. Pour valider notre proposition, une implémentation de notre approche dans un environnement de BDBO sous OntoDB est proposée. Enfin, dans le but d'accompagner l'utilisateur pendant le processus de conception, un outil d'aide à la conception des bases de données à partir d'une ontologie conceptuelle est présenté.

Mots-clés : Base de données à base ontologique, Conception, Dépendances, Modélisation, Méta-modélisation, Ontologie.

……

Abstract. Recently, ontologies have been widely adopted by small, medium and large companies in various domains. They have become central components in many applications. These models conceptualize the universe of discourse by means of primitive and sometimes redundant concepts (derived from primitive concepts). At first, the relationship between ontologies and database was loosely coupled. With the explosion of semantic data, persistence solutions providing high performance applications have been proposed. As a consequence, a new type of database, called ontology-based database (OBDB) is born. Several types of OBDB have been proposed including different architectures of the target DBMS and storage models for ontologies and their instances. At this stage, the relationship between databases and ontologies becomes strongly coupled. As a result, several research studies have been proposed on the physical design phase of OBDB. Conceptual and logical phases were only partially treated. To ensure similar success to that known by relational databases, OBDB must be accompanied by design methodologies and tools dealing with the different stages of the life cycle of a database. Such a methodology should identify the redundancy built into the ontology. In our work, we propose a design methodology dedicated to ontology-based databases including the main phases of the lifecycle of the database development: conceptual, logical and physical as well as the deployment phase. The logical design phase is performed thanks to the incorporation of dependencies between concepts and properties of the ontologies. These dependencies are quite similar to the functional dependencies in traditional databases. Due to the diversity of the OBDB architectures and the variety of the used storage models (triplet, horizontal, etc.) to store and manage ontological data, we propose a deployment 'à la carte. To validate our proposal, an implementation of our approach in an OBDB environment on OntoDB is proposed. Finally, in order to support the user during the design process, a tool for designing databases from a conceptual ontology is presented.

Key Words : Semantic database, Modeling, Dependencies, Metamodeling, Design, Ontology.
