Intégration de bases de données hétérogènes par articulation à priori d'ontologies: application aux catalogues de composants industriels

Dung Xuan Nguyen

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique

Ecole Doctorale des Sciences Pour l'Ingénieur ▪ Université de Poitiers

THESE

pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITE DE POITIERS

(Faculté des Sciences Fondamentales et Appliquées)
(Diplôme National — Arrêté du 25 Avril 2002)

Ecole Doctorale : Science Pour l'Ingénieur
Secteur de Recherche : INFORMATIQUE et APPLICATION

Présentée par

Dung NGUYEN XUAN

Intégration de base de données hétérogènes par articulation *a priori* d'ontologies: application aux catalogues de composants industriels.

Directeurs de Thèse : **Ladjel BELLATRECHE, Guy PIERRA**

Soutenance prévue le : 22 Décembre 2006

Devant la commission d'Examen

JURY

Rapporteurs :	**Anne DOUCET**	Professeur, Université Pierre et Marie-Curie, Paris
	Chantal REYNAUD	Professeur, Université Paris Sud, Paris
Examinateurs :	**Yamine AIT-AMEUR**	Professeur, ENSMA, Futuroscope
	Ladjel BELLATRECHE	Maître de Conférences, Université de Poitiers, Poitiers
	Marie-Christine LAFAYE	Maître de Conférences, Université de La Rochelle, La Rochelle
	Guy PIERRA	Professeur, ENSMA, Futuroscope
	Michel SCHNEIDER	Professeur, Université Blaise Pascal, Clermont Ferrand

Remerciements

Je tiens à adresser mes plus sincères remerciements à Guy Pierra, Directeur du LISI et Chef de l'équipe d'Ingénierie des données pour m'avoir accepté dans son laboratoire et intégré dans son équipe et de m'avoir encadré et aidé tout au long de mes études en France.

Je tiens également à remercier Ladjel Bellatreche, pour avoir encadré mon travail de thèse. Je le remercie pour toute la confiance qu'il m'a témoignée, pour son entière disponibilité, et pour tous ses conseils avisés.

Je souhaiterais remercier Prof. Anne Doucet et Prof. Chantal Reynaud qui ont eu la lourde tâche de rapporter ma thèse, ainsi que les autres membres du jury, Prof. Yamine Ait-Ameur, Marie-Christine Lafaye et Prof. Michel Schneider, qui m'ont fait l'honneur d'accepter d'être examinateurs.

Je remercie Kim Son Doan pour ses conseils dans ma décision de m'orienter vers le milieu de la recherche.

J'adresse mes remerciements à tout le personnel du LISI, et plus particulièrement Hondjack, Stéphane, Loé, Ahmed, Idir, Dago, Manu, Eric, Guillaume, JCP, Fred C, Claudine, Hieu, Fred R, Nicolas, Chimène, Karim, Sybille, Malo, Youcef, Michaël, David pour leur présence et leur soutien cordial.

Merci à mes amis, particulièrement Vinh Anh, Co Hanh, Kienco, Tuangà, Minhthuy, Hoangvu, Nam-Minh, a.Hung, Patrice, Cédric, Antoine, Sylvain, V.Cuong pour leur amitié désintéressée.

Une tendre pensée va à mon beau-père qui, malheureusement, n'a pas pu voir l'aboutissement de mon travail.

Mes derniers remerciements et non les moindres, iront à mes proches, et en particulier à mon père, qui m'ont toujours apporté leur soutien sans faille. Je les remercie de toute l'affection et tout l'amour qu'ils m'ont témoignés.

A ma grande mère, mes parents et ma femme.

Table des matières

Chapitre 1

Introduction générale

1.1 Contexte

Avec le développement d'Internet et des intranets, l'échange et le partage de l'information provenant de diverses sources de données réparties, autonomes et hétérogènes deviennent un besoin crucial. Dans un tel contexte, il est souvent nécessaire pour une application d'accéder simultanément à plusieurs sources, du fait qu'elles contiennent des informations pertinentes et complémentaires. Pour ce faire, la solution des systèmes d'intégration a été proposée. Elle consiste à fournir une interface uniforme et transparente aux données pertinentes via un schéma global. Cette solution émerge dans une variété de situations commerciales (quand deux compagnies semblables doivent fusionner leurs bases de données) et scientifiques (intégration des résultats de recherche de différents laboratoires en bioinformatique). Plusieurs systèmes d'intégration ont été proposés dans la littérature : on peut citer TSIMMIS développé au département d'informatique de l'Université de Stanford, Picsel développé par l'Université Paris Sud, MOMIS développé dans l'université de Modena et Reggio Emilia et l'Université de Milan, etc.

Intégrer les sources de données dans le but de fournir aux utilisateurs une interface d'accès uniforme est une tâche difficile. Cette difficulté concerne trois aspects : (1) l'hétérogénéité des données, (2) l'autonomie des sources, et (3) l'évolution des sources.

L'hétérogénéité de données concerne à la fois la structure et la sémantique. L'hétérogénéité structurelle provient du fait que les sources de données peuvent avoir différentes structures et/ou différents formats pour stocker leurs données. De nombreuses approches pour résoudre ce type d'hétérogénéité ont été proposées dans les contextes des bases de données fédérées et des multi-bases de données. L'hétérogénéité sémantique, par contre, présente un défi majeur dans le processus d'élaboration d'un système d'intégration. Elle est due aux différentes interprétations des objets du monde réel. En effet, les sources de données sont conçues indépendamment, par des concepteurs différents, ayant des objectifs applicatifs différents. Chacun peut donc avoir un point de vue différent sur le même concept. Afin de réduire cette hétérogénéité, de plus en plus d'approches d'intégration

associent aux données des ontologies qui en définissent le sens . Ces approches sont dites "**à base ontologique**". Deux catégories principales d'ontologies ont été utilisées dans l'élaboration de systèmes d'intégration : *linguistiques* ou *formelles*. Les ontologies linguistiques, comme WordNet, visent à définir le sens des mots et les relations entre ces mots. Ces relations étant *approximatives* et fortement contextuelles elles permettent seulement une automatisation partielle du processus d'intégration, sous la supervision d'un expert humain. Les ontologies formelles constituent des spécifications explicites de conceptualisations de domaines. Elles permettent de définir formellement les concepts de ce domaine et les relations entre ces concepts. Quelques systèmes ont été développés autour de cette hypothèse comme le projet COIN pour échanger les données financières, le projet Picsel2 pour intégrer les services Web.

Il existe différentes utilisations possibles des ontologies dans un système d'intégration. Les approches les plus courantes sont des approches *a posteriori*. Lorsque des *ontologies linguistiques* sont utilisées, on part des mots d'une source, on se rapproche d'un thésaurus (dit également ontologie linguistique) et on essaye de comprendre à travers les mots l'identité des concepts. Ceci demande une forte implication de l'expert et le résultat est très incertain car les décisions sont subjectives. Lorsque, méthode plus récente, des ontologies formelles sont utilisées, on suppose que chaque source contient sa propre ontologie. Le problème d'intégration sémantique devient alors un problème d'intégration d'ontologies. Comparée à la première méthode, la deuxième est :

1. plus rigoureuse car les concepts sont définis avec plus de précision, l'appariement peut être plus argumenté,
2. plus facilement outillable, en particulier, si les différentes ontologies sont basées sur le même modèle, par exemple OWL, PLib, etc. Une correspondance entre ontologies peut alors être exploitée par des programmes génériques pour intégrer les données correspondantes.

Mais on peut également, et c'est la proposition que nous avons développée dans le cadre de cette thèse prendre un point de vue *a priori*. En effet, dans un nombre croissant de domaines, des ontologies de domaine consensuelles commencent à exister. C'est en particulier le cas dans de nombreux domaines techniques où des ontologies de domaines **normalisées** sont en train d'émerger. Dans ces domaines, il est alors fréquent que l'objectif de chaque administrateur de base de données soit de rendre le contenu de celle-ci aussi facilement accessible que possible à tout utilisateur extérieur. C'est, par exemple, le cas dans du commerce électronique, où chaque fournisseur souhaite que son catalogue électronique soit très largement consulté, exploité et compris. C'est également le cas, beaucoup plus largement, pour l'échange électronique professionnel (souvent appelé B2B). Dans les cas où, à la fois, il existe une ontologie de domaine et où l'objectif des administrateurs de bases de données est de rendre celles-ci le plus accessible possible, il est clair (1) que la solution est de permettre l'accès à travers l'ontologie de domaine et (2) que les administrateurs des différentes sources sont prêts à faire les efforts nécessaires *a priori* - c'est-à-dire

lorsqu'ils rendent disponibles leurs bases de données de façon à faciliter *ensuite* l'accès à leurs bases de données par leur utilisateur.

Cela a été bien compris par tous les sites de commerces électroniques professionnels qui ont élaboré une première approche qui consistait à imposer à chaque concepteur de base de données l'utilisation de l'ontologie partagée comme *schéma* de ses propres données. Cette approche présente deux inconvénients majeurs.

1. Elle interdit de représenter des concepts qui n'existent pas dans l'ontologie partagée. Or, par définition, une ontologie partagée ne définit **que** ce qui est partagée par toute une communauté. Chaque membre de la communauté, et c'est trivialement vrai pour le commerce électronique, souhaite **aussi** définir des concepts non partagés. C'est ce qui constitue l'avantage compétitif de chaque fournisseur.
2. Elle impose de calquer la structure de la base de données sur la structure de l'ontologie, alors que chaque utilisateur peut souhaiter utiliser des systèmes de gestion différents (par exemple, un système de gestion de base de données relationnelles quand le modèle d'ontologie est orienté objet) ou des structures de schémas différents (par exemple, optimisés par rapport au seul sous-ensemble de l'ontologie pertinent pour chaque cas particulier).

Pratiquement, cette approche s'est avérée à la fois extrêmement coûteuse et inefficace. Et ceci a causé la mort de la plupart des places de marché professionnel (par exemple, COVISIN dans l'automobile, TRADE RANGER dans le pétrole, etc.) L'approche était coûteuse car l'utilisation d'un schéma commun par les différents fournisseurs étant pratiquement irréalisable, elle imposait à chaque fournisseur de convertir ses données dans le schéma de la place de marché. Et comme il existait plusieurs places de marché, il fallait faire cela plusieurs fois sans qu'il n'y ait le moindre lien, ni schématique ni sémantique entre les différents schémas. L'approche était inefficace car elle ne permettait en aucune façon à chaque fournisseur de se singulariser en définissant des nouvelles catégories de produits ou en décrivant des caractéristiques originales de ses produits.

Si cette approche s'est avérée inefficace, elle montrait néanmoins que dans bon nombre de domaines, il est raisonnable de penser que les administrateurs de bases de données sont prêts à faire des efforts *a priori* pour faciliter l'accès à leur base de données, mais qu'en contre-partie, cela n'est acceptable que s'ils peuvent, dans leur base, conserver leur propre schéma. C'est l'objectif de l'approche proposé dans cette thèse.

Cette approche vise à permettre à la fois : (1) à chaque concepteur de choisir librement son schéma, et (2) au système d'intégration de réaliser l'intégration automatique des différentes bases de données. Cette approche, appelée approche d'intégration *a priori* par articulation d'ontologies, est fondée sur les deux principes suivants :

1. chaque base de données contient, outre ses données, son schéma et l'ontologie (locale) qui en définit le sens, et
2. chaque administrateur fait ensuite que, *a priori* [1] chaque ontologie locale référence

[1] c'est-à-dire avant que la base de données ne soit rendue disponible

"autant que cela est possible" l'ontologie partagée de domaine. Nous définissons précisément, au chapitre 4, ce que signifie "référencer autant que cela est possible" une ontologie partagée.

Nous proposons successivement deux approches pour ce référencement *a priori*.

1. Lorsque l'ontologie partagée est unique dans le domaine et pré-existe à la création de la base de données, le plus naturel est que le concepteur de la base de données utilise directement, dans sa propre ontologie, les éléments pertinents de l'ontologie partagée. C'est l'approche que nous proposons au chapitre 4 et qui permet l'intégration automatiquement de sources de données hétérogènes et autonomes au niveau de leur schéma. C'est la première contribution de cette thèse.

2. Lorsqu'il n'existe pas, ou lorsqu'il existe plusieurs ontologies partagées, ou encore lorsque la base de données pré-existe à l'ontologie partagée, l'approche précédente n'est pas réalisable. Nous proposons alors au chapitre 6 une deuxième approche permettant de représenter les correspondances entre ontologie locale et ontologie globale à *l'extérieur* de l'ontologie locale. Une même ontologie locale peut alors être appariée avec *plusieurs ontologies partagées*, et cet appariement peut évoluer au fur et à mesure que l'ontologie partagée évolue, et cela sans que l'ontologie locale ait jamais besoin d'être modifiée. C'est la deuxième méthode d'intégration que nous proposons au titre de cette thèse et qui permet de représenter des relations arbitrairement complexes entre ontologies. C'est la deuxième contribution de notre travail.

Notons que la présence des ontologies (partagée et locales) permet une automatisation du processus d'intégration, mais elle rend plus difficile la gestion de l'évolution. Cela est dû à la prise en considération d'une nouvelle dimension d'évolution, à savoir les ontologies. On note que les fournisseurs des sources sont différents et que chaque source se comporte indépendamment des autres. En conséquence, la relation entre le système intégré et ses sources est faiblement couplée. Chaque élément évolue de façon asynchrone. Une source peut modifier sa structure et sa population sans en informer les autres. L'ontologie commune (partagée) peut évoluer indépendamment des sources. Ceci engendre des anomalies de maintenance. Dans un tel environnement asynchrone, gérer l'évolution consiste :

1. au niveau de l'ontologie, à permettre une évolution asynchrone des différentes ontologies tout en conservant les relations inter-ontologies.

2. au niveau du contenu, à reconnaître une instance même si elle est décrite par des propriétés un peu différentes (des informations locales peuvent lui être attachées), et à se souvenir (éventuellement) à quelles périodes une instance était valide.

Notre troisième contribution présentée au chapitre 5 consiste alors à proposer un modèle pour la gestion de l'évolution asynchrone des systèmes d'intégration à base ontologique, dont le résultat est matérialisé sous forme d'un entrepôt.

1.2 Organisation de la thèse

Cette thèse est organisée autour de sept chapitres.

Le chapitre 2 est un état de l'art portant sur le problème de l'intégration de sources de données réparties, autonomes, évolutives et hétérogènes et sur l'apport des ontologies conceptuelles dans les projets de l'intégration de données. La contribution principale de cette synthèse est de proposer une classification des systèmes d'intégration existants. Cette classification se base sur les trois critères orthogonaux suivants : (1) la représentation de données intégrées (virtuelle ou matérielle), (2) le sens de la mise en correspondance entre schéma global et schémas locaux, et (3) la nature du processus d'intégration. A propos des ontologies conceptuelles, nous citons d'abord quelques définitions, puis nous comparons trois modèles d'ontologies. Les deux premiers modèles, OWL et F-Logic, sont des modèles orientés inférences. Le troisième, PLIB, est orienté intégration de données et caractérisation et modularité. Avant de conclure ce chapitre, nous présentons quelques systèmes d'intégration à base ontologique selon deux axes : l'intégration *a posteriori*, et *a priori*.

Le chapitre 3 est consacré à la présentation de l'ensemble du contexte dans lequel notre travail de thèse a été développé. Nous décrivons d'abord le modèle d'ontologie PLIB. Pour l'essentiel, ce modèle permet de définir de façon formelle et traitable par machine toutes les catégories d'objets du monde réel que l'on peut avoir besoin de manipuler lors d'une transaction informatique, ainsi que les propriétés qui les caractérisent ou décrivent leur état. Nous présentons ensuite un nouveau modèle de base de données qui vise à représenter, en plus des données, l'ontologie locale qui en représente le sens, le modèle conceptuel qui en définit la structure, et, éventuellement les correspondances existant entre l'ontologie locale et des ontologies partagées. Une telle base de données est dite *B*ase de *D*onnées à *B*ase *O*ntologique (BDBO). Le modèle d'ontologie et d'instances PLIB étaient définis formellement dans le langage EXPRESS. Ils permettent d'échanger le contenu de BDBO. Nous présentons alors le langage EXPRESS et l'ensemble de la technologie qui lui associée. Nous présentons en particulier les outils que nous avons utilisés afin de valider notre processus d'intégration, à savoir l'environnement ECCO (EXPRESS Compiler COmpiler) pour implémenter les APIs d'intégration, puis l'éditeur PLIB pour créer/gérer les ontologies PLIB et les BDBOs à base PLIB.

Le chapitre 4 présente notre approche d'intégration de données qui nous permet d'intégrer d'une façon automatique des BDBOs ayant des articulations sémantiques *a priori* avec une ontologie normalisée. Dans cette approche, trois scénarii d'intégration sont proposés : (1)*FragmentOnto* où on suppose que l'ontologie locale de chaque source est un sous ensemble de l'ontologie partagée ; (2)*ProjOnto* où chaque source définit sa propre ontologie en référençant "autant que cela est possible" l'ontologie partagée, mais où les instances de chaque source sont exportées comme des instances de l'ontologie partagée ; (3)*ExtendOnto* où chaque ontologie locale est définie comme dans le scénario *ProjOnto*, mais où l'on souhaite enrichir automatiquement l'ontologie partagée ; toutes les instances de don-

nées peuvent alors être intégrées, sans aucune modification, au sein du système intégré. Pour chaque scénario, nous décrivons d'abord son contexte appliqué, puis son algorithme d'intégration et enfin ses applications réelles possibles. Nous présentons également dans ce chapitre une implémentation validant l'approche proposée. Cette implémentation est effectuée sur le langage EXPRESS dans l'environnement ECCO et JAVA. Elle consiste à étendre l'outil PLIBEditor en permettant l'échange de données entre des catalogues de composants à base modèle PLIB.

L'objet du chapitre 5 est de présenter une approche permettant l'évolution asynchrone de l'ensemble de sources, tout en maintenant la possibilité d'intégration automatique de ces dernières au sein d'un entrepôt. Nous introduisons d'abord quelques travaux antérieurs concernant l'évolution des données et des ontologies. Nous proposons ensuite un modèle pour la gestion des évolutions ontologiques, appelé *modèle des versions flottantes*. L'hypothèse fondamentale autour de ce modèle, appelée *le principe de continuité ontologique*, stipule qu'une évolution d'une ontologie ne peut infirmer un axiome antérieurement vrai. Ce principe a pour effet de simplifier considérablement la gestion de l'évolution des ontologies. Une méthode de gestion du cycle de vie des instances qui évite toute duplication de données est ensuite présentée. Le système d'intégration résultant, validé par plusieurs implémentations, présente alors la double originalité de permettre l'autonomie et l'évolution asynchrone des sources, tout en assurant une intégration automatique de leurs contenus.

Le chapitre 6 est consacré au problème de l'intégration dans le cas où l'ontologie locale est construite indépendamment de toute ontologie partagée. Nous identifions d'abord les relations qu'il est nécessaire de représenter entre deux ontologies dans le but de les intégrer. Nous proposons alors de représenter ces relations par une technique de réification des opérateurs algébriques successible d'être étendu à n'importe quel type d'opérateur. Cette représentation est *neutre*, c'est à dire indépendante de tout langage et de tout environnement. Elle peut donc être représentée au sein d'une BDBO, et échangée avec le contenu de celle-ci. Nous présentons alors l'algorithme d'intégration qui permet d'intégrer automatiquement de telles sources de données.

Le chapitre 7 présente les conclusions générales de ce travail et esquisse diverses perspectives.

Publications

La liste suivante représente les publications concernant le travail dans cette thèse.

1. Bellatreche, L., **Nguyen-Xuan, D.**, Pierra G., Dehainsala, H., *Contribution of Ontology-based Data Modeling to Automatic Integration of Electronic Catalogues within Engineering Databases*, Computers in Industry Journal 57 (8-9), pp. 711-724, 2006.
2. **Nguyen-Xuan, D.**, Bellatreche, L., Pierra, G., *Ontology Evolution and Source Autonomy in Ontology-based Data Warehouses*, Revue des Nouvelles Technologies

de l'Information (EDA'2006), pp. 55-76, Juin, 2006.

3. **Nguyen-Xuan, D.**, Bellatreche, L., Pierra G., *A Versioning Management Model for Ontology-Based Data Warehouses*, in 8th International Conference on Data Warehousing and Knowledge Discovery (DaWaK'06), pp. 195-206, Lecture Notes in Computer Science (LNCS), September 2006.
4. **Nguyen-Xuan, D.**, Bellatreche, L., Pierra, G., *Un modèle a base ontologique pour la gestion de l'évolution asynchrone des entrepôts de données*, in Proceedings of Modélisation, Optimisation et Simulation des Systèmes : Défis et Opportunités (MOSIM'06), pp.1682-1691, avril, 2006, Rabat - Maroc (sélectionnée pour la revue ISI).
5. Dehainsala, H., Jean, S., **Nguyen-Xuan, D.**, Pierra, G., *Ingénierie dirigée par les modèles en EXPRESS : un exemple d'application*, Actes des premières journées d'Ingénierie dirigée par les modèles, IDM'05, pp. 155-174, 30 juin - 1er juillet, 2005.
6. Bellatreche, L., Pierra, G., **Nguyen-Xuan, D.**, Dehainsala, H., Aït Ameur, Y., *An a Priori Approach for Automatic Integration of Heterogeneous and Autonomous Databases*, in proceedings of the 8th International Conference on Databases and Expert Systems (DEXA'04), pp. 475-485, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Zaragosa, September 2004.
7. Bellatreche, L., Pierra G., **Nguyen-Xuan, D.**, Dehainsala, H., *Integration de sources de données autonomes par articulation a priori d'ontologies*, Actes du Congrès d'Informatique Des Organisations et Systèmes D'Information et de Décision (INFORSID'2004), pp. 283-298, May, 2004.
8. Bellatreche, L., Pierra G., **Nguyen-Xuan, D.**, Dehainsala, H., *An Automated Information Integration Technique using an Ontology-based Database Approach*, in Proceedings of Concurrent Engineering Conference, CE'2003 - Special Track on Data Integration in Engineering (DIE'2003), pp. 217-224, July 2003, Madera.

Chapitre 2

Etat de l'art

2.1 Introduction

Le succès de l'Internet et l'Intranet a fait exploser le développement des sources d'informations sur le Web. Intégrer des informations issues de sources autonomes, hétérogènes et évolutives est devenu un besoin crucial pour un nombre important d'applications, comme le commerce électronique, business intelligence (OLAP, fouille de données), la bioinformatique, etc. Cette intégration permet à ces applications d'exploiter cette mine d'information. Pour réaliser une telle intégration, une solution naïve suppose une connaissance fine des sources de données participant dans le processus d'intégration (les schémas de données, les contenus, les localisations, etc.). Cette solution a deux inconvénients majeurs : (1) le non respect de la transparence d'accès aux données réparties et (2) un coût de mise en oeuvre dans le cas d'un nombre important de sources. Une deuxième solution consiste à fournir aux utilistateurs une interface d'accès *unique*, *homogène* et *transparente* aux données de sources.

Plusieurs problèmes doivent être pris en compte pendant la conception de systèmes d'intégration. Ces problèmes résultent de l'hétérogénéité structurelle et de l'hétérogénéité sémantique des données. L'hétérogénéité structurelle provient des différentes structures ou formats utilisés pour stocker les données de chaque source qui peut être non structurée (un texte par exemple), structurée (une base de données relationnelles) ou semi-structurées (un document XML). De nombreux travaux concernant ce type d'hétérogénéité ont été proposés dans les contextes des bases de données fédérées et des multi-bases de données. L'hétérogénéité sémantique, par contre, présente un défi majeur dans le processus d'élaboration des systèmes d'intégration. Elle est due aux différentes interprétations pour les objets du monde réel. En effet, les sources de données ont été conçues indépendamment par des concepteurs différents ayant des objectifs applicatifs différents. Chacun peut donc avoir un point de vue différent sur le même concept. Afin de réduire cette hétérogénéité, de plus en plus d'approches d'intégration visent à associer aux données des ontologies qui en définissent le sens. Ces approches sont dites "à base ontologique"'. Une ontologie est

définie comme une spécification formelle et explicite d'une conceptualisation. L'objectif de ces ontologies, dont chacune est normalement partagée par une communauté, est de représenter formellement la signification des données contenues. La référence à une telle ontologie est alors utilisée pour éliminer les conflits sémantiques entre les sources dans le processus d'intégration de données.

L'objectif de ce chapitre est de bien positionner notre travail par rapport à l'état de l'art sur le problème de l'intégration de sources de données à base ontologique.

Dans la section qui suit, nous présentons la problématique de l'intégration des données. Dans la section 3, une classification originale des approches d'intégration d'informations existantes est proposée. Cette classification nous permet de présenter de façon synthétique les méthodes d'intégration existantes. Elle nous permettra également de positionner précisément notre travail de l'intégration à base ontologique. La section 4 présente les ontologies conceptuelles et les différents modèles d'ontologies existants. Dans la section 5, nous nous concentrons sur les apports des ontologies conceptuelles dans l'élaboration des systèmes d'intégration de données. Avant de conclure le chapitre 2, nous positionnons précisément nos travaux et nos contributions dans la section 6.

2.2 Problématique de l'intégration de données

Avant de faire un état de l'art des systèmes d'intégration de données, il est important de définir la nature du problème à résoudre.

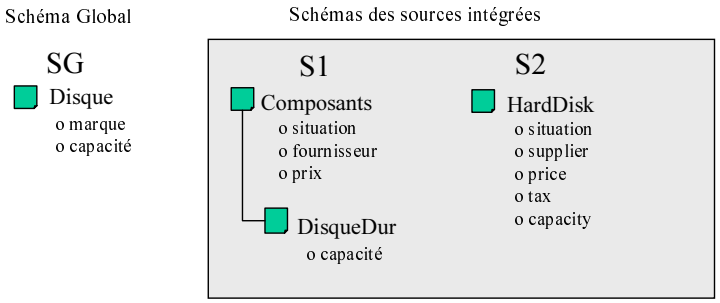

FIG. 2.1 – Exemple de catalogues électroniques hétérogènes

Les systèmes d'intégration doivent permettre à l'utilisateur d'accéder, via une interface d'accès unique, à des données stockées dans plusieurs sources de données. Ces sources de données ont été conçues indépendamment par des concepteurs différents. Cela entraîne l'hétérogénéité de données, c'est-à-dire que les données relatives à un même sujet sont représentées différemment sur des systèmes d'information distincts. Cette hétérogénéité provient des choix différents qui sont faits pour représenter des faits du monde réel dans un format informatique. En effet, les données des sources sont structurellement indépendantes mais sont toujours supposées relever de domaines similaires.

Exemple 1 *Pour illustrer ce problème d'hétérogénéité de sources, nous étudions l'exemple dans la figure 2.1. Supposons qu'une entreprise de commerce électronique vend des disques durs sur l'Internet. Les disques durs mis en vente sont fournis par des fournisseurs différents, où chacun organise ses produits dans son catalogue selon ses propres critères décidés localement. Cette entreprise doit donc traduire les catalogues de différents fournisseurs à son format, appelé schéma global. Pour des raisons de simplicité, nous considérons deux catalogues $S1$ et $S2$ de deux fournisseurs et le catalogue de l'entreprise SG (figure 2.1). Remarquons que les trois catalogues décrivent un disque dur différemment. Cette différence concerne le nombre de concepts utilisés pour définir chaque source, ainsi que l'aspect sémantique de chaque concept. Le tableau 2.1 décrit les différentes propriétés de SG, $S1$ et $S2$.*

	SG	**S1**	**S2**
Classes	**DisqueDur** : disque dur sous garantie	**Composants** : composant informatique **DisqueDur** : disque dur	**HardDisk** : disque dur
Propriétés		**situation** (domaine : *boolean*) : neuf ou occasion	**situation** (domaine : *boolean*) : disponible ou non
	marque (domaine : *string*) : marque de fabrique	**fournisseur** (domaine : *string*) : marque de fabrique	**supplier** (domaine : *string*) : marque de fabrique
		prix (domaine : *number*) : le prix total (qui inclut la TVA) d'un matériel neuf	**price** (domaine : *number*) : le prix hors taxe d'un disque dur
			tax (domaine : *number*) : la TVA d'un disque dur
	capacité (domaine : *integer*) : total de méga octets	**capacité** (domaine : *integer*) : total de méga octets	**capacity** (domaine : *number*) : total de giga octets

Tab. 2.1 – Sémantique de données de trois catalogues dans la figure 2.1

La question fondamentale lorsque l'on veut faire interopérer des bases de données hétérogènes est d'une part, l'identification de conflits entre les concepts dans des sources différentes qui ont des liens sémantiques, d'autre part, la résolution des différences entre les concepts sémantiquement liés.

Une taxonomie des conflits sémantiques a été proposée dans [37] : (1) conflits de représentations, (2) conflits de noms, (3) conflits de contextes, et (4) conflits de mesure de valeur que nous allons détailler dans les sections suivantes.

2.2.1 Conflits de représentation

Ces conflits se trouvent dans le cas où on utilise des propriétés différentes ou des schémas différents pour décrire le même concept. Cela signifie que le nombre de classes&propriétés et les classes&propriétés représentant un concept C dans les sources ne sont pas égaux.

Exemple 2 *Reprenons la Figure 2.1, où le fournisseur $S1$ utilise deux classes : $Composants$ et $DisqueDur$ et **4** propriétés : $situation$, $fournisseur$, $prix$, et $capacité$ pour décrire un disque dur. Tandis que le fournisseur $S2$ utilise une seule classe : $HardDisk$ et **5** propriétés : situation, supplier, price, tax, et capacity.*

Un autre exemple de conflit de representation entre les deux fournisseurs à mettre en évidence, c'est le cas où le fournisseur $S2$ utilise deux propriétés : price et tax pour calculer le prix d'un disque dur, tandis que $S1$ n'en utilise qu'une seule, à savoir prix.

2.2.2 Conflits de noms (termes)

Ces conflits se trouvent dans le cas où on utilise soit des noms différents pour le même concept ou propriété (*synonyme*), soit des noms identiques pour des concepts (et des propriétés) différents (*homonyme*).

Exemple 3 *Le même concept de disque dur est nommé par $DisqueDur$ dans la $S1$, et par $HardDisk$ dans la $S2$. La propriété Situation se trouve dans les deux sources, mais avec deux significations différentes (voir Table 2.1).*

2.2.3 Conflits de contextes

Le contexte est une notion très importante dans les systèmes d'information répartis. En effet, un même objet du monde réel peut être représenté dans les sources de données par plusieurs représentations selon un *contexte local* à chaque source. Ces conflits de contexte se trouvent dans le cas où les concepts semblent avoir la même signification, mais ils sont évalués dans différents contextes.

Exemple 4 *La propriété prix de disque dur ne s'applique que pour un disque dur neuf dans $S1$, mais peut l'être pour tous les disques soit neufs ou d'occasions dans $S2$.*

2.2.4 Conflits de mesure de valeur

Ces conflits sont liés à la manière de coder la valeur d'un concept du monde réel dans différents systèmes. Ils se trouvent dans le cas où on utilise des unités différentes pour mesurer la valeur de propriétés.

Par exemple, l'unité de mesure de la capacité dans la $S1$ est le méga octets tandis que celle dans la $S2$ est le giga octets.

2.3 Classification des approches d'intégration

L'intégration de données consiste à éliminer d'abord les conflits entre les données (présentés dans la section ci-dessus) et ensuite les représenter dans un seul schéma cohérent. Tout système d'intégration doit fournir les solutions aux problèmes suivants : (i) l'identification et la spécification des correspondances entre des données sémantiquement liées, (ii) fournir une vue globale intégrée des données représentées à travers des conceptualisations différentes, (iii) la gestion des mises à jour des données de différentes sources et leurs répercussions sur le schéma global.

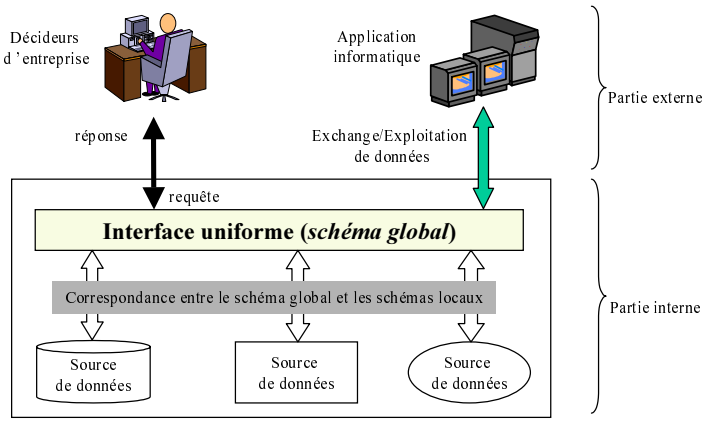

FIG. 2.2 – Système d'intégration de données

Généralement, un système d'intégration comporte deux parties principales (voir la figure 2.2) : une *partie externe* et une *partie interne*.

1. **La partie externe** correspond aux utilisateurs du système intégré comme, par exemple, les décideurs d'une entreprise, ou des autres systèmes,
2. **La partie interne** comprend d'une part les sources d'informations participant dans le processus d'intégration et d'autre part une interface (schéma global) permettant aux éléments de la partie externe d'accéder d'une manière transparente aux sources de données. Cette interface peut être une couche sans données propres (*approche médiateur*) ou une couche contenant sous une forme qui lui est propre une duplication des données pertinentes des sources (*approche entrepôt*). Les éléments externes au système ne voient pas les sources internes. Celles-ci sont *encapsulées*, *cachées* par l'interface. Les éléments externes doivent pouvoir agir sur le système d'intégration d'une manière transparente via un *schéma global*.

Plusieurs approches et systèmes d'intégration ont été proposés dans la littérature. A notre connaissance, aucune classification de ces derniers n'a été proposée. Pour faciliter la

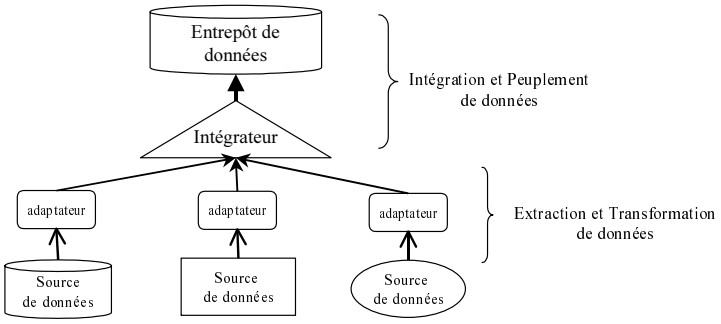

FIG. 2.3 – Intégration par l'entrepôt

compréhension des caractéristiques et les avantages et les faiblesses des systèmes existants, nous proposons une classification basée sur trois critères orthogonaux, à savoir : (1) la *représentation de données intégrées* : virtuelle ou matérialisée, (2) le *sens de la mise en correspondance entre schéma global et les schémas locaux*, (3) la *nature du processus d'intégration*. Cette classification va nous permettre de mieux positionner notre travail par rapport à l'existant.

2.3.1 Représentation de données intégrées

Ce critère permet d'identifier le type de stockage de données du système d'intégration. Les données intégrées peuvent être matérialisées (elles sont dupliquées dans une entrepôt de données), ou virtuelles (elles restent dans les sources d'origines et sont accédées via un médiateur).

2.3.1.1 Architecture matérialisée

Cette architecture consiste à centraliser physiquement, dans un *entrepôt de données*, l'ensemble de données consolidées à partir de diverses sources (catalogues électroniques, bases de données relationnelles, Web, etc.) (voir la figure 2.3).

Giraldo [38] considère que la conception d'un système d'intégration selon une architecture matérialisée est similaire à celle des entrepôts de données. Le processus de construction d'un système d'intégration matérialisé se compose en quatre étapes principales [43] : (1) l'extraction des données des sources de données, (2) la transformation des données au niveau structurel, (3) l'intégration sémantique des données, et (4) le stockage des données intégrées dans le système cible. Ces étapes correspondent aux outils d'ETL (extract, transform and load) [104].

Les deux premières étapes sont habituellement prises en charge par le même composant logiciel, appelé *adaptateur*. L'objectif d'un adaptateur est d'aboutir à des données représentées dans un même format. Chaque adaptateur fournit ainsi une interface d'accès et de requêtes sur la source. Les données extraites sont ensuite intégrées en éliminant les conflits, puis stockées dans l'entrepôt.

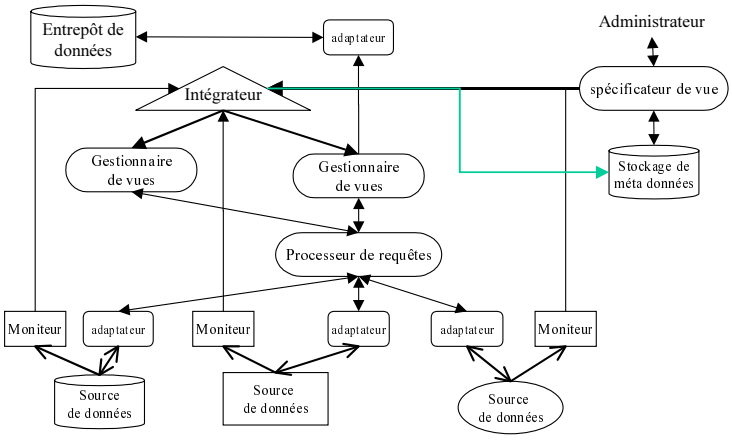

FIG. 2.4 – Architecture du système WHIPS pour la maintenance de l'entrepôt de données

L'avantage principal d'une telle approche est que l'interrogation d'un entrepôt de données se fait directement sur les données de l'entrepôt et non sur les sources originales. On peut donc utiliser les techniques d'interrogation et d'optimisation des bases de données traditionnelles. Par contre cette approche exige un coût de stockage supplémentaire et surtout un coût de maintenance causé par les opérations de mises à jour au niveau de sources de données (toute modification dans les sources locales doit être répercutée sur l'entrepôt de données). Le problème de maintenance du système d'intégration matérialisée est similaire au problème de maintenance des vues matérialisées [26].

Au titre de cette thèse, nous sommes dans le contexte où des entreprises veulent stocker localement toutes les données de divers sources au sein d'une base de données d'entreprise unique pour des raisons privées. Nous nous intéressons donc à l'architecture matérialisée.

Quelques projets spécifiques d'entrepôts de données servent actuellement de références, en particulier, le projet européen DWQ (Data Warehouse Quality) [48] et le projet WHIPS à l'université de Stanford - USA [44, 54]. Le but du WHIPS est de développer des algorithmes pour collecter, intégrer et maintenir des informations émanant de diverses sources. Une architecture a été développée ainsi qu'un prototype pour identifier les changements de données des sources, les transformer et les synthétiser selon les spécifications de l'entrepôt, et les intégrer de façon incrémentale dans l'entrepôt. La figure 2.4 présente les modules principaux du WHIPS [54] :

- Les données de l'entrepôt sont représentées selon le modèle relationnel : les vues sont définies dans ce modèle et l'entrepôt stocke les relations. Ces vues sont formulées par l'administrateur à travers le *spécificateur de vue*.
- Chaque vue d'entrepôt est associée à une *gestion de vue*. La gestion de vue est utilisée pour synchroniser des modifications survenues à cette vue en même temps. Pour chaque modification effectuée, une requête de mise à jour est créée et envoyée au *processeur de requête*.
- Le *moniteur* est chargé de détecter des modifications des sources. Chaque modifi-

cation d'une source sera signalée à l'*intégrateur* pour analyser son influence sur les vues matérialisées dans l'entrepôt.
- Les *adaptateurs*, quant à eux, traduisent les requêtes exprimées sous forme d'une représentation relationnelle en des requêtes dans le langage natif de la source. L'utilisation d'un adaptateur par source cache les détails d'interrogation (spécifique à la source) au processeur de requêtes. L'adaptateur de l'entrepôt reçoit les définitions de vues et les modifications sur les données des vues dans un format canonique, et les traduit dans la syntaxe spécifique du système de gestion de base de données de l'entrepôt. Tous les adaptateurs supportent la même interface de requêtes bien que leur code interne dépende de leur source.
- L'*intégrateur* a pour rôle principal de faciliter la maintenance des vues en calculant les modifications des sources qui ont besoin d'être propagées dans les vues. Ces modifications sont ensuite transférées aux *gestionnaires de vues*.
- Le *processeur de requête* reçoit les requêtes de mise à jour des *gestionnaires de vues* et génère les requêtes appropriées aux adaptateurs de chaque source.

2.3.1.2 Architecture virtuelle

Cette architecture consiste à développer une application chargée de jouer le rôle d'interface entre les sources de données locales et les applications d'utilisateurs. Ce type d'approche a été utilisé par un nombre important de projets [24, 36, 57, 63, 88, 106, 59]. Il repose sur deux composants essentiels : le *médiateur* et l'*adaptateur*.

1. Le médiateur : chargé de la localisation des sources de données et des données pertinentes par rapport à une requête, il résout de manière transparente les conflits de données. Un ensemble de connaissances sur les sources permet au médiateur de générer un plan d'exécution pour traiter les requêtes d'utilisateurs ;
2. L'adaptateur : comme dans l'approche entrepôt, c'est un outil permettant à un (ou plusieurs) médiateur(s) d'accéder au contenu des sources d'informations dans un langage uniforme. Il fait le lien entre la représentation locale des informations et leur représentation dans le modèle de médiation.

Dans cette approche, les données ne sont pas stockées au niveau du médiateur. Elles restent dans les sources de données et ne sont accessibles qu'à ce niveau. L'intégration d'information est fondée sur l'exploitation de vues abstraites décrivant de façon homogène et uniforme le contenu des sources d'information dans les termes du médiateur. Les sources d'information pertinentes, pour répondre à une requête, sont calculées en fonction de la définition de ces vues. Le problème consiste à trouver une requête qui est équivalente à la requête de l'utilisateur. Les réponses à la requête posée sont ensuite obtenues en évaluant cette requête sur les extensions des vues. Le problème d'hétérogénéité sémantique entre les différentes sources d'information ne se pose pas pour l'utilisateur puisqu'il est résolu par le médiateur. L'approche médiateur présente l'intérêt de pouvoir construire un système

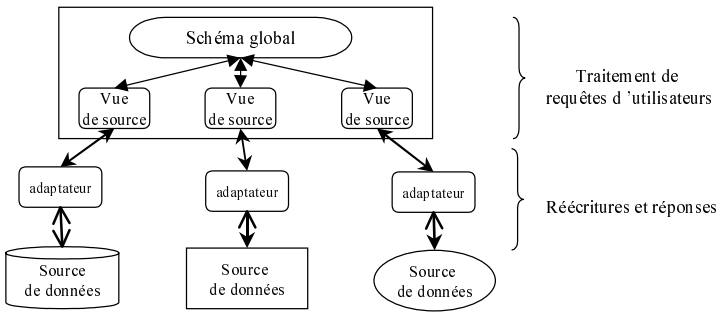

FIG. 2.5 – Intégration par le médiateur

d'interrogation de sources de données sans toucher aux données qui restent stockées dans leur source d'origine. Par contre le médiateur ne peut pas évaluer directement les requêtes qui lui sont posées car il ne contient pas de données, ces dernières étant stockées de façon distribuée dans des sources indépendantes. Dans cette approche, deux problèmes apparaissent : comment construire le schéma intégré et comment effectuer la mise en correspondance entre les termes utilisés dans ce schéma et les structures des schémas des sources qui contiennent les données.

De nombreux travaux ont porté sur l'architecture virtuelle. Hacid et al. [43] a identifié trois grands problèmes ayant fait l'objet d'études : (1) Des études ont porté sur les *langages* pour modéliser le schéma global, pour représenter les vues sur les sources à intégrer et pour exprimer les requêtes provenant des utilisateurs humains ou d'entités informatiques ; (2) d'autres travaux ont porté sur la conception et la mise en oeuvre d'algorithmes de réécriture de requêtes en termes de vues décrivant les sources de données pertinentes, et (3) certains travaux portent sur la conception d'interfaces intelligentes assistant l'utilisateur dans la formulation de requêtes, l'aidant à affiner une requête en cas d'absence de réponses ou en cas de réponses beaucoup trop nombreuses.

Nous présentons ici un exemple du système médiateur : le projet PADOUE [2]. L'objectif du PADOUE est de développer une architecture de partage de données environnementales, qui doit permettre aux chercheurs de l'environnement de mutualiser leurs ressources et de les exploiter.

Cette architecture de médiation du PADOUE est illustrée dans la figure 2.6. Elle se constitue en deux phases de construction suivantes :

1. la construction des *schémas publiés*. Il s'agit d'une pré-intégration des sources visant à homogénéiser autant que possible le modèle des données partagées, et ce grâce à la contribution humaine des fournisseurs de données [58]. Pour construire un schéma publié d'une source, le fournisseur exploite d'abord la *taxinomie de thèmes* spécifiant le contexte thématique du réseau et la connaissance sur les *schémas déjà publiés* afin de trouver le maximum de propriétés existantes dans le réseau et qui peuvent être utilisées pour décrire cette source. Ce schéma publié est ensuite complété par des

[2] urlhttp ://www-poleia.lip6.fr/padoue/

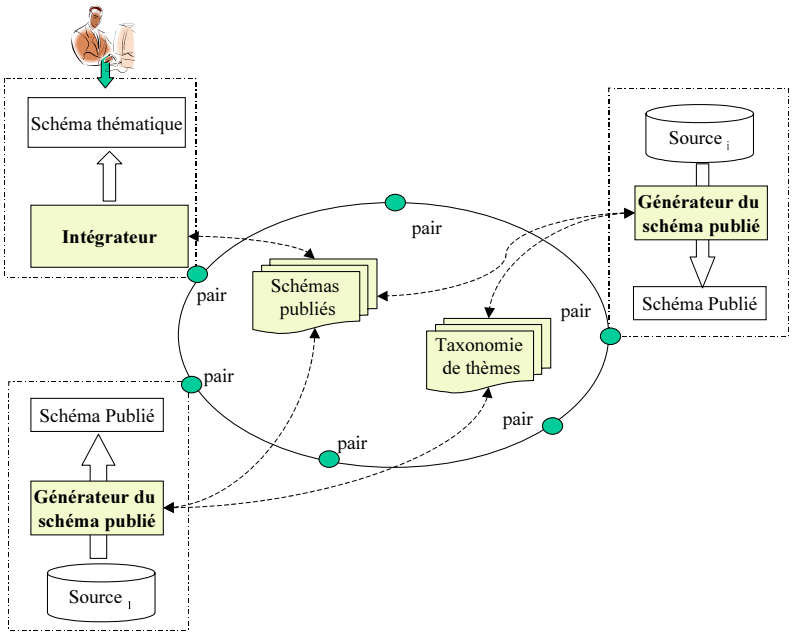

FIG. 2.6 – une vue sur le système pair-à-pair du Projet Padoue [58].

propriétés spécifiques qui sont nécessaires pour la source publiée, mais qui n'existent pas encore sur le réseau.

2. la construction des *schémas thématiques*. Un schéma thématique est une interface générique d'un thème sur laquelle les utilisateurs peuvent écrire des requêtes concernant ce thème. Il résulte de l'intégration des schémas publiés qui ont été localisés à travers le réseau. Le système collecte d'abord les schémas publiés qui concernent le thème du schéma thématique, et en intègre ensuite ces schémas collectés [58].

Ces deux types de schéma (médiateur) cités ci-dessus sont deux instances d'un modèle de schémas, nommé PAXschema (Peer Active Xschema). PAXschema est une extension du XSchema [3]. La modélisation d'un PAXschema passe par la création de quatre composants de métadonnées [58] :

1. *métadonnées identifiantes* qui est l'association à schéma d'un rôle et d'un domaine applicatif. Le rôle spécifie s'il s'agit d'une schéma publié ou d'un schéma thématique. Le domaine applicatif spécifie le thème auquel sont rattachées les données modélisées par ce schéma.
2. *métadonnées localisatrices* qui est l'association à un schéma de la localisation physiques des données modélisées (l'adresse web du médiateur permettant l'accès aux données) et la correspondance entre le nom publié d'un attribut et celui réel dans la source.
3. *métadonnées qualitatives* qui est l'association à chaque schéma des critères de réutilisation et de comparaison des schémas.

[3] `http://www.w3.org/XML/Schema.html`

Un objet OEM est structuré comme : <oid, étiquette, type, valeur>, où:
• **oid**: un identifiant,
• **étiquette**: une chaîne de caractères qui décrit ce que l 'objet représente,
• **type :** le type de la valeur de l 'objet,
• **valeur** : la valeur de l 'objet.

Description d 'un disque dur:
<o1, 'DisqueDur', set, {o2,o3,o4,o5}>
<o2, 'Situation', boolean, true>
<o3, 'marque', string, ' IBM '>
<o4, 'garantie', integer, 3>
<o5, 'Capacité', integer, 80>

FIG. 2.7 – Exemple de données OEM

4. *métadonnées descriptives* qui est l'association à chaque élément du schéma d'une description sémantique de l'élément.

Un autre exemple est le projet TSIMMIS (The Stanford-IBM Manager of Multiple information Source) [24] de l'Université de Stanford qui est considéré comme un des premiers projets d'intégration de données appliquant cette architecture. TSIMMIS a pour objectif de produire des outils pour l'accès intégré à des sources d'information. Il utilise une hiérarchie de médiateurs pour intégrer des sources de données hétérogènes. Le modèle commun utilisée dans TSIMMIS est OEM (Objet Exchange Model) [76]. Les données OEM sont représentées par un graphe, dont les noeuds sont des objets permettant de stocker les données et leur schémas (voir la figure 2.7). Chaque source d'information est équipée d'un adaptateur qui encapsule la source, en convertissant les objets de données en OEM. Chaque médiateur obtient d'abord les informations (sous forme d'OEM) de plusieurs adaptateurs ou d'autre médiateurs. Il traite ensuite ces informations en intégrant et en résolvant les conflits entre les fragments d'information des différentes sources, et enfin fournit l'information résultante à l'utilisateur ou aux autres médiateurs. Les médiateurs sont traités comme des vues sur les données trouvées dans les sources qui sont convenablement intégrées et traitées . Le médiateur est défini en termes de langage logique appelé MSL. MSL est essentiellement un Datalog étendu pour supporter les objets OEM. Les médiateurs réalisent typiquement des vues virtuelles puisqu'ils ne stockent pas les données localement. Quand un utilisateur pose une question au système, un médiateur concernant cette question est sélectionné. Ce médiateur décompose la requête et puis, propage les sous-requêtes. Lorsqu'une nouvelle source est ajoutée, TSIMMIS nécessite de changer tous les médiateurs qui l'utilisent. C'est précisément ce problème qui peut être évité en changeant le sens de définition des vues qui fait l'objet du critère suivant.

2.3.2 Sens de mise en correspondance entre schéma global et schéma local

Ce critère permet de définir les relations existant entre le schéma global et les schémas locaux. Deux principales approches ont été proposées : GaV (Global as View) et LaV

(Local as View).

2.3.2.1 Global As View (GaV)

L'approche GaV a été la première à être proposée pour intégrer des informations. Elle consiste à définir à la main (ou de façon semi-automatique) le schéma global en fonction des schémas des sources de données à intégrer puis à le connecter aux différentes sources. Pour cela, les prédicats du schéma global, aussi appelés relations globales, sont définis comme des vues sur les prédicats des schémas des sources à intégrer.

Comme les requêtes d'un utilisateur s'expriment en termes des prédicats du schéma global, on obtient facilement une requête en terme du schéma des sources de données intégrées, en remplaçant les prédicats du schéma global par leur définitions.

Parmi les systèmes utilisant GaV, on peut citer TSIMMIS [24] et MOMIS [13]. En effet, dans TSIMMIS, les médiateurs sont conceptuellement représentés par des vues définies sur une ou plusieurs sources. Dans le cas d'ajout d'une nouvelle source, TSIMMIS reconstruit les médiateurs (voir Section 2.3.1.2).

2.3.2.2 Local As View (LaV)

L'approche LaV est l'approche duale, elle suppose l'existence d'un schéma global et elle consiste à définir les schémas des sources de données à intégrer comme des vues du schéma global.

Les principaux systèmes développés autour de cette approche sont : Infomaster [36], PICSEL [38], Information Manifold [57].

InfoMaster [36], par exemple, est un entrepôt virtuel de catalogues développé par l'Université de Stanford (voir la figure 2.8). Il présente à l'utilisateur un catalogue virtuel, appelé *schéma référencé*, sur lequel il formule ses requêtes. Ce système utilise le langage KIF (Knowledge Interchange Format [4]) pour définir une liste de règles qui associe les schémas des sources intégrées aux vues du *schéma référencé*. Ces règles permettent aussi de convertir l'information afin de la mettre sous format adéquat et de l'adapter aux sources.

Lors de l'ajout d'une nouvelle source, le système d'intégration ajoute des nouvelles règles représentant la vue de la nouvelle source.

2.3.2.3 Bilan sur les approches GaV et LaV

On considère souvent que l'adjonction d'une nouvelle source est plus facile dans l'approche LaV que dans l'approche GaV [43]. En fait, cela dépend de l'hypothèse faite concernant (1) les concepts contenus dans le schéma global, et (2) les concepts existants (auxquels le système d'intégration doit donner accès) dans la nouvelle source. Si l'on fait l'hypothèse

[4] `http://www-ksl.stanford.edu/knowledge-sharing/kif/`

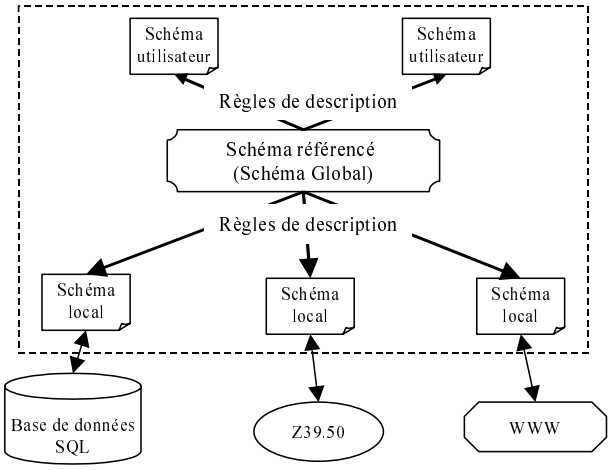

FIG. 2.8 – Architecture d'un système InfoMaster

qu'une nouvelle source ne doit pas modifier le schéma global (hypothèse faite dans les système LaV), soit qu'elle ne contienne aucun nouveau concept, soit comme présenté dans la Figure 2.9, que le schéma global ne représente que partiellement les différentes sources, l'effet d'adjonction est tout à fait similaire. Le coût de modification de l'implémentation de la vue globale, dans l'approche GaV, est contre balancée, dans l'approche LaV, par le coût de définition du schéma local comme vue du schéma global, puis par le coût de réécritures de requêtes en fonction des vues.

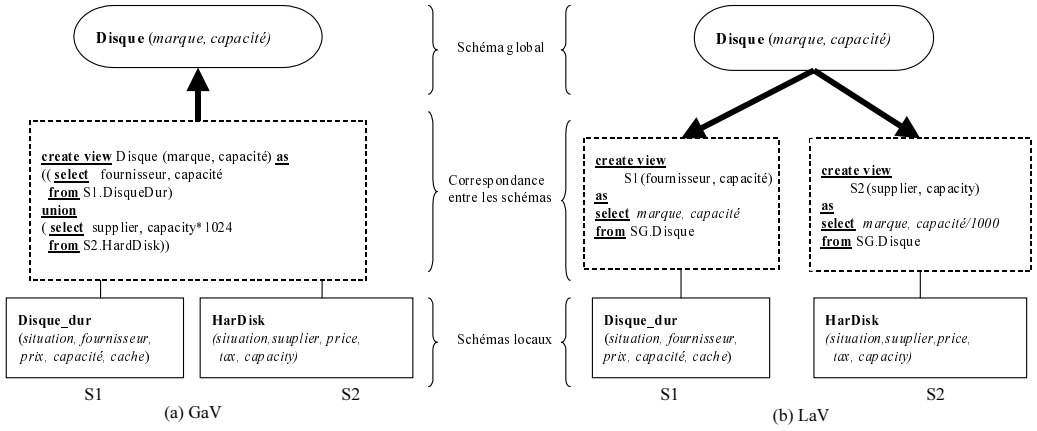

FIG. 2.9 – Exemple du sens de mise en correspondance entre schéma global et schéma local

2.3.3 Nature du processus d'intégration

Ce dernier critère spécifie si le processus d'intégration de données est effectué d'une manière manuelle, semi-automatique ou automatique. Ce critère devient essentiel lorsque l'on veut intégrer un nombre important de sources de données indépendantes.

2.3.3.1 Intégration manuelle de données

Les méthodes d'intégration manuelles de données correspondent à la première génération de systèmes d'intégration. Cette génération était apparue dans les années 90 [21, 64, 77, 96]. A cette époque, les travaux d'intégration de données visaient essentiellement à automatiser l'interopérabilité syntaxique de données. Par contre, les conflits sémantiques étaient résolus d'une façon manuelle. Puisque seul un observateur humain pouvait interpréter la sémantique des données, ces approches laissaient à l'administrateur (ou l'utilisateur) la responsabilité du processus d'intégration.

D'après Parent et al. [77], les approches d'intégration manuelles visent à fournir des langages de manipulation de schémas que l'administrateur ou l'utilisateur peuvent utiliser pour construire lui-même (si le langage est procédural) ou spécifier (si le langage est déclaratif) le schéma intégré. Les langages procéduraux offrent des primitives de transformation de schéma qui permettent de restructurer les schémas initiaux jusqu'à ce qu'ils puissent être fusionnés en un seul schéma. Le système génère alors automatiquement les règles de traduction entre les schémas initiaux et le schéma intégré. Les langages déclaratifs sont plus faciles à utiliser, mais l'établissement des règles de traduction par le système est plus délicat. Les stratégies manuelles supposent la connaissance par l'administrateur de la structure du schéma intégré.

Nous citons ici trois méthodes typiques. La première, nommée *système de multi-bases de données* [21], vise à construire un système d'accès à de multiples bases de données. La deuxième, nommée la fédération des bases de données [96], consiste à fédérer les bases de données intégrées. La troisième consiste à intégrer les données en utilisant des standards de représentation ou de formalisation comme Corba/IDL [5], ou XML [6], EXPRESS [94].

Le système de multi-base de données [21] permet de créer une véritable interopérabilité entre les bases de données sans pour cela avoir besoin de générer explicitement un schéma global intégré. Le processus d'intégration est conçu au niveau du langage d'interrogation. De cette manière, l'utilisateur spécifie sa requête en précisant les sources de données interrogées. L'implémentation du système (langage d'interrogation) est aisée et toutes les hétérogénéités sont traitées par les utilisateurs. Les interfaces d'interrogation sont textuelles. Ce processus d'intégration est donc complètement manuel. Par contre, ils assurent l'autonomie et l'intégrité de chaque base de données tout en permettant un accès partagé aux données.

La fédération des bases de données [96] vise à fournir un schéma conceptuel global représenté par l'union de toutes les bases de données. Le schéma global donne une vision homogène de toutes les sources et facilite l'interrogation de leurs données. Tout d'abord, un schéma externe est défini pour chaque base de données qui permet de définir ce que l'on vise à exploiter. Ces schémas sont ensuite convertis à une représentation commune, un formalisme relationnel, par exemple ; puis les schémas canoniques sont intégrés dans

[5] `http://www.omg.org/gettingstarted/omg_idl.htm`

[6] `http://www.w3.org/XML/1999/XML-in-10-points.html`

un schéma global. Les requêtes d'utilisateurs sont spécifiées sur le schéma global et sont ensuite décomposées en sous-requêtes. Ces sous-requêtes sont envoyées aux sources de données. Le processus de la construction de schéma global est fait *manuellement*.

Une troisième technique utilisée dans les approches d'intégration manuelle de données est l'utilisation des standards de représentation et d'accès pour faciliter l'interopérabilité de systèmes hétérogènes. XML et EXPRESS, par exemple, qui sont les standards pour la représentation et l'échange de données, sont utilisés pour résoudre les problèmes liés à l'hétérogénéité syntaxique (uniformisation de la structure des données) et produire une vue logique des données. Les données, et éventuellement les schémas de différentes sources une fois convertis dans un tel langage peuvent être réunis dans une même base de données adoptée au langage (SDAI EXPRESS [86] ou base de données XML [7], par exemple). L'intégration sémantique doit entièrement être faite par l'utilisateur. Dans l'architecture Corba [64], le processus d'intégration est réalisé en utilisant le langage IDL (Interface Definition Language) comme interface de communication de haut niveau entre les "composants" d'un logiciel, indépendante de leurs langages de programmation et de leur implémentation physique.

Dans les environnements qui nécessitent d'intégrer un nombre important de sources de données réparties qui évoluent fréquemment, l'intégration de données manuelle devient très coûteuse et souvent même impossible. Les traitements plus automatisés ont donc été conçus pour faciliter la résolution des conflits sémantiques. Ceci correspond aux deux classes étudiées ci-dessous.

2.3.3.2 Intégration semi-automatique à l'aide d'ontologies linguistiques (ou thésaurus)

Afin d'atteindre une intégration au moins partiellement automatique de différentes sources de données, plusieurs groupes de recherche ont développé des techniques d'intégration basées sur des notions d'ontologies, issues de la communauté de l'intelligence artificielle. Le rôle d'une ontologie est de servir de pivot pour définir la sémantique des différentes données à l'aide de concepts communs et formalisés, compréhensibles et admis par tous les participants (utilisateurs).

La deuxième génération de systèmes d'intégration utilise des ontologies linguistiques (*OL*) (qui sont également appelées des thésaurus) pour identifier automatiquement ou semi-automatiquement quelques relations sémantiques entre les termes utilisées dans les sources de données.

> *Une OL définit le sens des mots utilisés dans un domaine d'étude. Ces mots sont essentiellement liés par des relations linguistiques telles que la synonymie, l'antinomie, l'hyperonymie, etc.*

[7] `http://exist-db.org/index.html`

On peut alors automatiquement comparer les noms de relations ou d'attributs et essayer d'identifier les éléments similaires en utilisant des OL. Les OL restent cependant orientés "terme" et non "concept". Cela pose notamment des problèmes d'homonymie (par exemple, le cas de la propriété "situation" dans la figure 2.1) et de dépendance vis-à-vis d'un langage. Ce type d'ontologies est également très lourd à développer (dizaine de milliers de concepts) et à mettre à jour. Enfin, les relations entre termes sont très contextuelles. Ainsi le traitement par OL est nécessairement supervisé par un expert et ne peut être que partiellement automatique. Certains travaux d'intégration à base OL utilisent des mesures d'affinité et de similarité afin de calculer la vraisemblance de relations entre concepts [14, 13, 55]. Ces mesures sont associées aux seuils définis par l'administrateur de la base de données utilisés pour décider si la relation est retenue. De telles procédures compromettent la qualité du système d'intégration. C'est pourquoi ce type d'approche ne peut être complètement automatique que pour l'interprétation, automatique, mais approximative de documents.

Deux thésaurus assez connus sont utilisées dans ce genre d'approche : WORDNET et MeSH. MeSH (Medical Subject Heading) [8] est un thésaurus médical. C'est le thésaurus d'indexation de la base bibliographique MEDLINE. Le MeSH offre une organisation hiérarchique et associative et comprend jusqu'à neuf niveaux de profondeur. L'étude de MeSH montre que l'on est en face d'un thésaurus développé pour l'indexation et non pour les inférences [23]. WORDNET [65] est une base de données lexicales. Les termes y sont organisés sous formes d'ensembles de synonymes, les synsets. Chaque synset est un concept lexicalisé. Ces concepts lexicalisés sont reliés par des relations linguistiques. WORDNET est un énorme dictionnaire hypermédia de l'anglais-américain (plus de 100 000 synsets). Sa richesse et sa facilité d'accès le positionnent comme un intéressant outil pour la recherche d'information ou d'autres tâches comme le traitement du langage naturel mais ce n'est pas une ontologie car les relations ne sont en aucun cas formelles. L'utiliser tel quel, dans un système formel est donc voué à l'échec. Sa seule utilisation dans le cadre de l'intégration ne peut donc être que d'assister un expert humain.

Le projet MOMIS [13] est un exemple de projet d'intégration utilisant des OL qui vise à intégrer semi-automatiquement des données de sources structurées et semi structurées. Pour cela, chaque source de données est associée à un adaptateur qui consiste à traduire le schéma de cette source dans un modèle orienté objet commun ($OQLi$[3]) [12]. Un thésaurus global représentant les relations terminologiques entre les schémas des sources est d'abord construit, à l'aide de WordNet, en extrayant les termes utilisés dans les schémas des sources. Une vue intégrée est ensuite créée semi-automatiquement en se basant sur le thésaurus et les mesures d'affinité de concepts. Ces dernières calculent d'abord les similitudes entre deux concepts : affinité de nom, affinité structurelle, et affinité globale. Puis, le résultat de ce calcul est comparé avec un *seuil pré-choisi* afin d'introduire les concepts plus généraux correspondant à la vue intégrée. Les concepts introduits dans la vue intégrée

[8] `http://www.chu-rouen.fr/cismef/`

sont des subsumants communs à plusieurs termes apparus dans plusieurs schémas.

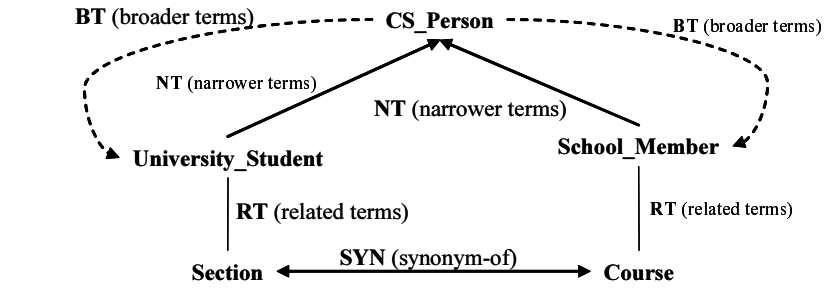

FIG. 2.10 – Un exemple de thésaurus [14]

La figure 2.10 montre l'exemple d'un thésaurus global [14]. Le dernier est représenté sous forme d'un graphe orienté et étiqueté (voir la Figure 2.10) :

- les noeuds représentent les termes du thésaurus (*CS_Person, University_Student, School_Member, Section, Course*),
- les arcs représentent les relations terminologiques entre les termes,
- les étiquettes définissent la nature des relations terminologiques comme : SYN (synonym) , BT(broader terms), NT(narrower terms) et RT(related terms).

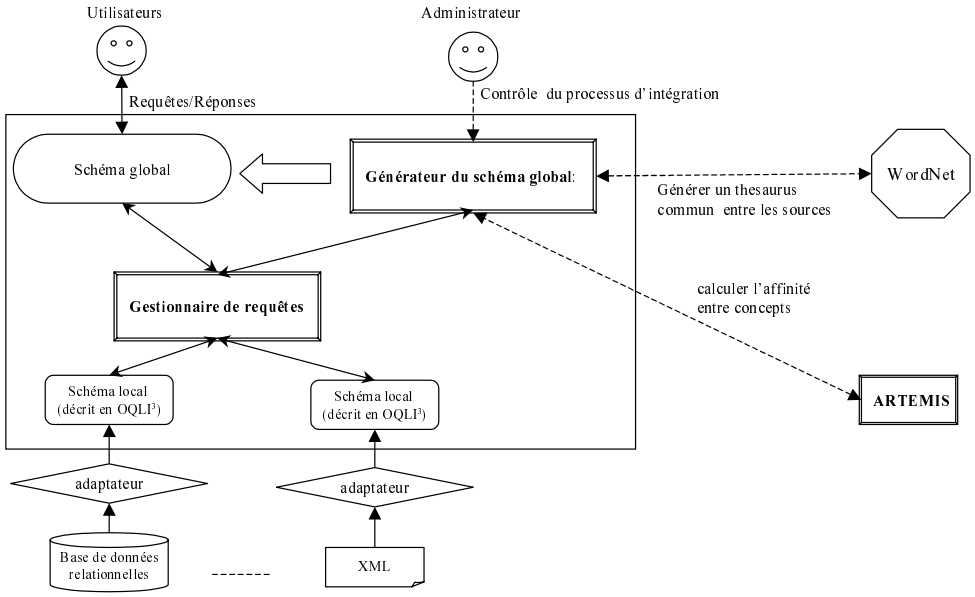

FIG. 2.11 – Architecture du système d'intégration de MOMIS

L'affinité terminologique entre deux termes t_i et t_j est calculée dans MOMIS par la fonction suivante :

$$A_{thes}(t_i, t_j) = \begin{cases} 1, & si\ t_i = t_j \\ \sigma_{i1_R} * \sigma_{12_R} ... * \sigma_{(k-1)j_R}, & si\ t_i \rightarrow^k t_j \text{ (voir ci-dessus)} \\ 0, & sinon \end{cases}$$

avec :

- σ_R représente le coefficient d'affinité d'une relation terminologique (R) entre deux termes du thésaurus. Ce coefficient est fixé par l'administrateur. Par exemple, dans le système d'intégration MOMIS, les différentes valeurs de σ sont :
  1. σ_{SYN}=1,
  2. σ_{RT} =0.5,
  3. $\sigma_{BT} = \sigma_{NT}$ =0.8
- $t_i \rightarrow^k t_j$: représente le chemin le plus court entre les termes t_i et t_j dans le thésaurus. Dans la figure 2.10, le chemin entre le terme $University\_Student$ et le terme $School\_Member$ est :
  "$University\_Student \rightarrow^{NT} CS\_Person \rightarrow^{BT} School\_Member$".
  Sur ce chemin, la fonction d'affinité entre les termes $University\_Student$ (U_S) et $School\_Member$ (S_M) est appliquée comme suit :

$$\begin{aligned} A_{thes}(U\_S, S\_M) &= \sigma_{NT}(U\_S, CS\_Person) * \sigma_{BT}(CS\_Person,\ S\_M) \\ &= 0.8 * 0.8 \\ &= 0.64. \end{aligned}$$

Le médiateur du système MOMIS est composé de trois modules principaux (voir la figure 2.11) : (1) "Générateur du schéma global" qui consiste à générer semi-automatiquement le schéma global présenté à l'utilisateur à partir des descriptions des sources ; (2)"ARTEMIS" qui consiste à calculer l'affinité entre les concepts dans le thésaurus global et (3) "Gestionnaire de requêtes" qui génère automatiquement à partir de la requête de l'utilisateur, des requêtes en $OQLi$[3], qui sont envoyées aux adaptateurs.

2.3.3.3 Intégration automatique à l'aide d'ontologie conceptuelle

La troisième génération des systèmes d'intégration consiste à associer aux données une ontologie conceptuelle (OC) qui en définit le sens (la définition d'une OC sera détaillée dans la section 2.4). Une ontologie conceptuelle est "*une spécification explicite et formelle d'une conceptualisation faisant l'objet d'un consensus*" [80]. En fait, dans une conceptualisation, le monde réel est appréhendé à travers des concepts représentés par des classes et des propriétés. Des mots d'un langage naturel peuvent être associés, mais ce ne sont pas eux qui définissent le sens des concepts. C'est l'ensemble des caractéristiques associées à un concept ainsi que ses liens avec les autres concepts qui en définissent le sens. Une OC regroupe ainsi les définitions d'un ensemble structuré de concepts. Ces définitions sont traitables par machine et partagées par les utilisateurs du système. Elles doivent, en plus,

être explicites, c'est-à-dire que toute la connaissance nécessaire à leur compréhension doit être spécifiée.

La référence à une telle ontologie est alors utilisée pour éliminer automatiquement les conflits sémantiques entre les sources dans le processus d'intégration de données. L'intégration de données est donc considérée comme automatique. Nous pouvons citer le projet PICSEL [35], OBSERVER [63], OntoBroker [28], KRAFT [106], COIN [37], etc. Comparées aux approches d'intégration utilisant des ontologies linguistiques, ces approches sont :

1. plus rigoureuses : car les concepts sont définis avec plus précision, l'appariement peut être plus argumenté,
2. facilement outillables : si les différentes ontologies sont basées sur le même modèle, comme OWL [62], PLib [82], etc. Une correspondance entre ontologies peut être exploitée par des programmes génériques pour intégrer les données correspondantes.

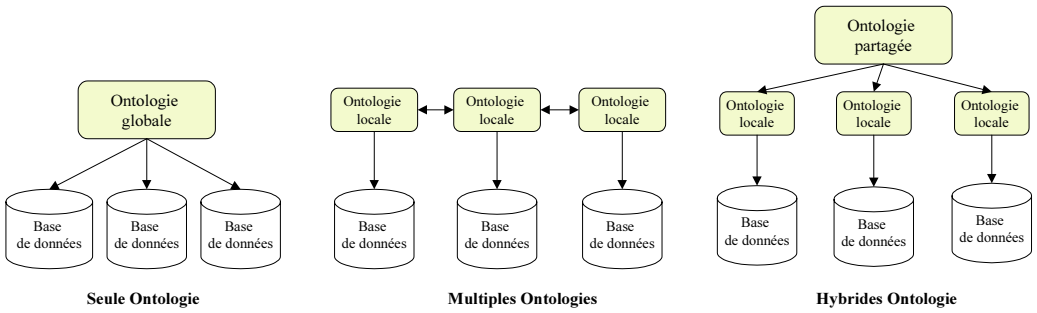

FIG. 2.12 – Différentes architectures d'intégration à base ontologique proposées par Wache et al. [107]

Plusieurs approches d'intégration à base ontologique ont été développées [107]. Ces dernières peuvent être divisées en trois catégories : approches avec une seule ontologie, approches avec ontologies multiples et approches hybrides (voir la Figure 2.12). Dans l'approche avec une seule ontologie, chaque source référence la même ontologie globale de domaine. Les systèmes d'intégration SIMS [7] et COIN [37] sont des exemples de cette approche . En conséquence, une nouvelle source ne peut ajouter aucun nouveau concept sans exiger le changement de l'ontologie globale. Dans l'approche à multiples ontologies (exemple du projet OBSERVER [63]), chaque source a sa propre ontologie développée indépendamment des autres sources. Dans ce cas, les correspondances inter-ontologies sont difficiles à mettre en oeuvre. L'intégration des ontologies est donc faite d'une façon manuelle ou semi-automatique. [63]. Pour surmonter l'inconvénient des approches simples ou multiples d'ontologies, l'approche hybride a été proposée. Dans cette dernière, chaque source a sa propre ontologie, mais toutes les ontologies utilisent un vocabulaire partagé commun (exemple du projet KRAFT [106]).

Dans cette thèse, nous divisons les approches d'intégration à base ontologique en deux

catégories : (1) les approches sémantiques *a posteriori*, et les (2) approches sémantiques *a priori*.

Notre travail se positionnant dans le domaine de l'intégration à base d'ontologies conceptuelles, nous discuterons ces approches de façon plus détaillée dans la section 2.5.2.

2.4 Ontologie conceptuelle

Nous avons présenté dans les sections précédentes la problématique de l'intégration de données. Nous avons proposé également une classification des approches d'intégration. Parmi ces dernières, l'utilisation d'ontologie conceptuelle apparaît comme une approche assurant l'automatisation du processus d'intégration sémantique de données.

Dans cette section, nous allons d'abord donner une définition de l'ontologie conceptuelle. Nous présenterons ensuite les points communs entre les modèles d'ontologies. Leurs principales différences seront illustrées à travers trois modèles : (1) le logique de description, (2) F-Logic, et (3) PLIB. Cette illustration nous permet d'identifier le but, les points forts et faibles de chacun.

2.4.1 Définition

L'origine de la notion d'ontologie se trouve dans une branche de la philosophie, initiée par Aristote, traitant de la science de l'être : "Partie de la métaphysique qui s'applique à l'être en tant qu'être indépendamment de ses déterminations particulières". En informatique, cette notion semble être apparue, pour la première fois, dans la communauté Intelligence Artificielle par le projet ARPA Knowledge Sharing Effort [71].

La définition communément admise d'une ontologie est énoncée par T. Gruber [39] comme la "spécification explicite d'une conceptualisation". Le but d'une ontologie informatique est de permettre de représenter formellement la sémantique d'un domaine et de supporter certains mécanismes de traitement automatique.

Avec le très fort développement des études sur les ontologies au cours des dix dernières années, de nombreuses définitions ont été proposées. Chaque communauté adopte sa propre interprétation selon l'usage qu'elle en fait et le but qu'elle vise et le mot ontologie recouvre en fait des réalités assez différentes. Pour beaucoup d'auteurs, néanmoins, tels par exemple Berners-Lee [16] à propos du Web sémantique, une ontologie réunit à la fois des éléments, concepts ou mots, et des règles permettant de manipuler ces éléments ou d'effectuer un certain nombre d'inférences. Un point commun à tous les modèles d'ontologies est la distinction entre les concepts *primitifs*, dont l'ontologie ne fournit pas de définition complète mais seulement quelques conditions nécessaires, la définition complète " reposant sur une documentation textuelle et un savoir préexistant partagé avec le lecteur " [39], et les concepts *définis* par l'ontologie, dont elle fournit des conditions nécessaires et suffisantes de reconnaissance en termes de concepts primitifs.

2.4.2 Caractéristiques communes des modèles d'ontologies

Même si les terminologies et objectifs de chaque modèle d'ontologie peuvent différer, pratiquement tous les modèles sont basés sur les mêmes notions. Ces notions sont les suivantes.

- **concepts** (ou classes, ou entités). Ils représentent des groupes d'individus partageant les mêmes caractéristiques. Ils correspondent aux entités "génériques" d'un domaine d'application. Les concepts d'une ontologie sont organisés hiérarchiquement par une relation d'ordre partiel qui est la relation de subsomption (*"est-une"*) permettant d'organiser sémantiquement les concepts par niveau de généralité : intuitivement, un concept C_1 *subsume* un concept C_2 si C_1 est plus général que C_2 au sens où l'ensemble d'individus représenté par C_1 contient l'ensemble d'individus représenté par C_2. Par exemple, le concept *Personne* subsume le concept *Femme*. Formellement, $C1$ subsume $C2$ si dans tout contexte : $x \in C2 \Rightarrow x \in C1$. Les concepts définis dans une ontologie peuvent être classés en deux catégories [39] :
  1. les **concepts primitifs** dont l'ontologie ne fournit pas les définitions complètes mais seulement des conditions nécessaires d'appartenance d'une instance à ce concept. Leurs définitions complètes s'appuient sur une documentation textuelle et un savoir préexistant partagé par le lecteur ;
  2. les **concepts définis** dont les conditions nécessaires et suffisantes de reconnaissance en termes de concepts primitifs sont fournies par une ontologie.
- **propriétés** (ou rôles ou attributs ou associations). Elles permettent de décrire et de caractériser des instances appartenant à une (ou plusieurs) classes de l'ontologie par des valeurs d'éléments caractéristiques ou des associations avec d'autres concepts. Une ontologie conceptuelle n'est pas seulement l'identification et la classification des concepts (comme dans une ontologie linguistique), mais aussi la représentation des caractéristiques qui leur sont attachées. Les propriétés peuvent être valuées soit dans un domaine de valeur simple, soit dans une autre classe décrivant alors ce qui est également appelé une association.
- **relations**. Elles constituent des types d'associations pré-définis entre les concepts. La relation commune qui est supportée par n'importe quel formalisme d'ontologie est *la subsomption : "est-un"*. Elle organise les concepts en une hiérarchie, où tout concept se compose d'une description propre définie par des propriétés *locales* et d'une description *partagée* avec ses subsumants comme c'est le cas entre classes dans un langage à objets [70]. La relation de subsomption est *transitive* (c'est-à-dire si C_1 subsume C_2 et C_2 subsume C_3 alors C_1 subsume C_3), *réflexive* (C_1 subsume C_1 lui même), et *antisymétrique* (si C_1 subsume C_2 et C_2 subsume C_1, alors $C1{=}C2$).
- **instances** (ou individus ou objets). Elle représentent des individus du domaine d'ontologie.
- **axiomes**. Ils explicitent les énoncés conceptuelles toujours vrais dans le contexte de l'ontologie. Ils peuvent être utilisés pour contrôler la correction des concepts ou des

relations, ou pour déduire de nouveaux faits.

Toutes ces entités ontologiques, c'est-à-dire les concepts, les relations, les axiomes et les instances, doivent être définis explicitement à l'aide d'un langage ou d'un modèle ayant une sémantique précisément définie. Cette sémantique peut être plus ou moins formelle, le degré de formalisation dépendant du but des applications qui utilisent l'ontologie.

Il existe plusieurs modèles d'ontologies proposés dans la littérature comme F-Logic [75], RDF/RDF Schéma [61], DAML+OIL [34], OWL [62], PLIB [82], etc.

Nous présentons ici une taxonomie de type d'ontologies conceptuels. Nous distinguons ici deux catégories d'ontologies conceptuelles : (1) les **ontologies canoniques** qui ne contiennent que des concepts primitifs, et (2) les **ontologies non canoniques** qui contiennent à la fois des concepts primitifs et des concepts définis.

2.4.2.1 Ontologie canonique

Pour pouvoir représenter tous les concepts existants dans un certain domaine, une ontologie conceptuelle n'a besoin de décrire que les concepts primitifs qui ne peuvent pas être dérivés d'autres concepts. En effet, les concepts définis n'augmentent nullement le domaine couvert par l'ontologie puisque, par définition, ces concepts pouvaient déjà se représenter à l'aide de la terminologie définie par les concepts primitifs. Comme dans le domaine des bases de données (où toute redondance est exclue), dans le vocabulaire technique (où un et un mot seul doit être utilisé pour chaque notion) et dans les formats d'échange (où il existe une seule représentation possible pour chaque information échangée), l'existence d'un langage *canonique*(une représentation pour chaque notion) constitue un avantage par rapport au langage contenant des synonymies.

Nous appelons *ontologie conceptuelle canonique* (OCC) une ontologie conceptuelle ne contenant que des éléments primitifs.

Partant donc du constat qu'on ne peut définir de nouveaux termes qu'à partir de termes primitifs, et que les termes primitifs d'un domaine technique sont eux-mêmes très nombreux et difficiles à appréhender, l'objectif essentiel d'une ontologie PLIB est de définir, mais de la façon la plus précise et la plus concise possible, les classes et propriétés primitives qui caractérisent les objets d'un domaine du monde réel, et les abstractions que les différentes communautés peuvent construire. L'objectif de concision signifie que PLIB ne définit de nouvelles classes que lorsque celles-ci ne peuvent complètement se définir en termes d'autres classes et de valeur de propriétés : il s'agit de classes primitives. L'objectif de précision signifie deux choses. D'une part, face à un objet matériel ou un artefact donné appartenant au domaine ciblé par une ontologie, un utilisateur humain de l'ontologie doit savoir décider : (1) à quelles classes l'objet appartient et n'appartient pas, (2) quelles propriétés s'appliquent à l'objet, et (3) quelles grandeurs ou valeurs caractéristiques correspondent à chaque propriété applicable. D'autre part, l'ontologie doit définir de façon formelle toutes les conditions nécessaires qui peuvent être vérifiées par un agent informatique.

Pour PLIB, une ontologie est donc une collection de descriptions explicites, formelles et consensuelles de l'ensemble des concepts d'un domaine [80] :

- dans le contexte le plus large où ces concepts ont un sens précis, et,
- sans aucune restriction ou règle correspondant à une utilisation particulière.

Dans ce contexte,

- explicite signifie que les types de concepts utilisés et les contraintes sur leurs usages sont explicitement définis.
- formel signifie exploitable, compréhensible, par la machine pour rendre certains traitements.
- partagé signifie que le contenu est consensuel, accepté par un groupe ; ceci est assuré par l'existence d'un processus normatif.

2.4.2.2 Ontologie non canonique

Des relations d'équivalence entre concepts peuvent également être représentées dans une *OC*. Ce type de relations permet de représenter les concepts définis. Ceci peut être fait en utilisant une ou des relations formelles de classe, que nous appelons également les *opérateurs de classes*, comme les opérateurs orientés ensemble (comme l'union, l'intersection et la différence) et les opérateurs de restrictions sur les values de propriétés. Ce peut également être fait en définissant des relations entre propriétés, que nous appelons aussi les *opérateurs de propriétés*, tels que les relations algébriques ou logiques.

Une telle *OC* que nous appelons *ontologie conceptuelle non-canonique*(OCNO) permet de représenter à la fois différentes conceptualisations d'un même domaine, et les correspondances entre celles-ci. Ceci permet, en effet, aux différents partenaires d'un échange d'associer leurs données à leurs propres ontologies. Puisque les partenaires ont exprimé leurs points de vue sur le même domaine d'étude, ils doivent pouvoir exprimer leurs concepts en terme des concepts de l'ontologie partagée.

Parmi des modèles d'ontologies qui supportent la définition de concepts définis, nous pouvons citer les logiques de description, F-Logic. Issu des logiques de description, OWL [62], par exemple, est un modèle d'ontologie permettant la définition de concepts primitifs et définis. Fondé sur la syntaxe de RDF/XML, OWL offre un moyen d'écrire des ontologies web. Si RDF et RDFS apportent à l'utilisateur la capacité de décrire des classes et des propriétés [61], OWL intègre, en plus, des outils de comparaison des propriétés et des classes : identité, équivalence, contraire, cardinalité, symétrie, transitivité, disjonction, etc. Dans OWL, la déclaration d'une classe se fait par le biais du mécanisme de "description de classe", qui se présente sous diverses formes. Une classe peut être définie comme suit [62] :

1. **l'indicateur de classe**. La définition d'une classe se fait directement par le nommage de cette classe. Par exemple, une classe "Doctorat" se déclare de la manière suivante :

   <owl :Class rdf :ID="Doctorat" />

2. **l'énumération des individus composant la classe**. La définition d'une classe se fait en énumérant les instances de la classe, à l'aide de la propriété *owl* : $oneof$. Par exemple :

```
<owl :Class>
  <owl :oneOf rdf :parseType="Collection">
    <owl :Thing rdf :about="#Dung"/>
    <owl :Thing rdf :about="#Hondjack"/>
  </owl :oneof>
</owl :Class>
```

3. **les opérateurs de restrictions sur les valeurs de propriétés**. Ce type de définition consiste à utiliser une restriction OWL qui définit une classe (un *concept défini*) en spécifiant une contrainte associée à une propriété. Cette classe est l'ensemble des instances satisfaisant cette contrainte. Celle-ci peut être de trois sortes :
   - une contrainte de cardinalité ; elle définit le nombre minimum, exact ou maximum de valeurs que doivent définir les instances pour la propriété contrainte ;
   - une contrainte de codomaine ; *allValuesFrom* (resp. *someValuesFrom*) permet de créer une classe dont les instances ne peuvent prendre pour valeur de la propriété contrainte que des (resp. au moins une) instances d'une classe spécifiée ;
   - une contrainte de valeur ; *hasValue* permet de créer une classe dont les instances ont une valeur spécifiée pour la propriété contrainte.
4. **les opérateurs orientés ensemble**. Ce type de définition est d'appliquer des opérations booléennes à une ou plusieurs classes déjà définies (*intersectionOf*, *unionOf*, *complementOf*) pour formuler une nouvelle classe (un *concept défini*). Par exemple, une "doctoratCIFRE" qui est un doctorat travaillant officiellement pour deux établissements se déclare de la manière suivante :

```
<owl :Class rdf :ID="doctoratCIFRE">
<owl :intersectionOf rdf :parsetype="Collection">
  <owl :Class rdfs :about="Doctorat">
  <owl :Restriction>
    <owl :onProperty rdf :resource="travaillePour"/>
    <owl :cardinality> 2 </owl :cardinality>
  </owl :Restriction>
</owl :intersectionOf>
</owl :Class>
```

OWL fournit trois constructeurs pour lier des classes qui sont : *subClassOf*, *equivalentClass*, *disjointClass*. Le constructeur "subClassOf" représente la relation de subsomption entre deux classes. Les "equivalentClass" et "disjointClass" permettent d'indiquer que les deux classes ont les mêmes ensembles d'instances ou que ceux-ci sont disjoints.

Dans la section ci-dessous, nous allons présenter de façon plus précise les différences entre trois modèles d'ontologies : PLIB, F-Logic, et les logiques de description. Leurs

différences nous permet d'identifier leurs applications cibles.

2.4.3 Différences entre les modèles d'ontologies

En règle générale, tout modèle d'ontologie fournit les moyens de définir les éléments de base (communs) cités dans la section 2.4.2. Mais plus particulièrement, selon son domaine prévu d'utilisation, chaque modèle possède ses propres opérateurs de définition de concepts. Cela permet de créer des relations entre concepts autre que la subsomption *est-un*. Ces opérateurs particuliers influencent non seulement le pouvoir de description du, domaine, mais aussi la complexité et la nature des inférences. Cette section présente de façon plus précise trois modèles d'ontologies : (1) OWL et les logiques de description, (2) F-Logic et (3) PLIB. Les deux premiers modèles sont des modèles orientés inférences. PLIB est, quant à lui, orienté intégration de données et caractérisation.

2.4.3.1 Logique de description et opérateurs de classes

Les logiques de description (LD) sont les logiques basées sur un formalisme de représentation qui s'apparente à la logique du premier ordre. Elles contiennent deux catégories d'éléments : les *concepts* (classes) et les *rôles*(propriétés). Un concept dénote un ensemble d'objets (l'extension du concept). Les rôles sont des relations binaires sur les objets. Une description est une expression obtenue par application d'*opérateurs* de construction sur des termes (noms) de classes ou de propriétés.

La LD structure une base de connaissances en deux parties : (1) la partie terminologique appelée $Tbox$ représentant la connaissance des descriptions ou des concepts, et (2) la partie assertionnelle appelée $Abox$ représentant la connaissance des instances. Les instances définies dans la $Abox$ sont basées sur des descriptions dans la $Tbox$. Une sémantique est associée à chaque description de concept et de rôle par l'intermédiaire d'une interprétation $I = (\Delta^I, f^I)$ qui est définie comme suit :

- Δ^I est le domaine de l'interprétation,
- f^I est une fonction d'interprétation qui associe à chaque description l'ensemble de ses instances : $f^I(C) = C^I \subseteq \Delta^I$, $f^I(R) = R^I \subseteq \Delta^I * \Delta^I$, avec C comme concept et R comme rôle.

La table 2.2 présente des opérateurs de base de logique de description pour formuler les concepts (prédicats unaires) et les rôles (prédicats binaires) dans un $Tbox$.

Les concepts, dans la LD, peuvent être primitifs ou définis. Les concepts primitifs sont comparables à des atomes et servent de base à la construction des concepts définis. On peut seulement leur associer des conditions nécessaires. Les concepts définis sont définis explicitement.

La LD supporte deux constructeurs permettant de définir des concepts complexes :

1. le constructeur de subsomption ($\sqsubseteq$) : Concept_définis $\sqsubseteq$ *expression_complexe*
2. le constructeur d'équivalence ($\equiv$) : Concept_définis $\equiv$ *expression_complexe*

Opérateurs	**Syntaxe**	**Sémantique**
Concepts	C,D	$C^I = \{x \in \Delta^I / x : C\}$
Rôles	R,T	$R^I = \{(x,y) \in \Delta^I * \Delta^I / (x,y) : R\}$
Négation	$\neg C$	$\Delta^I - C^I$
Quantificateur existentiel	$\exists R.C$	$\{x \in \Delta^I / \exists y, (x,y) \in R^I \wedge y \in C^I \}$
Quantificateur universel	$\forall R.C$	$\{x \in \Delta^I / \forall y, (x,y) \in R^I \Rightarrow y \in C^I \}$
Restriction inférieure	$(\leq nR).C$	$\{x \in \Delta^I / \|\{y, (x,y) \in R^I \wedge y \in C^I\}\| \leq n\}$
Restriction supérieure	$(\geq nR).C$	$\{x \in \Delta^I / \|\{y, (x,y) \in R^I \wedge y \in C^I\}\| \geq n \}$
Rôle inverse	S^-	$\{(x,y) \in \Delta^I * \Delta^I / (y,x) : R^I\}$
Conjonction	$C \prod D$	$C^I \bigcap D^I$
Disjonction	$C \coprod D$	$C^I \bigcup D^I$

TAB. 2.2 – Opérateurs (*de base*) de construction de concepts de la logique de description (LD)

Les *expressions complexes* sont construites en utilisant les opérateurs présentés dans la table 2.2 sur les concepts primitifs ou les autres concepts complexes. Par exemple :

$Femme \sqsubseteq Personne$

$\acute{E}tudiant \equiv (\geq$ *1 inscrit_à*$).\acute{E}cole \prod \neg Femme$

La *description complexe* du constructeur de subsomption n'est qu'une condition nécessaire, mais non suffisante pour vérifier si une instance appartient à la classe définie. Par contre, celle du constructeur d'équivalence est une condition nécessaire et suffisante.

Dans la *Abox*, les assertions représentant les instances des concepts dans la $Tbox$ sont exprimées comme suit :

– $a : C$, définissant une instance du concept C,

– $(a, b) : R$, représentant le lien entre deux instances.

Par exemple :

Nguyen :Étudiant

(Nguyen,ENSMA) :inscrit_à

La puissance des LD réside dans leur capacité à effectuer certains raisonnements. Les quatre principales opérations liées au raisonnement dans un système à base LD présentées par Amedeo Napoli [70] sont les suivantes :

1. Le *test de subsomption* permet de vérifier qu'un concept C_1 subsume un concept C_2.
2. Le *test de satisfiabilité d'un concept* C permet de vérifier qu'un concept C admet des instances (il existe au moins une interprétation I telle que $C^I \neq \phi$).
3. Le *test de satisfiabilité d'une base de connaissances* $\sum$ permet de vérifier que $\sum$ admet un modèle I : tout concept et toute assertion sont satisfaits par I.
4. Le *test d'instanciation* consiste à retrouver les concepts les plus spécifiques dont une instance i est instance.

Les langages comme DAML+OIL, OWL sont fondés sur les logiques de description. Quelques projets d'intégration de données utilisant LD afin de construire leurs ontologies qu'on peut citer sont : OBSERVER, SHOE.

Le modèle d'ontologie OWL [62], déjà cité, est un modèle permettant la définition de concepts primitifs et définis. Issu des logiques de description, les notions de classes, instances et propriétés sont faiblement couplées et peuvent même être définies en toute indépendance. Les instances ne respectent ainsi aucune structure définie. Elles peuvent prendre des valeurs pour des propriétés qui n'ont pas pour domaine leurs éventuelles classes de définition [49].

OWL fait la distinction entre deux types de propriétés : (1) les propriétés d'objet permettent de relier des instances à d'autres instances ($owl : ObjectProperty$), (2) les propriétés de type de données permettent de relier des instances à des valeurs de données ($owl : DatatypeProperty$). La définition des caractéristiques d'une propriété se fait à l'aide d'un axiome de propriété qui, dans sa forme minimale, ne fait qu'affirmer l'existence de la propriété :

<owl :ObjetctProperty rdf :ID="Père_de"/>

Si on considère que l'existence d'une propriété pour une instance donnée de l'ontologie constitue une fonction faisant correspondre à cette instance une autre instance ou une valeur de donnée, alors on peut préciser le domaine et le co-domaine de la propriété :

<owl :ObjetctProperty rdf :ID="Père_de">
<rdfs :domain rdf :resource="#$Personne$">
<rdfs :range rdf :resource="#$Personne$">
</owl :ObjetctProperty>

La définition d'une instance consiste à énoncer un axiome d'individu (ou un fait). Un axiome d'individu concerne généralement la déclaration de l'appartenance à une classe et les valeurs de propriété de cette instance. Un axiome d'individu s'exprime de la manière suivante :

<Personne rdf :ID="Philip">
<Marié_avec rdf :parseType="Françoise">
<Père_de rdf :parseType="Paul">
</Personne>

Le fait écrit dans l'exemple ci-dessus exprime l'existence d'une $Personne$ nommée "Philip" qui est le père de "Paul" et se marie avec "Françoise". Une difficulté qui peut apparaître dans le nommage des instances concerne les conflits de nom attribués aux instances. C'est la raison pour laquelle OWL propose un mécanisme permettant de lever cette ambiguité, à l'aide des opérateurs : $owl : sameAs$, $owl : differentFrom$, $owl : allDifferent$. Par exemple :

<rdf :Description rdf :about="dung_Nguyen">
<owl :sameAs rdf :resource="Nguyen_dung">
</rdf :Description>

Cet exemple permet de déclarer que les noms "dung_Nguyen" et "Nguyen_dung" dési-

gnent la même personne.

En résumé, le monde de LD se formalisme essentiellement sur l'inférence et la déduction : la relation de subsomption et la relation d'équivalence entre concepts. Pour cela, elles fournissent non seulement des opérateurs exprimant les éléments ontologiques canoniques (classes, propriétés, instances), mais également des opérateurs de classes (par exemple, $C \equiv C_1 \bigcup C_2$ ou $C^* \equiv (\leq nR).C$) permettant de définir des équivalences conceptuelles (afin de définir une $OCNC$).

Cependant, à notre connaissance, la plupart des langages issus des LD ne fournissent aucune expression sur les domaines concrets de données (comme *Real...*), ni aucune contrainte d'intégrité (par exemple, diamètre_interne < diamètre_externe). Il existe une extension de LD, *Domaine Concret* de la LD, permettant d'exprimer des expressions sur les domaines concrets. Jusqu'à aujourd'hui, aucun système de raisonnements ne supporte vraiment des inférences sur cette extension. Nous considérons donc que les domaines concrets ne sont guère utilisables dans les LD. De plus, les concepts primitifs d'ontologie sont juste décrits par un "commentaire" facultatif ce qui est tout à fait insuffisant pour permettre à un utilisateur humain de comprendre, sans ambiguïté, les concepts primitifs. Les ontologies basées sur les logiques de descriptions sont donc bien adaptées si l'objectif est de réaliser des inférences pouvant se ramener aux tests de subsomption ou d'instanciation. Elles sont par contre peu adaptées pour caractériser, de façon compréhensible par un utilisateur humain, l'ensemble des notions apparaissant dans les domaines assez vastes. Elles semblent également peu adaptées pour des ensembles d'instances importants : la seule proposition de système que nous connaissons et qui ne travaille pas en mémoire centrale gère jusqu'à un million d'instances, mais ne gère pas les propriétés. Enfin elles ne permettent pas de définir une propriété qui peut se déduire à partir de la valeur d'une autre propriété, par exemple : le grand père est le père du père ou de la mère, ou bien une longueur en centimètres est exactement égale à la longueur de la même caractéristique en mètre multipliée par 100.

2.4.3.2 F-Logic et opérateurs de propriétés

La F-Logic (appelée ainsi Frame-based Logic) est proposée par Michael Kifer et al. [52] pour combiner des possibilités de représentation des langages de "frame" et du calcul des prédicats. Elle est l'exemple le plus connu de langage opérationnel à base de frames dont le concept de frame a été introduit par Minsky en 1970 [68]. Les langages de frame sont une modélisation de base pour la représentation de connaissances dans le domaine de l'Intelligence Artificielle.

Un frame représente un objet, une collection d'objets ou un concept. Il est constitué d'une liste de slots. Un slot est identifié par son nom et peut avoir pour valeur une instance d'un type de donnée primitif (nombre, chaîne de caractères...) ou une référence à un ou plusieurs autres frames. Il s'agit, d'une part, de représenter une propriété du frame et, d'autre part, une association entre frames. Un frame est, dans ce contexte, un objet nommé

qui est utilisé pour représenter un certain concept dans un domaine. Entre les frames, il y a aussi la spécialisation qui entraîne l'héritage entre frames.

Opérateurs	**Syntaxe**	**Sémantique**
ET/OU/Négation logique	$AND/OR/NOT$	
Opérateur existentiel	$EXISTS$	
Quantificateur universel	$FORALL$	
Séparateur dans les règles entre la tête et le corps	$C_1 \leftarrow C_2$ $C_1 \rightarrow C_2$ $C_1 \leftrightarrow C_2$	(NOT C_2) OR C_1 (NOT C_1) OR C_2 $(C_1 \rightarrow C_2)$ AND $(C_2 \leftarrow C_1)$
Domaine d'une relation	$R =>> D$	le domaine de R est D
Déclaration/utilisation de relations d'un concept	$C[R =>> D]$	concept C a une relation R dont le domaine est D
Instance de	$o : C$	o est une instance de C
Sous-concept de	$C_2 : C_1$	C_2 est sous classes de C_1
Valeur d'une relation	$R- >> n$	R porte la valeur n
@	a_1[R@(a_2) $- >>$ b]	R de a_1 **avec** a_2 porte la valeur b

TAB. 2.3 – Opérateurs principaux de construction de concepts de la $F - Logic$

La table 2.3 présente les opérateurs de la F-Logic. Un concept dans la F-Logic est construit à travers les deux formulations ci-dessous :

1. **relation de subsomption** : "$C_2 : : C_1$", où le concept C_2 est subsumé par le concept C_1.
2. **description de concepts** : $C_1[R_1 =>> C; R_2 =>> REAL; R_3- >>$ "$Nguyen$"], où le concept C_1 a pour la relation R_1 qui prend ses valeurs dans les instances du concept C, la relation R_2 dont le type est $REAL$, et R_3 ayant la valeur "$Nguyen$".

Des instances de classes, correspondant à l'extension de chaque concept, peuvent être ajoutées, ainsi que des fonctions qui sont des types particuliers de relations liant un ensemble de classes à une valeur calculée à partir des valeurs des attributs des classes. Les instances sont décrites comme suit :

1. **création d'une instance** : "$o_1 : C_1$", signifiant que o_1 appartient à l'ensemble d'instances de C_1,
2. **description de l'instance** : $o_1[R_1- >> c; R_2- >> 3; R_3- >>$ "$Nguyen$"], signifiant que l'instance o_1 est décrite à travers les valeurs qui sont l'instance c, 3, "Nguyen" pour R_1, R_2, R_3 respectivement.

Dans la F-Logic, on utilise les règles afin de formuler des descriptions complexes. Les règles sont constituées de deux parties, la tête et le corps. La tête contient la connaissance

inférée par la règle ainsi que la déclaration des variables. Le corps contient les informations pour obtenir cette connaissance :

FORALL variables résultat ← condition
Par exemple : FORALL X,Y X[fils − >> Y] ← Y :man[père -> X]

Une autre formule logique présentée dans F-Logic est la requête. La structure d'une requête est proche de celle d'une règle, sauf que la requête n'infère pas de nouvelle connaissance. La tête de la requête ne comporte donc que les variables permettant d'obtenir le résultat de la requête :

FORALL variables ← requête
Par exemple : FORALL X,Y ← X : femme[son − >> Y[père -> abraham]]

F-Logic supporte également des prédicats qui correspondent à des fonctions déclarées dans le langage. Et il est possible d'utiliser dans le corps des prédicats comme des conditions. Basé sur la logique, la F-Logic offre peu de support pour raisonner. Par contre un des systèmes les plus connus basé sur la F-Logic OntoBroker [75], comporte un ensemble de prédicats prédéfinis qui contient en particulier :

- les **prédicats de manipulation de chaînes** : *isString, concat, constant2string, string2number,*
- les **prédicats mathématiques** : *less, lessorequal, greater, greaterorequal, +,-,*,/, sin, cos, tan,asin, acos, exp, sqrt, min, max, pow, ...*

  Par exemple, *FORALL X ← X is +(3,2).*

Les ontologies basées sur la F-Logic sont donc bien adaptées si l'on veut réaliser des inférences qui correspondent à datalog et aux bases de données déductives. En particulier, on peut très aisément exprimer le fait que le père du père est toujours le grand père. On sait également exprimer des contraintes d'intégrité qui doivent respecter les données. Au contraire, les F-Logic ne disposent pas des opérateurs de classes que l'on trouve dans les logiques de description.

2.4.3.3 PLIB et opérateurs de modularités

Conçu initialement pour définir précisément les concepts (classes et les propriétés) qui caractérisent les produits des domaines techniques, PLIB ne se concentre pas du tout sur l'équivalence entre concepts. Par contre, il offre un ensemble d'attributs permettant d'identifier et de définir précisément les concepts primitifs d'un domaine afin de fournir un vocabulaire canonial (pas redondant) permettant d'en décrire avec précision chacun des éléments. PLIB s'intéresse particulièrement aux propriétés numériques. Il permet d'expliquer dans quel contexte une valeur de propriété a été évaluée, et ce qui est l'unité de mesure de cette propriété. PLIB permet également d'exprimer toute sorte de contrainte

d'intégrité.

Dans ce modèle d'ontologie, les concepts sont modélisés en termes de classes hiérarchisées , caractérisées par des propriétés , typées par des domaines de valeurs. Chaque catégorie, propriété et type est défini dans le contexte le plus large où ils ont un sens précis, ce qui élimine le caractère très contextuel des modèles conceptuels usuels et également de OWL où il n'est pas possible de représenter le contexte d'évaluation d'une valeur. Enfin une catégorie est associée à toutes les propriétés qui la caractérisent, même si dans chaque contexte particulier, seul un petit sous-ensemble de propriétés sera en général utilisé.

Comparé aux deux modèles précédents, le modèle PLIB présente les caractéristiques suivantes :

1. visant à permettre l'échange de données, PLIB n'offre pour l'instant aucun opérateur pour exprimer les équivalences conceptuelles. Les ontologies PLIB sont toujours canoniques. Ce sont des ontologies de caractérisation et non de déduction.

2. le modèle est très riche pour décrire avec précision les concepts primitifs

3. visant le développement d'ontologies très vastes (pour couvrir tout le domaine technique) PLIB offre des mécanismes originaires de modularité d'ontologies qui interfacent une ontologie à une autre ontologie et importent les propriétés de la dernière dans la première. Cet opérateur de modularité s'appelle *"is_case_of"* : il s'agit d'une subsomption sans héritage.

4. PLIB possède plusieurs primitives qui sont apparues nécessaires dans certains domaines (en particulier les domaines techniques) et ne sont pas disponibles dans les autres modèles, par exemple la définition contextualisée de valeurs, les contraintes d'intégrité, la gestion des points de vue.

Le modèle d'ontologie sera détaillé dans la chapitre 3.

2.4.3.4 Conclusion sur les modèles d'ontologies

Cette présentation synthétique montre que si tous les modèles d'ontologie partagent les même concepts fondamentaux, chacun possède des opérateurs spécifiques adaptés aux problèmes qu'ils visent à résoudre. Certaines ontologies visent essentiellement l'inférence quand d'autres visent essentiellement la caractérisation. Une question qui reste actuellement ouverte est de savoir s'il convient (et s'il est possible) de réunir tous les constructeurs des différents modèles en un modèle unique, ou s'il est préférable, comme pour les langages de programmation, d'avoir différents modèles adaptés à des objectifs différents.

Pour résumer cette section, nous proposons la table 2.4 de comparaison des trois modèles d'ontologies.

Capacité d'expression	**LD**	**F-Logic**	**PLIB**
classe	+	+	+
subsomption	+	+	+
propriété	+	+	+
propriété dépendant du contexte	−	+	+
instances	+	+	$-/+^{(1)}$
opérateurs de classes	+	−	−
opérateurs de propriétés	−	+	−
contrainte d'intégrité	−	−	+
domaine concret	$-/+^{(2)}$	+	+
unité de mesure	−	−	+
opérateur de modularité	−	−	+
description lisible pour homme	−/+	−/+	+
(1) : *instances dans une ontologie à base PLIB n'est pas identifiée universellement.* (2) : *certains outils de raisonnement des LD commencent à être équipés d'un module lui permettant de travailler avec les domaines concrets de données*			

TAB. 2.4 – Comparaison de trois modèles d'ontologies : *LD,F-Logic,PLIB*. Les éléments en gras mettent en évidence les spécificités.

2.5 Ontologie conceptuelle et Intégration des données

Les ontologies conceptuelles représentent un outil intéressant pour résoudre les problèmes liés à l'hétérogénéité sémantique. L'ontologie fournit une représentation explicite de la sémantique des données pouvant servir de base à la mise en correspondance des modèles différents. Elle est utilisée dans la plupart des systèmes à la fois comme un schéma global de données et comme une interface d'interrogation. L'utilisation d'une ontologie spécifique au domaine permet de ramener les concepts à un référentiel unique et de mesurer la distance et le recouvrement entre eux. Les concepts de l'ontologie sont liés aux termes des sources de données dans le méta-modèle global. Ces liens sont utilisés pour identifier les sources de données pertinentes et pour transformer les requêtes en sous-requêtes locales sur les sources originelles.

Dans cette section, nous présentons d'abord les différences entre l'ontologie conceptuelle et le modèle conceptuel. Ceci permet de rendre clair l'intérêt d'ontologie conceptuelle pour intégrer des données hétérogènes. Ensuite, nous présentons une classification des approches d'utilisation d'ontologie conceptuelle dans l'intégration de données.

2.5.1 Différence entre ontologie/modèle conceptuel

Le mot "Ontologie" est devenu un mot à la mode qui est utilisé souvent au lieu du mot "modèle conceptuel". En effet, un modèle conceptuel par exemple un schéma EXPRESS développé dans le contexte de certaines normes comme ISO 10303 (STEP), est "une spécification explicite d'une conceptualisation". La définition habituelle n'est pas assez précise. Ainsi, comme remarqué par Guarino et Welty [42] :

> *"today (...ontology) is taken as nearly synonymous of knowledge engineering in AI, conceptual modeling in databases and domain modeling in OO design".*

Mais nous savons bien que la conception du modèle d'une base de données, en particulier, est réalisée selon une approche *prescriptive*. Cette approche a plusieurs implications [49] :

- seules les données pertinentes pour l'application cible sont décrites ;
- les données doivent respecter les définitions et contraintes définies dans le modèle conceptuel ;
- aucun fait n'est inconnu : *c'est l'hypothèse du monde fermé* ;
- la conceptualisation est faite selon le point de vue des concepteurs et avec leurs conventions ;
- le modèle conceptuel est optimisé pour l'application cible.

Une telle approche engendre les problèmes d'hétérogénéité que nous avons présentés dans la section 2.2. Pourtant la construction d'ontologie conceptuelle vise la conception d'un composant informatique générique capable de comprendre l'hétérogénéité des sources de données. Il est important donc d'expliquer la différence entre une ontologie et un modèle.

Une définition de MINSKY clarifie le rôle des modèles et souligne leur multiplicité et leur nécessité [67] : "*Pour un observateur B, un objet A^* est un modèle d'un objet A si B peut utiliser A^* pour répondre aux questions qui l'intéressent au sujet de A*". Cette définition souligne le caractère ternaire d'un modèle relationnel : elle dépend de l'objet (A) et de l'observateur (B), mais elle dépend également des questions que l'observateur se pose au sujet de A. Appliqué à la conception d'un modèle conceptuel, ceci montre que le modèle conceptuel dépend du contexte dans lequel le modèle conceptuel a été conçu. Le problème lorsque l'on conçoit un modèle informatique est que le contexte de modélisation est défini par les buts et l'environnement de ce système. Or, les buts et l'environnement de systèmes ne sont jamais exactement identiques. Les modèles conceptuels seront donc toujours au moins légèrement différents, et ces différences sont suffisantes pour les rendre *assez importants pour qu'il y ait des incompatibilités* entre eux.

Au contraire, le but d'une ontologie d'après la définition philosophique : "*the nature and the essence of things*" est d'être indépendant de tout contexte particulier. A la différence d'un modèle qui *prescrit* une base de données, l'ontologie décrit ce qui existe. Une ontologie sera ainsi conçue selon une *approche descriptive*. Cette approche vise à définir l'ensemble des concepts présents dans un domaine particulier et non pour une application

particulière. Les données d'une application référençant une ontologie ne satisferont pas forcément l'ensemble des définitions et contraintes exprimées dans celle-ci. Ces données manquantes sont considérées comme inconnues dans le cadre de l'application. C'est *l'hypothèse du monde ouvert* [49]. Enfin, contrairement à une approche prescritive, la conception d'une ontologie doit se faire dans un cadre *consensuel* sans contexte de conception ou alors en le définissant explicitement.

Dans ce but, il existe de plus en plus d'organismes internationaux dont l'objectif est de proposer des ontologies consensuelles (normalisées). Nous citons ci-dessous deux exemples d'ontologies normalisées :

- **l'OTA** (Open Travel Alliance [`http://www.opentravel.org/`] : a été créé en 1999. Il s'agit d'un consortium qui regroupe plus de 150 organisations relatives à l'industrie du voyage. Parmi ces organisations, on trouve d'une part des fournisseurs tels que : des compagnies aériennes, des hôtels, des agences de location de voitures, des compagnies d'assurance, etc., d'autre part des intermédiaires comme les GDS (Global Distribution Systems), les agences de voyages et les fournisseurs de technologie Web [38]. L'OTA vise à aider ces organisations à définir *un vocabulaire et une structure de messages commune*. En association avec la DISA (Data Interchange Standards Association), cet organisme a développé des normes de communication basées sur XML pour faciliter l'emploi du commerce électronique. L'OTA a publié environ 77 schémas XML [38] permettant de décrire des documents portant sur les besoins et les préférences des voyageurs pour des voyages d'affaires, des vacances, des voyages internationaux, des voyages en avion, des locations de voitures, des séjours en hôtel, etc. Ces schémas XML sont exploités pour créer des ontologies dans plusieurs projets concernant E-commerce ou l'intégration de données [88].
- Dans les domaines techniques, initiés en 1987 au niveau européen, l'objectif de **PLIB** était de permettre une modélisation informatique des catalogues de composants industriels afin de les rendre échangeables entre fournisseurs et utilisateurs. Pour réaliser ceci, le langage de modélisation EXPRESS [ISO 10303-11 :1994] a été utilisé pour représenter un catalogue sous forme d'instances d'un modèle. PLIB a en particulier donné lieu à un ensemble de normes ISO dans la série 13584 (Parts Library) dont les différentes parties ont été publiées entre 1996 et 2004. Parallèlement, des ontologies de domaines conformes à ce modèle ont été développées et normalisées au niveau international. La première publiée en 1998, décrit les principales catégories et propriétés de composants électroniques IEC 61360-4 :1998). Aujourd'hui plusieurs dizaines sont actuellement publiées ou en cours de développement. Certaines sont, ou visent à être, des normes (ISO 13399, ISO 10303-226, ISO 13584-501 et -511). D'autres sont développées par des consortiums industriels. L'ensemble des ontologies PLIB sont disponibles au [`www.plib.ensma.fr`].

La section suivante présentera quelques systèmes d'intégration de données à base ontologique antérieurs.

2.5.2 Utilisation d'ontologies conceptuelles pour l'intégration de données

Nous présentons dans cette section comment une ontologie conceptuelle (OC) est utilisée dans un système d'intégration de données. L'intégration de données à base ontologique peut être divisée en trois étapes. Ces dernières sont les suivantes :

1. la **représentation de la sémantique des données** qui vise à interpréter le sens de chaque source en l'associant à une ontologie locale. Ce processus est effectué d'une façon manuelle ou semi-automatique. En réalité, la sémantique de chaque source peut être découverte semi-automatiquement [87, 97] grâce aux techniques de fouilles de données proposées dans le domaine de l'Intelligence artificielle, par exemple les techniques linguistiques [13, 15, 103], la technique de machine d'apprentissage [30], etc. Mais son résultat est approximatif, il nécessite donc la présence de l'administrateur pour valider ce résultat.

   Cette étape peut être éliminée si chaque source contient *a priori* une ontologie locale, c'est-à-dire que la sémantique d'une source est déjà sauvegardée au sein de cette source quand on la crée.

2. l'**intégration sémantique** qui vise à intégrer les ontologies des sources. Elle consiste à établir les relations sémantiques (équivalence, subsomption) entre les concepts des ontologies. L'automatisation du processus d'intégration sémantique dépend des méthodes d'utilisation d'ontologies qui sont précisées ci-dessous.

3. l'**intégration de données** qui vise à peupler les données dans un entrepôt pour les systèmes matérialisés ou à construire des interfaces de requêtes pour les systèmes virtuels qui fournissent une vue unique sur les données. Cette étape peut être faite par des programmes génériques qui exploitent la correspondance ontologique établie dans l'étape précédente.

Nous nous concentrons ci-dessous, sur les méthodes d'utilisation d'ontologies au sein des systèmes d'intégration. Basé sur la nature de l'étape d'intégration sémantique, nous proposons de classifier les systèmes d'intégration à base ontologique en deux classes : (1) intégration sémantique *a posteriori*, et (2) intégration sémantique *a priori*.

2.5.2.1 Intégration sémantique *a posteriori*

L'approche d'intégration sémantique *a posteriori* est caractérisée par les principes suivants :

- les ontologies locales sont indépendantes ;
- l'étape d'intégration sémantique est donc effectuée d'une *façon manuelle* ou *semi-automatique*. Elle consiste à établir la correspondance entre les concepts primitifs des ontologies.
- dans ce contexte, deux structures d'intégration d'ontologies sont possibles :

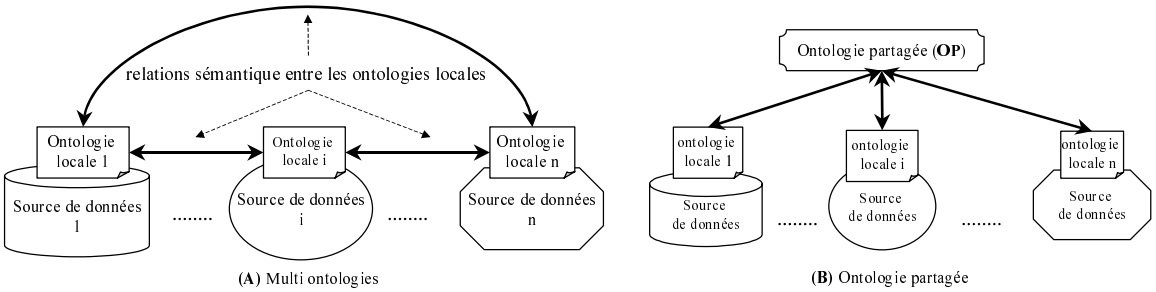

FIG. 2.13 – Utilisation d'ontologies conceptuelles dans l'approche d'intégration sémantique *a posteriori*

1. la structure "**multi-ontologies**" (voir la figure 2.13-A) :
   - la correspondance entre deux ontologies locales est établie directement de l'un à l'autre. Supposons qu'il existe n ontologies locales, il faut alors créer $[n * (n-1)/2]$ correspondance.
   - l'interface utilisateur peut être créée directement à partir d'une ontologie locale quelconque.
2. la structure "**ontologie partagée**" (voir la figure 2.13-B) :
   - la correspondance entre deux ontologies locales est établie indirectement à travers une ontologie référencée. Cette ontologie est appelée également *l'ontologie partagée* du système. Une ontologie locale n'est mise en correspondance qu'avec l'ontologie partagée. Et seuls les concepts locaux intéressés par le système sont mis en correspondance avec les concepts partagés. Supposons qu'il existe n ontologies locales, il faut alors créer $[n]$ articulations d'ontologies.
   - l'ontologie partagée est soit une ontologie spécifique au système, soit une ontologie normalisée indépendante du système.
   - l'interface utilisateur est créée à partir de l'ontologie partagée.

L'avantage d'une telle approche est l'indépendance des sources de données intégrées. Par contre, l'intégration sémantique n'est pas automatique. Pour la structure "multi-ontologies", le nombre de mise en correspondance entre les ontologies est important. Pour la structure "ontologie partagée", ce nombre est diminué ; et l'interface utilisateur correspond à l'ontologie partagée.

Parmi les systèmes d'intégration qui suivent cette approche, nous pouvons citer OBSERVER [63] pour la structure "multi-ontologies", et KRAFT [106] pour la structure "ontologie partagée" :

- **OBSERVER** (Ontology Based System Enhanced with Relationships for Vocabulary hEterogeneity Resolution) [63]. Le but de ce projet est de fournir un système permettant l'accès transparent et uniforme à des données indépendamment de leurs localisations, leurs formats de stockage et de leurs conceptualisations. OBSERVER associe à chaque source de données une ontologie qui conceptualise son contenu.

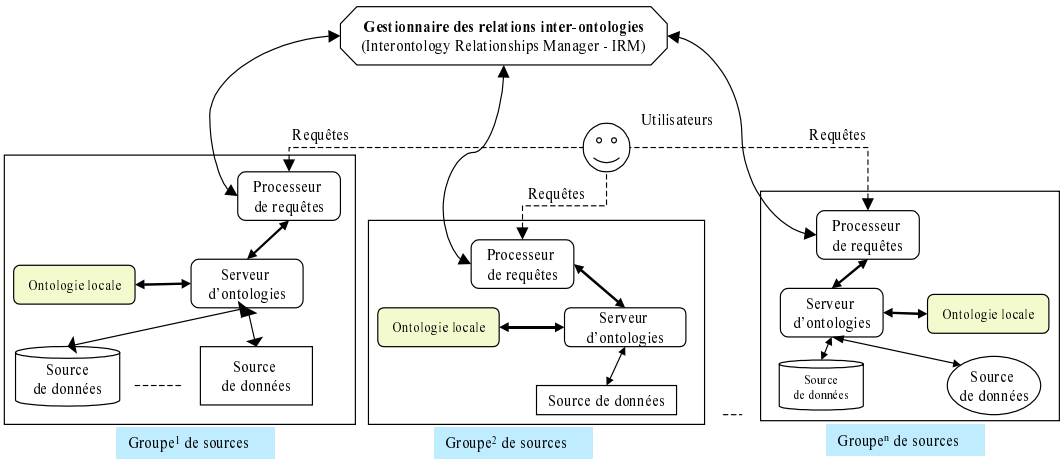

FIG. 2.14 – L'architecture de OBSERVER

Cette association ontologie-source de données est matérialisée par des liens entre les concepts de l'ontologie et les termes de la source de données. L'intégration de plusieurs sources de données se fait par les liens sémantiques entre les concepts des ontologies. Les ontologies sont représentées dans OBSERVER en utilisant CLASSIC [63]. Le système d'intégration OBSERVER (voir la figure 2.14) se compose de groupes de sources de données. Chaque groupe possède une ontologie locale, un serveur d'ontologies et un processeur de requêtes.

1. l'ontologie locale : contient les définitions des concepts locaux qui représentent la sémantique des sources de données de ce groupe.
2. le serveur d'ontologie : contient les explications sur les termes utilisés dans l'ontologie locale. Il garde également les liens des concepts ontologiques vers les données (ou plutôt vers les structures de données). Cela permet de retrouver les informations à partir des sources de données et de fournir les informations nécessaires à l'administrateur pour établir les relations sémantiques entre les ontologies locales des groupes de sources différents.
3. le processeur de requêtes : présente à l'utilisateur une interface lui permettant de formuler sa requête en termes des concepts présents au niveau de chaque groupe.

Le lien entre groupes de sources est assuré par le module de gestionnaire des relations inter-ontologies. OBSERVER stocke dans ce module toutes les informations concernant les relations sémantiques (subsomption/équivalence) entre des concepts appartenant à différentes ontologies. Les requêtes du système sont d'abord formulées à partir des concepts d'une ontologie choisie par l'utilisateur. Elles sont ensuite analysées par le système en utilisant les mécanismes d'inférence ontologique afin de déterminer les sources pertinentes et de transformer la requête dans le langage de la source. D'après OBSERVER, sa stratégie permet d'éviter de mettre en place une on-

tologie globale et de développer des ontologies spécifiques aux besoins d'utilisateurs qu'on peut extraire à partir des ontologies locales.

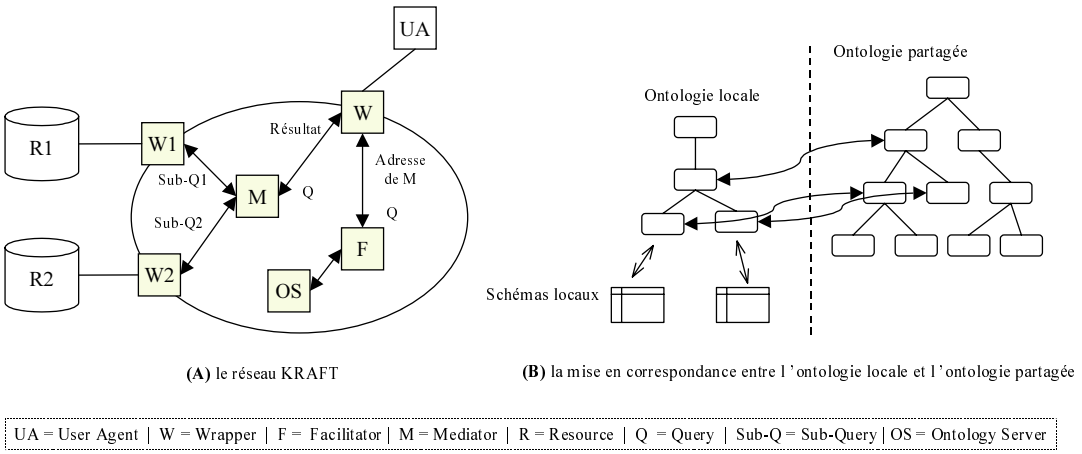

FIG. 2.15 – Architecture du système d'intégration proposée par le projet KRAFT

– **KRAFT (Knowledge Reuse and Fusion/Transformation)** [106, 85], par exemple, est un projet de recherche coopérative entre trois universités britanniques (l'Université de Aberdeen, l'Université de Cardiff et l'Université de Liverpool) avec BT (British Telecommunication). KRAFT propose une architecture d'agents dans le but d'intégration des sources de données hétérogènes. Dans KRAFT, les sources de données possèdent leurs ontologies propres. Ces ontologies sont décrites indépendamment. Elles ne sont pas obligées d'avoir un même format de représentation. Au contraire d'OBSERVER, une ontologie locale est mise uniquement en correspondance avec l'ontologie partagée du système (voir la figure 2.15). Cette mise en correspondance est faite d'une façon manuelle. KRAFT utilise le langage CIF (Constraint Interchange Format) pour spécifier la correspondance ontologique. Un système d'intégration KRAFT se compose des types d'agents ci-dessous (voir la figure 2.15) :

  1. User Agent (UA) : représente l'interface utilisateur.
  2. Wrapper (W) : représente l'interface entre une (des) source(s) de données ou une (des) UA et l'intérieur du système. W fournit des services traduisant des formats externes en un format interne (KRAFT message). W contient également la correspondance (en le format CIF) entre l'ontologie partagée et l'ontologie de la source. Cette correspondance est exploitée ensuite pour convertir les concepts locaux vers les concepts communs. A l'intérieur du système KRAFT, toutes les informations sont donc homogènes.
  3. Facilitator (F) : s'occupe d'identifier les agents médiateur (M) et/ou les agents W (correspondant aux sources de données) qui peuvent être utiles pour répondre à une requête.

4. Mediator (M) : se charge de répondre des requêtes complexes. Il reçoit des requêtes, puis les décompose en requêtes simples. Ces requêtes simples sont ensuite envoyées aux W correspondants. Pour les requêtes qu'il ne peut pas décomposer, il les signale à F. F cherche ensuite un autre M qui peut répondre à ces requêtes.

5. Ontology Server (OS) : est l'agent qui contient des informations concernant l'ontologie partagée du système. Il répond aux questions concernant les concepts du domaine posées par F.

En résumé, l'intégration de données par l'intégration sémantique *a posteriori* est proposée pour le cas où les sources de données intégrées possèdent des ontologies locales indépendantes. Une ontologie locale respecte uniquement sa source de données. Ainsi, l'intégration sémantique est faite de **façon manuelle**. Elle exige donc une bonne compréhension de l'administrateur du système sur chaque source. Ceci limite la scalabilité du système.

2.5.2.2 Intégration sémantique *a priori*

L'approche d'intégration sémantique *a priori* est caractérisée par les principes suivants (voir la figure 2.16) :

- les administrateurs veulent une communication directe entre les sources de données intégrées. Chaque source reprend *a priori* des concepts dans une ontologie de domaine pré-existante pour construire son ontologie locale. Cette ontologie de domaine est considérée comme *l'ontologie globale* du système. L'ontologie locale peut être vue comme un sous-ensemble de l'ontologie globale.
- l'intégration sémantique est donc naturellement **automatique** grâce aux relations sémantiques **a priori** entre les ontologies locales (ces relations sont celles entre les concepts dans l'ontologie globale).
- l'interface utilisateur est créée à partir de l'ontologie globale.
- un système d'intégration de données suivant cette approche n'intègre que les données dont la sémantique est présentée par l'ontologie globale.

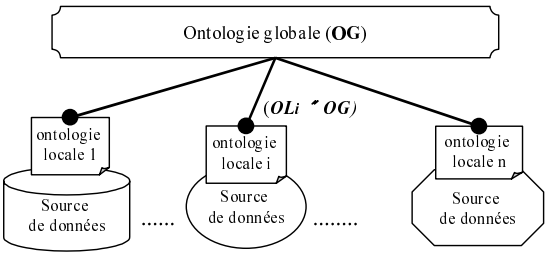

FIG. 2.16 – Utilisation d'ontologies conceptuelles dans l'approche d'intégration sémantique *a priori*

Cette approche permet d'intégrer facilement une nouvelle source dans le système si la sémantique de cette source est couverte par *OG*. Par contre, cet aspect limite les sources de données au niveau de leurs indépendances. Parmi les projets utilisant cette méthode, nous pouvons citer SIMS[7], PICSEL [35, 38], OntoBroker [28], BUSTER [105], COIN [37].

PICSEL, par exemple, propose une structure médiateur (voir la figure 2.17) qui permet d'interroger des sources d'information multiples, hétérogènes et éventuellement réparties. Les systèmes médiateurs auxquels PICSEL s'intéresse regroupent un ensemble important de sources d'information XML relatives à un même domaine d'application. Dans le PICSEL : "*une ontologie est un élément central. Son rôle est double. D'une part, elle fournit aux utilisateurs un vocabulaire de base approprié pour formuler leurs requêtes et interroger les sources auxquelles le médiateur a donné accès. D'autre part, elle permet la description des connaissances contenues dans les sources d'information interrogeables à l'aide d'un même vocabulaire et établit, de ce fait, une connexion entre elles* [38]".

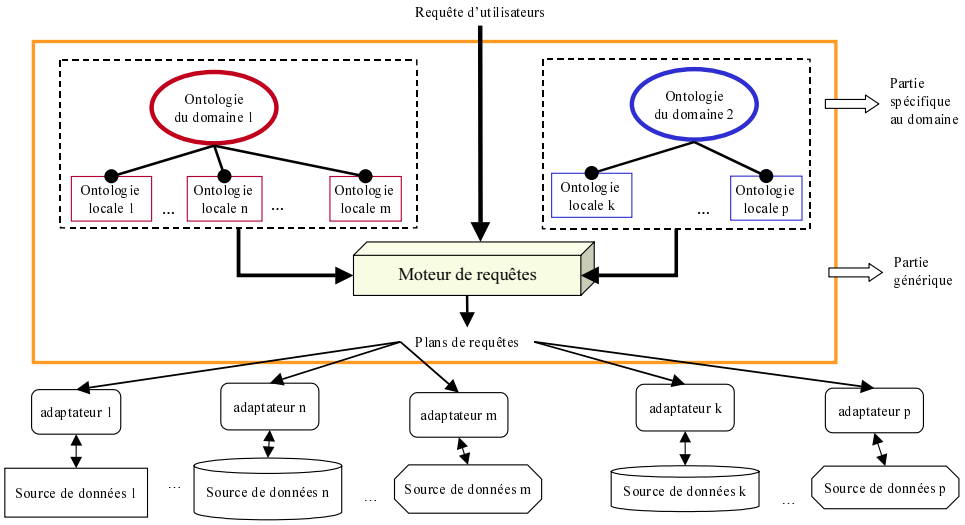

FIG. 2.17 – Architecture du système médiateur PICSEL

Le système de médiateur PICSEL comporte un moteur de requêtes générique et une partie de base de connaissances spécifique au domaine :

1. le moteur de requêtes est conçu d'une façon générique pour être utilisable quel que soit le domaine d'application.
2. la base de connaissances est spécifique au domaine appliqué. Elle se compose d'une ontologie du domaine et des ontologies locales. L'ontologie du domaine pré-existante modélise le domaine d'application et fournit ainsi, un vocabulaire structuré servant de support à l'expression des requêtes. Les ontologies locales décrivant le contenu des sources d'information sont formées *a priori* à partir des termes de l'ontologie du domaine.

PICSEL utilise CARIN [35] comme le langage de représentation de connaissances. Ce langage est un formalisme qui combine dans un cadre logique et homogène un langage de règles et un langage de classes. L'ontologie du domaine est représentée dans le formalisme CARIN-ALN, à l'aide de deux composantes : la composante terminologique et la composante déductive [38]. PICSEL possède également un langage de vues [38] et un langage de requêtes permettant d'exprimer, en termes de l'ontologie du domaine, respectivement, le contenu des sources et les requêtes des utilisateurs. L'ontologie globale (l'Ontologie du domaine) dans PICSEL est créée à travers l'ONTOMEDIA (Ontologie pour un MEDIAteur)[38]. Le système ONTOMEDIA permet de construire de façon semi-automatique une ontologie du domaine. Il s'agit d'un analyseur syntaxique dont la fonctionnalité consiste à appliquer un ensemble d'heuristiques pour identifier, parmi un ensemble de DTDs relatives à un domaine, les termes qui vont composer l'ontologie de ce domaine, puis de les organiser entre eux.

Un autre exemple est **OntoBroker** [28]. L'approche menée dans ce projet consiste à annoter les documents HTML du Web par les concepts d'une ontologie, de collecter les informations et les stocker d'une façon centralisée dans une base de connaissance qui sera interrogée par un langage de requêtes dédiée. L'idée centrale dans cette approche est d'utiliser une ontologie pré-existante pour décrire la base de connaissance, formuler les requêtes, ajouter de la sémantique aux documents HTML du Web et pour la conception de DTD de document XML. L'ontologie est utilisée par un moteur d'inférence pour inférer de nouveaux faits dans la base de connaissance. Le langage de requêtes utilise l'ontologie pour assister l'utilisateur dans le choix des termes de sa requête.

En résumé, l'intégration de données par l'intégration sémantique *a priori* permet **une automatisation effective** pour autant que chaque source référence exactement la même ontologie, sans possibilité d'extension ou d'adaptation. L'ontologie partagée est en fait un schéma global, et, en conséquence, chaque source locale a moins d'autonomie.

2.6 Positionnement de notre approche

Notre domaine d'application cible est le commerce électronique professionnel et l'échange de données techniques dans le domaine des composants industriels (e-ingénierie). Il s'agit, d'une part, de pouvoir rechercher de façon entièrement automatique quel fournisseur présent sur la Toile est susceptible de fournir un roulement à billes ayant des caractéristiques techniques données (par exemple, supporter une charge radiale de 100 Newton et axiale de 6 Newton avec une durée de vie de 2000 H en tournant à 500 t/s), et ceci quelle que soit la structure particulière du "catalogue" susceptible de le contenir. Il s'agit, d'autre part, de pouvoir intégrer automatiquement les catalogues de différents fournisseurs au sein d'une base de données d'entreprise utilisatrice, en offrant, sans aucune programmation, des possibilités d'accès ergonomiques.

L'état de l'art que nous avons présenté montre que nous pouvons atteindre une inté-

gration automatique de différentes sources de données. Il s'agit d'utiliser des ontologies conceptuelles dans une perspective d'intégration sémantique *a priori*. Une telle approche n'élimine pas la nécessité d'une réflexion humaine pour identifier deux conceptualisations différentes d'une même réalité. Mais, elle demande que cette réflexion soit faite *a priori*, lors de la mise à disposition de la source de données, et non *a posteriori*, pendant la phase d'intégration sémantique. Elle exige donc l'existence *a priori* d'une ontologie de domaine consensuelle. Les approches d'intégration sémantique *a priori* existant dans la littérature ont néanmoins deux inconvénients majeurs :

1. elles interdisent de représenter des concepts qui n'existent pas dans l'ontologie partagée, or, par définition, une ontologie partagée ne définie que ce qui partagée par toute une communauté. Mais chaque membre de la communauté, et c'est trivialement vrai pour le commerce électronique, souhaite aussi définir des concepts non partagés.
2. elle impose de calquer la structure de la base de données sur la structure de l'ontologie alors que chaque utilisateur peut souhaiter utiliser des systèmes de gestion différents (par exemple relationnel quand le modèle d'ontologie est orienté objet) ou des structures de schémas différents (par exemple optimiser par rapport au seul sous-ensemble de l'ontologie pertinent pour chaque cas particulier).

Ce sont les problèmes que nous visons à résoudre dans le travail de cette thèse. Ce travail s'inscrit dans le contexte d'une approche d'intégration sémantique *a priori*, et est associé à trois hypothèses :

- il existe une ontologie de domaine recouvrant la totalité des termes consensuels, et
- les administrateurs (fournisseurs) des sources de données veulent communiquer entre eux, mais
- les besoins sont différents et ils souhaitent une grande autonomie schématique.

Dans un tel contexte, l'approche d'intégration que nous proposons est basée sur les deux principes suivants :

1. chaque base de données contient outre ses données, son schéma et l'ontologie (locale) qui en définit le sens. Une telle source est appelée base de données à base ontologique : BDBO (nous la détaillerons dans le chapitre 3) ;
2. chaque ontologie locale s'articule *a priori* avec une (ou des) ontologie(s) partagée(s) de domaine. Elle référence "autant que cela est possible" cette ontologie partagée (nous allons définir précisément, au chapitre 4, ce que signifie "référencer autant que cela est possible" une ontologie partagée).

Nous proposons d'appeler une telle approche l'approche d'intégration sémantique *a priori* par articulation d'ontologies. Notre système d'intégration consiste à permettre à la fois : (1) à chaque concepteur de choisir librement son schéma, et (2) à réaliser l'intégration automatique des différentes bases de données.

Notons que ce point de vue *a priori* est assez similaire de celui adopté dans le projet Padoue pour une architecture de médiation dans un système de pair à pair. Dans ce

projet également, chaque pair doit a priori adapter son schéma public sur l'existant. Cette approche qui peut se classifier comme une approche virtuelle, GaV, et automatique possède une différence avec notre propre approche :

1. Dans notre approche, l'exportation des instances de chaque source peut être faite soit par rapport au schéma intégré (voir opérateur ProjOnto dans le chapitre 4) soit par rapport au schéma natif de la source (voir opérateur ExtendOnto dans le chapitre 4) alors que dans le projet Padoue elle doit être faite par rapport au schéma intégré.
2. L'approche que nous proposons permet non seulement d'étendre les propriétés existantes (comme Padoue), mais aussi d'étendre les classes existantes.

Le modèle d'ontologie PLIB est le noyau de notre système d'intégration. L'articulation d'ontologies proposée dans cette approche est construite en utilisant l'opérateur de modularité d'ontologies de PLIB : "is-case-of", il s'agit d'une subsomption sans héritage (ce type de subsomption sera étudié plus en détail dans le chapitre 3). Ce mécanisme de modularité interface chaque ontologie locale à l'ontologie partagée et importe les propriétés de la dernière dans la première. Cette approche orientée entité dans sa représentation de la correspondance existante entre le niveau global et le niveau local, permet alors l'adjonction complètement automatique d'une nouvelle source dans un système intégré, que ce soit dans une perspective d'entrepôt, ou dans une perspective médiateur.

2.7 Conclusion

Dans ce chapitre, nous avons défini la problématique de l'intégration de données, à savoir l'hétérogénéité de données. Cette hétérogénéité provient des choix différents qui sont faits pour représenter des faits du monde réel dans un format informatique. La question fondamentale lorsque l'on veut faire interopérer des bases de données hétérogènes est d'une part, l'identification de conflits entre les concepts dans des sources différentes qui ont des liens sémantiques, d'autre part, la résolution de ces conflits entre les concepts sémantiquement liés. Une taxonomie des conflits sémantiques qu'il convient de résoudre a été présentée : (1) conflits de représentations, (2) conflits de noms, (3) conflits de contextes, (4) conflits de mesure de valeur.

Nous avons proposé une classification des approches d'intégration de données existantes. Cette classification est faite en se basant sur les trois critères orthogonaux suivants : (1) la représentation de données intégrées, (2) le sens de la mise en correspondance entre schéma global et schéma local, et (3) la nature du processus d'intégration. Le premier critère nous permet de déterminer le type de stockage de données du système d'intégration qui est virtuel ou matériel. Le deuxième nous permet d'identifier le sens de mise en correspondance entre schéma global et schéma local. Ce sens influence directement la possibilité de passage à l'échelle et la complexité de traitement de requête du système d'intégration.

Le dernier spécifie si le processus d'intégration de données est effectué d'une façon manuelle, semi-automatique ou automatique. Ce critère devient essentiel lorsque l'on veut intégrer un nombre important de sources de données indépendantes.

Parmi les approches d'intégration de données, l'utilisation d'ontologies conceptuelles apparaît comme la seule approche assurant l'automatisation du processus d'intégration sémantique de données. Nous avons discuté les points communs et les différences principales entre les modèles d'ontologies conceptuelles. Cette présentation synthétique a montré que si tous les modèles d'ontologies partagent les même concepts fondamentaux, chacun possède des opérateurs spécifiques adaptés aux problèmes qu'ils visent à résoudre. Certaines ontologies visent essentiellement l'inférence quand d'autres visent la caractérisation et le partage de l'information.

L'utilisation d'ontologies conceptuelles pour développer des systèmes d'intégration est récente, mais devient populaire. Nous avons présenté comment les approches existantes utilisent des ontologies conceptuelles pour résoudre les conflits sémantiques de données. Nous classifions ces approches dans deux catégories : (1) l'intégration sémantique a posteriori, et (2) l'intégration sémantique a priori. La première approche permet de garder l'indépendance de chaque source de données, mais l'intégration sémantique est faite manuellement. Au contraire, la deuxième approche n'intègre que les sources de données dont la sémantique est représenté *a priori* en connexion avec une ontologie de domaine, et en conséquence son processus d'intégration sémantique est automatique. Notre travail dans cette thèse se situe dans la deuxième approche. Mais l'approche d'intégration que nous proposons consiste à *concilier l'intégration automatique* et *l'autonomie schématique des sources*.

Nous présentons dans les chapitres suivants la mise en oeuvre de cette approche dans **une perspective d'entrepôt**.

Chapitre 3

Modèle PLIB et Base de Données à Base Ontologique

3.1 Introduction

Nous avons présenté dans le chapitre précédent l'objectif de notre application cible, à savoir l'intégration automatique de données techniques. Notons que, dans les domaines techniques, les notions d'interchangeabilité et de normalisation sont très développées. Un vocabulaire technique consensuel existe donc déjà, de façon informelle, pour les termes essentiels de chaque domaine. L'intégration de données dans ce domaine présente néanmoins trois difficultés : (1) ces vocabulaires techniques n'existent pas, *a priori*, sous une forme exploitable automatiquement, (2) ces vocabulaires ne couvrent pas les innovations qui apparaissent de façon continue, et (3) chaque base de données, fournisseur ou utilisateur, et chaque catalogue électronique même s'il référence le vocabulaire commun, utilise une structure et une terminologie qui lui est propre.

Afin de résoudre ces difficultés, tout au long des années 90, un modèle d'ontologie adapté au domaine (ISO 13584-42, 1998), ainsi qu'un modèle d'échange d'instances d'entités décrites en termes de ces ontologies ont été développés au sein du laboratoire LISI. Ceci constituait le projet PLIB dont les résultats ont en particulier donné lieu à un ensemble de normes ISO dans la série 13584 (Parts Library) dont le dernier document vient d'être publié (ISO 13584-25, 2004). Basé sur le modèle d'ontologies PLIB, le prototype d'intégration proposé dans cette thèse permet d'intégrer d'une façon automatique les sources de données ayant une structure spécifique qui consiste à représenter dans les sources non seulement leurs ontologies et leurs schémas propres, mais également l'articulation avec la ou les ontologie(s) partagée(s) sur lesquelles elles s'engagent, et la capacité de répondre à une requête en terme de cette ou ces ontologies. C'est la notion de **B**ase de **D**onnées à **B**ase **O**ntologique (BDBO) [84] qui est développée au sein du LISI et est en train d'être validée dans différents environnements.

Ce chapitre est consacré à la présentation du modèle d'ontologie PLIB et de la structure

de BDBO que nous proposons de donner *a priori* aux différentes sources pour rendre leur intégration automatique faisable. Le contenu de ce chapitre est le suivant.

Dans la deuxième partie, nous présentons les principes de bases de la représentation d'une ontologie dans le modèle PLIB. Ces principes permettent de montrer les capacités du modèle PLIB ainsi que les caractéristiques d'une ontologie PLIB. La troisième partie présente la structure d'une BDBO. Cette structure présente deux caractéristiques. D'une part, elle permet de gérer à la fois des ontologies et des données. D'autre part, elle permet d'associer à chaque donnée le concept ontologique qui en définit le sens. Nous décrivons ensuite, dans la quatrième partie, le langage EXPRESS qui est le formalisme de modélisation utilisé par PLIB. Trois dimensions de modélisation de EXPRESS sont ici abordées : (1) la dimension structurelle (classe et type), (2) la dimension descriptive (attributs et relations), et (3) la dimension procédurale (contraintes d'intégrité et fonctions de dérivation). Cette présentation explique pourquoi EXPRESS a été choisi comme le formalisme de modélisation pour PLIB. Cette partie introduit également l'environnement ECCO qui est l'environnement de compilation d'EXPRESS et qui comporte également une base de données. ECCO est l'outil d'implémentation principal que nous utilisons afin de valider notre prototype d'intégration. Avant de conclure ce chapitre, nous présentons l'outil PLIBEditor qui est utilisé pour éditer les ontologies et les BDBOs.

3.2 Modèle PLIB

Initiée en 1987 au niveau européen [78], puis développée depuis 1990 au niveau ISO, la norme ISO 13584 (PLIB) est la norme destinée à permettre une modélisation informatique des catalogues de composants industriels afin de les rendre échangeables entre fournisseurs et utilisateurs. Un tel catalogue propose une classification des classes de composants industriels représentés, ainsi qu'une description de toutes les propriétés s'appliquant à celle-ci. En conséquence, un catalogue est une ontologie permettant de représenter une partie des concepts présents dans le domaine des composants industriels. Le modèle, publié en 1998, [ISO13584-42 :98] est complété récemment par des extensions dans les normes ISO 13584-24 et -25 est donc un modèle d'ontologies.

Dans cette section, les concepts fondamentaux du modèle PLIB seront présentés.

3.2.1 Modèle PLIB : modèle de l'ontologie de domaine

Avant de détailler PLIB, nous présentons, dans la partie suivante, une taxonomie des concepts modélisés dans ce modèle.

3.2.1.1 Taxonomie des concepts modélisés

Une ontologie PLIB permet la description de **classes** (class), de **propriétés** (property), de **domaines de valeurs** (data_type) et d'**instances**. Une classe est une collec-

tion d'objets définie en intention. Une propriété est une relation binaire entre deux classes ou entre une classe et un domaine de valeurs. Un domaine de valeurs est un ensemble mathématique défini en extension ou en intention. Une instance représente un objet appartenant à une classe. Enfin, toute définition ontologique émane d'une certaine **source** (supplier) qui en assume la responsabilité.

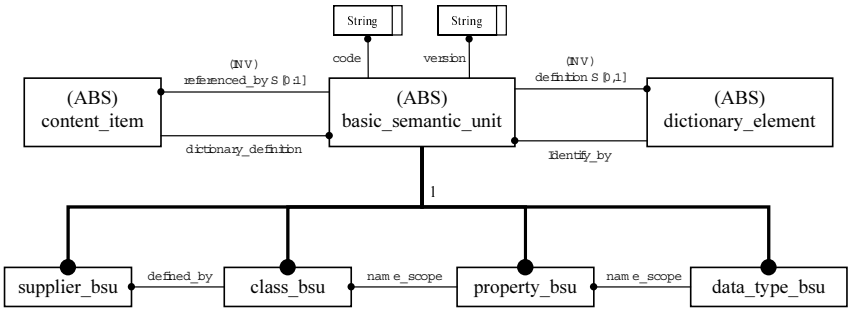

FIG. 3.1 – Identification d'un concept PLIB par BSU

Chaque concept d'une ontologie PLIB est défini par trois éléments :

1. une entité d'identification, appelée unité sémantique atomique (**basic_semantic_unit** ou **BSU**), qui permet une identification unique de ce concept ;
2. sa définition (**dictionary_element**) qui décrit ce concept par un ensemble de propriétés ;
3. ses instances (**content_item**).

La notion de BSU est modélisée par le schéma EXPRESS-G de la figure 3.1 [9], où :
– les traits gros représentent l'héritage,
– les traits fins représentent les attributs.

On y remarque qu'un BSU n'est pas forcement associé à sa définition (la cardinalité de *definition* est S[0 :1] ce qui *signifie* [0 :1]), un BSU permet donc une référence externe.

3.2.1.2 Identification d'un concept

Afin de pouvoir référencer de façon non ambiguë et multilingue n'importe lequel de ces concepts, PLIB comporte un schéma d'identification universel (GUI : "Globally Unique Identifier"). Chaque source potentielle est associée à un identifiant unique (en général préexistant pour toute organisation ou établissement ; par exemple en France il est construit sur les codes SIRET ou SIRENE). Chaque source doit alors attribuer un code unique à chacune des classes qu'elle définit. Enfin le code d'une propriété doit être unique pour une classe et toutes ses sous-classes. La concaténation de ces codes permet alors d'identifier de façon unique et universelle chacun des concepts ci-dessus. C'est le simple code, appelé un BSU (Basic Semantic Unit), qu'il sera suffisant de référencer pour caractériser une classe ou une propriété.

[9] un résumé des notions EXPRESS est présenté dans la section 3.4

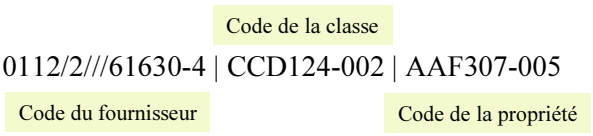

FIG. 3.2 – Identification du facteur de perméabilité d'un matériau magnétique à une fréquence donnée (le code 0112/2///61630-4 caractérise en effet la norme IEC 61360-4 en tant que source)

L'exemple dans la figure 3.2 représente le BSU d'une propriété technique très précise définie dans l'ontologie PLIB normalisée dans la norme IEC 61360-4. Cette identification permet de référencer des concepts qui ne sont pas définis dans la même ontologie. Ceci est notamment utile lors de l'échange d'ontologies, pour ne transférer que les définitions utiles pour le site receveur. Ce code pourra donc être utilisé pour résoudre les problèmes de conflits de noms évoqués dans la section 2.2.2 du chapitre précédent.

L'identification des concepts par un BSU fournit un mécanisme qui permet d'identifier de façon universelle les concepts. Cette identification est associée à une définition de concept qui est appelée en PLIB la "définition dictionnaire". La définition dictionnaire consiste à décrire sous forme informatique chacun des concepts identifiés de façon à la fois compréhensible par un humain et traitable par une machine les différents concepts de PLIB. Cette définition est multi-langues et la figure 3.3 en montre un exemple très simplifié (seuls les noms sont représentés). Dans cette figure, deux langues (anglais et français) sont utilisées pour une ontologie de composants informatiques. Grâce au mécanisme BSU, on peut traiter automatiquement des conflits de noms.

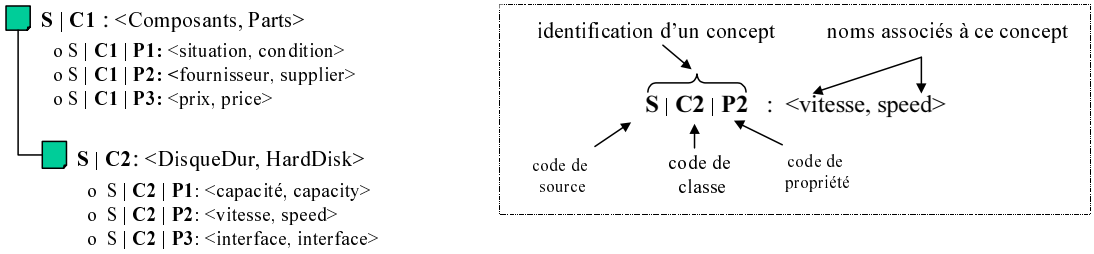

FIG. 3.3 – Exemple de multi langages

Nous présentons dans les parties qui suivent comment un concept est défini dans PLIB.

3.2.1.3 Définition d'un concept : connaissance structurelle

D'un point de vue structurel, un univers de composants, et, en particulier, un catalogue de composants, est naturellement appréhendé au travers de classes et de hiérarchies de classes. C'est la méthode de base pour organiser la connaissance pour un être humain [27]. Tous les catalogues papiers sont structurés de cette façon. Dans PLIB, les catégories de

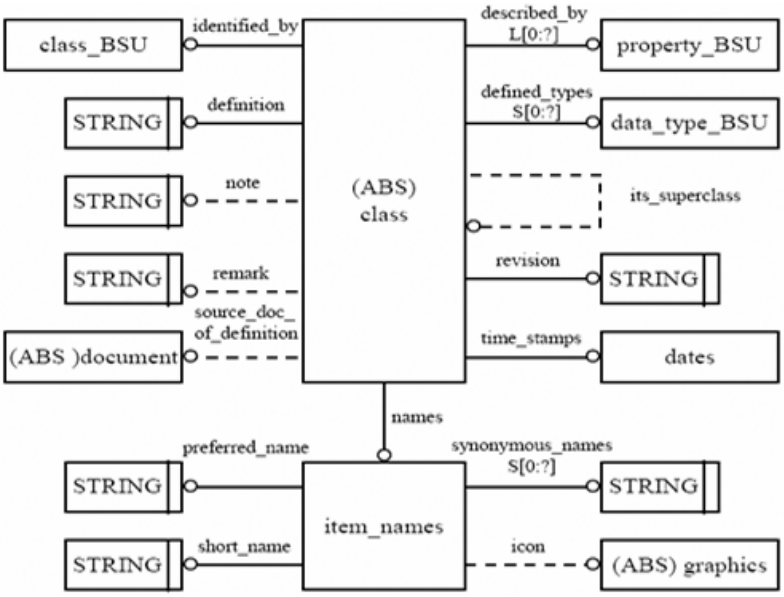

FIG. 3.4 – Schéma EXPRESS-G d'une classe en PLIB

composants doivent être modélisées par des hiérarchies de classes. La figure 3.4 présente le schéma EXPRESS-G d'une classe PLIB. Dans cette notation, un trait fin pointillé signifie qu'un attribut, ou une relation, est optionnel (notation équivalente à S[0 :1]). L[0 : ?] et S[0 : ?] représentent des cardinalités [0 :n] avec (L :List), ou sans (S :Set), relation d'ordre. Parmi les éléments du modèle qui permettent de caractériser une classe PLIB, citons : les relations et attributs suivants.

- Une classe est identifiée par un class_BSU (l'attribut *identified_by*) et est caractérisée par une liste de *property_BSU* (l'attribut *described_by*). Elle peut avoir éventuellement une super-classe (l'attribut *its_superclass*) ;
- *time_stamps*, *revision* et *version* permettent d'indiquer une date de version et de révision pour un concept. Si un changement sur la définition d'un concept ne modifie ni son sens, ni son utilisation, il sera qualifié de révision ; sinon, c'est un changement de version du concept.
- *names*, *definition*, *note* et *remark* permettent d'associer des mots d'une ou plusieurs langues naturelles à un concept.

Afin de structurer les classes d'une ontologie, le modèle PLIB implémente deux types de subsomption entre deux classes : l'une est la relation d'héritage et l'autre est la relation d'importation. Nous allons détailler ci-dessous ces deux types de relation.

3.2.1.3.1 Relation d'héritage *OOSub* (**is-a**) : est l'habituelle relation de d'héritage de l'approche objet. Une classe de composants qui spécialise une autre classe de composants par la relation *OOSub* hérite de toutes les propriétés qui caractérisent les composants de cette classe (attribut *described_by*). On peut ainsi définir différents niveaux d'abstraction dans la conception d'une ontologie PLIB

3.2.1.3.2 Relation d'importation *OntoSub* (**is-case-of**) est une relation de subsomption qui n'est pas associée à un mécanisme d'héritage automatique. Cette relation est différente de la relation "is-a" dans le sens où la classe se déclarant "is-case-of" doit explicitement importer les propriétés qu'elle souhaite à partir de la classe dont elle est "is-case-of". Cette différence donne un plus haut degré d'indépendance entre les classes qui sont destinées à être définies par des sources différentes ou qui ont des cycles de vie différents.

La relation de subsomption *OntoSub*("is-case-of") est l'opérateur de modularité utilisé par PLIB pour articuler une ontologie avec une autre ontologie. Ce mécanisme de modularité vise le développement d'ontologies très vastes (pour couvrir tout le domaine technique).

3.2.1.4 Définition d'un concept : connaissance descriptive

La figure 3.5 synthétise, par un schéma EXPRESS-G, les relations et attributs définis par le modèle PLIB pour les propriétés.

3.2.1.4.1 Relations entre classes et propriétés :

Le schéma dans la figure 3.5 montre que le modèle d'ontologies PLIB impose un couplage fort entre classes et propriétés :

- une propriété ne peut être définie sans indiquer son champ d'application par l'intermédiaire d'une classe (relation *name_scope*) ;
- une classe ne peut être définie sans indiquer les propriétés essentielles pour ses instances (relation *described_by*, voir la figure 3.4).

En pratique (1) le champ d'application d'une propriété est souvent composé de plusieurs classes qui peuvent se trouver dans plusieurs branches voisines d'une hiérarchie sans que la propriété soit applicable à toutes les classes de cette hiérarchie, et (2) ce n'est pas parce qu'une propriété est applicable, ontologiquement, à tous les composants d'une classe qu'elle doit nécessairement être effectivement utilisée pour décrire chaque composant dans toute base de données. Pour clarifier cette distinction, PLIB propose de distinguer trois types de relations entre classes et propriétés.

3.2.1.4.1.1 Propriétés visibles : dans une ontologie PLIB, une propriété doit être définie au plus haut niveau de la hiérarchie où l'on peut la définir sans ambiguïté. Elle est dite visible la classe où elle est ainsi définie, et, par héritage pour tout le sous-arbre correspondant. Ceci est formalisé par la relation *name_scope*.

3.2.1.4.1.2 Propriétés applicables : une propriété visible en un noeud peut y devenir applicable ; cela signifie que toute instance de cette classe doit présenter une caractéristique ou un aspect qui correspond à cette propriété. La propriété devient alors un

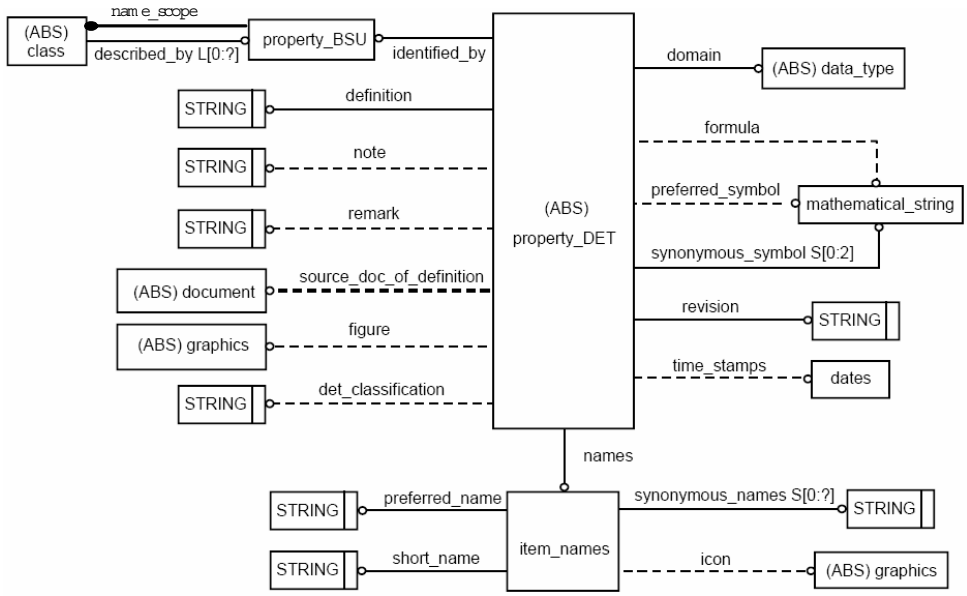

FIG. 3.5 – Schéma EXPRESS-G d'une propriété en PLIB

critère ontologique d'appartenance à la classe. Ceci est formalisé par la relation *describe_by* qui est également héritée.

3.2.1.4.1.3 Propriétés fournies : une propriété applicable à une classe peut être ou non utilisée dans la représentation d'une instance particulière de cette classe dans un univers formel particulier (base de données, échange informatisé,...). Ce dernier qualificatif montre comment PLIB aborde l'hypothèse du monde ouvert inhérente au domaine des ontologies. Dans une approche ontologique PLIB, les instances peuvent utiliser qu'un sous-ensemble clairement précisé des propriétés définies dans l'ontologie (à savoir les propriétés applicables à sa classe). Les données non renseignées sont considérées comme inconnues dans l'univers formel considéré.

3.2.1.4.2 Définition du codomaine d'une propriété :

Le co-domaine d'une propriété est défini par la relation *domain*. Ce peut être une classe ou un type de données. Le modèle d'ontologie PLIB fournit un système de type très complet. L'ensemble de ces types de données est présenté dans la figure 3.6. Parmi eux, nous pouvons citer :

1. les **types simples** comme :
   - *int_type, real_type, string_type, boolean_type* ;
   - *int_currency_type* et *real_currency_type* : somme entière et décimale en unité monétaire ;

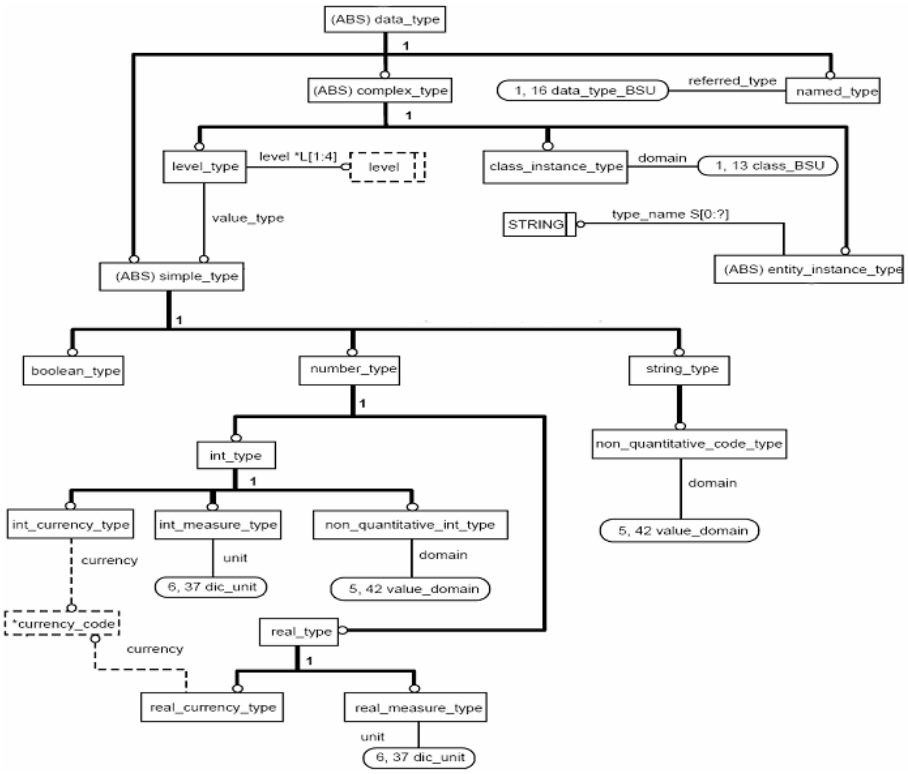

FIG. 3.6 – Schéma EXPRESS-G des types de données PLIB [ISO13584_IEC61360]

- *int_measure_type* et *real_measure_type* : grandeur physique exprimée dans une unité bien définie ;
- *non_quantitative_int_type* et *non_quantitative_code_type* : type énuméré, et valeur représentée par un entier ou par un code ;

2. les **types complexes** comme :
   - *level_type* : valeur numérique avec tolérances (min, max, nominal, typical) ;
   - *entity_instance_type* : type quelconque défini par un modèle EXPRESS ;
   - *class_instance_type* : type de données permettant de définir le co-domaine d'une propriété comme d'une classe.

3.2.1.5 Gestion du contexte

Permettre la définition et l'échange de catalogues de composants industriels a rapidement posé le problème de l'intégration de telles sources de données. Cette problématique a mis en évidence l'importance de la représentation du contexte de définition des concepts d'une ontologie. Pour la résoudre, Pierra [82] a identifié deux éléments de contexte qu'une ontologie doit représenter :

- le contexte d'évaluation d'une valeur, par exemple, le poids d'une personne dépend de la date à laquelle la mesure est faite ; une valeur peut aussi dépendre d'une unité ou d'une monnaie à expliciter.

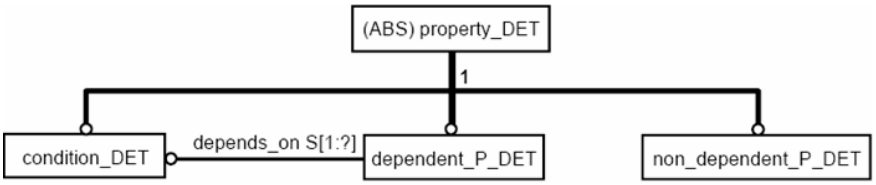

FIG. 3.7 – Trois catégories de propriétés PLIB

– le contexte de modélisation dans lequel chaque classe ou propriété est définie ; par exemple, le poids d'un objet ne peut être défini que dans le contexte d'un ensemble de classes bien défini. La même définition ne peut s'appliquer à un objet atomique (une vis) et à un objet composite (une voiture) pour lequel l'état de la composition doit être précisé (recevoir d'essence vide ou plein, avec ou sans GPS, radio, climatisation, etc.).

3.2.1.5.1 Contexte d'évaluation. Pour le contexte d'évaluation des valeurs, le modèle d'ontologies PLIB propose deux mécanismes. Le premier permet d'associer à toutes propriétés qui représentent une quantité mesurable, une unité définie par un schéma EXPRESS standard (ISO 10303-41 :2000). Le second propose de distinguer les propriétés en fonction de leurs dépendances vis-à-vis d'autres paramètres. Le modèle PLIB définit ainsi trois catégories de propriétés représentées en EXPRESS-G à la figure 3.7.

Propriétés non-dépendantes de contexte(non_dependent_P_DET), également appelées les propriétés caractéristiques. Ce sont les propriétés essentielles d'une classe qui possèdent *une valeur* pour chaque instance, par exemple le diamètre intérieur, ou sa vitesse de rotation maximale d'un roulement. Elles sont indépendantes des autres propriétés.

Propriétés dépendantes de contexte (dependent_P_DET). Ce sont les propriétés essentielles d'une classe dont la valeur pour chaque instance s'explique en réalité comme *une fonction* de paramètres caractérisant un contexte d'utilisation. Cette catégorie de propriétés permet de caractériser les comportements des composants dans un contexte particulier. Par exemple, la durée de vie d'un roulement à billes dépend directement et fondamentalement des forces axiales et radiales qui s'appliquent, ainsi que de sa vitesse de rotation.

Paramètres de contexte (condition_DET). Ce sont les propriétés qui représentent par des paramètres les contextes possibles d'utilisation ou d'évaluation d'une propriété. Cette catégorie de propriétés permet de caractériser le contexte dans lequel sera inséré un composant (par exemple, la charge réelle qu'un roulement va supporter dans un cas d'utilisation particulier) ou les conditions d'une mesure (par exemple la température à laquelle la longueur d'un matériel dilatable est mesurée). Ceci permet à la fois, dans une

démarche d'intégration, de distinguer des valeurs identiques, mais correspondant à différents contextes, et de représenter le savoir-faire des fournisseurs sur les composants qu'ils fournissent en donnant les grandeurs spécifiques caractérisant un problème particulier. Ainsi, les concepteurs, se basant sur ces grandeurs, peuvent résoudre d'une manière efficace les problèmes de conception et de choix de composants techniques. De façon beaucoup plus générique, cette notion permet de représenter au niveau ontologique, non seulement des propriétés qui s'expriment par des valeurs caractéristiques, mais également celles des fonctions caractéristiques.

La figure 3.8 présente quelques exemples de propriétés caractéristiques et dépendantes du contexte dans les domaines différents.

Concepts / **Propriétés**	Vin	Personne	Roulement à bille
Caractéristiques	couleur	date de naissance	diamètre intérieur
Dépendantes du contexte	qualité	couleur de cheveux	durée de vie
Paramètres de contexte	année de récolte	date	charge, vitesse d'utilisation

FIG. 3.8 – Exemples des différentes catégories de propriétés PLIB [49]

La deuxième colonne de ce tableau indique que la couleur est une propriété caractéristique de la classe Vin. La qualité (exécrable, moyen, très bon) en est une propriété dépendante du contexte. Elle dépend de la propriété *année de récolte*.

3.2.1.5.2 Contexte de modélisation. Le couplage fort imposé entre classes et propriétés (voir 3.2.1.4.1) contribue à la modélisation précise des propriétés, et des classes en imposant de définir :

- pour les propriétés, leurs domaines d'application, et
- pour les classes, les critères d'appartenance.

En complément, le modèle d'ontologies PLIB propose de distinguer les propriétés essentielles d'une classe, c'est-à-dire rigides [40] (celles dont la valeur ne peut être modifiée sans que l'objet ne soit modifié) de celles qui sont contingentes et dépendantes du point de vue sur le concept représenté. Cette distinction est mise en place dans le modèle PLIB via trois catégories de classes :

- les classes de définition (*item_class*) qui contiennent les propriétés essentielles d'une classe ;
- les classes de représentation (*functional_model_class*) qui contiennent les propriétés qui n'ont de sens que par rapport à un point de vue ;
- les classes de vue (*functional_view_class*) qui définissent la perspective dans laquelle les propriétés des classes de représentation sont définies.

La figure 3.9 présente ces modèles avec les relations sémantiques introduites dans la section 3.2.1.3 ainsi que deux nouvelles *is-view-of* et *created-view* induites par la modélisation du contexte présenté ci-dessus.

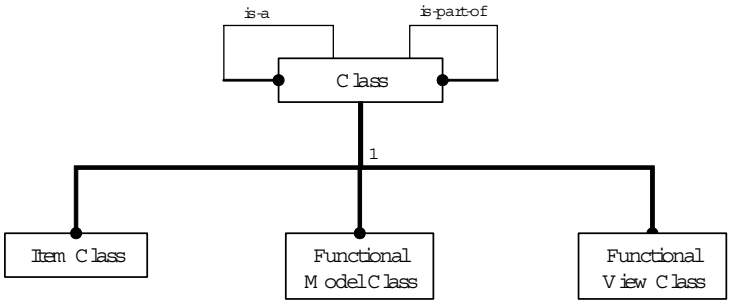

FIG. 3.9 – Les trois catégories de classes PLIB

Les propriétés définies dans une classe de représentation dépendent du point de vue non rigide, dite *representation_P_DET*, qui sont les propriétés non essentielles et dont la valeur peut changer sans que cela ne change l'objet décrit.

Exemple 5 *Nous étudions ici un exemple de mise en place de ces solutions. Considérons une ontologie portant sur des disques durs. La capacité est une propriété essentielle et rigide pour un disque dur : tout disque dur a une capacité. Par contre, le prix d'un disque dur n'est pas une propriété rigide. Elle n'a de sens que si le concept de disque est défini dans une perspective marchande. Et cette perspective mérite des précisions comme par exemple le vendeur ou la date de vente, etc. Sans l'ensemble de ces informations, la sémantique de la propriété prix n'est définie que par un contexte implicite de modélisation.*

En conséquence, si une personne souhaite avoir une idée du prix minimum auquel elle peut acheter un disque dur, sa recherche ne produira des résultats pertinents que si toutes les sources de données qui référencent notre ontologie associent à leurs disques durs un prix correspondant à leurs propres prix de vente.

Par contre, si un vendeur utilise cette ontologie pour indiquer un prix de revient, ou le prix de gros qu'il propose aux grandes surfaces, notre acheteur risque d'être fort surpris de la différence entre les prix pratiqués par son vendeur préféré et celui trouvé lors de sa recherche.

Cet exemple montre l'importance de définir explicitement le contexte d'une propriété non rigide même au sein de l'ontologie. D'ailleurs, dans le cas où les acteurs sont des agents automatiques, la compréhension implicite du contexte n'a pas de sens. En utilisant le modèle PLIB, notre concept de disque dur sera représenté par trois classes :

1. une classe de définition qui sera le domaine de la propriété *capacité* ;
2. une classe de représentation définie comme vue de celle de définition qui sera le domaine de la propriété *prix* ;
3. une classe de vue liée à celle de représentation qui sera le domaine, par exemple, des propriétés *date de vente*, *type d'acheteur* et *quantité minimale* ;

3.2.1.6 La méthodologie PLIB

La norme PLIB définit un ensemble de règles [49] qui vise à faciliter la conception d'ontologies par une communauté de personnes.

Règle PLIB 1. *Une classe ne doit être introduite dans la hiérarchie que si elle constitue le domaine d'une nouvelle propriété qui n'aurait pas de sens au dessus de cette classe.* Cette règle permet de modéliser un domaine en utilisant le nombre minimal de classes. C'est pour cela que PLIB est dit "orienté propriété".

Une autre règle PLIB précise également le critère à privilégier pour concevoir la hiérarchie :

Règle PLIB 2. *Le critère à privilégier pour concevoir une hiérarchie est celui permettant la factorisation maximale des propriétés.*

Encore faut-il que la propriété factorisée ait la même sémantique dans toute la hiérarchie où elle est définie. Une nouvelle règle précise cette notion :

Règle PLIB 3. *Deux propriétés définies dans deux classes différentes possèdent la même sémantique si (i) les objets définis par les deux classes sont, dans certaines circonstances particulières, interchangeables, et, dans ce cas, si un objet d'une classe est remplacé par un objet de l'autre classe, les valeurs de la propriété, pour les deux objets doivent être identiques ou si (ii) les propriétés jouent un rôle identique lorsqu'un traitement global est appliqué sur un ensemble d'instances pouvant appartenir à l'ensemble des deux classes.*

3.2.1.7 Représentation formelle d'une ontologie PLIB

Formellement (voir [82] pour un modèle plus complet), une ontologie PLIB peut être définie comme un 4-uplet : $O :< C, P, Sub, Applic >$, avec :

- C : l'ensemble des classes utilisées pour décrire les concepts d'un domaine donné (comme les services d'une agence de voyages, les pannes des équipements, les composants électroniques, etc.). Chaque classe est associée à un identifiant universel globalement unique (BSU) .
- P : l'ensemble des propriétés utilisées pour décrire les instances de l'ensemble des classes C. P définit toutes les propriétés susceptibles d'être présentes dans une base de données dont le modèle adhère à l'ontologie O. Chaque propriété est associée à un identifiant universel globalement unique (BSU). P est factorisé à trois sous-ensembles :
  1. P_{val} représentant l'ensemble des propriétés caractéristiques.
  2. P_{func} représentant l'ensemble des propriétés dépendantes de contexte.
  3. P_{cont} représentant l'ensemble des paramètres de contexte.

- $Sub : C \rightarrow 2^C$, est la relation de subsomption [10] qui, à chaque classe c_i de l'ontologie, associe ses classes subsumées directes. Sub définit un ordre partiel sur C.
  1. $OOSub$ ("is-a") : représente la relation de subsomption avec héritage ; elle est seulement utilisée entre deux classes d'une même ontologie et doit définir des hiérarchies simples.
  2. $OntoSub$ ("is-case-of") : représente la relation de subsomption sans héritage ; elle est en particulier utilisée entre deux classes de deux ontologies. $OntoSub$ permet alors d'articuler *a priori* une ontologie locale avec une ontologie partagée.
- $Applic : C \rightarrow 2^P$, associe à chaque classe de l'ontologie les propriétés qui sont applicables pour chaque instance de cette classe. Les propriétés qui sont applicables sont héritées à travers la relation $OOSub$ et elles peuvent être importées de façon explicite à travers la relation de $OntoSub$.

Exemple 6 *Pour illustrer la formalisation de O, étudions l'ontologie partagée O_p portant sur les disques durs dans la figure 3.3, elle est formulée comme suit :*

1. $C = \{Composant,\ DisqueDur\}$;
2. $P = \{situation, fournisseur, prix, capacité, vitesse, interface\}$;
3. $OOSub(Composant) = \{DisqueDur\}$;
4. $Applic(Composant) = \{situation, fournisseur, prix\}$;
   $Applic(DisqueDur) = Applic(Composant) \cup \{capacité, vitesse, interface\}$

Soulignons que les définitions ontologiques sont intentionnelles. Le fait qu'une propriété soit applicable pour une classe signifie qu'elle est rigide [40] c'est-à-dire essentielle pour chaque instance de la classe. Cela ne signifie pas qu'une valeur sera explicitement représentée pour chaque instance dans la base de données. Dans notre approche, le choix des propriétés effectivement représentées est fait au niveau du choix du schéma parmi les propriétés applicables. La section suivante discutera précisément de la représentation des instances.

3.2.2 Modèle PLIB : modèle des instances de classe

Le domaine d'application initial du modèle PLIB était celui des composants industriels, c'est-à-dire ceux qui sont définis à l'extérieur de l'entreprise utilisatrice d'où la nécessité d'échanger la signification de ces objets. Ces objets ont les particularités suivantes [32] :

- Ils sont décrits dans des catalogues,
- un catalogue regroupe les objets par classes, les propriétés de description de l'ensemble des objets d'une classe sont les mêmes et sont spécifiques de cette classe,

[10] 2^C désigne l'ensemble des parties de C

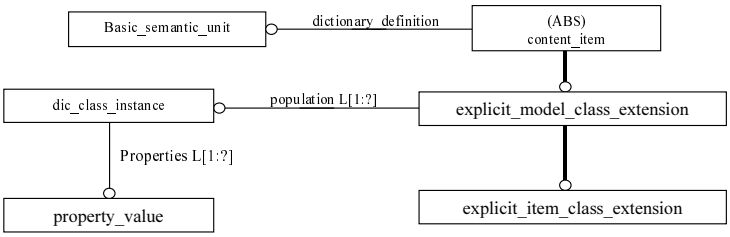

FIG. 3.10 – Représentation de population d'une ontologie PLIB selon la représentation explicite.

- les différents objets de chaque classe sont alors décrits par les valeurs de ces propriétés, ces valeurs étant définies soit explicitement (par une table) soit implicitement (par un ensemble de tables, des opérateurs de combinaison et/ou des formules de calcul),
- les différentes classes sont organisées selon une hiérarchie de classes, les propriétés applicables pouvant être factorisées à différents niveaux de cette hiérarchie.

Un tel ensemble d'objets constitue, selon la terminologie PLIB, l'extension du concept. Celle-ci est modélisée au travers d'une troisième catégorie d'entités (à côté de l'*identification* et la *dictionary_definition*) appelée *content_item*. Cette troisième catégorie d'entités permet de définir les instances d'un concept (par exemple les objets réels qui appartiennent à une classe et qui peuvent être physiquement commandés par un client). Le modèle d'extension défini par le modèle PLIB peut s'exprimer sous deux formes (publiées dans deux parties différentes de la norme PLIB) :

1. le modèle implicite (ISO 13584 :25) ;
2. le modèle explicite (ISO 13584 :24).

Le modèle implicite est fondé sur la définition algébrique d'un ensemble de tables élémentaires, de contraintes et de fonctions de dérivation, permettant de caractériser d'une manière implicite les populations des différentes classes. Il permet une représentation riche et compacte des instances d'une classe. En fait, elle permet non seulement la représentation des propriétés caractéristiques des composants, mais également des propriétés dépendantes du contexte dans des contextes d'insertion particuliers. De plus, elle permet de réduire la taille des tables de définition d'une famille qui peuvent être très volumineuses dans certains cas. Cette représentation très expressive est aussi assez complexe à mettre en oeuvre informatiquement. Elle est donc actuellement peu utilisée, et nous ne l'utiliserons pas dans le cadre de notre travail.

Le second modèle, dit représentation explicite (voir la figure 3.10 et 3.11), est le modèle que nous utiliserons. Il est basé sur la définition explicite de chaque instance, en énumérant les valeurs de ses propriétés. En effet, dans nombreux domaines, et notamment dans le domaine de catalogues de composants industriels, les propriétés dépendant du contexte sont définies pour un ensemble très limité de valeurs des paramètres de contexte. Dans de tels cas, une représentation explicite devient possible. Elle présente l'avantage de simplifier

Méta-modèle d'instances	Instances du méta-modèle

TYPE valeur_simple = SELECT (INTEGER, REAL, STRING,...) ;
END_TYPE ;
TYPE valeur_type = SELECT (Instance, valeur_simple,...) ;
END_TYPE ;
ENTITY Class_extension
Classe : String;
Population : LIST[0:?] OF Instance ;
...
END_ENTITY ;
ENTITY Dic_classe_instance
Classe : String;
Proprietes : LIST[0:?] of property_value ;
...
END_ ENTITY ;
ENTITY Property_value;
sa_propriete : String;
valeur : valeur_type ;
...
END_ ENTITY;
	// Les instances des vis
	#1 = Class_extension('VIS', (#2,#3)) ;
	#2 =Dic_class_instance('VIS',(#20,...)) ;
	#20= Property_value('longueur',5) ;
	#3 = Dic_class_instance ('VIS',(#21,...)) ;
	#21= Property_value ('longueur',6) ;
	
	// Les instances des écrous
	#4 = Class_extension ('ECROU', (#5,#6)) ;
	#5 = Dic_class_instance ('ECROU',(#10,...)) ;
	#10= Property_value ('diametre',10) ;
	#6 = Dic_class_instance ('ECROU',(#11,...)) ;
	#11= Property_value ('diametre',17) ;

FIG. 3.11 – Méta-modèle des instances

le modèle de données des instances et de faciliter l'intégration de données de composants décrites dans plusieurs catalogues. De plus, cela facilite la gestion des données de catalogues de composants dans des bases de données relationnelles. La partie 24 de la norme modélise ces instances sous forme de méta-modèle, ce qui permet de représenter toute classe selon la même structure (voir la figure 3.11). L'approche est identique donc à celle du méta-modèle d'ontologie décrit dans la partie 42 de la norme.

La figure 3.11 donne une vue très simplifiée (les identifications sont, en particulier, remplacées par des noms) d'un méta-modèle d'instances et des instances de ce (méta-) modèle qui donne une vue succincte de l'extension d'un concept en PLIB. La deuxième colonne du tableau instancie le méta-modèle d'instances avec l'exemple ci-dessous :

- VIS(5,...) : Une vis de longueur 5 et ...
- VIS(6,...) : Une vis de longueur 6 et ...
- ECROU(10,...) : Un écrou de diamètre 10 et ...
- ECROU(17,...) : Un écrou de diamètre 17 et ...

On remarque que de façon analogue à une instance OEM dans le projet TSIMMIS [24], mais à la différence des individus dans les systèmes de classification ou dans les logiques terminologiques (OWL, par exemple), toute instance conforme à une ontologie PLIB possède une *classe de base*. Cette classe est la classe la plus spécialisée à laquelle l'instance appartient et le modèle PLIB en impose l'unicité. Les instances peuvent donc être représentées de façon générique sous forme d'un fichier dans lequel chaque instance PLIB (voir la figure 3.12) consiste en :

- un identifiant d'objet ;

- une référence à la classe de base de l'objet (par l'intermédiaire de l'identifiant universel de la classe) ;
- éventuellement des références avec autres classes auxquelles l'instance appartient suite à des relations *OntoSub* (*is-case-of*).
- une liste de couples :
  1. Référence à une propriété applicable à la classe (par son identifiant universel).
  2. Valeur de la propriété (type simple, identifiant d'objet ou collection).

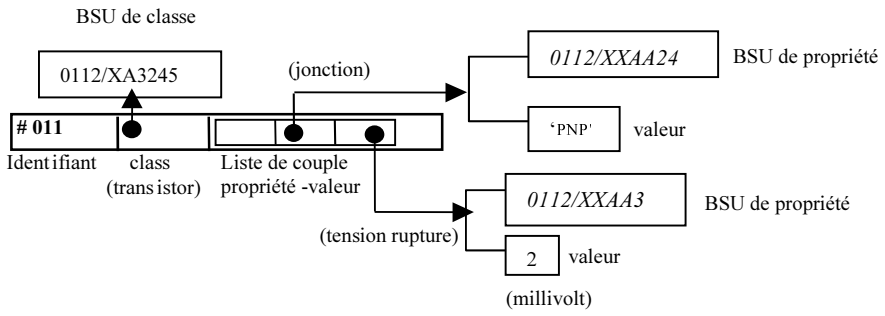

FIG. 3.12 – Représentation d'une instance décrite en termes d'une ontologie PLIB

PLib définit donc un modèle à deux niveaux :

1. Au niveau des concepts manipulés : le modèle d'ontologie permet de décrire ces concepts de façon formelle, de les échanger sous forme informatique, de les référencer par des identifiants universels. Ce modèle permet également de spécialiser et d'étendre une ontologie normalisée par des concepts spécifiques tout en gardant explicitement le lien entre un concept spécifique et le concept normalisé qu'il spécialise.
2. Au niveau du contenu : les instances des concepts sont décrits par leur classe de base et les valeurs de leurs propriétés. Notons que PLIB ne définit pas d'identifiants d'instances, ceci est éventuellement défini dans chaque classe.

3.2.3 Bilan sur le modèle d'ontologie PLIB

Nous avons décrit dans cette section le modèle PLIB et ses spécificités. Ce modèle permet la description de sources d'information, de classes, de propriétés, et de domaines de valeurs. Une classe est une collection d'objets définie en intention. Une propriété est une relation binaire entre deux classes ou entre une classe et un domaine de valeurs. Un domaine de valeurs est un ensemble mathématique défini en extension ou en intention. PLIB offre également une identification non ambiguë pour n'importe quel concept défini dans une ontologie PLIB. C'est le mécanisme BSU. Cela nous permet d'identifier les différents concepts présents dans un domaine. Toutes les références entre concepts sont effectuées en

direction d'un BSU, cela permet également d'extraire un sous-ensemble quelconque d'une ontologie PLIB tout en assurant l'intégrité référentielle du sous-ensemble extrait. Enfin la définition intentionnelle des classes est séparée de la définition en extension comme présenté dans le tableau 3.1.

Concepts	**Identification**	**Définition/Intention (méta-données)**	**Extension (données)**
Super-type abstrait	basic_semantic_unit	dictionary_element	content_item
Source d'information	supplier_bsu	supplier_element	-
Classe	class_bsu	class	class_extension
Propriété	property_bsu	property_det	-

TAB. 3.1 – Le triptyque du modèle PLIB

Visant à permettre l'échange de données, le modèle d'ontologies PLIB n'offre pour l'instant aucun opérateur pour exprimer des équivalences conceptuelles. Les ontologies PLIB sont toujours canoniques. C'est-à-dire qu'elles ne contiennent que des concepts *primitifs* [70] qui ne peuvent pas se définir en terme d'autres concepts de la même ontologie. Ce sont donc des ontologies orientées caractérisation (elles permettent de décrire n'importe quel objet du domaine) et non orientées déduction (elles ne permettent pas d'inférer des équivalences conceptuelles). Le modèle PLIB est très riche pour décrire avec précision les concepts primitifs. Il offre des mécanismes originaux de modularité d'ontologies qui permettent le développement d'ontologies très vastes (pour couvrir tout le domaine technique).

Le modèle d'ontologie PLIB est lui-même défini sous forme d'un schéma décrit dans le langage de spécification de données EXPRESS. Nous allons décrire dans la section 3.4 les aspects principaux du *formalisme de spécification* EXPRESS. L'avantage essentiel de ce langage pour notre propos est qu'il possède un langage d'expression de contraintes très puissant permettant d'assurer l'intégrité des ontologies et des instances échangées.

La section qui suit décrit le concept de Base de Données à Base Ontologique que nous utilisons pour les sources de données dans notre contexte d'intégration de données.

3.3 Base de données à base ontologique

Le développement du modèle PLIB s'est achevé en 2001, un nouveau projet a été lancé depuis 2002, appelé OntoDB, dont l'objectif était de permettre la gestion, l'échange, l'intégration et l'interrogation de données structurées de grande taille associées à des ontologies conceptuelles. C'est dans ce contexte qu'a été développé le concept de Base de Données à Base Ontologique(BDBO).

3.3.1 Introduction

Les travaux menés dans le domaine des bases de données (BD) dans la dernière décennie ont surtout visé à augmenter les capacités de représentation de l'information. Ont ainsi été proposées, après le modèle relationnel, les BDs déductives, actives, objets et relationnel-objet. Ces travaux n'ont, par contre, guère contribué à réduire l'hétérogénéité entre BDs portant sur un même domaine. Ceci rend l'intégration de sources de données hétérogènes toujours aussi difficile.

Parallèlement, les travaux menés en modélisation des connaissances faisaient émerger la notion d'ontologie comme un modèle conceptuel consensuel et partageable d'un domaine. Contrairement au modèle conceptuel qui *prescrit* les informations qui doivent être représentées dans une base de données pour répondre à un cahier des charges applicatif, une ontologie vise à décrire de façon consensuelle l'ensemble des informations permettant de conceptualiser des domaines d'application assez larges. Divers langages de définitions d'ontologies ont été développés. Ils permettent la représentation et l'échange informatique à la fois d'ontologies, et d'objets définis en termes de ces ontologies. De plus, un nombre croissant d'ontologies a pu être développé et faire l'objet de consensus dans des communautés plus ou moins larges.

Dans les domaines où il existe une ontologie, il est possible d'exprimer les différents modèles conceptuels correspondants à différents cahiers des charges applicatifs en termes de sous-ensembles ou de spécialisation de cette ontologie. La représentation explicite de cette ontologie au sein de chaque BD permet alors (1) leur intégration plus facile, voire automatique, (2) une génération automatique d'interfaces d'accès au niveau de connaissance, c'est à dire au niveau ontologique, pour les données contenues dans chacune des bases. C'est le concept de Bases de Données à Base Ontologique ($BDBO$) que nous présentons dans cette section, et qui possède deux caractéristiques : (1) ontologie et données sont toutes deux représentées dans la BD et peuvent faire l'objet des mêmes traitements (insertion, mise à jour, requêtes, etc.) ; (2) toute donnée est associée à un élément ontologique qui en définit le sens.

Notons que dans le reste de ce travail toutes les sources considérées contiendront leur propre ontologie et obéiront à la définition ci-dessus. Ce seront donc des $BDBOs$.

Nous présentons d'abord ci-dessous le modèle d'architecture, appelé $OntoDB$, qui est proposé pour une $BDBO$, et ensuite nous proposons une description formelle pour une $BDBO$ dont l'ontologie obéit au modèle défini en 3.2.1.7.

3.3.2 Architecture OntoDB pour base de données à base ontologique

Afin de représenter explicitement leurs ontologies, les BDBOs doivent avoir une structure différente de celle des bases de données usuelles. L'architecture que suggère le modèle PLIB est alors un modèle en quatre parties, appelé $OntoDB$ qui permet de traiter, de

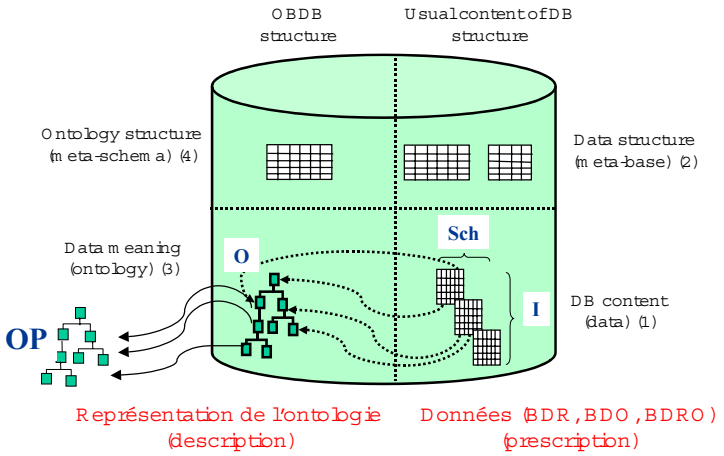

FIG. 3.13 – Architecture de BDBO : Modèle OntoDB

façon générique, aussi bien les données que les ontologies, mais aussi les liens entre ontologies et données. Comme les bases de données traditionnelles les BDBO conformes à *OntoDB* possèdent deux parties (partie droite de la Figure 3.13) : une partie contenu et une partie méta-données qui décrit les tables, les colonnes, les clés étrangères, etc. Mais OntoDB possède, en plus, deux autres parties (partie gauche de la Figure 3.13). Celles-ci représentent l'ontologie et la structure de l'ontologie qui permet tous les traitements génériques sur les ontologies.

Le modèle PLIB définit séparément la présentation de l'ontologie (Onto) et celle des données (DB). Nous présentons successivement ci-dessous les différentes parties du modèle OntoDB.

3.3.2.1 Représentation des ontologies : la partie *ontology*

Les ontologies supportées par le modèle d'architecture *OntoDB* sont celles susceptibles d'être représentées sous forme d'un modèle au sens de Bernstein [19], c'est à dire d'un ensemble d'objets accessibles à partir d'un objet racine par des relations de connexité particulière, dite de composition. Cette définition correspond à la plupart des modèles d'ontologies récents tels que, OWL, PLIB. Une telle ontologie est donc représentée comme instance d'un schéma objet (ex. appelé méta-modèle) dans un formalisme de modélisation particulier (XML-Schema pour OIL et OWL, EXPRESS pour PLIB). Cette représentation, à son tour, fournit un format d'échange pour les ontologies visées (document XML pour OWL, fichier physique d'instances EXPRESS pour PLIB), et, par transformation de modèles, un schéma logique de représentation au sein du SGBD cible.

Identification des besoins. Afin de préciser la structure de ce schéma cible, nous nous appuyons sur l'analyse des besoins présentée dans [84] concernant une BDBO. D'après Pierra et al. [84], les fonctions souhaitées pour une base de données à base ontologique

incluent, en particulier, les suivantes :

- **F1** : capacité de stockage interne des ontologies au sein d'un schéma logique adapté au SGBD cible (par exemple, l'ontologie des composants électroniques) ;
- **F2** : capacité de lire les ontologies représentées dans leur format d'échange et de les stocker dans la BDBO, et, inversement, d'exporter des ontologies ;
- **F3** : interface générique (i.e. indépendante du schéma d'ontologie) d'accès par programme aux entités définissant une ontologie et ce, indépendamment du schéma logique de la base de données (une telle interface générique, c'est à dire pouvant être utilisée quel que soit le schéma d'ontologie, est usuellement appelée "API à liaison différée").
- **F4** : interface d'accès par programme aux entités de l'ontologie par des fonctions spécifiques à la fois du modèle d'ontologie et du langage de programmation (une telle interface, qui permet donc de contrôler la correction syntaxique des appels, est usuellement appelée "API à liaison préalable").
  Afin d'implémenter une ontologie définie, comme nous l'avons vue, comme instance d'un modèle objet au sein d'une base de données, deux solutions peuvent être envisagées et sont comparées dans [84] :

  1. la première approche consiste à appliquer des règles de transformations particulières (manuellement ou automatiquement) au modèle objet afin de créer le modèle logique du SGBD cible permettant de stocker les instances du modèle objet. Les applications sont alors écrites en fonction de ce modèle résultat.
  2. la deuxième approche consiste à définir des règles de transformation automatique et à représenter en plus le modèle objet lui-même (classes, propriétés, relations, héritages, etc.) dans la BD, sous forme d'instances d'un méta-modèle objet (méta-schéma).

La figure 3.14 présente une comparaison de ces deux solutions lorsque le modèle d'ontologie est susceptible d'évoluer. Cette comparaison montre alors le caractère indispensable, si l'on souhaite que le système de gestion de la BDBO puisse s'adapter automatiquement à des évolutions de modèle d'ontologie, de représenter explicitement son méta-modèle. Cette représentation constitue alors la deuxième partie du modèle OntoDB appelée méta-schéma.

3.3.2.2 Représentation du méta-modèle d'ontologie : la partie *meta-schema*

Comme nous l'avons vu dans la section précédente, le principal objectif de la partie méta-schéma est d'offrir une interface de programmation permettant l'accès au modèle d'ontologie courant par parcours des instances représentées dans le méta schéma. Ceci permet de rendre générique, par rapport aux modèles d'ontologies, un certain nombre de fonctions devant être réalisées par le système. Lorsque le SGBD utilisé : (1) est un SGBD

Fonction	Sans représentation explicite du méta-modèle	Avec représentation explicite du méta-modèle
F1 - Définition du modèle logique de représentation des ontologies.	Définition manuelle du schéma (ou génération par programme externe)	- Définition de règles d'implantation (R1) pour toute instance du méta-modèle (exemple: tout schéma XML pour OWL, ou tout modèle EXPRESS pour PLIB). Programme de génération qui interprète les instances du méta-modèle correspondant au modèle courant et met en œuvre (R1).
F2 - Lecture / écriture de fichier	Ecriture d'un programme spécifique	- Des règles génériques (R2) de représentation externe existent déjà pour les instances du méta-modèle (par exemple: document XML pour OWL, fichier d'instances EXPRESS pour PLIB) - Programme générique qui interprète les instances du méta-modèle correspondant au modèle courant en exploitant (R1) et (R2)
F3 - API à liaison différée	Tout le modèle doit être codé dans l'API	L'API devient un interprète des instances du méta-modèle en exploitant (R1)
F4 - API à liaison préalable	Toute l'API doit être codée à la main	- Définition de règles génériques (R3) de représentation d'un modèle EXPRESS ou d'un schéma XML dans le langage cible (par exemple java). - Programme de génération interprétant le méta-modèle en exploitant (R1) et (R3)

FIG. 3.14 – Implémentation des fonctions nécessaires, avec et sans métaschéma [84]

Objet, et (2) permet de représenter tous les mécanismes existant dans le formalisme de définition du modèle d'ontologie, alors cette partie préexiste en général dans le SGBD et s'identifie avec la méta-base du système. Dans tous les autres cas, il s'agit d'une partie nouvelle, nécessaire dans une BDBO.

Identification des besoins. Comme la partie ontologie, la partie méta-schéma doit offrir les fonctions de :

- **F1** : génération du schéma logique de gestion des modèles d'ontologie (par exemple, le modèle PLIB) ;
- **F2** : lecture / écriture de fichiers d'instances représentant des modèles d'ontologie.
- **F3** : parcours des modèles d'ontologie par une API à liaison différée ;
- **F4** : parcours des modèles d'ontologie par une API à liaison préalable.

Les exigences existant jusqu'alors sur le méta-schéma, étaient d'être capable de représenter l'ensemble des variantes des modèles d'ontologie envisagés . Si on impose alors au méta-schéma d'être, de plus, réflexif, c'est-à-dire de pouvoir se représenter lui-même, on pourra représenter le méta-schéma comme instance de sa propre structure. Ceci permettra à nouveau d'écrire, de façon générique, l'ensemble des quatre fonctions envisagées.

Notons de plus que si le formalisme utilisé pour définir le modèle d'ontologie et le modèle du méta-schéma est le même (par exemple : UML, ou alors EXPRESS) alors le même code générique pourra être réutilisé pour les fonctions F1, F2, F3 et F4, à la fois pour la partie ontologie, et pour la partie méta-schéma [84].

Notons enfin que la représentation explicite du méta-schéma n'empêche pas d'optimiser les programmes générés afin qu'ils y accèdent le moins possible. C'est d'ailleurs ce qui est fait, dans l'implantation effectuée au sein du LISI, concernant l'API à liaison préalable permettant l'accès, en Java, aux ontologies PLIB.

3.3.2.3 Représentation des instances : la partie *data*

Une ontologie vise à représenter la sémantique des objets d'un domaine en les associant à des classes, et en les décrivant par des valeurs de propriétés. Selon les modèles d'ontologies utilisés, plus ou moins de contraintes existent sur ces descriptions. Ainsi si on n'introduit pas de restrictions particulières sur les descriptions OWL, un objet peut appartenir à un nombre quelconque de classes, par exemple parce que plusieurs points de vue ont été pris en compte dans la conception de l'ontologie [79], et être décrit par n'importe quelles propriétés. Ceci donne à chaque objet du domaine une structure qui peut lui être spécifique. A contrario, un schéma de base de données vise à décrire des ensembles d'objets "similaires" par une structure logique identique de façon à pouvoir optimiser les recherches sur tel ou tel ensemble par des techniques d'indexation.

En l'absence de toute hypothèse particulière sur la représentation des objets du domaine, la seule structure commune possible consiste à pouvoir associer chaque objet :

- à un sous-ensemble quelconque de l'ensemble des classes ;
- à un nombre quelconque de propriétés

Cette structure entraînerait soit une perte de place très considérable (avec représentation systématique des appartenances ou propriétés non pertinentes pour une instance), soit des temps de traitements importants résultant de l'absence d'indexation ou de besoins de jointures nombreuses (si seules les appartenances et les propriétés pertinentes sont représentées).

Dans le cadre du modèle d'architecture OntoDB, il est proposé d'imposer deux restrictions désignées sous le terme d'*hypothèse de typage fort*. Notons que ces hypothèses semblent être respectées par un grand nombre d'application, dont celui des catalogues de composants industriels et sont formulées par d'autres applications et en particulier l'éditeur d'ontologie Protégé [11].

- **R1** : tout objet du domaine est décrit par son appartenance à une et une seule classe, dite de base qui est la borne inférieure unique de l'ensemble des classes auquel il appartient (il appartient bien sûr également à ses super-classes).
- **R2** : tout objet du domaine ne peut être décrit que par des propriétés applicables à sa classe de base (ceci signifie, si "c" est la classe de base, en PLIB : "appartenant à Applic(c)", et, en OWL "propriété associée à un domaine tel que ce domaine subsume c").

Il est alors possible de définir un schéma logique de représentation des instances, significatif pour la classe, qui est constitué de toutes les propriétés applicables et effectivement utilisées (soit, selon le choix de l'administrateur de la BD, par toutes les instances représentées, soit, par au moins une instance ayant cette classe comme classe de base).

Notons, et c'est une grande différence avec les BDOOs, qu'il est parfaitement possible qu'une propriété définie comme applicable au niveau d'une classe A soit effectivement représentée dans le schéma de deux sous classes (A1 et A2) de A, et non dans celui d'une troisième sous-classe (A3). Dans une BDBO, la relation de subsomption est intentionnelle et ne peut donc pas être simplement représentée par héritage au niveau du modèle logique de la partie données. Ainsi donc, malgré le caractère objet des données à représenter, et malgré la hiérarchie de classes existante au niveau ontologique, une représentation verticale au sein d'un modèle relationnel (représentation dans laquelle chaque propriété est représentée dans la table correspondant au niveau où la propriété est définie) s'avère peu adaptée.

La définition des classes à représenter en tant que classes de base (donc associées à des instances), et des propriétés devant être représentées pour chaque classe de base est effectuée soit par l'administrateur, soit automatiquement (selon le scénario d'utilisation défini pour la BDBO) lors de la lecture des fichiers d'instances. Ceci définit la relation devant être associée à chaque classe de base. De plus :

- la clé de cette relation (par défaut un identifiant d'objet généré par le système) peut être définie par l'administrateur ;
- le schéma de cette relation (par défaut une simple table) peut être également défini

[11] `http://protege.stanford.edu/`

par l'administrateur.

3.3.2.4 La partie *meta-base*

L'ensemble des schémas logiques des trois autres parties, et en particulier les schémas de représentation des instances de la partie *data*, sont alors représentées dans la partie méta-base (ou catalogue) qui existe dans tout SGBD. C'est la quatrième partie de l'architecture *OntoDB*.

Correspondances entre schéma et ontologie. Schémas et Ontologies étant gérés séparément (partie ontologie et méta-base), il est nécessaire d'établir un mécanisme de liaison bilatérale entre ces deux parties. Ce mécanisme de liaison est constitué de deux fonctions partielles :

- *Nomination* : classe ∪ propriété → relation ∪ attribut
- *Abstraction* : relation ∪ attribut → classe ∪ propriété

Ces deux fonctions sont partielles car :

- certaines classes et/ou propriétés peuvent ne pas être représentées ;
- certaines tables et/ou attributs, de nom prédéfinis, correspondent à des informations de type système.

Ces fonctions sont représentées en utilisant les BSUs de classe et de propriétés ontologiques comme identifiants des relations et attributs de représentation des instances.

3.3.3 Représentation formelle d'une BDBO

Une base de données à base ontologique (BDBO) s'appuyant sur une ontologie conforme au modèle proposé à la section 3.2.1.7 est définie formellement comme un 4-uplet : $BDBO_i :< O_i, I_i, Pop_i, Sch_i >$, avec :

- O_i représente son ontologie ($O_i :< C_i, P_i, Sub_i, Applic_i >$)
- I_i représente l'ensemble des instances de données de la base de données. La sémantique de ces instances est décrite par O en les caractérisant par des classes et des valeurs de propriétés définies dans l'ontologie partagée.
- $Pop_i : C_i \rightarrow 2^{I_i}$, associe à chaque classe les instances qui lui appartiennent (directement ou par l'intermédiaire des classes qu'elle subsume). Pop(c_i) constitue donc la population de c_i.
- $Sch_i : C_i \rightarrow 2^{P_i}$, associe à chaque classe c_i les propriétés applicables pour cette classe et qui sont effectivement utilisées pour décrire tout ou partie des instances de $Pop_i(c_i)$. Pour toute classe c_i, Sch(c_i) doit satisfaire : $Sch_i(c_i) \subseteq Applic_i(c_i)$.

Exemple 7 *Pour illustrer cette formalisation de $BDBO_1$, prenons l'exemple dans la figure 3.15. L'ontologie locale de la source 1 (S_1) : O_1, est définie en référençant l'ontologie partagée : O_p.*

La classe S1|C1 (appelée en français DisqueDur) de l'ontologie O_1 référence la classe S|C2 (appelée en français DisqueDur) de l'ontologie O_p, en important les propriétés suivantes : $\{S|C1|P2_{(fournisseur)}, S|C2|P1_{(capacite)}, S|C2|P2_{(vitesse)}, S|C2|P3_{(interface)}\}$. Localement, cette classe définit deux nouvelles propriétés : $\{S1|C1|P1_{(serie)}, S1|C1|P2_{(situation)}\}$.

En conséquence, les éléments de la source $S_1 =< O_1, I_1, Sch_1, Pop_1 >$ sont définis comme suit :

- *O_1 :*
  1. *C_1=$\{S1|C1_{(DisqueDur)}\}$;*
  2. *P_1=$\{S|C1|P2_{(fournisseur)}, S|C2|P1_{(capacite)}, S|C2|P2_{(vitesse)}, S|C2|P3_{(interface)}, S1|C1|P1_{(serie)}, S1|C1|P2_{(situation)}\}$.*
  3. *$Sub_1(S1|C1)=\phi$.*
  4. *$Applic_1(S1|C1)=\{S|C1|P2, S|C2|P1, S|C2|P2, S|C2|P3, S1|C1|P1, S1|C1|P2\}$.*
- *$Pop_1(S1|C1) = I_1$*
- *$Sch_1(S1|C1)$ =$\{S1|C1|P1, S|C2|P1, S|C2|P3, S|C2|P2, S1|C1|P2\}$*

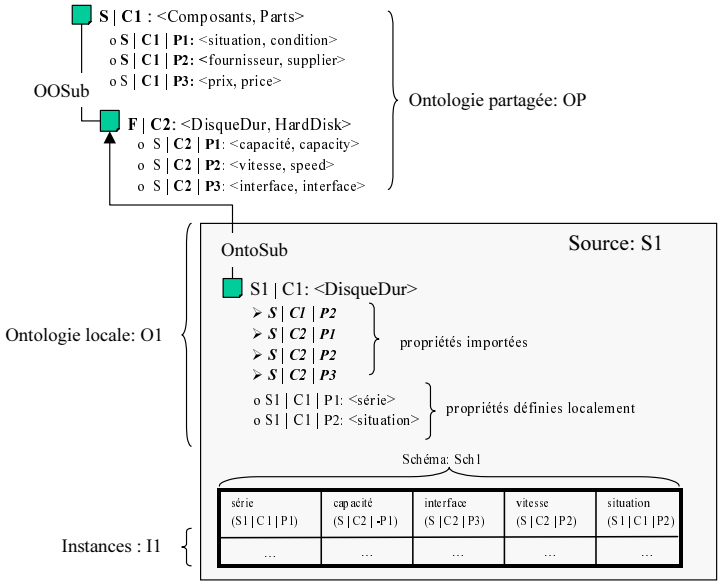

FIG. 3.15 – Exemple d'une BDBO définie à travers le modèle PLIB

3.3.4 Bilan sur les bases de données à base ontologique

Dans cette section, nous avons présenté un nouveau modèle de base de données qui vise à représenter, en plus des données, l'ontologie locale qui en représente le sens, le modèle conceptuel qui en définit la structure, et, éventuellement les correspondances existant entre l'ontologie locale et des ontologies partagées. Les bases de données qui possèdent ces caractéristiques sont appelées bases de données à base ontologique (BDBO) [84]. Nous

avons également présenté un modèle d'architecture complet pour les BDBOs. Ce modèle, appelé OntoDB, est constitué de quatre parties. Il contient d'abord les deux parties classiques des bases de données, données et méta-base, qui stockent, d'une part, les données d'instances, et, d'autre part, le schéma. A ceci s'ajoutent une partie qui représente l'ontologie, et une autre partie, appelée méta-schéma, qui représente sous forme d'instances, au sein d'un modèle réflexif, le modèle de données de l'ontologie.

Au sein du LISI, la notion *BDBO* et le modèle *OntoDB* ont été protypés, validés simultanément dans deux environnements différents : un environnement de base de données orientée objet ECCO (EXPRESS Compiler COmpiler) et un environnement relationnel-objet sous le SGBD POSTGRES-SQL. En ce qui concerne mes implémentations, l'environnement ECOO a été utilisé.

3.4 Langage EXPRESS

Le langage EXPRESS a été défini dans le cadre d'un projet de normalisation de l'ISO intitulé STEP (STandard for Exchange Product model data) initié à la fin des années 1980. STEP avait pour objectif la normalisation de modèles de données pour les différentes catégories de produits matériels ou immatériels. Aucun langage existant à l'époque ne permettait de représenter formellement des modèles d'une telle complexité, le projet STEP a donc commencé par développer à la fois un langage de modélisation et toute une technologie associée. Ceci a abouti à la production d'une série de normes définissant un environnement appelé EXPRESS. Cet environnement comprend :

- un langage de modélisation de l'information : EXPRESS (ISO 10303-11 :1994) ;
- un format d'instances qui permet l'échange de données entre systèmes (ISO 10303-21 :1994) ;
- une infrastructure de méta-modélisation associée à une interface d'accès normalisée, appelée SDAI, pour accéder et manipuler simultanément les données et le modèle de n'importe quel modèle EXPRESS. Cette interface associée à un méta-modèle d'EXPRESS en EXPRESS a d'abord été définie indépendamment de tout langage de programmation (ISO 10303-22 : 1998) puis des implémentations spécifiques ont été spécifiées pour le langage C++ (2000), Java (2000), C (2001) ;
- un langage déclaratif de transformation de modèles (ISO 10303-14 :2002).

Enfin, le langage EXPRESS possède un langage procédural complet (analogue à PASCAL) pour l'expression de contraintes. Au prix de quelques extensions mineures, ce langage peut également être utilisé comme langage impératif de transformation de modèles. Depuis une dizaine d'années, de nombreux environnements de modélisation EXPRESS sont commercialisés et le langage EXPRESS est utilisé dans de nombreux domaines, que se soit pour modéliser et échanger des descriptions de produits industriels, ou pour spécifier des bases de données dans des domaines divers, ou même pour fabriquer des générateurs de codes dans des ateliers de génie logiciel. Nous présentons succinctement dans la section qui

suit les concepts généraux du langage EXPRESS et son environnement de modélisation.

3.4.1 Connaissance structurelle

Ce type de connaissance est représenté, dans les modèles conceptuels, sous forme de concepts et de relations entre concepts. Les concepts sont souvent appelés *types d'entités* (ou, lorsque le contexte évite toute ambiguïté avec la notion d'instance, *entités*) et les relations entre concepts sont appelées des associations [25].

Le langage EXPRESS utilise la notion d'entité (ENTITY), pour réaliser l'abstraction et la catégorisation des objets du domaine de discours. Une entité est similaire à une classe des langages à objets à la différence qu'elle ne définit pas de méthodes. Les entités sont hiérarchisées par des relations d'héritage qui peuvent relever de l'héritage simple, multiple ou répété.

Dans le langage EXPRESS, une entité déclare ses super-types et peut déclarer aussi ses sous-types en utilisant trois opérateurs (ONEOF, AND, ANDOR) qui imposent ou permettent à une entité d'être instance d'une ou de plusieurs de ses sous-types. Illustrons ces opérateurs sur des exemples :

- Person SUPERTYPE OF ONEOF(male, female) : une instance de l'entité *person* peut être soit une instance de l'entité *male* soit une instance de l'entité *female*
- Person SUPERTYPE OF (employee ANDOR student) : une instance de *person* peut être une instance de l'entité *employee* mais aussi simultanément une instance de l'entité *student*.
- Person SUPERTYPE OF (ONEOF(female,male) AND ONEOF(citizen, alien)) : une instance de l'entité *person* peut être une instance de l'entité *male* ou de *female* et une instance de l'entité *citizen* ou de l'entité *alien*.

Les associations sont représentées sous forme descriptive, par des attributs.

3.4.2 Connaissance descriptive

La connaissance descriptive associe aux catégories des propriétés qui permettent de discriminer les différentes instances des catégories. Ce type de connaissance est représenté dans les modèles conceptuels sous forme d'attributs. Chaque attribut doit être défini dans le contexte d'une entité.

En EXPRESS, une entité est décrite par des attributs dont les co-domaines sont soit des ensembles de valeurs, soit des entités. Les attributs permettent de caractériser les instances d'une entité par des valeurs. Le langage EXPRESS distingue deux catégories d'attributs :

1. les attributs libres : ils ne peuvent pas se calculer à partir d'autres attributs ; leur valeur peut être déclarée optionnelle.
2. les **attributs dérivés ou calculés** : dépendent fonctionnellement d'autres attributs et la dépendance s'exprime par une fonction algébrique.

Chaque attribut est typé et ses valeurs appartiennent au type de données qui définit son co-domaine. Le langage EXPRESS définit quatre familles de types :

- les types simples : ce sont essentiellement les types *chaînes de caractères* (STRING), *numériques* (REAL, BINARY, INTEGER) ou *logiques* (LOGICAL, BOOLEAN) ;
- les types nommés : ce sont des types construits à partir de types existant auxquels un nom est associé. Un type nommé peut être défini par restriction du domaine d'un type existant. Cette restriction peut être faite par la définition d'un prédicat qui doit être respecté par les valeurs du sous domaine créé. Il peut également être défini par énumération (ENUMERATION) ou par l'union de types (SELECT) qui, dans un contexte particulier, sont alternatifs.
- les types agrégats : ce sont des types qui permettent de modéliser les domaines dont les valeurs sont des collections. Les types de collections disponibles sont les ensembles (SET), les ensembles multi valués (BAG), les listes (LIST) et les tableaux (ARRAY). Si un type collection n'a pas a être nommé : il apparaît directement dans la définition de l'attribut qu'il type.
- les types entités : un attribut d'un tel type représente une association.

3.4.3 Connaissance procédurale

La connaissance procédurale correspond aux règles de raisonnement qui peuvent être appliquées aux différentes instances de chaque catégorie. Ce type de connaissance est celui que l'on représente dans les règles, les fonctions et les procédures. Les formalismes de modélisation possèdent souvent, de ce point de vue, un pouvoir d'expression très limité. Par exemple, ils ne permettent pas d'exprimer une procédure de calcul des moyennes des notes des étudiants, ou encore d'apposer des contraintes du type intervalle sur les valeurs licites pour un attribut (par exemple $note \in [0, 20]$). Le langage EXPRESS permet de modéliser deux catégories de connaissances procédurales : le fonction de dérivation et la contrainte d'intégrité.

3.4.3.1 Dérivation d'attribut

Comme nous l'avons présenté à la section 3.4.2, EXPRESS supporte deux catégories d'attributs : (1) les attributs libres et, (2) les attributs dérivés (DERIVE).

Pour les attributs dérivés, la valeur de ceux-ci est définie soit directement par une expression de dérivation, soit indirectement par une fonction de dérivation à laquelle on doit fournir en paramètre d'entrée des informations accessibles à partir de l'instance dont on calcule l'attribut (par exemple, d'autres attributs de la même instance).

3.4.3.2 Contraintes logiques

Le langage EXPRESS est très expressif au niveau des contraintes. Les contraintes peuvent être classifiées selon deux grandes familles : les contraintes locales, qui s'appliquent

individuellement sur chacune des instances d'un type d'entité ou du type de valeur sur lequel elles sont définies, et les contraintes globales, qui nécessitent une vérification globale sur l'ensemble des instances d'une entité donnée.

Les contraintes locales (*WHERE*) sont définies au travers de prédicats auxquels chaque instance de l'entité, ou chaque valeur du type, sur lequel elles sont déclarées, doit obéir. Ces prédicats permettent, par exemple, de limiter la valeur d'un attribut en restreignant son domaine de valeurs, ou encore de rendre obligatoire la valuation d'un attribut optionnel selon certains critères.

Concernant les contraintes globales :

- La contrainte d'unicité (UNIQUE) contrôle l'ensemble de la population d'instances d'une même entité pour s'assurer que les attributs auxquels s'applique la contrainte possèdent une valeur unique sur toute cette population.
- La contrainte de cardinalité inverse (INVERSE), permet de spécifier la cardinalité de la collection d'entités d'un certain type qui référencent une entité donnée dans un certain rôle. L'attribut inverse exprime l'association entre une entité sujette et des entités référençant l'entité sujette par un attribut particulier.
- Les règles globales (RULE) ne sont pas déclarées au sein des entités mais sont définies séparément. Elles permettent d'itérer sur une ou plusieurs population(s) d'entités pour vérifier des prédicats qui doivent s'appliquer à l'ensemble des instances de ces populations.

3.4.3.3 Procédures et Fonctions

Le langage EXPRESS enrichit son pouvoir d'expression en proposant aux utilisateurs un ensemble de fonctions prédéfinies telles que : *QUERY* (requête d'itération sur des instances), *SIZEOF* (taille d'une collection), *TYPEOF* (introspection : donne le type d'un objet), *USED_IN* (calcul dynamique des associations inverses), etc. Les utilisateurs ont alors la possibilité de définir leurs propres procédures et fonctions en vue du calcul des attributs dérivés, ou encore de l'expression de contraintes d'intégrité.

La définition des fonctions et procédures repose sur un langage impératif structuré, proche de Pascal, incluant déclarations de types et structures de contrôle. Notons que EXPRESS autorise la récursivité, d'où une très grande souplesse d'utilisation.

Pour illustrer cette présentation d'EXPRESS considérons la figure 3.16. Dans cet exemple, nous définissons un schéma de nom *universitaire*. Ce schéma est constitué de cinq entités. Les entités *étudiant* et *salarié* qui héritent de l'entité *personne*. L'entité *étudiant_salarié* qui hérite des entités (héritage multiple) *étudiant* et *salarié*. Enfin l'entité *notes* qui est co-domaine de l'attribut *ses_notes* de *étudiant*. L'entité *personne* définit un attribut dérivé (*nom_prenom*) qui est associé à la fonction EXPRESS (*nom_complet*) retournant la concaténation de l'attribut *nom* et *prénom* d'une instance de l'entité *Personne*. On peut remarquer la contrainte locale dans l'entité notes qui permet de vérifier que les valeurs de l'attribut *note_40* sont comprises entre 0 et 40.

```
SCHEMA Universitaire ;
TYPE CLASSE = ENUMERATION OF (A1, A2, A3);
END_TYPE;
ENTITY Personne ;
SUPERTYPE OF ONEOF (Etudiant, Salairié);
  noSS : NUMBER;
  nom : STRING ;
  prénom : STRING;
  age : INTEGER;
  conjoint : OPTIONAL Personne;
DERIVE
 nom_prenom : STRING := Nom_complet (SELF);
UNIQUE url : NoSS;
END_ENTITY ;
ENTITY Notes ;
  module_ref : STRING;
  note_40 : REAL ;
INVERSE
 appartient_à : SET[0 :?] OF Etudiant FOR ses_notes ;
WHERE
  wr1: {0 <= note_40 <= 40};
```

```
DERIVE
  note_20 : REAL := note_40/2;
END_ENTITY ;

ENTITY Etudiant ;
SUBTYPE OF (Personne);
 sa_classe : CLASSE;
 ses_notes : LIST[0 : ?] OF NOTES;
END_ENTITY ;
ENTITY Salarié;
SUBTYPE OF (Personne)
  salaire : REAL;
END_ENTITY ;
ENTITY Etudiant_Salarié;
SUBTYPE OF( Salarié, Etudiant);
END_ENTITY ;

FUNCTION Nom_complet(per : Personne): STRING;
RETURN (per.nom + ' ' + per.prenom);
END_FUNCTION;

END_SCHEMA ;
```

FIG. 3.16 – Exemple d'un schéma EXPRESS

3.4.4 Représentation Graphique d'EXPRESS

EXPRESS possède également une représentation graphique appelée EXPRESS-G. Il permet une représentation synthétique d'un modèle de données EXPRESS. De plus, ce formalisme peut être utilisé dans les phases préliminaires de conception de modèles de données. EXPRESS-G permet une représentation des concepts structurels et descriptifs du modèle de données par une annotation graphique, ce qui augmente la lisibilité et la compréhensibilité. Par contre, les aspects procéduraux (dérivation et contraintes) ne peuvent être représentés. Une contrainte est seulement indiquée par une "*" sur le nom de(s) attribut(s) correspondant(s). La représentation graphique du schéma EXPRESS de la figure 3.16 est présentée dans la figure 3.17

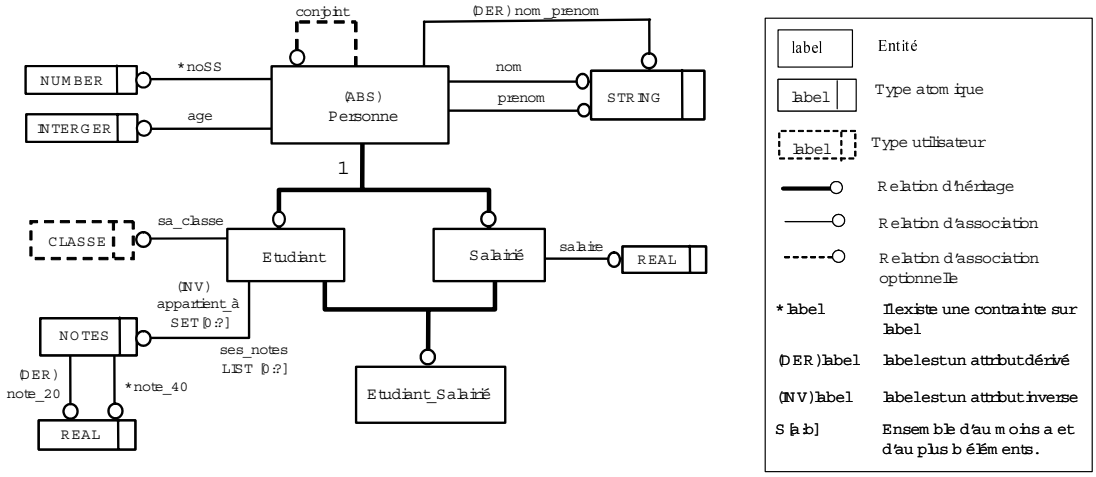

FIG. 3.17 – Exemple d'un Schema EXPRESS en EXPRESS-G.

Schéma cible	Schéma de mapping	Instances de résultat de mapping

SCHEMA Person ;
ENTITY Person ;
id : **NUMBER**;
name : **STRING** ;
minor : **BOOLEAN**;
UNIQUE ur1 : id;
END_ENTITY ;
END_SCHEMA
	SCHEMA_MAP mapping_exemple;
	REFERENCE FROM Universitaire **AS SOURCE**;
	REFERENCE FROM Person **AS TARGET**;
	MAP U_personne_map **AS** per : person;
	FROM p: personne;
	SELECT
	per.id := p.noSS;
	per.name := p.nom_prenom;
	per.minor := **IF** p.age > 17 **THEN** ***TRUE*** **ELSE** ***FALSE***;
	END_IF;
	END_MAP;
	END_SCHEMA;
		#101 = PERSON(02489, 'Nguyen Sylviane', 'TRUE');
		#102 = PERSON(13457, 'Jean Hondjack', 'TRUE');
		#103 = PERSON(23215, 'Jean Nathalie', 'TRUE');

FIG. 3.18 – Exemple de transformation avec EXRESS-X

3.4.5 Modularité

Destiné a concevoir des modèles de taille importante, par exemple l'ensemble des entités modélisant un avion, EXPRESS intègre des mécanismes de modularité permettant la décomposition d'un modèle complexe en plusieurs sous-modèles. Ceci permet de faciliter la conception, la maintenance et la réutilisation d'un modèle. Un modèle EXPRESS appelé schéma peut faire référence à un ou plusieurs autres schémas soit pour intégrer toutes ou une partie des entités définies dans ceux-ci (USE), soit uniquement pour typer des attributs des schémas référencés (REFERENCE). Le découpage de la modélisation d'un domaine donné peut se faire selon deux approches qui peuvent être combinées :

- horizontal : chaque schéma modélise un sous domaine du domaine considéré ;
- vertical : chaque schéma représente une modélisation du domaine à un niveau d'abstraction différent.

3.4.6 Transposition de modèle : EXPRESS-X

EXPRESS-X est un complément déclaratif du langage EXPRESS (ISO 10303-14) dont l'objectif est de permettre une spécification explicite des relations de correspondances (mapping) existant entre des entités de différents schémas EXPRESS. Ce langage supporte deux types de constructions spécifiques :

1. SCHEMA_VIEW qui permet de déclarer des Vues spécifiques des données d'un schéma EXPRESS,
2. SCHEMA_MAP qui permet de déclarer des correspondances ("mapping") de transformation entre des entités ou des vues d'un (ou plusieurs) schéma(s) EXPRESS source(s) vers un (ou plusieurs) schéma(s) EXPRESS cible(s).

Dans l'exemple de la figure 3.18, nous déclarons un mapping pour faire migrer les instances du schéma de la figure 3.16 en des instances du schema *person* (première colonne du tableau). La deuxième colonne du tableau montre la déclaration du mapping.

Il transforme les instances de l'entité *personne* en des instances de l'entité *person* de la manière suivante :

- le *noSS* de *personne* correspond à l'attribut *id* de l'entité *person* ;
- l'attribut *name* de l'entité *person* est la concaténation des attributs *nom* et *prénom* de l'entité *personne*.
- l'attribut *minor* de type BOOLEAN est dépendant de l'attribut *age* de l'entité *personne*. La valeur de l'attribut est VRAI si la valeur de l'attribut *age* est supérieur à 17 et FAUX dans le cas contraire.

La dernière colonne du tableau montre l'application du mapping sur deux instances de l'entité *salarié* décrites comme indiqué ci-dessous (section 3.4.7).

Notons que ce langage est essentiellement déclaratif, on peut néanmoins utiliser toute la puissance du langage procédural pour calculer des attributs (dérivés) du schéma source destinés a être transposés dans le schéma cible.

3.4.7 Représentation des instances

A tout modèle EXPRESS, est automatiquement associé un format de représentation textuel d'instances permettant de réaliser l'échange entre systèmes. Ce format est appelé fichier physique [ISO10303-21 :1994].

Une caractéristique très importante d'EXPRESS est que la structure du fichier physique est automatiquement définie dès que le modèle de données est écrit. Il existe même des générateurs de programmes qui, par compilation du modèle EXPRESS génèrent automatiquement les programmes capables de lire le format d'échange et d'alimenter une base de données orientée objets (Java, C++,...) et/ou de lire une base de données et d'écrire dans le format d'échange.

Les instances de la figure 3.19 sont celles des entités du schéma de la figure 3.16.

```
#2 = SALARIE(13457,'Jean','Hondjack',27,#3,1000);
#3 = SALARIE(23215,'Jean','Nathalie',25,#2,2500);
…
#100 = ETUDIANT(02489,'Nguyen','Sylviane',18,$,'A3',(#101,#102));
#101 = NOTE('A3_120',31);
#102 = NOTE('A3_121',28);
```

FIG. 3.19 – Un exemple de fichier physique conforme aux entités de la figure 3.17.

3.4.8 Un exemple d'environnement de modélisation EXPRESS : ECCO

EXPRESS étant un langage textuel, tout ensemble de schémas peut être compilé. Le résultat de cette compilation est de générer des programmes qui permettent :

- de lire des instances figurant dans un fichier physique et de les représenter en mémoire ;

- d'accéder à ces instances par une interface graphique ;
- d'accéder à ces instances par programme : directement en EXPRESS, ou via une API dans les autres langages (C, C++, Java) ;
- de vérifier les contraintes définies au niveau schéma et de calculer les attributs implicites (dérivés et inverses) pour une population d'instances ;
- de sauvegarder les instances sous forme d'un fichier physique.
- d'accéder simultanément au schéma lui-même sous forme d'instance d'un méta-schema ;

Différents environnements existent. Certains permettent de faire persister les instances créées et manipulées. Par exemple, EXPRESS Data Manager (EPM Technology) est lié à une base de données. Toute compilation d'un schéma EXPRESS génère automatiquement le schéma de base de données correspondant, les contraintes assurant l'intégrité de la base et l'interface SDAI permettant d'accéder à la fois aux schémas et aux instances.

Le travail effectué dans cette thèse se base principalement sur un autre environnement qui est l'environnement ECCO (EXPRESS Compiler COmpiler) [99]. ECCO est un environnement de développement EXPRESS qui offre beaucoup de possibilités, et, en particulier :

1. L'édition, vérification de syntaxe et de sémantique des modèles.
2. La manipulation graphique de la population des modèles.
3. La génération d'une librairie de fonctions en java et C++ pour :
   (a) l'accès aux données
   (b) la lecture/écriture de fichiers physiques. Une fois les traitements effectués, les instances doivent être sauvegardées sous forme d'un fichier physique pour être conservées.
   (c) la vérification des contraintes
   (d) l'accès à la description du schéma lui-même sous forme d'instances d'un méta schéma d'EXPRESS
4. La possibilité de programmer dans le langage EXPRESS-C qui est essentiellement le langage EXPRESS et permet donc d'accéder à un modèle EXPRESS et à ses instances mais possède également deux extensions :
   (a) La capacité de faire des entrées-sorties (par exemple lire et écrire dans un fichier)
   (b) La capacité de déclencher un programme par un événement (il s'agit donc d'une sorte de "programme principal") écrit en EXPRESS, ce qui n'existe pas en EXPRESS standard.

La Figure 3.20 illustre l'architecture ECCO. Le code source est passé en entrée d'un analyseur en vue de la vérification de la structure du code écrit, conformément à la grammaire du langage EXPRESS, et en vue de la vérification des règles de sémantique statique,

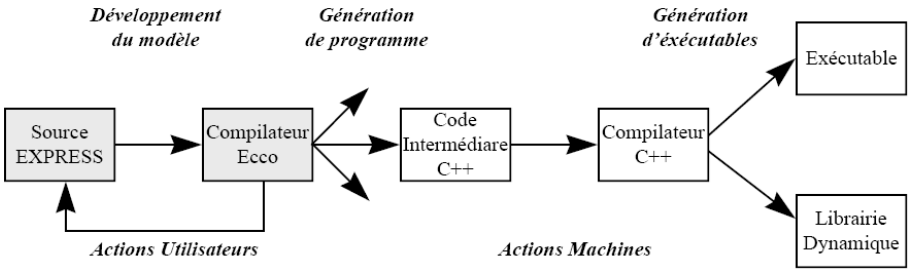

FIG. 3.20 – Définition d'un modèle de données EXPRESS par Ecco [93]

comme par exemple le contrôle de types. Des outils d'aide au développement permettent, d'une part, d'identifier précisément les erreurs au travers de messages clairs, et, d'autre part, de se positionner directement au niveau de l'erreur dans le code source en vue d'une correction. Lorsque cette première phase de vérification d'un modèle de données EXPRESS est terminée, un code intermédiaire (du C++ dans notre cas) est généré, puis compilé et soumis à un éditeur de liens afin de produire un code exécutable (ou une librairie dynamique) qui va permettre de créer des populations de données et ainsi offrir un outil de vérification sémantique propre au modèle de données sur ces populations tests.

3.4.9 Bilan sur la partie EXPRESS

Le langage EXPRESS constitue un outil formel permettant la modélisation de la connaissance selon les trois points de vue (structurel, descriptif et procédural) sous une forme traitable par machine. L'existence d'un langage procédural proche du PASCAL associé à de nombreuses fonctions prédéfinies (telles que QUERY) permettent l'expression de n'importe quel type de contrainte calculable et de n'importe quel type de fonction dérivation. Les modèles EXPRESS ont donc une sémantique (ensembliste) très précise, ce qui permet d'utiliser ce langage pour définir des modèles très grands et très complexes tels que les modèles de produits définis dans le projet STEP et les méta-modèles définis dans le projet PLIB pour la modélisation à base ontologique de catalogues de composants. Notons qu'EXPRESS possède de nombreux autres domaines d'applications. Par exemple, l'expression des contraintes fonctionnelles (via les attributs dérivés) permet de développer la programmation événementielle appliquée à EXPRESS. Ceci a été utilisé en particulier pour programmer des générateurs de code et des outils d'*I*ngénierie *D*irigée par les *M*odèles (IDM) [29].

Une caractéristique essentielle de la technologie EXPRESS est qu'elle est outillée par des environnements de haut niveau. Nous avons présenté dans cette section un environnement de modélisation EXPRESS, à savoir ECCO, qui sera utilisé pour développer notre prototype d'intégration et valider nos propositions. ECCO permet de générer à partir de la compilation d'un modèle EXPRESS une application C++ qui représente automatiquement toutes les instances EXPRESS sous forme d'instances C++, et toutes les contraintes

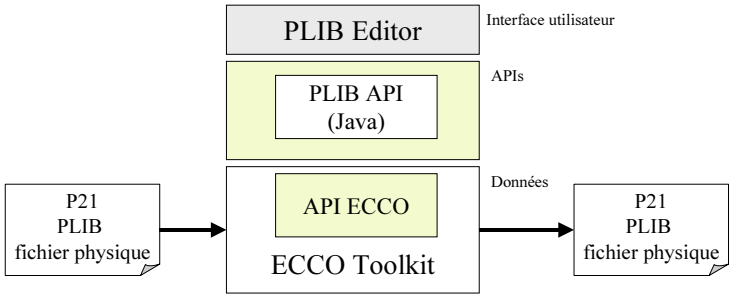

FIG. 3.21 – Structure globale de PLIBEditor

sous forme de méthodes C++. Le schéma lui-même est également représenté sous forme d'instances C++ d'un méta-schéma. Cette application, qui comporte en particulier les fonctions de lecture/écriture de fichiers physiques, peut être exploitée par une interface graphique, par une API utilisable en C, C++, Java et directement en EXPRESS par une application compilée avec le modèle. Une fois les traitements effectués, les instances doivent être sauvegardées sous forme d'un fichier physique pour être conservées.

La section qui suit présentera un outil qui a été développé au sein du LISI permettant d'éditer à la fois une ontologie PLIB et une *BDBO* basée sur une ontologie PLIB.

3.5 Outil PLIBEditor

Comme son nom l'indique, PLIBEditor est un éditeur d'ontologies (et de BDBOs) au format PLIB. Il est développé en JAVA. Dans sa version de base, PLIBEditor permet de visualiser, modifier ou créer une ontologie ou une BDBO basée sur une ontologie PLIB. Au sein du LISI, il existe deux versions de PLIBEditor, à savoir PLIBEditor ECCO-EXPRESS et PLIBEditor POSGRES-SQL. Le point commun des deux versions est l'interface utilisateur, qui est implémentée en Java. Les différences concernent la nature des données stockées et les APIs implémentées pour manipuler ces données. La version PLIBEditor ECCO-EXPRESS utilise ECCO pour implémenter les APIs, et les données dans ce cas sont donc stockées en mémoire centrale et archivées sous forme de fichiers physiques (voir la figure 3.21). Par contre, dans la version POSGRES-SQL, les APIs sont implémentés par le langage POSGRES-SQL, et les données sont donc stockées sous les tables SQL de POSGRES-SQL. Nous ne présentons ci-dessous que la partie commune, c'est à dire l'interface utilisateur des deux versions PLIBEditor.

3.5.1 Définition des classes et des propriétés ontologiques

La capture d'écran de la figure 3.22 représente l'écran principal de PLIBEditor. Cette fenêtre principale est composée de 4 zones :

1. zone 1 : regroupe les barres de menu et d'outils.

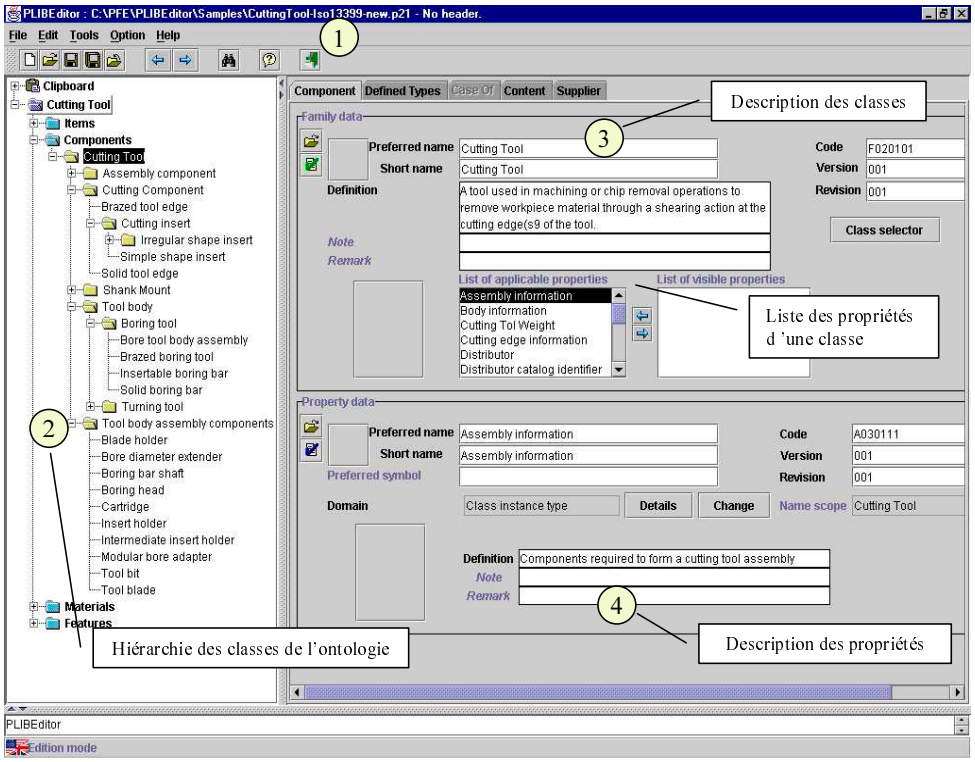

FIG. 3.22 – Édition des classes et des propriétés de l'ontologie

2. zone 2 : représente la hiérarchie des classes de l'ontologie. Il est possible de créer, supprimer, déplacer une classe. Une classe peut être définie soit comme une classe racine, soit comme une classe subsumée par une autre classe. Si une classe est subsumée par la relation *OOSub*, elle hérite automatiquement de toutes les propriétés de sa classe subsumante. Par contre, si une classe est subsumée par la relation *OntoSub*, l'administrateur doit choisir les propriétés à importer dans la classe définie parmi celles de sa classe subsumante (voir la figure 3.23).

3. zone 3 : affiche les attributs de la classe sélectionnée (nom, description, code, version, propriétés visibles et applicables) . Les onglets présents dans la partie supérieure de cette zone permettent de sélectionner les données à afficher : (1) la classe sélectionnée, (2) les types définis dans la classe, (3) le détail des classes subsumantes sélectionnées par la subsomption *OntoSub* (”is-case-of”) (voir la figure 3.23), (4) l'extension ou le contenu de la classe (voir la figure 3.24), ou (5) la source de la définition de la classe.

4. zone 4 : détaille la propriété sélectionnée dans la zone précédente (nom, description, code, version, co-domaine, etc.).

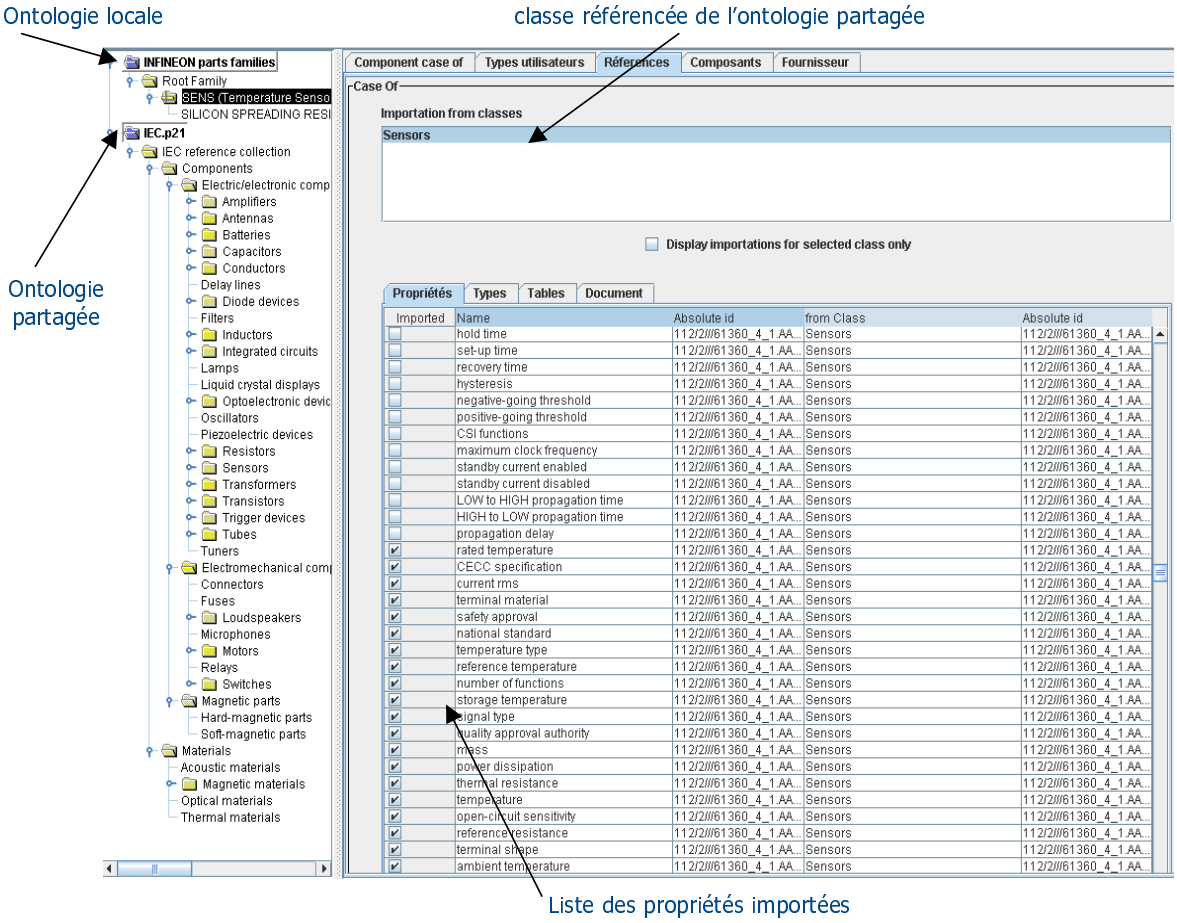

FIG. 3.23 – Création d'une ontologie locale qui référence une ontologie partagée avec PLIBEditor

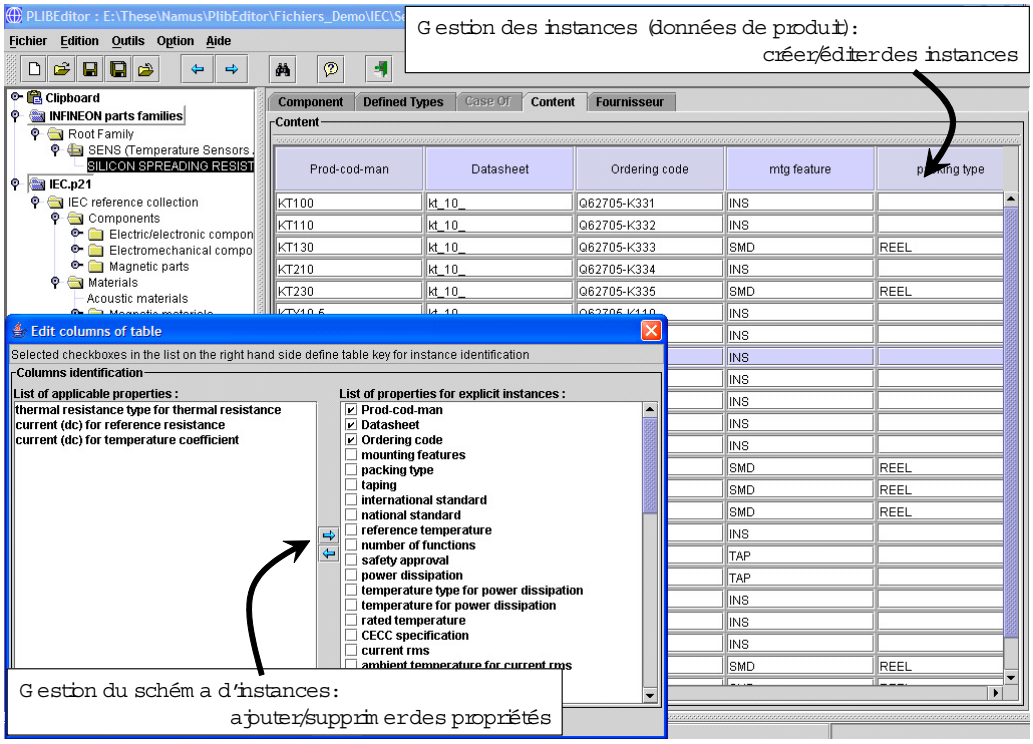

FIG. 3.24 – Édition des instances d'une classe dans une BDBO

3.5.2 Gestion des instances de classe

L'onglet "Content" (voir la figure 3.24) permet, d'une part, de définir ou de changer le schéma d'instances d'une classe et d'autre part, d'ajouter, supprimer et modifier des instances de classe. Le schéma d'instance est créé à partir des propriétés qui sont applicables pour cette classe.

C'est au sein de cet outil que nous implémenterons les différentes méthodes d'intégration de données que nous proposons dans cette thèse.

3.6 Conclusion

Dans ce chapitre nous avons présenté l'ensemble du contexte dans lequel notre travail de thèse a été développé. Nous avons d'abord décrit le modèle formel d'ontologie PLIB qui a été développé au cours des dix dernières années et qui est maintenant publié dans la norme ISO 13584-42 :1998 étendue part les normes ISO 13584-24 :2002 et ISO 13584-25 :2002.

Pour l'essentiel, ce modèle permet de définir de façon formelle et traitable par machine toutes les catégories d'objets que l'on peut avoir besoin de manipuler lors d'une transaction de type B2B, ainsi que les propriétés qui les caractérisent ou décrivent leur état. Il permet ensuite de référencer ces définitions par de simples codes. Comparé, par exemple,

à OWL, le modèle PLIB présente l'avantage de supporter le multi-linguisme, la multi-représentation des concepts et un système de type complet beaucoup plus riche que les types existant en OWL. Le modèle PLIB d'ontologie contient des contraintes d'intégrité et des relations de subsomption entre les classes d'objets modélisés mais ne permet pas d'exprimer d'autres règles de raisonnement sur ces objets. Il s'agit là d'un choix délibéré. D'abord parce que l'objectif d'une ontologie PLIB est de fournir un langage commun à une communauté d'utilisateurs pour permettre les échanges informatisés. Et un tel langage se doit, comme c'est toujours le cas aussi bien dans les domaines techniques que dans celui des bases de données, d'être canonique, c'est-à-dire sans redondance ni possibilité de représentation multiple de la même information. D'autre part, parce que, s'il est apparu possible de définir de façon consensuelle des ontologies d'objets assez générales et indépendantes des objectifs particuliers visés lorsque ces objets sont référencés par des agents informatiques, cela semble beaucoup plus difficilement être le cas pour les règles devant être utilisées par l'agent pour les manipuler. Celles-ci apparaissent, en effet, fortement dépendantes des objectifs de la tâche à accomplir, et il est apparu préférable de les laisser extérieures à l'ontologie.

Nous avons ensuite présenté un nouveau modèle de base de données qui vise à représenter, en plus des données, l'ontologie locale qui en représente le sens, le modèle conceptuel qui en définit la structure, et, éventuellement les correspondances existant entre l'ontologie locale et des ontologies partagées. Les bases de données qui possèdent ces caractéristiques sont appelées bases de données à base ontologique (BDBO). Nous avons proposé une architecture particulière de BDBO appelé OntoDB au développement de laquelle nous avons participé durant notre travail de thèse et que nous avons validé dans un environnement purement objet. Cette architecture comporte, outre les données et les ontologies, un méta-schéma réflexif qui permet de programmer de façon générique plusieurs modèles, ou variantes de modèles d'ontologies, et une méta-base qui décrit la structure logique de l'ensemble des autres parties. Notre approche d'intégration a été réalisée en considérant les sources de données ayant précisément la structure d'une BDBO.

Le modèle d'ontologie et d'instances PLIB était défini formellement dans le langage EXPRESS. Nous avons ensuite présenté ce langage et l'ensemble de la technologie qui lui est associée : modèle d'instances, langage de contrainte et de mapping, environnement de méta-modélisation et d'ingénierie dirigée par les modèles. Nous avons en particulier présenté les outils que nous avons utilisés afin de valider notre processus d'intégration, à savoir l'environnement ECCO pour implémenter les APIs d'intégration, puis l'éditeur PLIB pour créer/gérer les ontologies PLIB et les BDBOs à base PLIB, et dans lequel nous avons introduit les diverses techniques d'intégration proposées dans la suite de cette thèse.

Dans le chapitre suivant, nous allons présenter les principes de notre approche d'intégration, appelée, intégration des bases de données à base ontologique par articulation *a priori* d'ontologies.

Chapitre 4

Intégration automatique des BDBOs par articulation *a priori* d'ontologies

4.1 Introduction

De nombreux systèmes d'intégration ont été proposés dans la littérature (voir le chapitre 2). La principale difficulté de ces systèmes est l'interprétation automatique de la signification, la sémantique, des données hétérogènes et autonomes ce qui donne lieu à différents conflits. Dans la première génération de systèmes d'intégration, la signification des données n'est pas représentée explicitement. Les correspondances entre le schéma global et les schémas locaux sont réalisées manuellement et encodées dans des définitions de vues. Avec l'avènement des ontologies, un progrès important a pu être réalisé dans l'automatisation du processus d'intégration de sources hétérogènes grâce à la représentation explicite de la signification des données.

Plusieurs types d'ontologies ont été utilisés dans les systèmes d'intégration. Elles sont d'ordre linguistiques ou conceptuelles. Les ontologies linguistiques permettent une automatisation partielle du processus d'intégration avec la supervision d'un expert humain. Les ontologies conceptuelles permettent une automatisation effective pour autant que chaque source référence exactement la même ontologie, sans possibilité d'extension ou d'adaptation. La limite de ces systèmes réside dans le fait qu'une fois l'ontologie partagée définie, chaque source doit utiliser le vocabulaire commun. L'ontologie partagée est en fait un schéma global, et, en conséquence, chaque source locale peu, ou pas d'autonomie schématique.

Dans de nombreux domaines comme les Web service [88], l'e-commerce [6, 56], la synchronisation des bases de données réparties [1, 58], le nouveau défi consiste à permettre une intégration entièrement automatique des sources de données gardant une autonomie significative. La transformation du schéma à travers lequel une information est représentée sous forme de données nécessitant d'interpréter la *signification* des différentes données, nous pensons que l'automatisation complète de cette transformation n'est possible qu'à

deux conditions :

1. chaque source doit représenter explicitement la *signification de ses propres données* ; c'est la notion d'ontologie locale qui doit exister dans chaque source ; et
2. il doit exister une ontologie partagée du domaine, et chaque ontologie locale doit référencer explicitement l'ontologie partagée pour définir *les relations sémantiques existant entre les concepts des ontologies locales et globale (articulation).*

Ce chapitre est dédié à la présentation détaillée de notre approche d'intégration, appelée **intégration automatique des BDBOs par articulation *a priori* d'ontologies**. Notre approche n'élimine pas la nécessité d'une réflexion humaine pour identifier deux conceptualisations différentes d'une même réalité. Mais elle demande que cette réflexion soit faite *a priori*, lors de la mise à disposition de la source de données, et non *a posteriori*, pendant la phase d'intégration. C'est ce qui permettra à notre approche de pouvoir penser à très grande échelle.

Ce chapitre est organisé comme suit. La section 2 constitue la problématique à partir de laquelle notre approche a été développée. Dans la section 3, nous présentons d'une façon générale l'architecture de notre système intégré. Les sections 4, 5, 6 visent à décrire les trois scénarios d'intégration correspondant à trois opérateurs algébriques de composition de BDBO que nous considérons dans notre étude. Pour chaque scénario, nous présenterons tout d'abord son contexte, puis l'algorithme d'intégration qui lui correspond, et enfin son application dans le commerce électronique professionnel. Avant de conclure ce chapitre, la section 7 présentera la mise en oeuvre de notre approche d'intégration. Il s'agit d'implémenter dans PLibEditor une extension qui permet d'intégrer automatiquement des catalogues électroniques (conformant au modèle d'ontologie PLIB) au sein d'un entrepôt de composants techniques.

4.2 Problématique

Avec la croissance exponentielle du nombre de sources de données apparaissant sur le Web, les méthodes d'intégration traditionnelles imposant une activité manuelle de l'administrateur central apparaissent de moins en moins faisables ou acceptables. En ce qui concerne les données semi-structurées, le besoin d'intégration automatique est adressé à travers la notion de méta-données représentée par exemple en RDF [12] ou RDFS [13]. L'idée qui préside à cette approche est que, si le travail sémantique d'intégration ne peut être réalisé *a posteriori*, alors elle doit s'effectuer *a priori* par les auteurs de sources documentaires. Ceci en ajoutant à des documents (sémantiquement et terminologiquement hétérogènes) des méta-données qui référencent une ontologie commune et fournissent donc une interface intégrée et homogène destinée à la recherche de documents pertinents pour une requête appartenant au domaine de l'ontologie.

[12] `http://www.w3.org/TR/rdf-primer`
[13] `http://www.w3.org/TR/rdf-schema`

L'approche d'intégration que nous proposons correspond à la mise en oeuvre de la même idée dans l'univers des bases de données. Dès lors qu'un responsable de base de données connecte celle-ci au Web, c'est pour en rendre le contenu facilement accessible. A partir du moment où des ontologies de domaine existent et sont acceptées (e.g., ce sont normalisées) c'est bien dans les termes de ces ontologies qui fournissent une *interface générique*, que les utilisateurs vont rechercher l'information. En effet, dans un nombre important de domaines, des ontologies de domaine consensuelles commencent à exister. C'est en particulier le cas dans de nombreux domaines techniques, où des ontologies de domaine *normalisées* sont en train d'émerger (par exemple, un ensemble d'ontologies techniques normalisées est disponibles sur le site `www.oiddi.org`). Dans ces domaines, il est également fréquent que les concepteurs de base de données souhaitent **rendre l'accès le plus facile possible à des ensembles d'utilisateurs les plus vastes possibles**. C'est en particulier le cas du commerce électronique, où chacun souhaite que son catalogue électronique soit très largement consulté, exploité et compris.

Ebay (`www.ebay.com`), par exemple, est un e-marché supportant l'e-commerce dans plusieurs domaines. Il organise les produits selon une *hiérarchie de catégories* de produits. Pour mettre en vente un produit, son fournisseur doit d'abord choisir la catégorie à laquelle ce produit est associé. Puis il décrit son objet à l'aide des *propriétés standards* fournies par Ebay, qui caractérisent la catégorie choisie. Enfin, le vendeur peut ajouter des informations complémentaires (images, commentaires textuels, etc) sur la description de son produit. Lorsque le fournisseur complète les champs caractérisant son objet, les utilisateurs qui explorent la catégorie de mise en vente de son objet ont accès à un menu très pratique qui leur permettra de rechercher des objets en fonction de leurs attributs. D'après Ebay, cet aspect d'utilisation de propriétés standards est considéré comme très important dans la mesure où :

- chaque fournisseur gagne du temps lors de la rédaction des descriptions
- chaque objet est décrit de façon claire et compréhensible par les utilisateurs.
- les utilisateurs disposent ainsi d'un moyen simple et rapide de trouver les objets recherchés ;

Notons que l'e-marché Ebay est réparti sur plusieurs pays, et des pays différents peuvent utiliser non seulement des langages différents, mais également des propriétés légèrement différentes pour caractériser une catégorie de produits (voir la figure 4.1). Par contre, les fournisseurs ne peuvent utiliser que les catégories et les propriétés fournies par Ebay pour décrire leurs objets ce qui n'est pas acceptable dans le marché professionnel où chaque vendeur essaye précisément d'individualiser son produit par des caractéristiques spécifiques.

Dans l'exemple de l'e-marché Ebay ci-dessus, on constate que les fournisseurs n'ont pas beaucoup d'autonomie schématique comme par exemple l'ajout de nouvelles catégories. Ce problème se pose aussi pour les applications d'e-commerce existantes et c'est d'après nous, une des raisons de leur démarrage assez lent dans le domaine professionnel. Notre

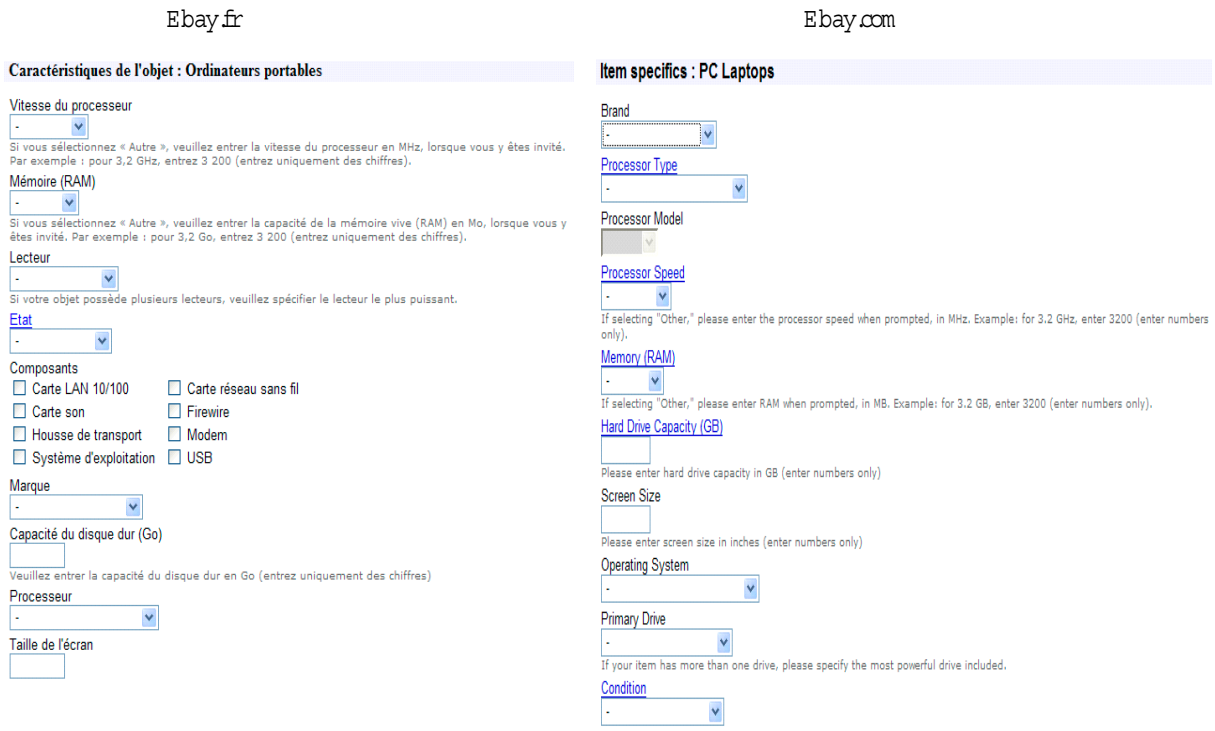

FIG. 4.1 – Exemples des propriétés standards caractérisant la catégorie "Ordinateur portable" sur les deux sites : *Ebay français* et *Ebay américain*

approche d'intégration s'inscrit également dans le contexte d'intégration des catalogues de composants d'un système d'e-commerce professionnel *B2B* autour d'une ontologie partagée. Mais elle est basée sur trois principes :

1. chaque source participante au processus d'intégration doit *contenir sa propre ontologie* (appelée base de données à base ontologique (BDBO) présentée dans la chapitre 3),
2. chaque ontologie locale s'articule *a priori* avec une (ou des) ontologie(s) partagée(s),
3. chaque ontologie locale *étend l'ontologie partagée* pour satisfaire ses besoins.

A notre connaissance, notre travail est le premier à traiter du problème d'intégration en proposant qu'une ontologie conceptuelle soit explicitement représentée dans chaque source de données, et que les articulations avec les ontologies partagées soient définies *a priori*. Dans cette thèse, comme dans [69], l'articulation entre l'ontologie globale et les ontologies locales sont exprimées par un ensemble de relations de subsomption.

Pour mieux comprendre notre problème, nous considérons ci-dessous l'exemple d'un e-marché de catalogues de composants industriels.

Exemple 8 *: Un marché B2B consiste à permettre l'échange des données de composants du domaine des matériaux informatiques. Les composants du marché sont issus de différents fournisseurs. Ils sont décrits dans les catalogues (représentés sous forme d'une base*

de données). Chaque catalogue contient également une ontologie qui explicite la sémantique des composants de ce catalogue. Nous supposons que les catalogues sont distribués et autonomes.

Afin de fournir une interface générique, l'administrateur propose au marché une ontologie partagée du domaine. Cette ontologie est soit fournie par une organisation indépendante du marché, soit créée par ce marché lui-même. Chaque fournisseur participant au marché construit son ontologie locale en réutilisant aussi souvent que possible les définitions existantes dans l'ontologie partagée. Cela permet de simplifier le processus de conception et de réduire le temps pour la construction de l'ontologie locale. Notons que chaque catalogue local est autonome. Ceci peut être compris en observant que :

1. *d'une part, un catalogue local peut avoir des concepts ontologiques propres qui sont définis localement pour s'adapter aux besoins privés, (autonomie schématique) ;*
2. *d'autre part, un fournisseur peut à volonté intégrer ou retirer son catalogue dans l'e-marché, (autonomie opérationnelle).*

Supposons qu'un tiers veuille intégrer les catalogues du marché au sein d'un catalogue unique et homogène. Le but du catalogue intégré est d'offrir un accès générique aux composants des différents fournisseurs. A cause de l'autonomie de chaque source de tel e-marché, le système intégré doit répondre à deux problèmes :

1. *l'automatisation, c'est-à-dire que le système doit intégrer automatiquement les catalogues de composants même si les ontologies locales contiennent des concepts différents de ceux dans l'ontologie partagée.*
2. *le passage à l'échelle (scalability), c'est-à-dire que le système doit supporter l'ajout de nouveaux catalogues lorsqu'ils participent au marché.*

Dans les sections qui suivent, nous allons présenter une méthode d'intégration répondant aux problèmes posés ci-dessus.

4.3 Architecture du système d'intégration de BDBOs

Notre architecture d'intégration part, en entrée, d'un ensemble de *BDBO* référençant *a priori* une ontologie partagée , et vise à produire, en sortie, un entrepôt ayant lui même la structure d'une BDBO. Cette architecture définit plusieurs opérateurs d'intégration correspondant à différents scénarii possibles qui dotent l'ensemble des BDBOs d'une structure d'algèbre. Cette architecture est illustrée dans la figure 4.2.

Nous allons d'abord présenter les conditions que chaque ontologie locale doit respecter lorsqu'elle s'articule avec l'ontologie partagée, ensuite les scénarii d'intégration considérés.

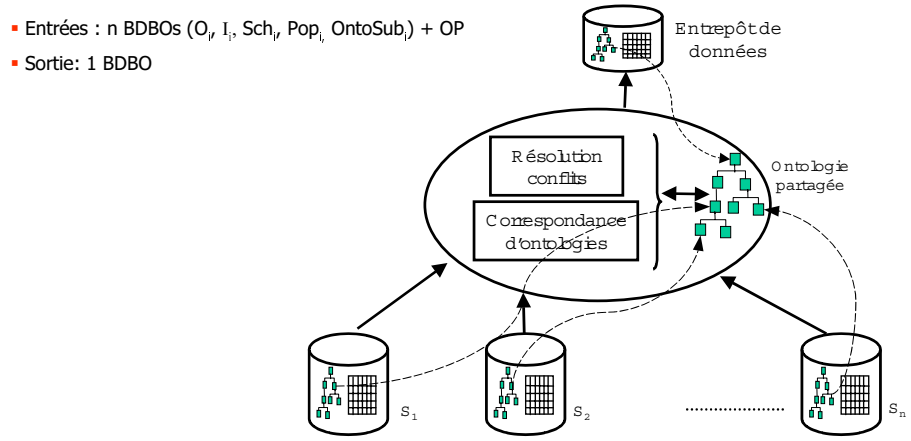

FIG. 4.2 – Système d'intégration des bases de données à base ontologique par articulation *a priori* d'ontologies : algèbre de composition.

4.3.1 Principe d'engagement sur une ontologie de référence

Une ontologie est dite "partagée" entre plusieurs sources, lorsque les sources s'engagent sur le sens et *sur le fait d'utiliser* les définitions ontologiques de l'ontologie partagée qui ont été acceptées et éventuellement normalisées. Afin de garder l'autonomie d'une source, cette source peut définir sa propre hiérarchie de classes, et, si besoin est, rajouter les propriétés qui n'existent pas dans l'ontologie partagée.

Plus précisément, s'engager sur une ontologie partagée signifie respecter la double contrainte suivante (appelée SSCR : Smallest Subsuming Class Reference [10]) :

- toute classe locale doit référencer, par la relation *OntoSub*, la plus petite classe subsumante existante dans la hiérarchie de référence si ce n'est pas la même que celle de sa propre super classe ;
- toute propriété nécessaire à l'ontologie locale et existant dans l'ontologie de référence doit être importée à travers la relation *OntoSub*.

Si une ontologie locale O_i est articulée avec l'ontologie partagée O_p en respectant le principe $SSCR$, nous disons qu'elle "*référence autant que cela est possible*" l'ontologie O_p.

Nous formulons l'articulation entre une $BDBO_i$ et l'ontologie O_p comme un triplet $\mathcal{A}_{i,p} :< BDBO_i, O_p, OntoSub_{i,p} >$, où

- $BDBO_i :< O_i, I_i, Pop_i, Sch_i >$ représente une base de données à base ontologique, dont $O_i :< C_i, P_i, Sub_i, Applic_i >$ est l'ontologie locale
- $O_p :< C_p, P_p, Sub_p, Applic_p >$ représente l'ontologie partagée portant sur le même univers du discours que $BDBO_i$,
- $OntoSub_{i,p} : C_p \rightarrow 2^{C_i}$ représente les relations de subsomption entre O_p et O_i qui associent à chaque classe c_p de C_p l'ensemble des classes $c_i \in C_i$ qui sont subsumés directement par c_p :
  $\forall c_p \in C_p, OntoSub_{i,p}(c_p) = \{c_i \in C_i | (c_p \text{ subsume } c_i) \wedge (\forall c'_i | c_i \in Sub_i(c'_i) \Rightarrow c'_i \notin OntoSub_{i,p}(c_p)) \wedge (\forall c'_p \in Sub_p(c_p) \Rightarrow c_i \notin OntoSub_{i,p}(c'_p))\}$.

Retournons l'exemple 7 dans la section 3.3.3 du chapitre 3, l'articulation entre l'ontologie O_1 et l'ontologie O_p est : $\mathcal{A}_{1,p} :< BDBO_1, O_p, OntoSub_{1,p} >$ avec : $OntoSub_{1,p}(S|C2) = \{(S1|C1)\}$.

4.3.2 Scénarii d'intégration de données

Soit $S = \{S_1, S_2, ..., S_n\}$ l'ensemble des sources de données participant au processus d'intégration. Chaque source S_i est définie comme suit : $S_i :< O_i, I_i, Sch_i, Pop_i >$. Notons que dans une BDBO, tout élément représenté dans le schéma, classe ou propriété doit appartenir à l'ontologie, de sorte que le schéma est un sous-ensemble de l'ontologie, chaque entité représentée correspondant à une classe et ses attributs correspondant aux propriétés applicables choisies. On fait abstraction ici du découpage éventuel résultant des opérations de normalisation, une vue étant, dans tous les cas, créée pour représenter la population de chaque classe.

Pour simplifier le propos, nous supposons désormais que seules les classes feuilles sont choisies comme classes de base et sont directement instanciables. Les classes non feuilles sont supposées "abstraites", c'est-à-dire que leur population est l'union des populations de leurs sous-classes.

Dans la méthode d'intégration par articulation *a priori* d'ontologie, nous supposons que l'ontologie partagée O_p pré-existe à la définition de la source $BDBO_i$. Notons que cette hypothèse est toujours faite lorsque l'on annote à l'aide de méta-données. On suppose alors que l'administrateur de la source (DBA) définit sa propre ontologie, et que cette ontologie référence *autant que cela est possible*, l'ontologie partagée. Ceci signifie que la source $BDBO_i$ est conçue en six étapes :

1. le DBA choisit la hiérarchie de classes (C_i, Sub_i) de sa propre ontologie O_i.
2. le DBA articule cette hiérarchie de classes avec celle de l'ontologie partagée C_p en définissant les relations de subsomption $OntoSub_{i,p}$ entre C_i et C_p.
3. A travers les relations de subsomption $OntoSub_{i,p}$, le DBA importe dans $Applic_i(c_i)$ les propriétés de $Applic_p(OntoSub_{i,p}^{-1}(c_i)) \subset P_p$ qu'il souhaite utiliser dans sa propre ontologie. Ces propriétés appartiennent alors à P_i.
4. le DBA complète éventuellement les propriétés importées par des propriétés supplémentaires, propres à son ontologie définissant ainsi l'ontologie locale : $O_i :< C_i, P_i, Sub_i, Applic_i >$.
5. le DBA de chaque source choisit pour chaque classe feuille les propriétés qui seront valuées en définissant $Sch_i : C_i \rightarrow 2^{P_i}$, et
6. Le DBA choisit une implémentation de chaque classe feuille (e.g., afin d'assurer la troisième forme normale), et il définit ensuite $Sch(c_i)$ comme une vue sur l'implémentation de c_i.

Dans ce cas, le schéma de chaque classe feuille est explicitement défini. Et celui d'une classe non feuille est calculé. Il est calculé comme étant l'intersection entre les propriétés

applicables de c_j et l'intersection des ensembles de propriétés associées à des valeurs dans toutes les sous classes $c_{i,j}$ de c_j :

$$Sch(c_j) = Applic(c_j) \cap (\cap_i Sch(c_{i,j})) \tag{4.1}$$

Une définition alternative qui nous semble préférable peut également être utilisée pour créer le schéma d'une classe non feuille. Elle consiste à prendre l'union des propriétés existant dans au moins une sous-classe, et en complétant par des valeurs nulles :

$$Sch'(c_j) = Applic(c_j) \cap (\cup_i Sch(c_{i,j})) \tag{4.2}$$

Le choix entre ces deux représentations doit être laissé à l'utilisateur.

Nous pouvons distinguer plusieurs scénarii d'intégration, correspondant à différentes articulations entre les ontologies locales et l'ontologie partagée du domaine. Dans cette thèse, trois scénarios sont étudiés [9, 10, 11] :

1. **FragmentOnto** : dans ce scénario, on suppose que les ontologies locales des bases de données sont directement extraites de l'ontologie partagée (chaque ontologie locale est un sous ensemble de l'ontologie partagée).
2. **ProjOnto** : chaque source définit sa propre ontologie (elle n'instancie aucune classe de l'ontologie partagée). Par contre l'ontologie locale référence l'ontologie partagée en respectant la condition SSCR. Dans ce scénario, on souhaite néanmoins intégrer les instances de chaque source comme des instances de l'ontologie partagée ;
3. **ExtendOnto** : chaque ontologie locale est définie comme dans le scénario ProjOnto, mais l'on souhaite enrichir automatiquement l'ontologie partagée. Ensuite toutes les instances de données sont intégrées, sans aucune modification, au sein du système intégré.

Nous allons détailler successivement ci-dessous ces trois scénarios d'intégration.

4.4 FragmentOnto

Le premier scénario d'intégration que nous proposons est nommé *FragmentOnto*. Nous présentons ci-dessous d'abord son contexte d'étude, ensuite l'algorithme d'intégration et enfin son application au domaine des composants industriels.

4.4.1 Contexte

Ce scénario d'intégration suppose que l'ontologie partagée est suffisante pour couvrir toutes les sources locales. Une hypothèse de ce type a déjà été utilisée. Nous pouvons citer par exemple le projet Picsel2 [88], COIN [37] et l'e-marché Ebay présenté plus haut. Dans ce cas, l'autonomie des sources se limite à (1) sélectionner un sous ensemble pertinent de

l'ontologie partagée (classes et propriétés) et (2) concevoir le schéma local de la base de données.

L'ontologie O_i de chaque source S_i étant un fragment de l'ontologie partagée O_p. Elle se définit comme le quadruplet $O_i :< C_i, P_i, Sub_i, Applic_i >$, avec :

- $C_i \subseteq C_p$;
- $P_i \subseteq P_p$;
- $\forall c \in C_i, Sub_i(c) \subseteq Sub_p(c)$;
- $\forall c \in C_i, Applic_i(c) \subseteq Applic_p(c)$.

Pour intégrer ces sources au sein d'une BDBO il suffit de trouver l'ontologie, le schéma et la population du système intégré. Le système intégré est donc défini comme $Int :< O_{Int}, Sch_{Int}, Pop_{Int} >$. Maintenant, il s'agit de calculer sur la structure de chaque élément de Int. Ce calcul est présenté dans la section qui suit.

4.4.2 Algorithme

Nous présenterons successivement ci-dessous comment les éléments de Int sont calculés.

O_{Int}. Dans le cas d'une intégration par $FragmentOnto$: $\forall i : O_i \subset O_p$. Nous pouvons donc utiliser l'ontologie partagée comme l'ontologie du système intégré :

$$O_{Int} = O_p \tag{4.3}$$

Cette définition assure que O_{Int} couvre toutes les sources.

Sch_{Int}. Le schéma du système intégré est défini pour chaque classe comme suit (l'intégration par Intersection) :

$$Sch_{Int}(c) = (\bigcap_{i \in 1..n | Sch_i(c) \neq \phi} Sch_i(c)) \tag{4.4}$$

Cette définition assure que les instances du système intégré ne seront pas complétées par des valeurs nulles. Pour chaque classe, seules les propriétés valuées dans toutes les sources de données seront préservées. Si dans certaines sources on trouve des classes vides, elles ne seront pas prises en compte pour calculer les propriétés fournies par toutes les sources.

Le schéma du système intégré peut être également défini comme suit (intégration par Union) :

$$Sch'_{Int}(c) = (\bigcup_{i \in 1..n | Sch_i(c) \neq \phi} Sch_i(c)) \tag{4.5}$$

Pour le deuxième calcul du schéma intégré, les instances du système intégré seront complétées par des valeurs nulles. Au contraire du cas précédent, pour chaque classe, toutes

les propriétés valuées dans au moins une source seront préservées.

Pop_{Int}. La population de chaque classe du système intégré est définie comme suit :

$$Pop_{Int}(c) = \bigcup_i Pop_i(c) \tag{4.6}$$

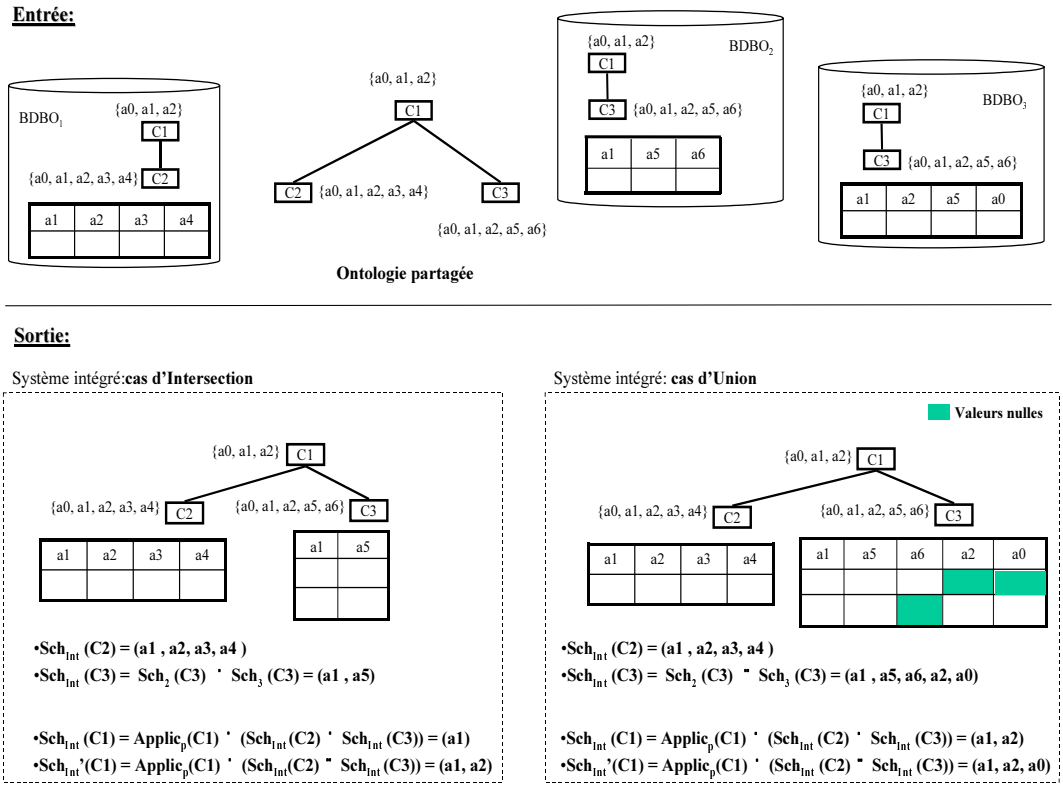

FIG. 4.3 – Exemple d'une intégration de BDBOs par *FragmentOnto*

Pour illustrer l'algorithme d'intégration par *FragmentOnto*, nous étudions l'exemple suivant.

Exemple 9 *: Supposons que l'on ait trois sources de données à base ontologique référençant une ontologie partagée comme dans la figure 4.3.*

La source $BDBO_1$ utilise la branche $\{C_1, C_2\}$ (un fragment de l'ontologie partagée) comme son ontologie locale. Puis la classe feuille C_2 est choisie comme la classe de base des instances de $BDBO_1$, avec $Sch_1(C_2) = \{a1, a2, a3, a4\}$. Quant aux sources $BDBO_2$ et $BDBO_3$, la branche $\{C_1, C_2\}$ est utilisée comme leur ontologie locale. Pour la $BDBO_2$, le schéma physique des instances de données est : $Sch_2(C_3) = \{a1, a5, a6\}$. Et pour la $BDBO_3$: $Sch_3(C_3) = \{a1, a2, a5, a0\}$.

Enfin, l'intégration de ces trois sources donne le résultat illustré dans la figure 4.3.

4.4.3 Application au domaine des composants industriels

L'opérateur d'intégration *FragmentOnto* peut être appliqué dans les systèmes d'intégration développés pour des grands groupes industriels ou des grands donneurs d'ordre (par exemple : PSA, Renault) dans le but de centraliser des catalogues de leurs différents fournisseurs. Il est facile d'imaginer le scénario suivant :

- ces groupes proposent d'abord aux différents fournisseurs leur propre ontologie,
- chaque fournisseur extrait ensuite une partie de cette ontologie selon ses besoins pour construire son catalogue.

Dans un tel environnement, l'intégration automatique de données est assurée. C'est effectivement ce scénario que le projet français PFI (dirigé par Renault) est en train de mettre en oeuvre dans le domaine des composants hors fabrication pour les industries manufacturières. Notons que cette approche fait reposer toute la difficulté sur les fournisseurs qui doivent décrire plusieurs fois leurs données, si plusieurs consortiums existent.

4.5 ProjOnto

Le deuxième scénario que nous étudions est nommé *ProjOnto*. Son contexte d'étude, l'algorithme d'intégration lui correspondant et son application réelle seront détaillés successivement ci-dessous.

4.5.1 Contexte

De nombreuses applications conçues autour de l'approche d'intégration *a priori* exigent plus d'autonomie. Dans le domaine du commerce électronique professionnel qui est le nôtre :

- la classification de chaque source doit pouvoir être complètement différente de celle de l'ontologie partagée, et
- certaines spécialisations de classe et certaines propriétés n'existant pas dans l'ontologie partagée doivent pouvoir être ajoutées dans les ontologies locales. Ce cas est très différent du précédent du fait que chaque source S_i a sa propre ontologie O_i et ses classes spécifiques. Néanmoins, l'ontologie O_i référence autant que possible l'ontologie partagée O_p à travers l'articulation $\mathcal{A}_{i,p} :< BDBO_i, O_p, OntoSub_{i,p} >$.

Dans ce scénario, les sources ont des concepts propres qui n'existent pas dans l'ontologie partagée, mais ces concepts ne sont pas supposés intéresser les utilisateurs du système intégré. Ainsi, le système intégré vise à intégrer les instances de données de chaque source comme des instances de l'ontologie partagée. Les instances de données de chaque S_i seront donc projetées sur l'ontologie partagée. Notons que, dans ce scénario, les sources ne peuplent que les classes qui lui sont propres, pas celles de l'ontologie partagée.

La section qui suit déterminera la structure de chaque élément du système intégré Int ($<Ont_{Int}, Sch_{Int}, Pop_{Int}>$).

4.5.2 Algorithme

Comme dans le scénario précédent ($FragementOnto$), l'ontologie du système intégré est exactement l'ontologie partagée : $O_{Int} = O_p$:

- $C_{Int} = C_p$,
- $P_{Int} = P_p$,
- $Applic_{Int}(c) = Applic_p(c)$,
- $Sub_{Int}(c) = Sub_p(c)$.

Cette définition montre qu'aucun concept défini localement n'est intégré dans le système intégré.

Pour ce scénario, chaque instance de données d'une source sera projetée sur les propriétés applicables de sa plus petite classe subsumante dans l'ontologie partagée. Cette plus petite classe subsumante devient donc la classe de base de l'instance intégrée. Contrairement au cas précédent, la classe de base d'une instance dans le système intégré n'est pas celle d'origine de cette instance dans sa source locale.

Soit $Pop^*(c)$ la population des instances projetées sur la classe c de l'ontologie partagée, $Pop^*(c)$ est calculée par l'union des populations de toutes classes locales référençant directement c [14]. $Pop^*(c)$ est donnée par l'équation suivante :

$$Pop^*(c) = \bigcup_{i\in[1:n]} \Big(\bigcup_{c_j \in OntoSub_i(c)} Pop_i(c_j) \Big) \tag{4.7}$$

Le schéma des instances projetées de la classe c est déterminé comme :

$$Sch^*(c) = Applic_p(c) \bigcap \Big(\bigcap_{i\in[1:n]} \Big(\bigcap_{(c_j \in OntoSub_i(c)) \wedge (Pop_i(c_j) \neq \phi)} Sch_i(c_j) \Big) \Big) \tag{4.8}$$

ou

$$Sch'^*(c) = Applic_p(c) \bigcap \Big(\bigcup_{i\in[1:n]} \Big(\bigcup_{c_j \in OntoSub_i(c)} Sch_i(c_j) \Big) \Big) \tag{4.9}$$

La population et le schéma de chaque classe feuille de l'ontologie partagée sont calculés par les équations 4.7 et 4.8, respectivement (ou 4.7 et 4.9). Autrement dit : $Pop_{Int}(c) = Pop^*(c)$ et $Sch_{Int}(c) = Sch^*(c)$.

En revanche, pour une classe non feuille, les trois équations 4.7,4.8 et 4.9 doivent être complétées comme suit :

$$Sch_{Int}(c) = Applic(c) \bigcap \Big(\Big(\bigcap_{(c_k \in Sub_{Int}(c)) \wedge (Pop*(c_k) \neq \phi)} Sch_{Int}(c_k) \Big) \bigcap_{Pop*(c) \neq \phi} Sch^*(c) \Big) \tag{4.10}$$

ou

[14] les classes locales d'une source S_i référençant directement la classe c de l'ontologie partagée sont les classes dans $OntoSub_i(c)$.

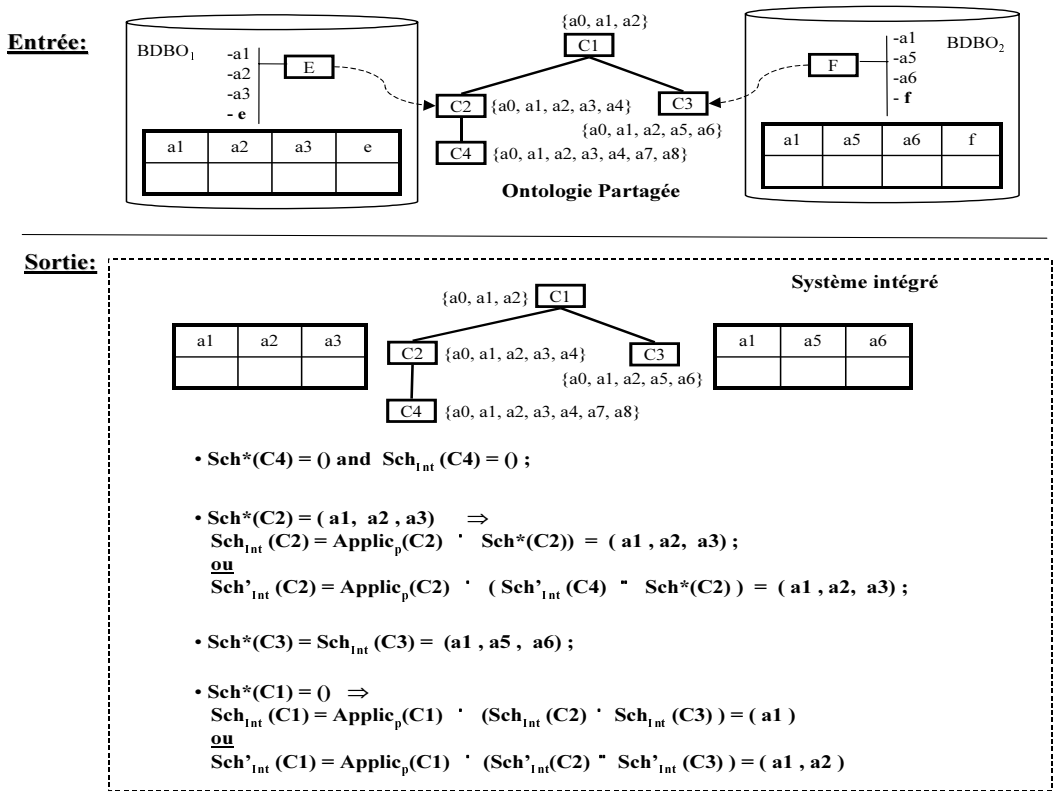

FIG. 4.4 – Exemple d'une intégration de BDBOs par ProjOnto

$$Sch'_{Int}(c) = Applic(c) \bigcap ((\bigcup_{c_k \in Sub_{Int}(c)} Sch'_{Int}(c_k)) \bigcup Sch^*(c)) \tag{4.11}$$

et dans les deux cas :

$$Pop_{Int}(c) = (\bigcup_{c_k \in Sub_{Int}(c)} Pop_{Int}(c_k)) \bigcup Pop^*(c) \tag{4.12}$$

L'exemple suivant illustrera l'algorithme d'intégration par *ProjOnto*.

Exemple 10 *: Supposons que l'on ait deux bases de données à base ontologique référençant une ontologie partagée comme dans la figure 4.4. La classe locale E dans S_1 ($BDBO_1$) référence la classe C_2 (en important les trois propriétés : a_1, a_2, a_3). Quant à la classe locale F de la source S_2 ($BDBO_2$), elle référence C_3 de l'ontologie partagée (en important : a_1, a_5 et a_6). De plus, les classes E et F ajoutent les propriétés e et f, respectivement.*

Le système intégré de deux sources S_1, S_2 selon l'approche ProjOnto est illustré dans la figure 4.4. La classe C2 est une classe non feuille, mais elle devient une classe de base.

4.5.3 Application au domaine des composants industriels

Le scénario d'intégration de données par *ProjOnto* est appliqué au domaine des composants industriels où :

1. il existe une ontologie normalisée du domaine (par exemple, l'ontologie normalisée *IEC 61360-4* sur le domaine des composants électroniques, ou *ISO 13399* sur le domaine des outils coupants, *ISO 13584-501* sur le domaine des matériels de mesure, etc.),
2. cette ontologie est acceptée comme une ontologie partagée par tous les participants de l'environnement "public" $B2B$ étudié. Cet environnement $B2B$ est dit *public*, parce que les fournisseurs gardent une autonomie significative. [15], et
3. chaque fournisseur décrit donc ses composants dans son catalogue en référençant le plus possible cette ontologie normalisée.

Ce scénario *ProjOnto* permet d'intégrer automatiquement tous les composants issus des catalogues de différents fournisseurs comme les instances de l'ontologie normalisée.

4.6 ExtendOnto

Le troisième scénario d'intégration que nous proposons dans cette thèse est nommé *ExtendOnto*.

4.6.1 Contexte

L'entrée du système intégré du scénario *ExtendOnto* est identique à celle de *ProjOnto* où chaque source a sa propre ontologie et ses classes spécifiques. En revanche, ce cas est différent du précédent du fait que le système intégré permet d'intégrer même les extensions de chaque S_i. L'ontologie partagée sera donc étendue en intégrant les ontologies locales. Pour un tel contexte, l'intégration de BDBOs consiste à intégrer d'abord les ontologies, puis les données.

Dans ce cas également, une automatisation du processus d'intégration est possible. Pour ce faire, nous devons trouver la structure finale de la $BDBO$ constituant le système intégré Int : $< O_{Int}, Sch_{Int}, Pop_{Int} >$.

4.6.2 Algorithme

Redéfinissons d'abord la structure de l'ontologie intégrée : O_{Int} : $<C_{Int}$, P_{Int}, Sub_{Int}, $Applic_{Int}>$, où chaque élément de O_{Int} est défini comme suit :

– $C_{Int} = C_p \bigcup_{(i \mid 1 \leq i \leq n)} C_i$,

[15] l'autonomie significative veut dire que chaque fournisseur peut avoir des concepts propres dans son catalogue, et est en plus indépendant de ses acheteurs.

- $P_{Int} = P_p \bigcup_{(i \mid 1 \leq i \leq n)} P_i$,
- $Applic_{Int}(c) = \begin{cases} Applic_p(c), \; si \; c \in C_p \\ Applic_i(c), \; si \; c \in C_i \end{cases}$
- $Sub_{Int}(c) = \begin{cases} Sub_p(c) \cup OntoSub_i(c), \; si \; c \in C_p \\ Sub_i(c), \; si \; c \in C_i \end{cases}$

Définissons ensuite la population du système intégré. Pop_{Int} de chaque classe (c) est calculée d'une manière récursive en utilisant un parcours poste fixé de l'arbre C_{Int}.

Si la classe c appartient à une C_i et n'appartient pas à C_p, sa population est donnée par : $Pop_{Int}(c) = Pop_i(c)$. Si non, i.e., c appartient à l'ontologie partagée, $Pop_{Int}(c)$ est définie par l'équation suivante :

$$Pop_{Int}(c) = \bigcup_{c_i \in Sub_{Int}(c)} Pop_{Int}(c_i) \tag{4.13}$$

Finalement, le schéma de chaque classe du système intégré est calculé en utilisant le même principe que la population en considérant les classes appartenant à C_i et les classes appartenant à C_p (rappelons que les classes de C_p ne sont directement peuplées par aucune source). Les schémas des classes appartenant à l'une des C_i sont explicitement définies ($Sch_{Int}(c)$=$Sch_i(c)$). Concernant les classes de C_p, le schéma de la classe c peut être calculé en appliquant la formule 4.1 (resp. 4.2) sur l'ontologie du système intégré O_{Int} :

$$Sch_{Int}(c) = Applic_{Int}(c) \bigcap (\bigcap_{(c_i \in Sub_{Int}(c)) \wedge (Pop_{Int}(c_i) \neq \phi)} Sch_{Int}(c_i)) \tag{4.14}$$

ou

$$Sch'_{Int}(c) = Applic_{Int}(c) \bigcap (\bigcup_{c_i \in Sub_{Int}(c)} Sch'_{Int}(c_i)) \tag{4.15}$$

Cela montre qu'il est possible d'offrir aux sources locales une large autonomie tout en permettant également une construction automatique du système intégré d'une manière déterministique et exacte.

Pour mieux comprendre l'algorithme d'intégration par *ExtendOnto*, nous observons l'exemple suivant.

Exemple 11 *: nous considérons les mêmes sources et la même ontologie que dans l'exemple du scénario ProjOnto. La structure du système intégré est décrite par la figure 4.5. Afin d'intégrer les deux sources, l'ontologie partagée doit être étendue en intégrant deux classes locales.On note que la vue offerte à l'utilisateur à travers l'ontologie partagée est la même. Mais celui-ci peut, en plus, descendre dans le contenu initial des sources.*

Il est important de noter que, lorsque toutes les sources de données utilisent une ontologie indépendante sans référencer une ontologie partagée, d'une part, l'intégration

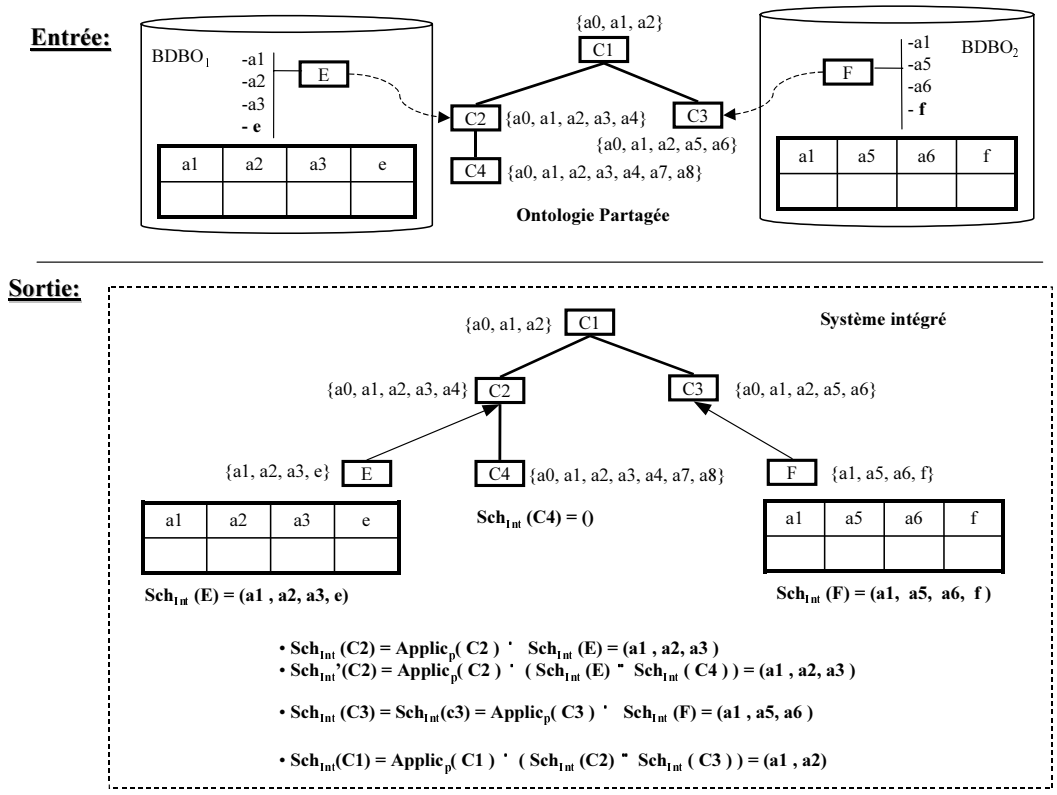

FIG. 4.5 – Exemple d'une intégration de BDBOs par *ExtendOnto*

automatique peut néanmoins se produire (i.e., lecture de toutes les données dans le même entrepôt), et d'autre part, la tâche d'articulation de ces ontologies sur l'ontologie du système receveur peut être faite manuellement, par le DBA. Ensuite une nouvelle intégration peut être réalisée automatiquement comme dans le cas *ExtendOnto* (ce cas est discuté au chapitre 6).

4.6.3 Application au domaine des composants industriels

Le scénario *ExtenOnto* est différent du scénario *ProjOnto* du fait qu'il permet de stocker au sein d'une base unique le contenu complet de chaque catalogue intégré (non seulement les composants, mais également les concepts propres).

Le système intégré dans ce cas supporte deux types d'accès possibles aux composants :

1. l'accès générique à travers l'ontologie normalisée, et
2. l'accès spécifique à chaque catalogue en descendant de l'hiérarchie de l'ontologie intégrée (voir la figure 4.10).

4.7 Mise en oeuvre

Nous avons présenté trois scénarios d'intégration pour l'approche d'*intégration des BDBOs par articulation a priori d'ontologies*. Dans cette section, nous allons présenter une implémentation qui consiste à réaliser effectivement l'intégration au sein d'une base de données orientée objet ECCO, puis à interfacer le résultat avec l'outil $PLIBEditor$, déjà abordé dans le chapitre 3. Cette extension permet de valider les trois scénarios d'intégration proposés ci-dessus.

Nous décrirons tout d'abord l'environnement dans lequel nous avons effectué notre implémentation. Le scénario et l'implémentation de cette mise en oeuvre seront ensuite détaillés.

4.7.1 Environnement de mise en oeuvre

Notre implémentation est réalisée dans l'environnement de base de données orientée objet ECCO où les ontologies et les catalogues de composants sont représentés sous la forme d'un fichier physique conformément au modèle PLIB. Rappelons que ECCO est un environnement qui permet :

- de compiler tout modèle EXPRESS (et donc en particulier PLIB),
- de générer un schéma de gestion des instances de ce modèle,
- de lire et écrire des fichiers physiques,
- d'accéder par programme aux instances et au modèle initial.

L'architecture de cette implémentation est illustrée dans la figure 4.6. Nous détaillons ci-dessous les modules principaux de cette figure.

PLIBAPI. Ecco dispose d'une API pour le langage Java permettant l'accès aux instances et au modèle du fichier physique. Cette API permet d'envoyer des commandes à Ecco sous la forme d'une chaîne de caractères et de récupérer le résultat sous la forme d'un tableau de chaînes de caractères. Par exemple, la commande suivante renverra la liste des entités du modèle instancié :

DBEngine.Exec(" list_entities " + NomDuSchema) ;

"DBEngine" étant un objet de classe Ecco définie dans l'API et "NomDuSchema" représentant le nom du schéma concerné (un fichier physique peut contenir plusieurs schémas). L'API fournit un ensemble de primitives permettant d'accéder au schéma et de manipuler les données, c'est-à-dire de réaliser des mises à jour, de créer ou détruire des instances, de valuer ou de modifier des attributs, de vérifier des contraintes, etc.

PLIBFolders. Le but de PLIBFolders est de fournir les outils permettant de manipuler indépendamment différents ensembles d'instances. En effet, dans l'environnement Ecco, les instances sont toutes réunies en un seul bloc.

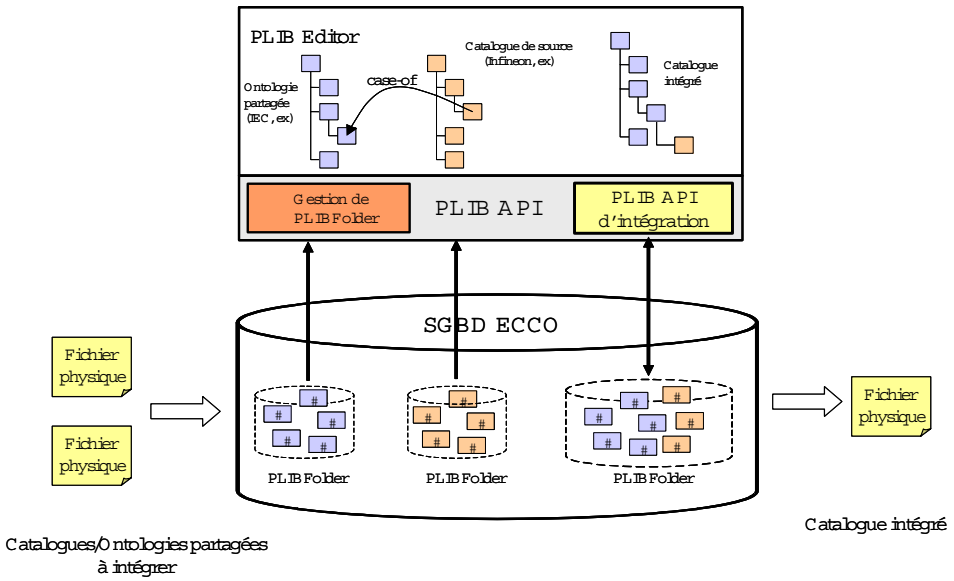

FIG. 4.6 – Architecture PLIBEditor pour la gestion un système d'intégration des catalogues électroniques dans ECCO.

Par exemple, dans les applications de la norme PLIB, si on essaie de lire un fichier physique représentant le catalogue de composants d'un fournisseur (ex : INFINEON), puis un autre fichier physique provenant d'un autre fournisseur (ex : IEC), il n'y a aucun moyen de différencier les instances, ce qui peut être gênant si l'on souhaite par exemple sauvegarder seulement les informations relatives au fournisseur INFINEON.

Des fichiers physiques peuvent être lus à l'aide de l'interface de gestion des folders. Un folder peut ensuite être associé au contenu de chacun de ces fichiers physiques (voir la figure 4.6). Du point de vue implémentation, lorsqu'on parle d'un catalogue (ou d'une ontologie), il s'agit du "*folder*" qui réunit les instances représentant le contenu de ce catalogue (ou de cette ontologie).

PLIB API d'Intégration. Ce module regroupe l'ensemble des fonctions d'intégration que nous devons implémenter pour valider les différents opérateurs de notre approche d'intégration. La section qui suit présentera un exemple de ces APIs.

4.7.2 Exemple des PLIB APIs d'Intégration

Avant de détailler un exemple des APIs d'intégration, nous résumons le processus d'intégration de catalogues que nous avons effectué. En effet, ce processus comporte les trois étapes suivantes :

1. la création d'un entrepôt de composants (catalogue intégré) dont le contenu initial est l'ontologie partagée,
2. l'intégration d'ontologies. Elle consiste :

- si les catalogues sont intégrés par *ProjOnto* ou *FragmentOnto*, à identifier les classes de l'ontologie partagée sur lesquelles les composants seront projetés,
- si les catalogues sont intégrés par *ExtendOnto*, à importer les concepts ontologiques locaux [16] à l'ontologie partagée.

3. l'intégration de composants. Elle consiste à :

   (a) mettre à jour le schéma intégré, et

   (b) projeter les composants à intégrer sur ce schéma.

Nous avons développé des primitives permettant, par composition, de réaliser les opérateurs d'intégration nécessaires. Les primitives fondamentales sont spécifiées dans la tableau 4.1.

En utilisant ces primitives, nous pouvons implémenter les différents *PLIB APIs d'Intégration* correspondant aux différents scénarios. Nous présentons dans cette section, trois APIs :

- *integration_projection* (Cat_i, Cat_{Int}). Cet API permet d'intégrer le catalogue Cat_i dans le catalogue intégré Cat_{Int} selon les deux scénarios d'intégration : *ProjOnto* ou *FragmentOnto*. Son implémentation est résumée dans la table 4.4.
- *integration_extension* (Cat_i, Cat_{Int}). Cet API permet d'intégrer le catalogue Cat_i dans le catalogue intégré Cat_{Int} selon le scénario d'intégration : *ExtendOnto*. Son implémentation est résumée dans la table 4.5.
- *export_ClassInstances* (I_i, c). Cette API permet d'exporter les instances d'entrée (I_i) comme des instances de la classe c de l'ontologie partagée. Notons que les instances I_i peuvent êtres issues des différentes classes de base. Les instances exportées sont projetées sur la classe c, c est alors la classe de base des instance I_i dans le catalogue intégré. Cette fonction consiste à des étapes suivantes :

  1. récupérer l'ensemble des classes de base des instances I_i, noté par $\bar{C}_i$,

  2. pour chaque $\bar{c}_i \in \bar{C}_i$: **si** la classe c subsume la classe $\bar{c}_i$ **alors** :
     - mettre à jour le schéma de l'extension de la classe c par les propriétés du schéma de l'extension de la classe $\bar{c}_i$ (voir la fonction *update_Schema* présentée dans la table 4.1).
     - pour chaque l'instance $i \in I_i \wedge i \in Pop(\bar{c}_i)$ [17] : projeter l'instance i dans l'extension de la classe c (voir la fonction *insert_ClassInstance* présentée dans la table 4.1).

[16] rappelons qu'un concept ontologique o dans le modèle PLIB peut être : soit un fournisseur (la description d'une source), soit une classe, soit une propriété, soit un type de données.

[17] $Pop(\bar{c}_i)$: l'ensemble des instances de la classe $\bar{c}_i$

Primitives	**Description**
Opérateurs sur l'ontologie	
$get\_Code\ (o_i \rightarrow id^{(1)}) : String$	Cette fonction calcule le code identifiant ($\bar{code}$) de l'identification (BSU) du concept o_i.
$get\_ID(o_i \rightarrow id, O_{Int}) : o \rightarrow id$	Cette fonction permet de déterminer s'il existe un concept o ($\in O_{Int}$) dont le code identifiant est égal au code identifiant du concept o_i. Si la condition est satisfaite, la fonction rend l'identification du concept o. Au contraire, elle rend la valeur $null$.
$get\_SSCR(c_i \rightarrow def^{(2)}, C_{Int}) : C$	Cette fonction permet de calculer l'ensemble des classes C qui sont les plus petites classes subsumantes de la classe c_i dans l'ensemble de classes C_{Int} ($C \in C_{Int}$).
$integrate\_Concept\ (o_i \rightarrow id, O_{Int})$	Cet opérateur permet d'intégrer le concept o_i dans l'ontologie O_{Int}. Il consiste : 1. à fusionner les identifications de o_i et o, s'il existe un concept o ($\in O_{Int}$) dont le code d'identifiant est égal à celui du o_i, 2. au contraire, à intégrer tous les éléments décrivant o_i en assurant l'intégrité référentielle [(3)].
Opérateurs sur le contenu	
$insert\_ClassInstance\ (i, tab_{Int})$	Cet opérateur consiste à insérer l'instance de données i dans la table d'instances tab_{Int} (extension d'une classe du catalogue intégré). (voir la tableau 4.3)
$update\_Schema$ $(tab_i, tab_{Int}, Applic_{Int})$	Cet opérateur permet de mettre à jour le schéma de la table tab_{Int} en la complétant par les propriétés dont chacune satisfait les deux conditions : 1. elle appartient à l'ensemble des propriétés applicables $Applic_{Int}$ pour la classe de base de la table "tab_{Int}", 2. elle figurent dans tab_i, mais pas dans tab_{Int}.

Note :

(1) : $o \rightarrow id$ *: représente l'identification du concept o, i.e. l'entité bsu. Cette notation est utilisée car un concept c est représenté par deux entités distinctes qui font l'objet de traitements différents.*

(2) : $o \rightarrow def$ *: représente la définition du concept o (dictionary_element). Le dictionary_element (définition) du concept o contient toutes les références aux autres concepts (exactement à leurs identifications) auxquels il est lié, et les références aux entités fournissant la description du concept o, telles que : item_names, graphics, etc.*

(3) : *assurer l'intégrité référentielle d'une entité qui référence, directement ou indirectement, un ensemble d'entités signifie assurer que celles-ci sont présentées dans le contexte considéré.*

TAB. 4.1 – Les fonctions primitives d'intégration

$get\_SSCR(c_i \rightarrow def, C_{Int}) : C$
Cette fonction est utilisée pour identifier les classes les plus bases ($\in C_{Int}$) sur lesquelles les instances de la classe c_i peuvent être intégrées : – si $c_i \in C_{Int}$, alors c_i est la classe trouvée (cela correspond au cas d'intégration par $FragmentOnto$) – si non (correspondant au cas d'intégration par $ProjOnto$), elle est implémentée d'une façon récursive : 1. récupérer toutes classes $\in C_{Int}$, et qui sont référencées directement par c_i à travers $OntoSub$, et 2. appliquer la fonction get_SSCR sur la super-classe de c_i (si elle existe).

TAB. 4.2 – Algorithme pour implémenter get_SSCR

$insert\_ClassInstance\ (i, tab_{Int})$
Cette fonction consiste à insérer dans la table tab_{Int} une nouvelle instance : – si une propriété figure dans i et aussi dans tab_{Int}, la valeur de cette propriété pour la nouvelle instance est celle de i. – si une propriété ne figure que dans tab_{Int}, la valeur de cette propriété pour la nouvelle instance est "*null*".
new_ins := insert_new_ClassInstance(tab_{Int}) (1) ; Sch_{Int} := get_schema (tab_{Int}) (2) ; Sch_i := get_schema (i) ; **Repeat Foreach** ($p_i \in Sch_i$) ; – p := get_ID(p_i, Sch_{Int}) ; – **If Not** (p = *null*) **Then** – val_i := get_value(p_i, ins) (3) ; – update_value(new_ins, p, val_i) (4) ; – **End If** ; **End Repeat** ;
Note : **(1)** : *insert_new_ClassInstance(tab_{Int}) consiste à créer pour la tab_{Int} une nouvelle instance de donnée de tab_i dont la valeur de chaque propriété est null.* **(2)** : *get_Schema(tab_i) (ou get_Schema(i)) consiste à rendre toutes identifications des propriétés figurant dans la table "tab_i" (ou dans l'instance "i").* **(3)** : *get_value(p_i, i) rend la valeur de la propriété p_i dans l'instance i.* **(4)** : *update_value(new_ins, p, val_i) consiste à insérer la valeur val_i pour la propriété p de l'instance new_ins.*

TAB. 4.3 – Algorithme pour implémenter l'API $insert\_ClassInstance$

integration_projection(Cat_i, Cat_{Int})

Repeat Foreach $tab_i \in Cat_i$;
- $c_i \rightarrow def$:= get_BaseClasse(tab_i) (1) ;
- C_s := get_SSCR($c_i \rightarrow def$) ;
- **Repeat Foreach** $c \rightarrow def \in C_s$;
  - tab_{Int} := get_table($c \rightarrow def$) (2) ;
  - **If** ($tab_{Int} = null$) **Then**
    - $Applic_{Int}$:= $c \rightarrow def$.described_by ;
    - tab_{Int} :=init_table($c \rightarrow def, Cat_{Int}$) (3) ;
  - **End If** ;
  - **update_Schema**($tab_i, tab_{Int}, Applic_{Int}$) ;
  - **Repeat Foreach** ($i \in tab_i$) ;
    - **insert_ClassInstance**(i, tab_{Int}) ;
  - **End Repeat** ;
- **End Repeat** ;

End Repeat ;

(1) : *get_BaseClasse(tab_i) rend la classe de base de la table (ensemble des instances) tab_i. Dans la base de données ECCO, get_BaseClasse(tab_i) = tab_i* ***.dictionary_definition***.

(2) : *get_table($c \rightarrow def$) rend :*
- *la table (extension) dont la classe de base est c, si* ***Exists***($c \rightarrow id$.referenced_by) = ***TRUE***.
- *la valeur "null", si non.*

(3) : *init_table($c \rightarrow def, Cat_{Int}$) crée une table des instances (extension) pour la classe c dans le catalogue "Cat_{Int}". Cette table (extension) ne contient initialement aucune instance. Dans la base de données ECCO, il consiste à créer une entité* ***explicit_item_class_extension***.

TAB. 4.4 – Algorithme pour implémenter l'API *integration_projection*

integration_extension(Cat_i, Cat_{Int})

Repeat Foreach $class_i \rightarrow id \in Cat_i$;
- **integrate_Concept**($class_i, Cat_{Int}$) ;

End Repeat ;

Repeat Foreach $tab_i \in Cat_i$;
- import_Table(tab_i, Cat_{Int}) (1) ;

End Repeat ;

Note :

(1) : *import_Table(tab_i, Cat_{Int}) consiste à ajouter la table d'instances (extension) "tab_i" dans le catalogue intégré Cat_{Int}. Pour la base de données ECCO, elle est implémentée simplement comme suit :*
- ***Repeat Foreach*** *($e \in bunch(tab_i)$) ;*
  - *import_Entity(e,Cat_{Int}) ;*
- ***End Repeat*** *;*

TAB. 4.5 – Algorithme pour implémenter l'API *integration_extension*

4.7.3 Applications implémentées

Nous avons présenté dans la section ci-dessus un exemple des PLIB APIs d'Intégration. En exploitant les APIs, deux versions d'intégration de composants ont été implémentées :

1. la première version applique les deux scénarios d'intégration *FragmentOnto* et *ProjOnto* et permet l'extraction de composants pour les exporter vers un autre utilisateur. Note : l'exportation en *ExtendOnto* ne pose pas de difficulté particulière puisque cela revient à exporter entièrement le catalogue ce que notre système sait déjà faire.
2. la deuxième version applique les scénario d'intégration *ProjOnto* et *ExtendOnto* qui permet l'intégration automatique au sein d'un entrepôt des catalogues des différents fournisseurs. Note : l'intégration en *FragmentOnto* ne pose pas de difficulté particulière puisqu'il s'agit seulement d'ajouter des composants sans modification d'ontologie ni de schéma, ce que notre système supporte sans difficulté.

Première Version d'implémentation. Sur la figure 4.7, l'interface à gauche montre la partie d'extraction de données (concepts ontologiques et composants). A partir d'un catalogue sélectionné, nous pouvons choisir les classes intéressantes ou/et les composants intéressants associés aux classes choisies pour les extraire comme un nouveau catalogue en assurant :

1. la cohérence (du point de vue intégrité référentielle),
2. la pertinence (par rapport aux besoins utilisateurs)

Ce scénario peut être utile dans le cas où :

- un grand donneur d'ordre veut fragmenter son ontologie partagée pour ses sous-traitants ou ses fournisseurs qui vont ensuite utiliser les fragments leur convenant comme leurs ontologies locales,
- un fournisseur ne veut fournir qu'une partie de son catalogue à un client.

Dans l'interface à droite, nous pouvons exporter les composants sélectionnés d'un catalogue dans un autre catalogue qui contient l'ontologie référencée par le premier. Ces composants exportés sont archivés comme les instances de l'ontologie du nouveau catalogue. Tout processus d'exportation est exécuté automatiquement en appliquant les deux scénarios d'intégration : *FragmentOnto* et *ProjOnto*. Cette implémentation peut être utile dans le cas où :

- un fournisseur voudrait exporter ses propres composants comme les instances de l'ontologie normalisée référencée par son ontologie locale,
- un grand donneur d'ordres voudrait centraliser les catalogues des différents fournisseurs au sein de son catalogue unique.

Deuxième Version d'implémentation. Cette version permet de manipuler plusieurs catalogues de sources en même temps. La figure 4.8 montre l'interface qui permet de choisir le scénario d'intégration pour chaque catalogue à intégrer :

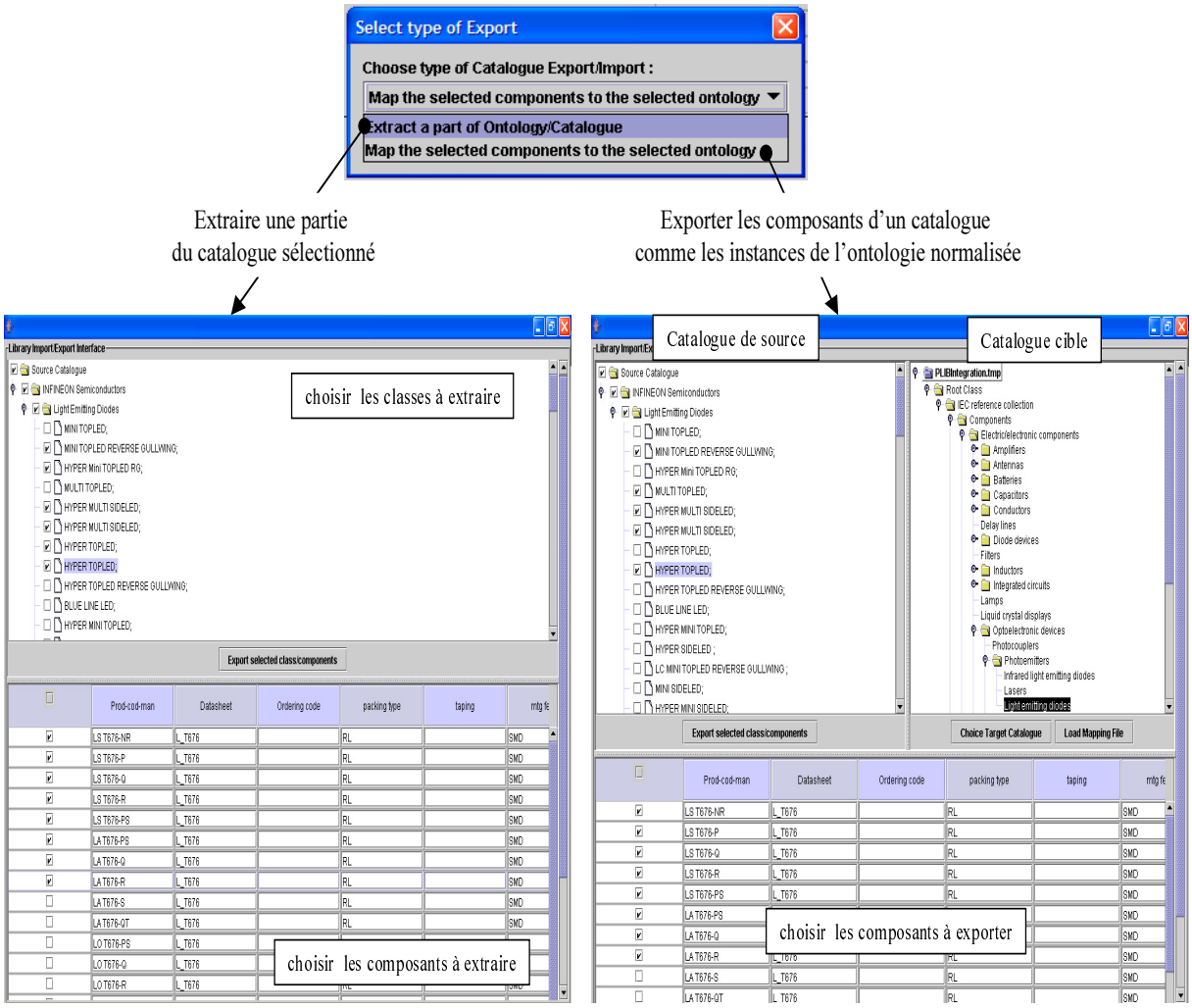

FIG. 4.7 – Interface graphique supportant l'échange de composants entre les catalogues

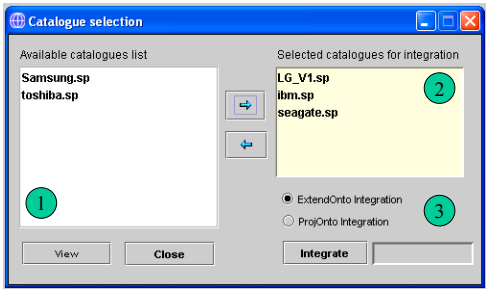

FIG. 4.8 – Interface graphique servant à choisir le scénario d'intégration pour les catalogues à intégrer

– la partie **(1)** présente la liste des catalogues disponibles dans la section courante de base de données ECCO et non encore intégrés dans le catalogue cible (ces catalogues sont chargés en même section d'ECCO à travers PLIBEditor),
– la partie **(2)** présente les catalogues choisis à intégrer selon le type d'intégration sélectionné (*ProjOnto* ou *ExtendOnto*) dans la partie **(3)**.

Cette mise en oeuvre peut être utile pour construire un entrepôt de composants dans un système d'e-commerce tel que celui dans l'exemple 8.

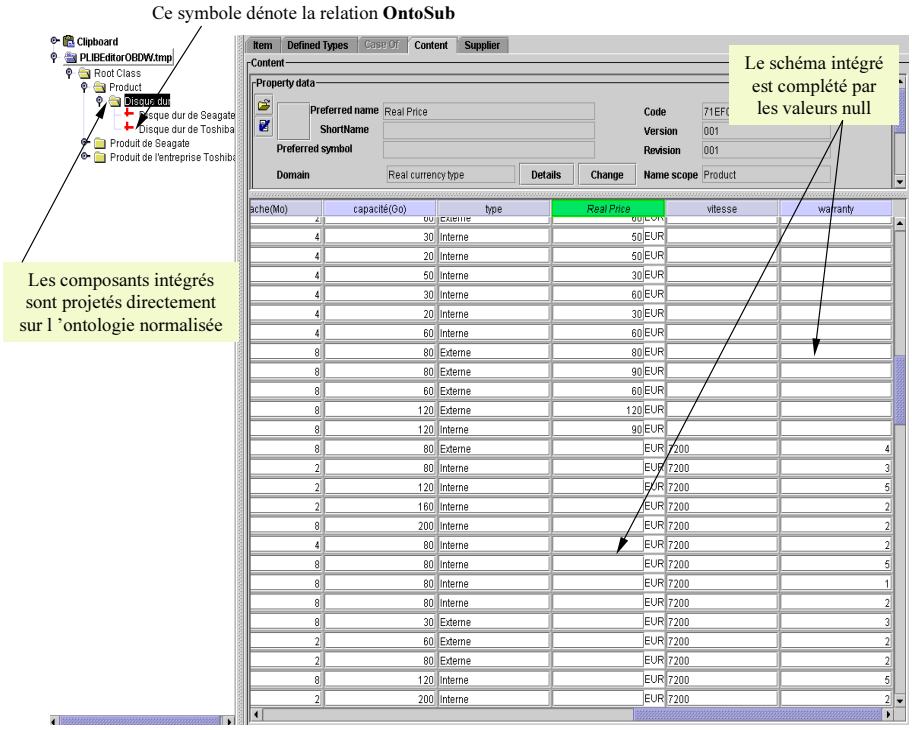

FIG. 4.9 – Un exemple d'intégration de catalogues par ProjOnto

La figure 4.9 présente un catalogue intégré par *ProjOnto*. Dans cet exemple, les disques durs des différents fournisseurs sont projetés sur l'ontologie partagée. Ils sont donc accessibles à travers la classe "*disque dur*" partagée. Cette figure ne montre pas l'origine de chaque instance intégrée (par exemple, son fournisseur, sa classe de base, etc.). Cet aspect du système intégré sera abordé dans le chapitre 5.

La figure 4.10 présente un catalogue intégré par *ExtendOnto*. Dans cet exemple, les ontologies locales, telles que l'ontologie d'IBM, celle de *LG*, celle de *Seagate*, sont également stockées dans le catalogue intégré. Ceci assure qu'aucune information ne soit perdue. Les composants intégrés sont conservés comme dans leurs sources d'origine. Par contre, ils peuvent être accessibles à travers l'ontologie partagée en parcourant les classes subsumées de la classe "*disque dur*" partagée.

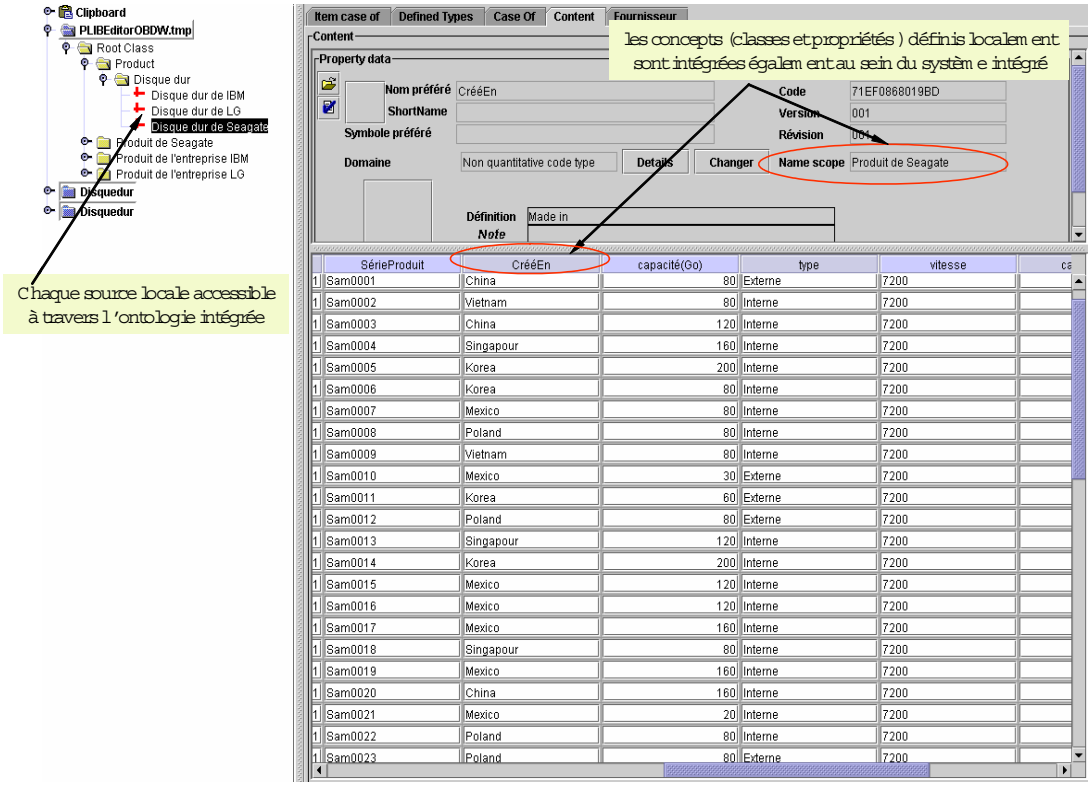

FIG. 4.10 – Un exemple d'intégration de catalogues par ExtendOnto

4.8 Conclusion

Dans ce chapitre, nous avons proposé une méthode complètement automatique d'intégration de sources de données structurées hétérogènes et autonomes. Cette approche, appelée intégration par articulation *a priori* d'ontologies, suppose l'existence d'une (ou plusieurs) ontologie(s) de domaine, mais elle laisse chaque source autonome quant à la structure de sa propre ontologie.

Au lieu de réaliser l'intégration des ontologies *a posteriori*, comme c'est le cas dans toutes les approches classiques, notre approche exige de l'administrateur de chaque source à intégrer que :

1. sa base de données contienne une ontologie, et
2. qu'il *s'engage sur l'ontologie de domaine*, c'est-à-dire qu'il ajoute *a priori* à cette ontologie les relations (articulations) existantes entre celle-ci et l'ontologie de domaine.

Cette hypothèse est réaliste dans tous les secteurs où des ontologies de domaines existent ou apparaissent, et où chaque administrateur qui publie sa base de données souhaite à la fois lui conserver sa structure propre, et la rendre accessible à des usagers de façon homogène à travers une ontologie de domaine. C'est en particulier le cas dans le cadre du commerce électronique professionnel. Elle exige également que chaque source publie

non seulement ses données, mais également son ontologie, ce qui correspond à une généralisation de l'approche de type meta-données utilisée pour les sources semi-structurées.

Nous avons présenté dans ce chapitre une architecture d'intégration sur l'ensemble de bases de données à base ontologique. Cette architecture définit plusieurs opérateurs d'intégration correspondant à différents scénarii possibles, à savoir $FragmentOnto$, $ProjOnto$ et $ExtendOnto$, qui dotent l'ensemble les BDBOs d'une structure d'algèbre.

L'ensemble de ces propositions a été validé sur des exemples issus du commerce électronique professionnel (B2B) et de l'intégration des catalogues de composants industriels (projet PLIB). Nous avons présenté ici la mise en oeuvre de cette approche dans une perspective d'intégration physique des données au sein d'un entrepôt. Nous utilisons des ontologies PLIB qui sont bien adaptés pour représenter les entités d'un domaine fortement structuré et les propriétés intrinsèques qui les caractérisent. Ce type d'ontologie semble bien adapté au domaine du commerce électronique professionnel. De plus, ce modèle d'ontologie, formellement défini dans le langage EXPRESS, est associé à une structure d'échange permettant de représenter de façon neutre tant des ontologies que des instances référençant ces ontologies. Les algorithmes que nous avons présentés permettent alors, à travers un tel échange, l'intégration automatique du contenu de toute nouvelle source autonome au sein d'un entrepôt déjà constitué à partir de l'ontologie de domaine. Cette intégration peut même étendre automatiquement, si on le souhaite, l'ontologie de domaine en respectant la compatibilité ascendante. Nous avons utilisé dans cette implémentation des ontologies PLIB qui permettent de typer finement les valeurs des propriétés des composants (unités explicites, représentation des dépendances entre propriétés) et de référencer une classe d'une autre ontologie sans importer entièrement cette dernière.

Chapitre 5

Gestion de l'évolution asynchrone d'un système d'intégration à base ontologique

5.1 Introduction

Dans de nombreux domaines comme les services Web, le commerce électronique ou la synchronisation des bases de données réparties, le nouveau défi est de permettre une intégration entièrement automatique de sources de données qui gardent, néanmoins, une autonomie significative tant au niveau de leurs schémas propres où les données doivent pouvoir présenter un certain dégré d'hétérogénéité, qu'au niveau de leurs évolutions qui doivent pouvoir se dérouler de façon asynchrone.

Nous avons déjà proposé une solution, dans le chapitre précédent, au problème de l'hétérogénéité des sources. C'est l'approche d'intégration sémantique par articulation *a priori* d'ontologies dans laquelle on exige que chaque classe locale référence sa plus petite classe subsumante dans l'ontologie partagée et en importe, sans les modifier, les propriétés utiles.

Mais les fournisseurs des sources de données étant différents, chaque source doit pouvoir, en plus, se comporter indépendamment des autres (tout particulièrement dans des environnements dynamiques comme le WWW [47]). En conséquence, la relation entre le système intégré et ses sources est faiblement couplée. Dans un tel contexte, il est difficile de synchroniser l'évolution de l'ontologie partagée et les évolutions des sources. Cela est dû aux facteurs suivants : (i) l'indépendance des sources et (ii) le temps nécessaire pour diffuser les évolutions des ontologies partagées dans une communauté. Les sources à intégrer peuvent donc ne pas référencer à une même version de l'ontologie partagée. La figure 5.1 montre un exemple de l'intégration de BDBOs dans un environnement asynchrone. Dans ce contexte, l'automatisation du processus d'intégration n'est pas assurée.

L'objet du chapitre est de présenter une approche permettant l'évolution asynchrone

de l'ensemble de sources, tout en maintenant la possibilité d'intégration automatique de ces dernières au sein d'un entrepôt. *Deux problèmes* doivent être résolus :

1. la gestion des évolutions des ontologies afin de maintenir la cohérence des relations entre ontologies et entre ontologies et données, et
2. la gestion de cycle de vie des instances, c'est-à-dire la représentation de la connaissance des instances qui existaient dans l'entrepôt à un instant donné.

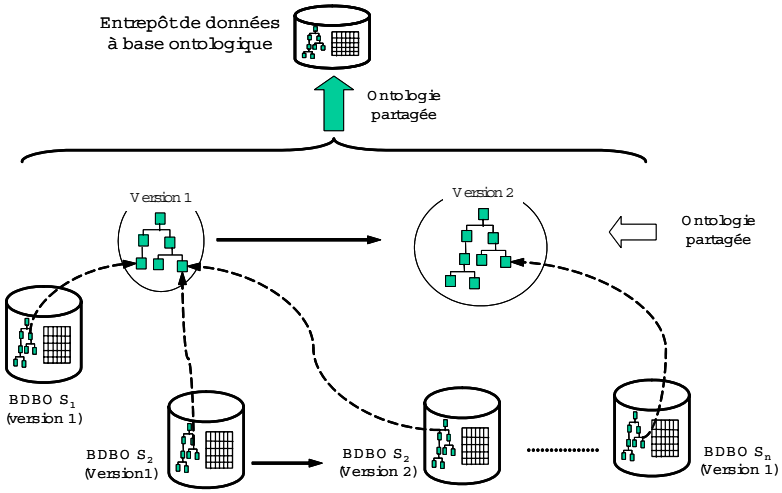

FIG. 5.1 – Exemple de l'intégration de bases de données à base ontologique dans un environnement asynchrone.

Le contenu de ce chapitre est le suivant. La partie qui suit introduit quelques travaux antérieurs concernant l'évolution des données et des ontologies. Cette partie permet de situer notre solution par rapport aux approches existantes. Dans la troisième partie, nous proposons un modèle pour la gestion de l'évolution asynchrone des ontologies. La section 4 présente une approche pour gérer le cycle de vie des instances à base ontologique. Avant de conclure ce chapitre dans la section 6, nous présentons les implémentations du prototype OntoDaWa (Ontology based Data Warehouse) validant les modèles de gestion proposés.

5.2 Travaux antérieurs sur les évolutions de données

Dans cette section, nous présentons brièvement les travaux antérieurs sur les problèmes d'évolution de données selon deux axes : (1) l'évolution de données et de schéma, et (2) l'évolution d'ontologies.

5.2.1 Évolution de données

Les changements que subissent au cours du temps les schémas de bases de données représentent un problème classique. Celui-ci a été abordé dans le contexte des bases de

données relationnelles et des bases orientées objet [89, 90, 95, 66, 108]. On peut distinguer deux approches de ces problèmes, à savoir, l'approche par *schéma versionné* et l'approche par *schéma évolutif*.

- Dans l'approche par *schéma versionné* [66, 108], toutes les versions de chaque table sont explicitement stockées (voir la Figure 5.6a). Cette solution a deux avantages principaux :(i) elle est facile à implémenter et permet une automatisation du processus de mise à jour, et (ii) elle offre un traitement des requêtes rapide dans le cas où l'on précise la ou les versions de recherche. Par contre, le coût peut devenir important si la requête nécessite un parcours de toutes les versions de données disponibles dans la base. Un autre inconvénient est le coût de stockage à cause de la duplication des données.
- Dans l'approche par *schéma évolutif* [89, 108], un seul schéma est représenté pour chaque table. Ce schéma est obtenu en faisant l'*union* de toutes les propriétés figurant dans les différentes versions. On y ajoute, à chaque rafraîchissement, toutes les instances existant dans la nouvelle version de la table. Les instances sont complétées par des valeurs nulles (voir Figure 5.6b). Cette solution évite la représentation de plusieurs versions de chaque table. Les inconvénients majeurs de cette solution sont : (i) le problème de duplication est toujours présent, (ii) l'implémentation est un peu plus difficile que l'approche précédente en ce qui concerne le calcul automatique du schéma des tables stockées ; (iii) le tracé du cycle de vie de données est difficile à mettre en oeuvre (*"valid time"* [108]) et (iv) l'ambiguïté sémantique des valeurs nulles : figuraient-elles dans la table insérée, ou résultent-elles de l'intégration ?.

Pour des entrepôts de données (data warehouse), la problématique d'évolution de données a également été abordée. Les travaux dans ce domaine visent à adapter les résultats obtenus précédemment aux caractéristiques particulières des entrepôts telles que la nature de données intégrées et leur caractère multidimensionnel. On peut identifier deux types de travaux portant respectivement sur l'évolution du schéma d'entrepôt [8, 20, 54], et sur la maintenance des vues d'entrepôt [92].

Les travaux sur les entrepôts WHIPS relèvent de la première problématique. Dans un entrepôt WHIPS [54], par exemple, les évolutions de données sont faites en ligne. En parallèle, l'entrepôt de données peut recevoir des changements issues des différentes sources. Si ces changements sont appliqués dans un ordre inadapté, cela peut produire des inconsistances de données dans l'entrepôt. WHIPS a proposé un algorithme de synchronisation (Strope [109]) permettant de calculer un ordre pertinent pour les changements.

Le système d'intégration EVE [92] relève du dernière type de travaux. Il vise à automatiser la maintenance des vues. Ce système propose une nouvelle extension de SQL (E-SQL) qui permet de définir *a priori* des évolutions acceptables pour une vue matérialisée. E-SQL ajoute des paramètres spécifiques à chaque vue. Ces paramètres déterminent *a priori* quelles informations (attributs/relations/conditions) sont indispensables, quelles informations sont remplaçables par des informations similaires provenant des sources, et

```
CREATE    VIEW Asia-Customer (Δ =⊇) AS
SELECT    Name (ε = false), Address (ε = false),
          PhoneNo (π = true, ε = true)
FROM      Customer C (θ = true), FlightRes F
WHERE     (C.Name = F.Passenger) (σ = false)
          AND (F.Destination = 'Asia') (σ = true)
```

View Evolution Parameter

Parameter		Symbol	Semantics	Default
Attribute-	dispensable (AD)	π	*true:* the attribute is dispensable *false:* the attribute is indispensable	false
	replaceable (AR)	ε	*true:* the attribute is replaceable *false:* the attribute is nonreplaceable	false
Condition-	dispensable (CD)	σ	*true:* the condition is dispensable *false:* the condition is indispensable	false
	replaceable (CR)	β	*true:* the condition is replaceable *false:* the condition is nonreplaceable	false
Relation-	dispensable (RD)	α	*true:* the relation is dispensable *false:* the relation is indispensable	false
	replaceable (RR)	θ	*true:* the relation is replaceable *false:* the relation is nonreplaceable	false
View-	extent (VE)	Δ	don't care: no restriction on the new extent $\equiv$: the new extent is equal to the old extent $\supseteq$: the new extent is a superset of the old extent $\subseteq$: the new extent is a subset of the old extent	$\equiv$

FIG. 5.2 – Un exemple de E-SQL du projet EVE [92]

quelles conditions doivent être respectées tout au long de l'évolution d'une vue (les paramètres de E-SQL sont présentés dans la figure 5.2). En se basant sur E-SQL, EVE permet de re-définir ses vues d'une façon automatique.

5.2.2 Évolution d'ontologies

Comme tout composant informatique, une ontologie évolue et nécessite donc une gestion de ses différentes versions. L'évolution d'une ontologie est normalement beaucoup plus problématique que les évolutions d'une base de données. Ceci est dû au caractère partageable de l'ontologie. Pour illustrer ce problème, nous citons Rogozan [91] qui pose la question suivante : *"comment préserver l'accès et l'interprétation des objets au moyen de leur référencement à une ontologie évolutive"*?

En effet, le changement d'une ontologie utilisée comme une interprétation sémantique peut produire des pertes des coordonnées sémantiques, c'est-à-dire des références d'objets établies par rapport à un certain référentiel sémantique. En conséquence, les objets utilisant ces références ne sont plus accessibles au moyen de l'ontologie modifiée [45, 72]. Pour cette raison, Noy et al. [72] considèrent que toutes les versions d'une même ontologie doivent être conservées. En conséquence, ils ont définit l'évolution de l'ontologie comme : "*la capacité de gérer les changements de l'ontologie et leurs effets sur les instances, en*

créant et en maintenant différentes versions d'une ontologie. Cette capacité consiste à différencier et à identifier les versions, à spécifier les relations qui explicitent les changements effectués entre versions et à utiliser des mécanismes d'accès pour les artefacts dépendants".

Ainsi, la plupart des approches s'intéressent surtout aux conséquences sur les individus de l'ontologie (que nous appellerons *instances à base ontologique*) que sur des évolutions de l'ontologie. Deux approches peuvent, néanmoins, être distinguées dans la littérature :

- La première approche [101, 102, 100] vise à supporter les processus d'évolution des ontologies. Elle étudie les différents opérateurs de modification d'ontologies (adjonction, suppression, modification de domaines) et les conséquences qui en résultent pour la cohérence des instances à base ontologique. Lorsque les modifications sont incompatibles avec l'état courant des instances, cette approche étudie comment modifier les instances pour les rendre cohérentes avec la nouvelle version de l'ontologie.
- La deuxième approche [53, 72, 74] ne vise pas à gérer le processus d'évolution des ontologies et les conséquences sur les instances existantes, mais plutôt à permettre la représentation et la comparaison des différentes versions d'une ontologie afin de faire migrer les instances à base ontologique de l'une vers l'autre. Cette approche analyse et représente la compatibilité entre les versions de l'ontologie.

Par exemple, le travail de Stojanovic [100] relevant de la première approche, propose un processus d'évolution d'une ontologie composée de six étapes principales :

1. *Capture du changement*. Le processus d'évolution de l'ontologie commence à capturer les changements à partir des exigences explicites ou du résultat des méthodes de découverte des changements qui induisent des changements à partir des données existantes.
2. *Représentation du changement*. Cette étape vise à représenter les changements dans un format approprié. Les changements peuvent être représentés à deux niveaux : (i) les changements élémentaires (l'adjonction, la suppression, la modification des entités ontologiques) et (ii) les changements complexes (par exemple, la fusion ou la séparation des entités ontologiques).
3. *Sémantique du changement*. L'ontologie doit évoluer d'un état consistant vers un autre état consistant [72]. Cette phase permet de résoudre des changements induits d'une manière systématique en assurant la consistance de toute l'ontologie. Afin de résoudre les inconsistances introduites par les changements, d'autres changements additionnels sont nécessaires.
4. *Propagation du changement*. Le but de cette étape est de modifier automatiquement les instances et les ontologies dépendantes (qui utilisent une partie de l'ontologie évolutive dans sa structure ontologique [91]) afin de préserver leur consistance avec l'ontologie évoluée.
5. *Implémentation du changement*. Cette phase consiste à (i) informer l'administrateur de toutes les conséquences d'un changement, (ii) à exécuter le changement, une fois

approuvé par l'administrateur.

6. *Validation du changement.* Cette phase permet la justification des changements exécutés et de les annuler suite à la demande de l'utilisateur.

Par ailleurs, dans OWL [62], les ontologies, les classes, et les propriétés peuvent être annotées pour tracer l'existence d'évolutions ontologiques. Elles peuvent toutes être associées à un numéro de version ($versionInfo$). De plus, trois étiquettes peuvent être utilisées pour représenter la compatibilité entre les différentes versions d'une ontologie : *priorVersion*, *backwardCompatibleWith* et *incompatibleWith* [46, 47]. Ceci permet, dans ce cas où la compatibilité est déclarée (*backwardCompatibleWith*), de savoir qu'il n'y a pas lieu de modifier les instances.

5.2.3 Synthèse d'intégration et évolution

Cet état de l'art montre que la prise en compte de l'évolution des ontologies dans les systèmes d'intégration à base ontologique est un problème crucial et que personne, tout au moins à notre connaissance, n'a encore proposé de méthode permettant simultanément l'intégration automatique de sources de données hétérogènes et la prise en compte de l'évolution asynchrone tant des ontologies que des données. Les conditions de faisabilité d'une telle approche, et la description de sa mise en oeuvre au sein d'un système d'intégration de type entrepôt constituent précisément l'objectif de la suite du chapitre.

Nous proposons dans la section suivante une approche permettant à l'ensemble de sources de données d'évoluer de façon asynchrone tout en maintenant la possibilité d'intégration automatique.

5.3 Gestion des évolutions des ontologies

Permettre à des ontologies que l'on souhaite intégrer d'évoluer au cours du temps nécessite sans aucun doute qu'une certaine cohérence soit maintenue tout au long de ces évolutions pour assurer une notion de "compatibilité" entre ontologies [47]. Nous définissons d'abord dans cette section les contraintes que nous proposons d'introduire pour limiter les évolutions autorisées et qui correspondent à l'étiquette *backwardCompatibleWith* de OWL. Nous présentons ensuite le modèle de gestion que nous proposons pour supporter de telles évolutions même lorsqu'elles sont réalisées de façon asynchrone.

5.3.1 Principe de continuité ontologique

Les contraintes que l'on peut définir pour régler l'évolution des entrepôts de données à base ontologique résultent des différences fondamentales existantes, du point de vue évolution, entre les modèles conceptuels et les ontologies. Un modèle conceptuel est un modèle, c'est-à-dire, selon Minsky [68], un objet qui permet de répondre à des questions

sur un autre objet, à savoir définir les informations représentées dans une base de données. Lorsque les questions changent [18], son modèle conceptuel est modifié en conséquence, et ceci, sans que cela signifie le moins du monde que le domaine modélisé a été modifié. Au contraire, une ontologie est une conceptualisation visant à représenter l'essence des entités d'un domaine donné sous forme consensuelle pour une communauté. C'est une théorie logique d'une partie du monde, partagée par toute une communauté, et qui permet aux membres de celle-ci de se comprendre. Ce peut être, par exemple, la théorie des ensembles (pour les mathématiciens), la mécanique rationnelle (pour les mécaniciens) ou la comptabilité analytique (pour les comptables). Pour de telles ontologies, deux types de changements doivent être distingués : (1) *l'évolution normale* d'une théorie et son approfondissement. Des vérités nouvelles, plus détaillées s'ajoutent aux vérités anciennes. Ce qui était vrai hier reste vrai aujourd'hui, (2) *révolution* : il peut également arriver que des axiomes de la théorie aient à être remis en cause. Dans ce cas, il ne s'agit plus d'une évolution mais d'une révolution, où deux systèmes logiques différents vont coexister ou s'opposer.

Les ontologies que nous visons correspondent à cette conception. Il s'agit d'ontologies soit normalisées, par exemple au niveau international (ISO 13584-511), soit définies par des consortiums importants, et qui formalisent de façon stable les connaissances d'un domaine technique. Les changements auxquels nous nous intéressons dans notre approche ne sont donc pas des révolutions, qui correspondent à un changement d'ontologie, mais les évolutions d'ontologie.

Nous imposerons donc aux ontologies manipulées, qu'elles soient globales ou locales de respecter la contrainte suivante dont nous détaillons les conséquences dans la section 4.2.

Principe de continuité ontologique : *si l'on considère chaque ontologie intervenant dans le système d'intégration à base ontologique comme un ensemble d'axiomes, tout axiome vrai pour une certaine version de l'ontologie restera vrai pour toutes les versions ultérieures.*

5.3.2 Contraintes sur les évolutions des ontologies

Nous détaillons ici les conséquences du principe de continuité ontologique.

5.3.2.1 Identification des classes et propriétés

Gérer l'évolution suppose de pouvoir désigner, et donc identifier, tous les éléments faisant l'objet d'évolution.

Nous avons déjà précisé que toute source, toute classe et toute propriété étaient associées à des identifiants universels (ID : universal identifier). En fait, dans un contexte

[18] les objectifs organisationnels auxquels répond un système d'information sont modifiés

multi-versions, ces identifiants doivent contenir deux parties : un code (unique) identifiant les concepts et une version (entière) identifiant les versions d'un concept :

$$ID ::= code\ version$$

Toute référence entre éléments utilisant le ID, la référence est elle-même versionnée par les versions de ses extrémités. Enfin, toute définition de classe ou de propriété contient, en particulier, la date à partir de laquelle cette version est valide.

Notons que cette caractéristique est systématiquement présentée dans les ontologies PLIB que nous manipulons [81]. Elle est également possible en OWL [62] à travers l'étiquette $versionInfo$.

Notons maintenant par un indice supérieur la version des différentes composantes d'une ontologie : $O^k =< C^k, P^k, Sub^k, Applic^k >$.

5.3.2.2 Permanence des classes

L'existence d'une classe ne pourra être infirmée à une étape ultérieure :

$$\forall k, C^k \subset C^{k+1}.$$

Pour tenir compte de la réalité, il pourra apparaître pertinent de considérer comme obsolète telle ou telle classe. Elle sera alors marquée en tant que telle ("deprecated"), mais elle continuera à faire partie des versions ultérieures de l'ontologie. Par ailleurs la définition d'une classe pourra être affinée sans que l'appartenance à cette classe d'une instance antérieure ne puisse être remise en cause. Cela signifie que : (i) la définition des classes pourra elle-même évoluer, (ii) chaque définition d'une classe sera associée à un numéro de version, et (iii) la définition (intensionnelle) de chaque classe englobera les définitions (intentionnelles) de ses versions antérieures.

5.3.2.3 Permanence des propriétés

De même :

$$\forall k, P^k \subset P^{k+1}.$$

Une propriété pourra, de même, devenir obsolète sans que la valeur existante d'une propriété pour une instance existante puisse être remise en cause. Une propriété pourra évoluer dans sa définition ou dans son domaine de valeurs, par contre le principe de continuité ontologique implique que les *domaines de valeurs* ne puissent être que *croissants*, certaines valeurs étant, éventuellement, marquées comme obsolètes.

5.3.2.4 Permanence de la subsomption

La subsomption est également un concept ontologique qui ne pourra être infirmé. Notons $Sub^* : C \rightarrow 2^C$ la fermeture transitive de la relation de subsomption directe Sub. On a alors :

$$\forall c \in C^k, Sub^{*k}(c) \subset Sub^{*k+1}(c).$$

Cette contrainte permet évidemment un enrichissement de la hiérarchie de subsomption des classes, par exemple en intercalant des classes intermédiaires entre deux classes liées par une relation de subsomption.

5.3.2.5 Description des instances

Le fait qu'une propriété $p \in Applic(c)$ signifie que la propriété est rigide [41] pour toute instance de c. Il s'agit encore d'un axiome qui ne pourra être infirmé :

$\forall c \in C^k, Applic^k(c) \subset Applic^{k+1}(c)$

Soulignons que ceci ne suppose pas que les mêmes propriétés soient toujours utilisées pour décrire les instances d'une même classe. Il ne s'agit plus là en effet d'une caractéristique de nature ontologique, mais seulement de nature schématique.

5.3.2.6 Cycle de vie des instances

Dans une ontologie PLIB, l'extension d'une classe (c'est-à-dire l'ensemble d'instances qui lui appartient à un moment donné) est également associé à un autre numéro de version, c'est la version de l'extension.

Nous supposons que le cycle de vie des instances est définie par ce graphe de la figure 5.3.

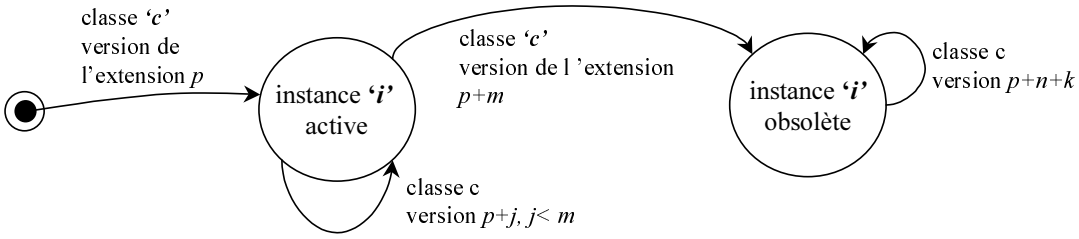

FIG. 5.3 – Cycle de vie d'une instance

Le cycle de vie d'une instance (apparition et obsolescence) peut donc être défini par les versions de l'extension de la classe de base auxquelles elle appartient.

5.3.2.7 Bilan

Les deux tables ci-dessous (5.1, 5.2) résument les contraintes d'évolutions ontologiques que nous proposons pour assurer le principe de continuité ontologique :

- **V** signifie que l'opération sur l'attribut affecte (augmente) la *Version* du concept,
- **X** signifie que l'opération n'est pas autorisée sur l'attribut du concept, et
- **-** signifie que l'opération sur cet attribut n'affecte pas la version.

Attribut	Ajout	Modification	Suppression
ID(Code)	X	X	X
Classe auquelle elle est définie	X	X	X
Type de données	X	-(étendu)/X(restreint)	X
Description (Name, Note, Icon,...)	-	-	X

TAB. 5.1 – Évolution de propriété en PLIB

Attribut	Ajout	Modification	Suppression
ID(Code)	X	X	X
Classe subsumante	X	X	X
Description (Name, Note, Icon,...)	-	-	X
Propriétés applicables (*Applic*)	V	V(ajout)/X(suppression)	X

TAB. 5.2 – Évolution de classe en PLIB

5.3.3 Modèle de gestion des évolutions

Les objectifs de ce modèle de gestion sont :

1. de permettre d'intégrer automatiquement dans l'entrepôt des sources qui sont elles-mêmes dans des versions différentes et qui référencent des *versions différentes de l'ontologie partagée*,
2. de permettre *l'accès uniforme à l'ensemble des instances* existant à un instant donné dans l'entrepôt via l'ontologie de l'entrepôt,
3. de connaître *l'historique des instances*, et
4. éventuellement, de savoir, pour chaque instance, *à quelle version de l'ontologie elle correspond*.

On décrit ci-dessous, d'abord l'hypothèse portant sur la méthode de rafraîchissement de l'entrepôt, ensuite comment les deux premiers objectifs peuvent être atteints par le mécanisme des versions flottantes. L'analyse des deux derniers objectifs sera menée dans la section 5.4.

5.3.3.1 Réalisation des mises à jour

Nous supposons que notre entrepôt de données est rafraîchi de la façon suivante. A des moments donnés, choisis par l'administrateur de l'entrepôt, la version courante d'une

source S_i est intégrée dans l'entrepôt. Cette version courante de S_i comporte son ontologie, ses références à l'ontologie partagée, et son extension. Notons que, dans cette extension, certaines instances pouvaient déjà exister dans l'entrepôt, d'autres peuvent être nouvelles, d'autres enfin peuvent avoir été supprimées (i.e., être devenues obsolètes).

Ce scénario correspond, par exemple, dans le domaine de l'ingénierie, à un entrepôt qui consolide les descriptions de composants d'un ensemble de fournisseurs. Un rafraîchissement est effectué chaque fois qu'une nouvelle version d'un catalogue électronique d'un fournisseur est reçue.

5.3.3.2 Accès uniforme aux instances courantes : Modèle des versions flottantes

On appelle *instances courantes* de l'entrepôt les instances résultant du plus récent rafraîchissement de chacune des sources. La principale difficulté qui résulte de l'autonomie de chaque source est que, lors de deux rafraîchissements simultanés par deux sources différentes, la même classe de l'ontologie partagée c peut être référencée par une articulation de subsomption dans des versions différentes. Par exemple les versions c^k et c^{k+j} peuvent être référencées, au même moment, par deux classes c_i^n et c_j^p.

Il convient de noter que, compte tenu du principe de continuité ontologique :

1. toutes les propriétés applicables à c^k sont également applicables à c^{k+j}, et
2. toutes les classes subsumées par c^k sont également subsumées par c^{k+j}.

Donc la relation de subsomption entre c^k et c_i^n entraîne l'existence d'une relation de subsomption entre c^{k+j} et c_i^n . La classe c^k n'est donc pas nécessaire pour accéder aux instances de c_i^n.

Cette remarque nous amène à proposer un modèle appelé, *modèle des versions flottantes*, qui nous permet d'accéder à toutes les instances courantes de l'entrepôt via une seule version de l'ontologie de l'entrepôt. Cette version, appelée "*version courante*" de l'ontologie de l'entrepôt, est telle que la version courante de chacune de ses classes c^f est supérieure ou égale à la plus grande version de cette même classe qui ait été référencée par une articulation de subsomption lors d'un quelconque rafraîchissement.

Pratiquement, cette condition est assurée de la façon suivante :

- si une articulation $\mathcal{A}$ définit une relation subsomption dans une classe c^f dans une version inférieure à f, alors la relation de subsomption $OntoSub_{i,p}^{-1}$, (voir section 3.2) est modifiée pour référencer c^f,
- si une articulation $\mathcal{A}$ définit une relation subsomption avec une classe c^f avec une version supérieure à f, alors l'entrepôt télécharge la dernière version de l'ontologie partagée et fait migrer toutes les références $\mathcal{A}_{i,n}$ (i = 1..nombre des sources) vers les nouvelles versions courantes de l'ontologie partagée.

Exemple 12 *: (cet exemple est illustré dans la Figure 5.4). Lors d'un rafraîchissement, une classe C_1^2 est déclarée subsumée par une classe partagée C^2 par l'articulation $\mathcal{A}_{1,p}$*

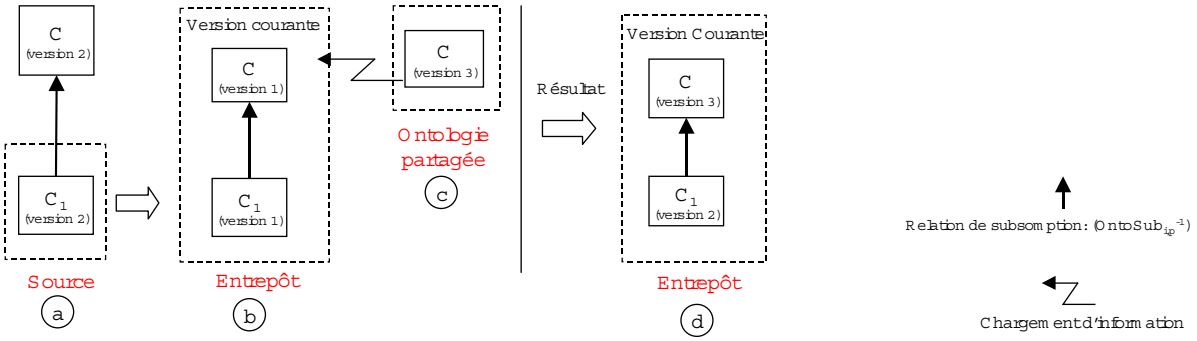

FIG. 5.4 – Exemple de comportement du modèle des versions flottantes

(a). Mais la version courante de C dans l'entrepôt est 1 (b). Alors l'entrepôt télécharge la version courante de l'ontologie partagée (c). Celle-ci étant 3, l'articulation $\mathcal{A}_{1,p}$ est modifiée pour définir la classe C_1^2 comme subsumée par C^3 (d).

Le modèle des versions flottantes permet donc bien d'accéder à toutes les instances de l'entrepôt quelles que soient les sources dont elles proviennent à l'aide de la seule version courante de l'ontologie de l'entrepôt. La section suivante présente le modèle de gestion du cycle de vie des instances.

5.4 Gestion de l'évolution des instances

Nous présentons dans cette section nos propositions pour gérer l'évolution des instances dans l'entrepôt. Nous présentons d'abord un mécanisme, la *clé sémantique*, qui permet de reconnaître une instance même si sa représentation change. Nous proposons ensuite une approche pour tracer le cycle de vie des instances.

5.4.1 Identification des instances

En règle générale, la durée de vie d'une instance peut être largement supérieure au cycle de mise à jour des sources de données. C'est en particulier le cas pour les instances des catalogues de composants industriels [9], dont certaines durent de nombreuses années alors que les catalogues sont mis à jour sur une base annuelle.

Afin de pouvoir identifier d'une façon non ambiguë une instance, toute source doit définir pour la population de chacune de ses classes de base une *clé sémantique*. Cette clé consiste à une ou plusieurs propriétés applicables de la classe dont les valeurs fournies pour les instances obéissent à une *contrainte d'unicité*. Ces valeurs devront toujours exister et être non "*null*" pour chaque instance, et elles ne devront jamais être modifiées d'une version à l'autre pour une même instance. Les clés sémantiques permettent donc non seulement de distinguer les instances d'une même version de l'extension d'une classe, mais également d'identifier la même instance dans différentes versions de cette extension.

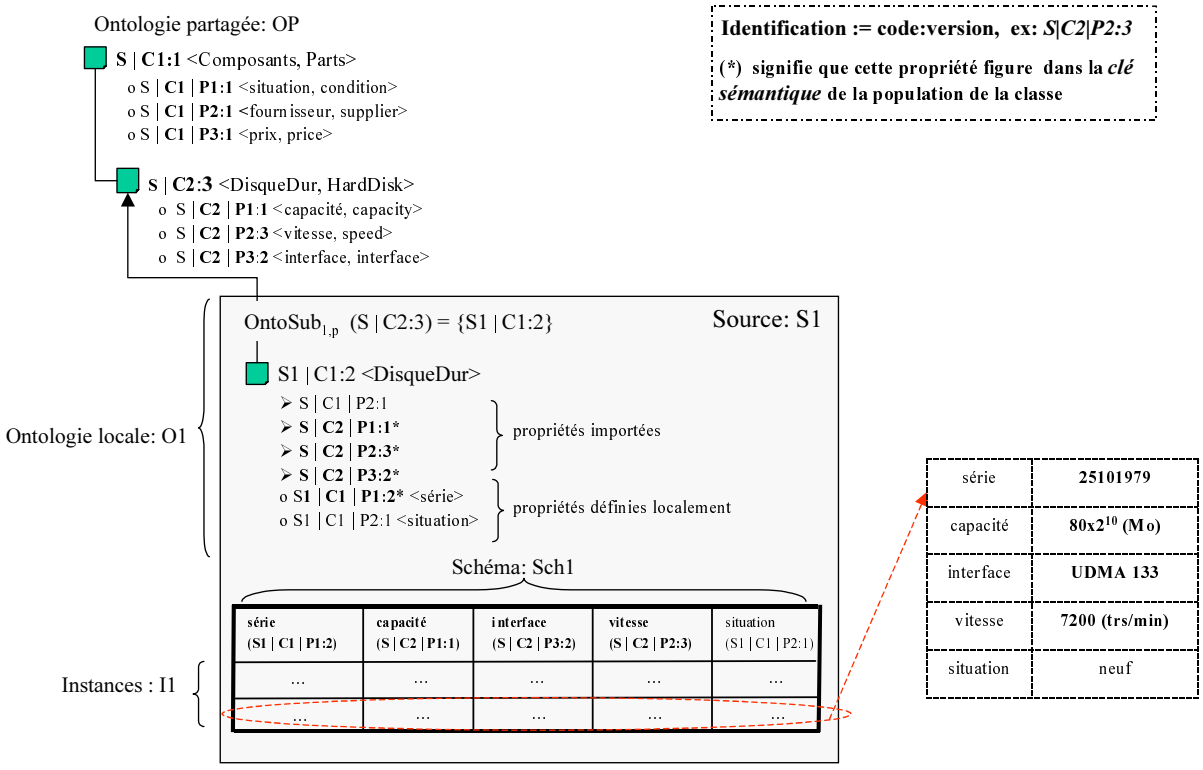

FIG. 5.5 – Un exemple de clé sémantique d'une classe ayant une population

L'exemple de la figure 5.5 montre la clé sémantique pour l'extension de la classe $S1|C1_{(DisqueDur)}$. Cette clé sémantique est formée des 4 propriétés : {*capacité*, *vitesse*, *interface*, *série*}. Pour l'instance "*i*" de *DisqueDur* dont la série est "25101979", la valeur de sa clé sémantique est : <80.2^{10}, 7200, UDMA 133, "25101979">. Notons que cette valeur est toujours la même dans n'importe quelle version de la classe *DisqueDur* contenant l'instance *i*. Si elle est changée, nous considérons l'instance avec la nouvelle valeur comme une nouvelle instance. Par contre, la valeur d'une propriété qui ne figure pas dans la *clé sémantique* peut apparaître ou ne pas apparaître d'une version à l'autre de l'instance "*i*". Par exemple, dans une nouvelle version, la valeur de la propriété *situation* peut être fournie alors qu'elle n'était pas fournie auparavant.

5.4.2 Gestion du cycle de vie des instances

On peut également souhaiter représenter dans un entrepôt l'historique des instances intégrées.

Dans les bases de données usuelles, ce problème a été étudié sous le nom de *version de schéma*(voir la section 2.1). Dans ce contexte, deux solutions ont été proposées : (1)l'approche par "*schéma versionné*" et (2)l'approche par "*schéma évolutif*" (voir la figure 5.6). Chaque approche possède des avantages et des inconvénients (voir la section 5.2.1). Un des inconvénients commun aux deux approches est le fait de dupliquer une instance si elle

apparaît successivement dans plusieurs versions du même schéma. Ceci provient du fait que, en l'absence d'ontologie et de clé sémantique, il n'est pas possible de reconnaître une instance si son schéma de représentation a été modifié.

Notre approche va tirer profit de l'existence de ces deux éléments pour mettre en oeuvre de façon originale l'approche *schéma évolutif*. Dans l'approche que nous proposons, toutes les instances des différentes versions d'une même classe de base seront représentées dans une unique table, mais de plus :

- Deux colonnes, appelées $version\_min$ et $version\_max$, permettent de définir entre quelles versions de l'extension de sa classe, une instance a été active. Ces colonnes nous permettent de savoir :
  1. la première version de l'extension de sa classe ($version\_min$), pour laquelle une instance est apparue, et
  2. la première version de l'extension de sa classe ($version\_max$), pour laquelle une instance a disparu.

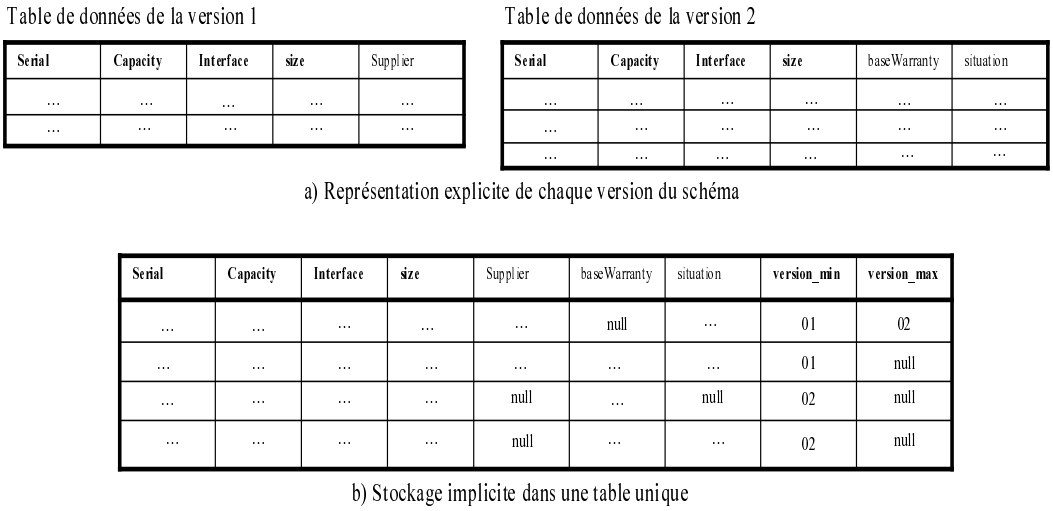

FIG. 5.6 – Le stockage par schéma versionnée et schéma évolutif

- Chaque instance n'est représentée qu'une seule fois en étant reconnue à chaque version par sa clé sémantique et par l'identification de sa classe de base. Retournons à l'exemple dans la figure 5.5, l'instance "i" est identifiée par :
  1. sa valeur de la clé sémantique : <80.2^{10}, 7200, UDMA 133, "25101979">,
  2. l'identification de sa classe de base : $S1|C1$.

  Si cette instance est définie par certaines propriétés différentes dans les différentes versions, toutes ses valeurs de propriétés sont accumulées dans sa description. Notons qu'il ne peut y avoir de conflit de valeurs pour une même propriété d'une même instance car nous ne nous intéressons dans le cadre de nos applications qu'aux

propriétés rigides [41], c'est à dire celles dont la valeur ne peut changer sans changer l'instance. Pour les propriétés dont la valeur dépend également du contexte, cette dépendance est représentée au niveau de l'ontologie ce qui évite toute ambiguïté [81].
– Le schéma (Sch) de chacune des versions d'extension est conservé dans l'entrepôt de façon à savoir, pour chaque version de l'extension, quelles propriétés étaient fournies. Ainsi, la sémantique de la valeur *null* est explicitée.

Cette approche élimine tous les inconvénients identifiés pour l'approche *schéma évolutif*, à savoir (1) la duplication de données, (2) l'absence de la représentation du cycle de vie des instances et (3) l'ambiguïté de la sémantique de la valeur *"null"*. Une instance implémentée selon la solution ci-dessus est dite *instance multi-versionnée*.

Dans la section qui suit, nous allons présenter notre implémentation validant nos propositions.

5.5 Mise en oeuvre de notre modèle

Nous présentons ci-dessous deux implémentations que nous avons réalisées dans notre modèle de gestion d'évolution, à savoir, (1) *entrepôt avec versionnement des instances mais sans historisation ontologique* et (2) *entrepôt avec versionnement et historisation ontologique*. Ces deux implémentations sont en fait disponibles au choix de l'utilisateur, au sein d'un même entrepôt de données, nommé OntoDaWa.

5.5.1 Entrepôt avec versionnement des instances mais sans historisation ontologique

Il peut être utile, dans certains cas, de pouvoir gérer les évolutions asynchrones et de savoir quelles instances existaient dans l'entrepôt à tout instant passé sans qu'il apparaisse nécessaire d'archiver toutes les versions successives d'ontologies. En effet, la version courante de l'ontologie est compatible avec toutes les instances passées, elle suffit pour interpréter le contenu.

Un tel entrepôt comporte deux parties, illustrées sur la partie à droite de la figure 5.8 : zones (1) et (2) :

1. la partie d'*Ontologie*, contient la version courante de l'ontologie intégrée.
2. la partie de *Contenu*, est composée des *tables multi-versionnées* dont chaque instance est une *instance multi-versionnée*.

Soulignons que, seules sont représentées les versions courantes des diverses classes de l'ontologie, un tel entrepôt peut néanmoins être rafraîchi de façon asynchrone par diverses sources qui référencent des versions différentes de l'ontologie partagée.

5.5.1.1 Gestion des versions de la partie d'ontologie

Nous résumons ci-dessus le processus pour créer et mettre à jour la version courante d'un concept (c_i) de l'entrepôt. Il consiste à intégrer le concept c_i en respectant le modèle des versions flottantes. Avant de décrire ce processus, notons que :

- $c_i^{current}$ représente la version courante du concept c_i dans l'entrepôt, si elle existe.
- c_i^{new} représente la nouvelle version à intégrer du concept c_i.
- $O_{Int}^{current} \rightarrow ID$ représente l'ensemble des identifications ($BSUs$) de la version courante de l'ontologie intégrée.

L'intégration du concept c_i dans l'entrepôt se compose de trois étapes :

1. ***vérifier l'existence*** du concept c_i dans l'entrepôt. Il s'agit de trouver un concept dans l'entrepôt qui lui équivaut. Grâce au mécanisme de BSU et la particularité de l'entrepôt, nous ne comparons que BSU du c_i^{new} avec les $BSUs$ dans $O_{Int}^{current} \rightarrow ID$ sans tenir compte de leurs versions.
2. ***créer*** du concept $c_i^{current}$ dans l'entrepôt. Il s'agit d'ajouter le concept c_i, s'il n'existe pas dans l'entrepôt. Cette étape consiste à :
   (a) ajouter le concept c_i^{new} dans l'entrepôt, et
   (b) mettre à jour c_i^{new} pour qu'il référence les autres concepts de la version courante.
3. ***mettre à jour*** du concept $c_i^{current}$. Il s'agit de mettre à jour la version courante du concept c_i, s'il existe déjà dans l'entrepôt :
   (a) **si** ($c_i^{current}$.version) > (c_i^{new}.version) ou ($c_i^{current}$.version) = (c_i^{new}.version) **alors** : on ne fait rien.
   (b) **si** ($c_i^{current}$.version) < (c_i^{new}.version) **alors** remplacer ($c_i^{current}$ par c_i^{new}) :
      i. supprimer $c_i^{current}$ de l'entrepôt,
      ii. intégrer c_i^{new} dans la place de $c_i^{current}$.

5.5.1.2 Gestion des versions de la partie contenu

Concernant l'implémentation des tables *multi-versionnées*, nous associons chaque table à un ensemble de propriétés qui permet de représenter explicitement à la fois l'origine et aussi le cycle de vie des instances dans cette table. Ainsi, une classe racine de toute classe ontologique de l'entrepôt a été implémentée (voir la figure 5.7).

La classe racine est caractérisée par les propriétés suivantes :

1. "*SupplierCode*" permettant de savoir le code du fournisseur de chaque instance intégrée i.
2. "*ClassCode*" permettant de savoir le code de la classe de base (d'origine) de l'instance i.
3. "*InstanceCode*" permettant de savoir la clé sémantique de l'instance i. La valeur de la propriété *InstanceCode* est constituée, de la représentation sous la forme

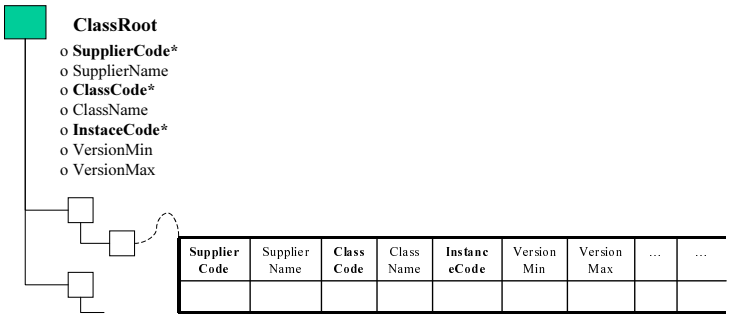

FIG. 5.7 – La racine de toute classe ontologique se présentant dans un entrepôt de données à base ontologique

d'une chaîne de caractères, la clé sémantique d'origine de l'instance i. Reprenons l'exemple 5.5, la valeur de la propriété "*InstanceCode*" créée pour l'instance "i" est : "$80.2^{10}|7200|UDMA133|25101979$".

Remarquons que dans un entrepôt, le schéma intégré des instances n'est pas leur schéma d'origine. Il nécessaire donc de choisir, pour chaque schéma intégré, une clé sémantique pouvant identifier d'une façon globale les instances de ce schéma intégré. Dans notre implémentation, les trois premières propriétés ci-dessus permettent non seulement de tracer l'origine des instances intégrées, mais également de formuler la clé sémantique de toute classe de base de l'entrepôt.

4. "*VersionMin*" qui indique la première version de la population d'origine, pour laquelle i est valide.
5. "*VersionMax*" représentant la valeur de la première version de la population d'origine, pour laquelle i n'est plus valide.
6. "*SupplierName*" qui représente le nom du fournisseur de i.
7. "*ClassName*" qui représente le nom de la classe de base d'origine de i.

Notons les deux dernières propriétés offrent simplement la possibilité d'accéder rapidement le nom de la source et le nom de la classe de base d'origine des instances intégrées.

5.5.2 Entrepôt avec versionnement des instances et historisation ontologique

Il est clair que le mécanisme des versions flottantes fait que l'articulation stockée dans la version courante de l'ontologie d'entrepôt entre une ontologie locale et l'ontologie partagée peut ne pas être sa définition originale (voir la Figure 5.4).

Dans le cas où il apparaît nécessaire de pouvoir accéder à une instance à travers les définitions ontologiques qui existaient lorsque cette instance était elle-même active, il est nécessaire d'archiver également toutes les versions successives de l'ontologie de l'entrepôt.

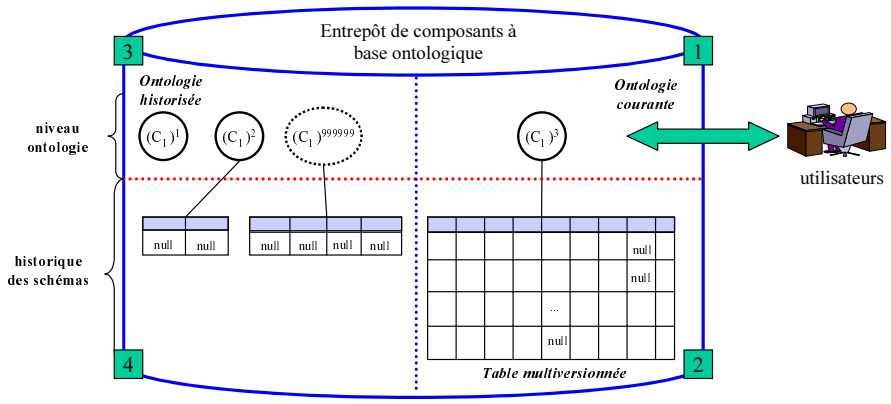

FIG. 5.8 – Structure d'un entrepôt de composants à base ontologique complètement historisé

Cela est utile, pour connaître ce qu'était, à l'époque de l'instance, le domaine précis d'une de ses propriétés énumérées (dont le domaine peut, ensuite, avoir été étendu).

Nous avons également implémenté cette possibilité en offrant d'archiver, dans l'entrepôt OntoDaWa, les définitions de toutes les versions de classes ayant existées dans la vie de l'entrepôt, ainsi que toutes les relations sous leur forme originale.

Cette structure est constituée de quatre parties (voir la figure 5.8) :

1. L'ontologie courante : elle contient la version flottante de l'ontologie de l'entrepôt. Elle constitue également l'interface générique d'accès aux données.

2. Les tables multiversionnées contiennent toutes les instances intégrées ainsi que leurs cycles de vie.

   Les deux premières parties sont les deux parties que nous avons présenté dans l'architecture précédente. Lorsqu'on veut historiser complètement les instances et les ontologies intégrées, deux nouvelles parties sont ajoutées.

3. L'archivage des ontologies : qui contient toutes les descriptions ontologiques des différentes versions de chaque classe et de chaque propriété de l'ontologie de l'entrepôt. Cette partie fournit aux utilisateurs les vraies définitions des versions de chaque concept si cela leur est nécessaire.

4. L'archivage des schémas : Les versions du schéma de chaque ensemble des instances I_i sont également historisées en archivant la fonction $Sch^k(c_i)$ de chaque version k de c_i où c_i est la classe de base des I_i. Dans notre implémentation, cette historisation est faite en conservant une instance unique dont toutes les valeurs seront à "*nulle*" (Instance dite "formelle") (voir la figure 5.11).

Notons que le principe de la continuité ontologique semble rendre rarement nécessaire ces deux derniers archivages qui sont à la fois complexes et coûteux.

5.5.3 Architecture de OntoDaWa

Les deux fonctionnalités ci-dessus [19] d'entrepôt ont été développée sous JBuilder et ECCO au sein de même entrepôt OntoDaWa.

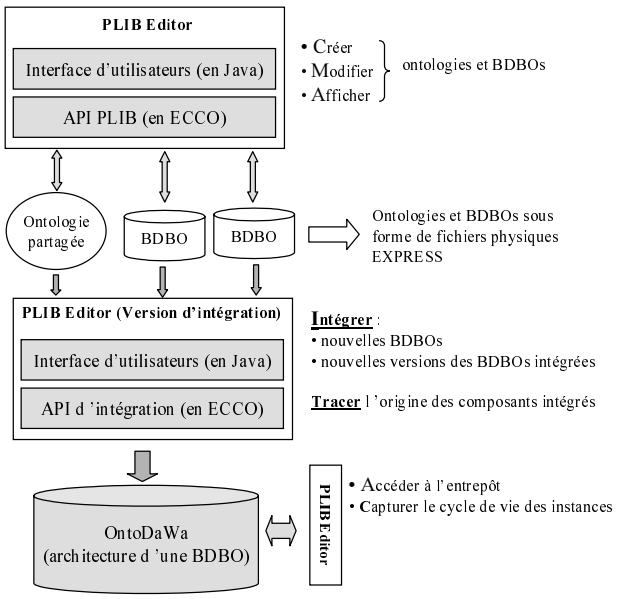

FIG. 5.9 – Prototype d'intégration et de gestion d'évolution asynchrone des entrepôt de données à base ontologique

L'architecture de notre prototype est décrite par la figure 5.9. Les ontologies et les BDBOs auxquelles nous nous intéressons sont celles définies par le modèle PLIB spécifié en langage EXPRESS. Elles sont donc échangeables sous forme de fichiers d'instances EXPRESS ("fichier physique"). Pour créer les ontologies et les BDBOs, nous avons utilisé PLIBEditor, présenté dans le chapitre 3, qui est un éditeur permettant non seulement de créer des ontologies et des BDBOs, mais également de les modifier ou les faire évoluer.

JBuilder est utilisé pour créer des interfaces d'utilisateurs. ECCO sert à implémenter les APIs d'intégration. Ces APIs nous permettent :

- d'une part, d'intégrer les différentes BDBOs selon les différents scénarii d'intégration définis dans le chapitre précédent,
- d'autre part, d'intégrer différentes versions d'une BDBO au sein de l'entrepôt en respectant le modèle des versions flottantes.

La table 5.3 présente un exemple des primitives principales que nous avons implémenté.

La figure 5.10 représente l'interface graphique intégrée dans *PLIBEditor* qui sert à choisir le mode d'intégration : sans historisation ontologique, ou avec l'historisation ontologique.

[19] (1) entrepôt avec versionnement des instances mais sans historisation ontologique et (2) entrepôt avec versionnement et historisation ontologique

Primitive	**Description**
$create\_RootClass(supplier \rightarrow id) : c_{root}$	Cette fonction crée la classe racine pour l'entrepôt de composants. Dans notre implémentation, les 7 propriétés, citées dans la section 5.5.1.2, sont implémentées au niveau de cette classe racine.
$double\_Entity(e, O_{Int} \rightarrow ID)$: Generic	Dans le but de création de la version courante, nous créons la fonction *double_Entity*. L'idée ici est qu'une entité au niveau quelconque doit référencer les autres au même niveau. Par exemple : un "dictionary_element" du concept au niveau de version courante ne référence que des entités $BSUs$ de la version courante. Cette fonction vérifie tout d'abord la nécessité de duplication de l'entité e (*les seules entités qui sont dupliquées sont les $BSUs$ et les entités qui référencent directement les BSUs*) : 1. si ce n'est pas nécessaire, la fonction *double_Entity* rend l'entité e même, 2. sinon, elle copie e, puis modifie les attributs de la copie, dont le type est BSU, afin que la copie référence les entités dans l'ensemble des $BSUs$ ($O_{Int} \rightarrow ID$ qui contient les $BSUs$ que la copie de l'entité e, si elle existe, doit référencer), et enfin retourne cette copie.
$integrate\_Concept\_with\_trace$ $(c_i^{new} \rightarrow def, O_{Int})$	Dans cette fonction, il s'agit d'une part à implémenter la version courante de c_i et d'autre part historiser sa dernière version. Elle est utilisée pour une entrepôt de composants avec l'historisation ontologique.
$InstanceIdentification\_to\_String$ $(i, SK_i) : STRING$	Cette fonction calcule la valeur de la propriété $InstanceCode$ pour chaque instance intégrée.
$init\_table(c \rightarrow def, Cat_{Int})$	Cette fonction consiste à initier une table multiversionnée, notre implémentation ici consiste à : 1. ajouter les 7 propriétés de la classe racine dans cette table mutltiversionnée, et 2. choisir les 3 propriétés : $SupplierCode$, $ClassCode$, $InstanceCode$, comme la clé sémantique de cette table.

TAB. 5.3 – Exemple des primitives implémentées pour gérer l'évolution d'un entrepôt de données à base ontologique.

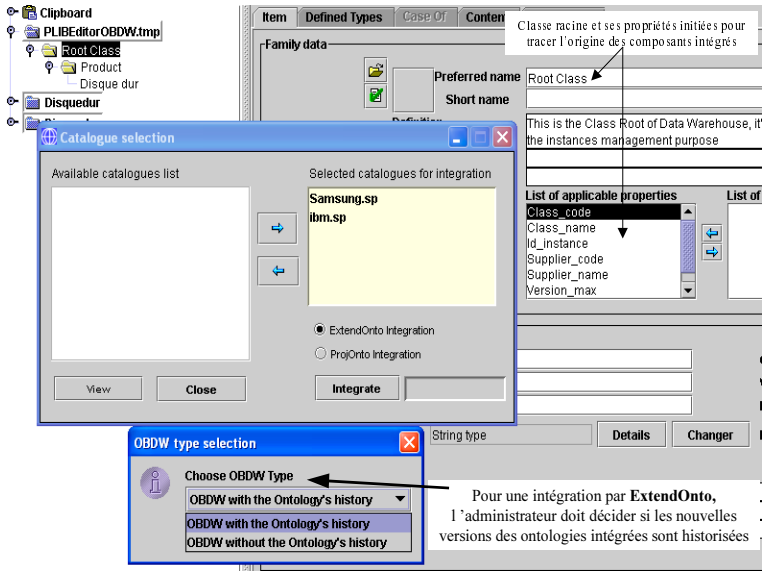

FIG. 5.10 – L'interface graphique montre l'option qui permet de choisir le mode d'intégration : soit sans historisation ontologique, soit avec l'historisation ontologique.

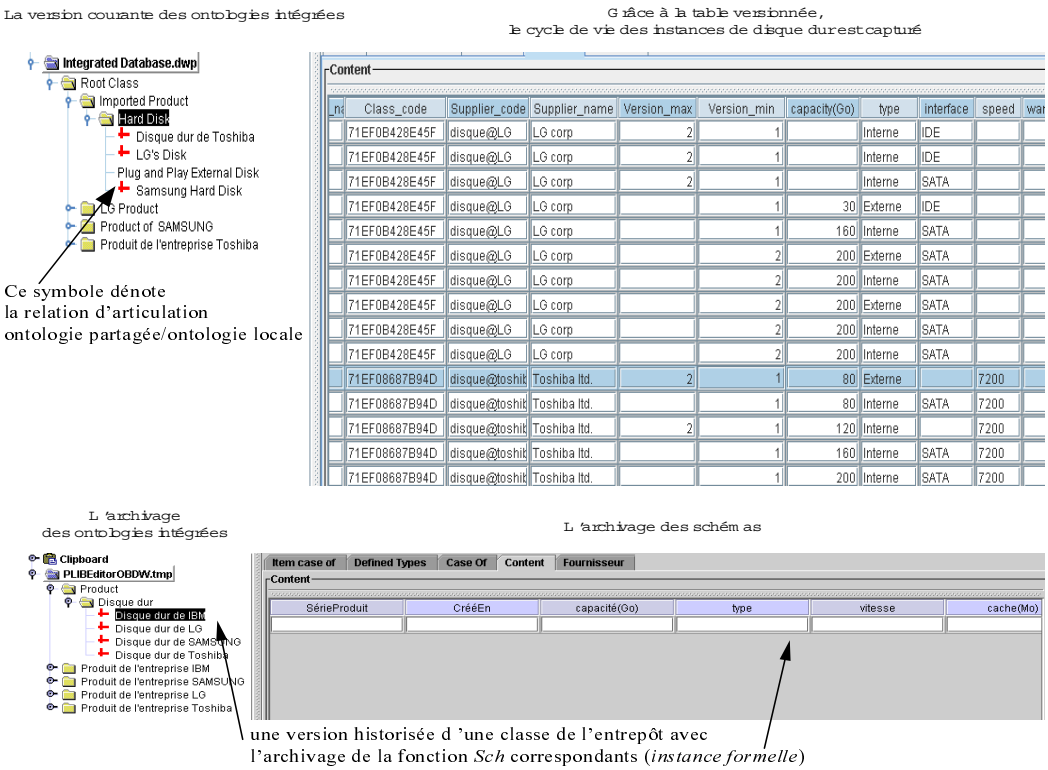

FIG. 5.11 – Un exemple de l'entrepôt de disques durs avec l'historisation ontologique

La figure 5.11 illustre un exemple de l'entrepôt de disque dur avec le stockage de l'origine des composants intégrés. Le haut de cette figure montre la version courante et le bas présente la partie historisée de l'entrepôt. A l'aide de PLIBEditor, nous pouvons également accéder aux instances de données dans l'entrepôt de deux manières. D'une part on peut accéder globalement au contenu intégré à travers la version courante de l'ontologie partagée et accéder de façon *spécifique à chaque catalogue* en descendant la hiérarchie de l'ontologie courante de l'entrepôt. D'autre part, on peut également accéder de façon directe à chaque catalogue intégré, ceux-ci apparaissant également explicitement. Le cycle de vie de chaque instance est également représenté explicitement dans la table de données (voir la figure 5.11).

5.6 Conclusion

Dans ce chapitre, nous avons présenté le problème de l'évolution asynchrone des données et des ontologies dans un système d'intégration de type entrepôt de données. Nous considérons des sources de données à base ontologique, dont chacune contient une ontologie locale qui référence une ontologie de domaine partagée. Ces sources sont autonomes ; elles évoluent de façon asynchrone et peuvent étendre ou/et spécialiser, en cas de besoin, l'ontologie de domaine. Notre processus d'intégration intègre d'abord les ontologies puis les données. En l'absence d'évolution, la présence de ces ontologies permet une automatisation complète du processus d'intégration et la résolution des conflits usuels dans l'intégration de bases de données hétérogènes. Lorsque les ontologies évoluent de façon incohérente entre les différentes sources, l'intégration automatique devient impossible. Pour résoudre ce problème, nous avons proposé d'encadrer les évolutions d'ontologies autorisées par le principe de continuité ontologique. Ce principe stipule qu'une ontologie ne peut infirmer un axiome qui se trouvait vérifiée dans une version antérieure de l'ontologie. Ce principe nous a alors permis de proposer un mécanisme, dit *de version flottante*, qui permet de ne conserver dans l'entrepôt qu'une seule version de chaque classe et de chaque propriété, appelées versions courantes (en fait, la plus grande version connue). Ceci permet de consolider entre les versions courantes toutes les relations ayant existé entre les différentes versions des différents concepts. L'ensemble des concepts en version courante constitue l'ontologie courante, et cette ontologie permet d'interpréter toutes les instances existant dans le système d'intégration, quelles que soient les versions de classe pour lesquelles elles avaient été définies.

Nous avons ensuite discuté la possibilité d'historisation des instances figurant dans les différentes sources et réintroduites lors de chaque rafraîchissement de l'entrepôt. Afin d'identifier les instances ontologiques bien qu'elles puissent, au cours du temps, être décrites par des ensembles différents de propriétés, nous avons proposé la notion de *clé sémantique*. Il s'agit d'un sous-ensemble de propriétés applicables d'une classe qui sont suffisantes pour en identifier les instances tout au long de l'évolution. Cette notion nous

a alors permis de proposer une méthode d'historisation des instances qui évite toute duplication : chaque instance n'est représentée qu'une seule fois, elle consolide toutes les propriétés qui ont existées au cours du temps et deux attributs identifient les versions de classe pour lesquelles l'instance est ou a été valide (instance *multi-versionnée*).

Au total la structure de notre système d'intégration, appelé OntoDaWa, est celle d'un entrepôt à base ontologique, qui référence également l'ontologie partagée et dont les instances sont multi-versionnées. Elle est constituée de quatre parties, à savoir, (1) l'ontologie courante qui contient la version courante de l'ontologie de l'entrepôt, (2) l'archivage des ontologies qui contient, si besoin, toutes les versions des définitions ontologiques de chaque classe et propriété, (3) l'historique des schémas de représentation des instances des différentes classes, et (4) les tables multi-versionnées contenant toutes les instances ainsi que leur première et dernière versions d'activité. Cette structure permet à la fois d'historiser les ontologies, les schémas, et les instances. L'historisation des ontologies peut être évitée dans la plupart des cas, l'ontologie courante suffisant à la fois pour intégrer de façon automatique les sources, et pour accéder à l'ensemble des instances.

Notre modèle a été validé par l'outil *PLIBEditor* de la version ECCO. Il s'agit de compléter le système intégré que nous avons implémenté dans le chapitre 4 en prenant en compte la version de données intégrées.

Notons que notre approche est la première, à notre connaissance, qui permet à la fois d'intégrer de façon complètement automatique des sources de données disposant d'une grande autonomie de représentation, et de supporter l'évolution asynchrone de ces sources.

Chapitre 6

Réification des correspondances entre ontologies pour l'intégration des BDBOs

6.1 Introduction

Notre approche d'intégration *a priori* par articulation d'ontologies contraint les sources locales à référencer *a priori* l'ontologie partagée. Cette approche est intéressante pour les applications comme e-gouvernement, l'e-commerce en direction d'un grand donneur d'ordre où une autorité oblige ses partenaires (les sources) à référencer l'ontologie partagée. Mais dans certaines situations, cette approche n'est pas souhaitable. Parmi ces situations, nous pouvons citer trois cas de motivation :

1. une ontologie normalisée du domaine n'existe pas, puis apparaît sans que l'on veuille changer la base de données,
2. il existe plusieurs ontologies de domaine qui se chevauchent (par exemple : une ontologie des USA, une ontologie de l'européen), et
3. on veut pouvoir livrer simultanément plusieurs "mappings", laissant le client les activer lui-même (par exemple, un fournisseur veut publier un unique catalogue avec plusieurs connexions vers les différentes ontologies).

Dans ces trois cas, la correspondance avec une ontologie partagée ne peut pas, ou ne doit pas, être intégrée dans l'ontologie locale. Elle doit donc être représentée à l'extérieur, comme un modèle à part entière comme le suggère des travaux récents sur la gestion des modèles [18, 60, 17]. C'est ce que nous appelons une *réification des correspondances entre ontologies*. Ceci est en particulier indispensable si l'on souhaite pouvoir échanger non seulement nos données à base ontologique, mais également les modèles de correspondance de façon à permettre à chaque utilisateur d'effectuer l'intégration avec la (ou les) ontologie(s) qu'il utilise. Nous souhaitons donc :

1. mettre la "correspondance" à l'extérieur du modèle pour permettre plusieurs "mappings",
2. représenter la "correspondance" elle même comme un modèle [17], et
3. réifier également les opérateurs mis en jeu dans les correspondances de façon à rendre leur exploitation neutre, c'est-à-dire indépendante de tout langage ou environnement particulier.

Ce chapitre est consacré à l'étude du problème d'intégration de base de données à base ontologique par réification des correspondances entre ontologies. L'hypothèse fondamentale est l'existence d'une ontologie normalisée non référencée *a priori* par des sources. Cette situation engendre des problèmes d'hétérogénéité entre les ontologies (locales et partagée) : un concept peut être explicité différemment dans deux ontologies différentes. Par exemple, dans une ontologie de voitures utilisée pour les applications de tourisme, le concept "*Voiture*" est subsumé par "*Moyen de transport*", et dans une application de commerce, le concept "*Voiture*" est subsumé par le concept "*Produit*".

Notons que l'hétérogénéité au niveau des modèles d'ontologies n'est pas étudié dans le cadre de cette thèse. Les ontologies articulées sont supposées basées sur le même modèle d'ontologie.

Dans une intégration par *réification des correspondances entre ontologies*, l'articulation d'ontologie consiste à identifier et à modéliser les relations sémantiques entre les différents concepts ontologiques. Lorsque ces relations sont modélisées, il est possible soit de les utiliser directement pour projeter les données d'une ontologie sur l'autre, soit de les échanger pour permettre à un utilisateur d'utiliser les données en termes d'une quelconque ontologie.

L'objectif de ce chapitre est (1) de proposer une approche permettant d'une part, de formuler les correspondances entre ontologies, et, d'autre part, de les exploiter pour projeter automatiquement des données à base ontologique sur une autre ontologie, et (2) de proposer un modèle neutre afin de représenter ces correspondances.

Ce chapitre est organisé en sept sections. La section 2 rappelle d'abord quelques travaux antérieurs sur la mise en correspondance entre ontologies ("mapping"), et puis identifie les correspondances possibles entre deux ontologies dans le but d'échanger de données entre les sources dont les sémantiques sont représentées par ces deux ontologies. La section 3 propose une formulation des correspondances entre une BDBO et une quelconque ontologie. Cette formulation est présentée sous forme d'une projection entre cette BDBO et l'ontologie mise en correspondance. La quatrième section est dédiée à l'algorithme de projection qui permet l'intégration automatique des BDBOs en exploitant la réification des correspondances entre ontologies. Dans la section 5, nous proposons un modèle de "mapping" pour modéliser les correspondances entre ontologies. Ce modèle est basé principalement sur la méta-représentation des expressions en EXPRESS et la réification des opérateurs. Une implémentation permettant de valider les propositions du chapitre est présentée dans la section 6. La section 7 conclut ce chapitre.

6.2 Problématique et correspondances entre ontologies

Dans cette section, nous présenterons d'abord quelques travaux antérieurs sur la mise en correspondance entre ontologies. Nous discutons ensuite sur les relations possibles entre ontologies.

6.2.1 Travaux antérieurs sur la mise en correspondance entre ontologies

Une mise en correspondance entre ontologies définit les relations conceptuelles entre des concepts (classes et propriétés) définis dans deux ontologies. De nombreux travaux sur la mise en correspondance entre les ontologies ont été présentés dans la littérature [33, 50, 98]. On peut identifier deux types de travaux :

1. le premier [31, 73, 55, 51] vise à automatiser la découverte des éléments équivalents dans des ontologies. Les travaux dans un tel type utilisent des techniques de fouilles de données proposées dans le domaine d'intelligence artificielle, par exemple : les techniques linguistiques utilisant des mesures de similarité entre les termes linguistiques [55, 51], la technique d'apprentissage [31], etc. Généralement, les solutions proposées se limitent au traitement des relations (concept source-concept cible) de type 1 : 1 [31, 73].
2. le deuxième se concentre sur la modélisation des correspondances ontologiques [17, 22], car il suppose que les modèles d'ontologies sont hétérogènes [85], ou pas suffisants pour exprimer des correspondances.

Pour mieux comprendre le premier type de travaux, nous détaillons l'algorithme ASCO proposé par Le et al. [55]. Ce dernier s'exécute en deux phases :

1. dans la première phase, dite *phase linguistique*, la similarité entre deux entités (concepts ou relations) provenant des deux ontologies est calculée à partir de différentes informations disponibles sur leurs noms, leurs étiquettes (labels), et leurs descriptions qui décrivent textuellement les concepts afin de faciliter leurs compréhensions aux utilisateurs. Le calcul de la valeur de similarité linguistique est effectué en utilisant des métriques comme "string-distance", TF/IDF (Terme Frequency/Inverse Document Frequency). ASCO intègre également WordNet dans son système pour préciser les relations de synonymie ou hyperonymie entre termes.
2. la deuxième phase, dite *phase structurelle* exploite les informations taxonomiques dans les structures des ontologies pour calculer la similarité structurelle entre deux entités. Cette dernière est calculée en combinant la similarité (linguistique) entre leurs voisins dans l'arbre de l'ontologie (super-entités directes, sous-entités directes, et les entités au même niveau) et la similarité entre leurs chemins (le chemin d'un

concept est défini comme l'ensemble des concepts rencontrés en partant du concept racine jusqu'au concept considéré).

Un travail concernant la modélisation des correspondances, développé par Bullig et al. [22], consiste à mettre en correspondance deux ontologies dans le but d'échanger les données de différents catalogues de produits. Cette correspondance est réalisée en deux étapes (voir la figure 6.1) :

1. la correspondance entre classes associe une classe source à n classes cibles. Cette relation permet de déterminer la classe cible de chaque instance i d'une classe c_s de la *source*. La classe cible de i est sélectionnée parmi les classes "*default target*" et "*Target bound to a condition*". Chaque classe c_i de "*Target bound to a condition*" est accompagnée par une condition (*Evaluated Properties*) sur une propriété de la classe c_s.
   Si i satisfait cette condition, elle sera intégrée comme une instance de la classe c_i. Sinon, sa classe cible est la classe "*default target*".
2. la correspondance entre propriétés associant m propriétés sources à une propriété cible. Cette relation est exprimée par une fonction de conversion spécifiant le calcul de la valeur d'une propriété cible à partir de propriétés sources.

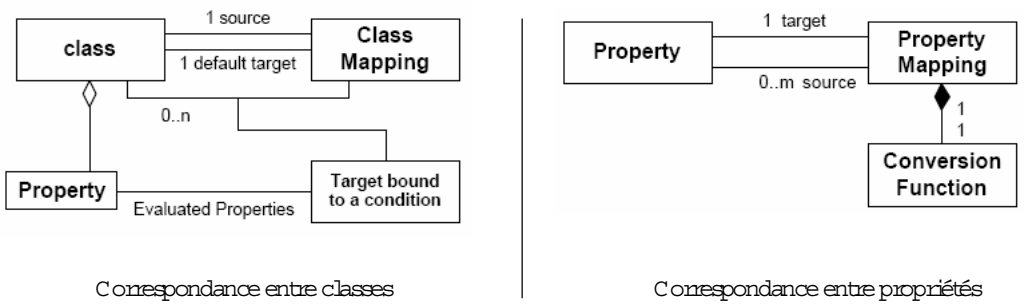

FIG. 6.1 – Les correspondances entre ontologies de produits proposées par l'Université de Hagen [22]

Dans ce travail, la correspondance entre classes présente une relation ($source - cible$) de type $1-n$ et celle entre propriétés présente une relation ($cible - source$) de type $m-1$. Ces correspondances sont implémentées manuellement par des experts du domaine.

Plusieurs questions vont être étudiées dans ce chapitre :

1. quelles sont les relations sémantiques nécessaires pour établir un "mapping" entre deux ontologies indépendantes ?
2. comment modéliser ce "mapping" ?
3. une fois un "mapping" entre ontologies établi, comment intégrer les BDBOs ?

Notre travail proposé dans ce chapitre concerne la modélisation des correspondances.

Nous identifions ci-dessous les relations entre ontologies que nous voulons exploiter dans le but d'intégration.

6.2.2 Correspondances entre ontologies

Nous distinguons deux types de correspondances : (1) les relations entre classes et (2) les relations entre propriétés que nous allons détailler dans les sections suivantes.

6.2.2.1 Relation entre classes

La correspondance entre classes la plus générale est une relation de type $n : m$, notée par $\{c_{s_i}\}_{i\in[1:n]} \rightarrow^{n:m} \{c_{t_j}\}_{j\in[1:m]}$ où :

- $\{c_{s_i}\}_{i\in[1:n]}$ représente l'ensemble des classes sources.
- $\{c_{t_j}\}_{j\in[1:m]}$ représente l'ensemble des classes cibles.

Prenons l'exemple dans la figure 6.2. Afin d'intégrer les instances des deux classes sources : $Travailleur$ et $NonTravailleur$ dans les deux classes cibles : $Homme$ et $Femme$, une correspondance entre eux est nécessaire. Pour le cas le plus général, cette correspondance est une relation de type 2 : 2. Mais, cette dernière peut se décomposer en deux relations de type 1 : 2 :

- $Travailleur \rightarrow^{1:2} \{Homme, Femme\}$ et
- $NonTravailleur \rightarrow^{1:2} \{Homme, Femme\}$,

Notons que la relation $\{Travailleur, NonTravailleur\} \rightarrow^{2:2} \{Homme, Femme\}$ peut se décomposer également en deux relations de type 2 : 1 :

- $\{Travailleur, NonTravailleur\} \rightarrow^{2:1} Homme$ et
- $\{Travailleur, NonTravailleur\} \rightarrow^{2:1} Femme$.

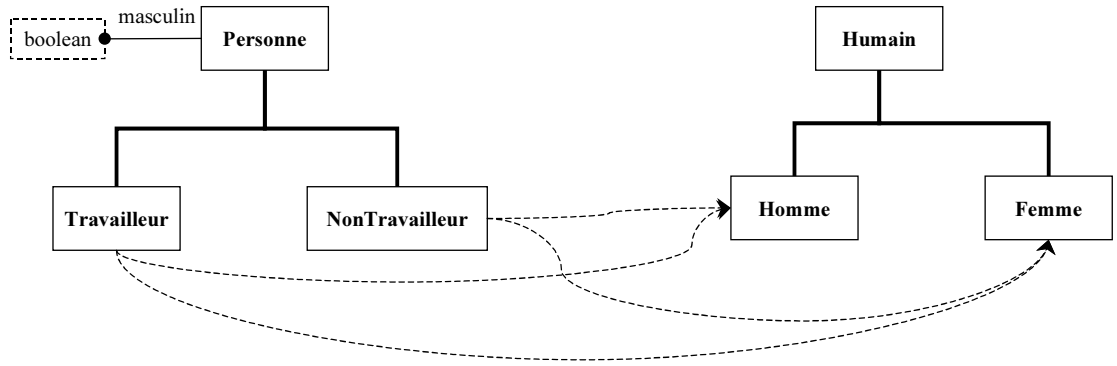

FIG. 6.2 – Exemple des relations entre classes

Dans le cas où une classe est mappée sur m classes, il faut trouver un prédicat qui permet d'identifier l'appartenance de chaque instance.

Exemple 13 *La relation* $Travailleur \rightarrow^{1:2} \{Homme, Femme\}$ *est décrite comme suit (voir la figure 6.2) :*

- *si la valeur de la propriété masculin d'une instance $i \in Travailleur$ est égale à $True$, cette instance i est intégrée dans la classe $Homme$,*
- *si non ($i.masculin = False$), cette instance i est intégrée dans la classe $Femme$.*

Ainsi, on peut toujours se ramener :

1. à une relation (classe source : classe cible) de type $1 : m$,
2. un prédicat permet pour chaque classe source de filtrer ses instances,
3. plusieurs classes sources peuvent contribuer à la même cible.

6.2.2.2 Relations entre propriétés

Pour caractériser les instances intégrées en fonction des propriétés de leurs classes cibles, une correspondance entre les propriétés est nécessaire.

La relation entre propriétés la plus générale est une relation de type $n : m$, notée par $\{p_{t_i}\}_{i\in[1:n]} \leftarrow^{n:m} \{p_{s_j}\}_{j\in[1:m]}$ où :

- $\{p_{t_i}\}_{i\in[1:n]}$ représente l'ensemble des propriétés cibles.
- $\{p_{s_j}\}_{j\in[1:m]}$ représente l'ensemble des propriétés sources.

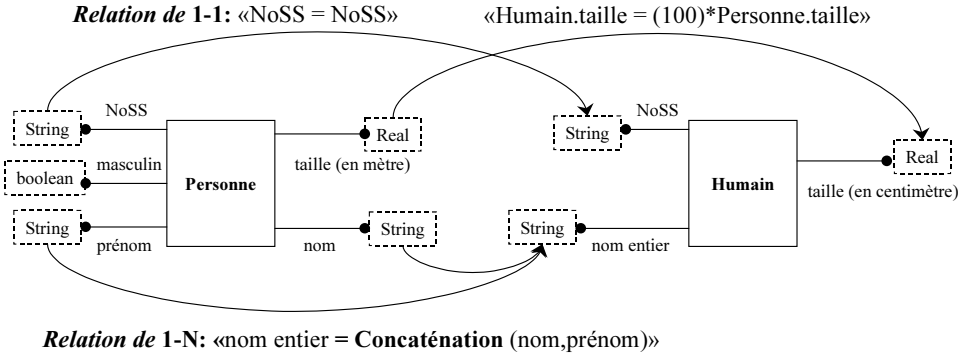

FIG. 6.3 – Exemple de la relation a posteriori entre propriétés

Prenons l'exemple de la propriété *"Humain.nom entier"* de la figure 6.3. Cette dernière est calculée par la concaténation de deux propriétés : *"Personne.nom"* et *"Personne.prénom"*. Elle illustre une relation (*cible* : *source*) de type 1 : 2. Dans le cas inverse, la relation {*"Personne.nom"*,*"Personne.prénom"*}$\leftarrow^{2:1}$*"Humain.nom entier"* représente une relation de type 2 : 1. Notons que cette dernière peut se décomposer en deux relations de type 1 : 1 :

- *"Personne.nom"* $\leftarrow^{1:1}$ *"Humain.nom entier"*.
- *"Personne.prénom"* $\leftarrow^{1:1}$ *"Humain.nom entier"*.

Autre exemple, l'unité de mesure de la propriété $Personne.taille$ est le "mètre", tandis que la propriété $Humain.taille$ est en "centimètre". Dans ce cas, lorsqu'une instance (i) de la classe $Personne$ est intégrée dans l'ontologie $Humain$, la valeur de $Personne.taille$ de l'instance i doit être convertie en centimètre pour qu'elle devienne la valeur de $Humain.taille$

de i. Cette relation entre $Humain.taille$ et $Personne.taille$ illustre une relation (*cible* : *source*) de type 1 : 1.

Ainsi, on peut toujours se ramener à :
- une relation (*cible* : *source*) de type $1 : n$,
- une fonction permet de calculer, *pour une classe source donnée*, comment *un attribut cible* donné est calculé pour chaque instance.

6.2.2.3 Bilan sur les correspondances entre ontologies

Nous avons présenté ci-dessus les correspondances entre ontologies, à savoir la relation de classes et la relation de propriétés. Dans le cas le plus général, les cardinalités de la relation entre classes sont de type $n : m$. Il en est de même pour la relation entre propriétés.

Nous remarquons que les deux relations de type $n : m$ pouvaient en fait se décomposer en relations de type $n : 1$ ou $1 : m$.

Nous présenterons dans la section suivante la formalisation de la projection entre une BDBO et une ontologie.

6.3 Formalisation de la projection entre une BDBO et une ontologie

Nous supposons donnés les deux éléments :
- $BDBO_s :< O_s, I_s, Sch_s, Pop_s >$ avec $O_s :< C_s, P_s, Sub_s, Applic_s >$, étant la base de données à base ontologique source.
- $O_T :< C_T, P_T, Sub_T, Applic_T >$, étant l'ontologie cible.

Nous supposons également que l'ontologie O_T est canonique et basé sur le principe de mono-instanciation :

Hypothèse H1 : $\forall$ *ins* $\in I_s$, *il existe au plus un élément minimal* c_j *pour la relation de subsomption tel que :* $c_j \in C_T$ *et ins* $\in c_j$.

Enfin, nous noterons $proj(x)$ la projection d'une instance de I_s sur l'ontologie O_T définie par la transformation suivante.

Une projection $\mathcal{P}$ entre $BDBO_s$ et O_T est définie par un triplet : $< ClassMap, \mathfrak{B}, \mathfrak{F} >$ où :
- $ClassMap : C_s \rightarrow 2^{C_T}$ est la fonction partielle qui associe à toute classe de O_s les classes de O_T auxquelles appartiennent une ou plusieurs de ses instances.
- $\mathfrak{B}$ est un ensemble de prédicats qui définit si la projection d'une instance de $c_i \in C_s$ appartient à $c_j \in ClassMap(c_i) \subset C_T$:

- $\forall c_i \in C_s$, $\forall c_j \in ClassMap(c_i)$, $Belongs_{c_i,c_j} : c_i \rightarrow Boolean$;
- si $ins \in c_i$, $Belongs_{c_i,c_j}(ins) \Leftrightarrow proj(ins) \in c_j$.

– $\mathfrak{F}$ est un ensemble de fonctions qui spécifient la valeur des propriétés de la projection d'une instance sur l'ontologie O_T :

- $\forall c_i \in C_s, \forall p_k \in \bigcup_{c_j \in ClassMap(c_i)} Applic(c_j) : PropMap_{c_i,p_k} : c_i \rightarrow Range(p_k) \bigcup NULL$
- si $ins \in c_i$:

– $PropMap_{c_i,p_k} = NULL$, signifie que p_k n'est pas valuée pour $proj(ins)$,

– $PropMap_{c_i,p_k} = \alpha$, signifie que $p_k(proj(ins)) = \alpha$.

Avec ces notations, l'hypothèse H1 se formalise comme suit :

$$(H_1) : \quad \forall c_i \in C_s, \forall c_j, c_k \in ClassMap(c_i), \forall ins \in c_i :$$
$$c_j \neq c_k \Rightarrow \neg(Belongs_{c_i,c_j}(ins) \wedge Belongs_{c_i,c_k}(ins))$$

On notera que, dans cette définition :

1. certaines instances peuvent ne pas avoir de projection (si $BDBO_s$ dépasse le domaine sémantique couvert par O_T),
2. certaines propriétés de O_s peuvent ne pas être représentées dans la propriété de O_T.
3. les instances d'une même classe source peuvent se répartir dans plusieurs classes cibles.
4. les instances d'une classe cible peuvent provenir de plusieurs classes sources.

Une projection $\mathcal{P}$ présente également une *mise en correspondance* entre $BDBO_s$ et O_T.

Nous présenterons dans la section suivante un algorithme de projection qui permet d'intégrer automatiquement la source $BDBO_s$ dans l'ontologie cible O_T.

6.4 Algorithme de projection

Le résultat de la projection de la $BDBO_s$ dans l'ontologie O_T est de produire un $BDBO_T$ avec : $BDBO_T :< O_T, I_T, Sch_T, Pop_T >$. Définir un algorithme d'intégration revient donc à calculer ces quatre éléments.

Pour simplifier les notations, nous introduisons d'abord les fonctions suivantes :

– $Typeof$ est la fonction générique qui associe à toute instance d'une $BDBO$ sa plus petite classe d'appartenance.

– $ValuedProp$ est la fonction générique qui associe à toute instance d'une $BDBO$ l'ensemble de ses propriétés valuées.

– $InsTargetClass$ est la fonction qui associe à toute instance de I_s la classe de base unique de sa projection dans I_T : $\forall ins \in I_s$,
$InsTargetClass(ins) = \{c_j \in ClassMap(TypeOf(x)) | Belongs_{Typeof(x),c_j}(ins)\}$

Avec ces notations, les quatre éléments de $BDBO_T$ se calculent comme suit :

1. O_T est déjà définie,
2. $Pop_T(c_j) = \bigcup_{c_i \in ClassMap^{-1}(c_j)} \{proj(ins) | ins \in c_i \wedge InsTargetClass(ins) = c_j\}$,
3. $I_T = \bigcup_{c_j \in C_T} Pop_T$,
4. $Sch_T(c_j) = \bigcup_{x \in Pop(c_j)} ValuedProp(x)$;
   les propriétés des instances O_T sont alors calculées en utilisant les fonctions $\mathfrak{F}$ et la correspondance $O_s \times O_T$.

Cet algorithme permet donc très simplement de calculer la projection de la population d'une BDBO dans le contexte d'une autre ontologie. Concrêtement l'algorithme se passe de la façon suivante (hors optimisation) :

⊙ l'algorithme prend en entrées $BDBO_s$, O_T et la projection $\mathcal{P}$ et rend en sortie la $BDBO_s$ et la relation $\mathcal{R} \subset I_s \times I_T$ qui associe à chaque instance source son image dans la cible. L'algorithme est utilisé deux fois :

- pour la première fois, la fonction créer($projection(x, c_j)$)$^{(1)}$ crée à la fois $\bar{x}$ dans I_T et insère (x,$\bar{x}$) dans $\mathcal{R}$ et la fonction créer($\bar{x}, p, PropMap_{c_i,p}(x)$) $^{(2)}$ crée la valeur pour la propriété p. Si p est une association, sa valeur est $NULL$.
- pour la deuxième fois, la fonction (1) ne crée plus rien. La fonction (2) traite seulement les propriétés p de type association et remplace la valeur $NULL$ par l'image de $p(x)$ dans $\mathcal{R}$.

◇ *Pour* $c_i \in C_s$
- *calculer* CM :=$ClassMap(c_i)$;
- *pour* $x \in c_i$
  - *pour* $c_j \in CM$
    - *si* $Belongs_{c_i,c_j}(x)$ *alors*
      - $\bar{x}$:= *créer* $(projection(x, c_j))$; $^{(1)}$
      - *pour* $p \in Applic(c_j)$
        - *créer* $(\bar{x}, p, PropMap_{c_i,p}(x))$;$^{(2)}$
      - *fin pour* ;
    - *fin si* ;
  - *fin pour* ;

◇ *Fin Pour* ;

Nous avons formulé le problème d'intégration des données à base ontologique par réification des correspondances entre ontologies. Nous avons proposé également un algorithme de projection qui permet de projeter automatiquement la population d'une $BDBO_s$ dans une ontologie en exploitant les correspondances entre eux.

Dans la section suivante, nous proposons une solution pour modéliser les correspondances entre ontologies.

6.5 Représentation du "mapping" en tant que modèle

Dans cette section, nous allons présenter un modèle du "mapping" pour représenter les correspondances entre ontologies. Le modèle proposé est basé sur la méta-représentation des expressions en EXPRESS qui permet de spécifier un schéma EXPRESS pour la représentation et l'échange non ambigus des expressions interprétables par l'ordinateur. L'objectif est de représenter ces correspondances de façon neutre indépendante du langage d'exécution, et cela afin de pouvoir éventuellement échanger les expressions.

Nous présentons tout d'abord l'existant du modèle PLIB concernant la mise en correspondance entre ontologies.

6.5.1 L'existant du PLIB : A_posteriori_case_of

Dans le modèle PLIB [83], il existe déjà une entité qui représente une subsomption *a posteriori*. Il s'agit de l'entité *a_posteriori_case_of*. Cette entité est modelé dans la figure 6.4 :

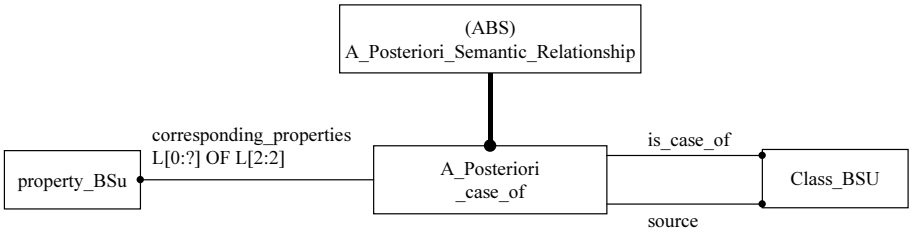

FIG. 6.4 – Relation A_Posteriori_Case_Of

- *a_posteriori_case_of* est un sous-type de *A_Posteriori_Semantic_Relation* qui est le supertype abstrait de toutes relations sémantiques *a posteriori*.
- l'attribut *source* : représente la classe cible (classe subsumante) de la correspondance.
- l'attribut *is_case_of* : représente la classe de source qui est "is-case-of" de la classe cible.
- l'attribut *corresponding_properties* : représente l'ensemble des paires de propriétés similaires dans deux classes articulées. La première propriété est à la classe cible et la deuxième est à la classe source. Ces deux propriétés sont applicables pour leurs classes. Et elles doivent être compatibles (c'est-à-dire qu'on peut mapper la propriété source sur celle cible sans modification de la valeur de propriété).

Remarquons que la relation entre propriétés définie dans cette correspondance représente une relation de type 1 : 1.

Avant de présenter un schéma du "mapping" générique entre ontologies, nous présentons ci-dessous l'approche que nous proposons pour modéliser les expressions qui permet de décrire d'une part les prédicats $\in \mathfrak{B}$, d'autre part les fonctions $\in \mathfrak{F}$ pour une quelconque projection $\mathcal{P}$.

6.5.2 La méta-représentation des expressions en EXPRESS

L'approche que nous proposons d'utiliser pour modéliser les expressions est basée sur la technique de la méta-modélisation. Cette technique a été appliquée au langage EXPRESS dans [3, 4]. D'un point de vue structurel, à l'image des interpréteurs dans les langages de programmation, ces expressions sont modélisées par un graphe orienté acyclique dont les noeuds sont des opérateurs et les feuilles sont des littéraux. Cette représentation correspond à un arbre de syntaxe abstraite dans la théorie de compilation [2].

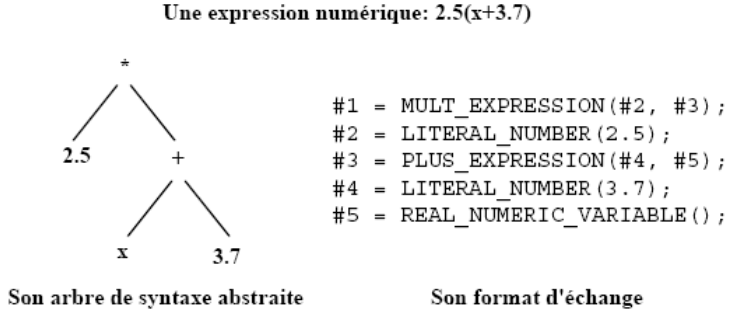

FIG. 6.5 – modélisation de la connaissance procédurale par méta-modélisation [93].

La Figure 6.5 illustre l'arbre de syntaxe abstraite d'une expression numérique et son format d'échange représenté par des instances d'entités EXPRESS.

Dans un tel format d'échange, chaque instance d'entité est associée à un identifiant (par exemple #2), le nom de son type d'entité (par exemple $LITERAL\_NUMBER$), et à une valeur explicite pour un littéral (par exemple "2.5"), ou calculée pour un variable ou une expression.

Notons que la variable "x" de l'expression numérique, représentée par "#5" dans le format d'échange, est juste un symbole pour la variable. Ce symbole doit être associé, en dehors de l'expression, à une sémantique définie par le résultat d'une fonction d'interprétation.

La représentation des expressions par méta-modélisation permet de représenter et d'échanger des expressions de calcul algébrique. En conséquence, elle permet de :

1. exprimer des conditions (contraintes d'intégrité) sur les données représentées (par exemple, "*masculin = TRUE*").
2. représenter la dérivation de propriétés qui dépendent d'autres propriétés en utilisant des fonctions (par exemple : *"nom entier = concaténation (nom, prénom)"*).

Notre modèle de correspondances entre ontologies est basé sur la partie 20 de la norme ISO-13584 [5]. Cette dernière propose un modèle logique d'expressions en EXPRESS. Ce modèle est fondé sur la méta-modélisation. Il fournit un ensemble très riche d'expressions, telles que les *expressions numériques*, les *expressions booléennes* et les *expressions de chaîne de caractères*. Nous résumons ci-dessous les concepts fondamentaux du modèle.

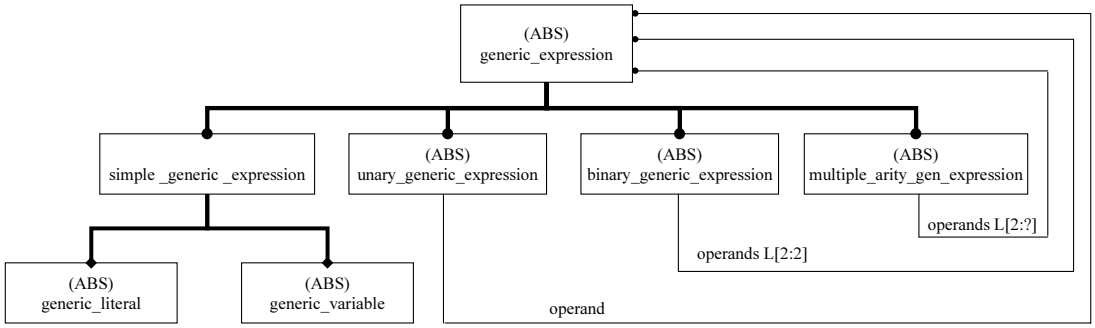

FIG. 6.6 – modèle des expressions génériques présenté dans [5]

Expression générique. La figure 6.6 présente le schéma EXPRESS-G des expressions génériques dans [5]. L'entité **generic_expression** est le supertype abstrait de toutes expressions possibles. Afin d'assurer l'acyclicité des expressions, *generic_expression* est sous-typée selon son arité [20].

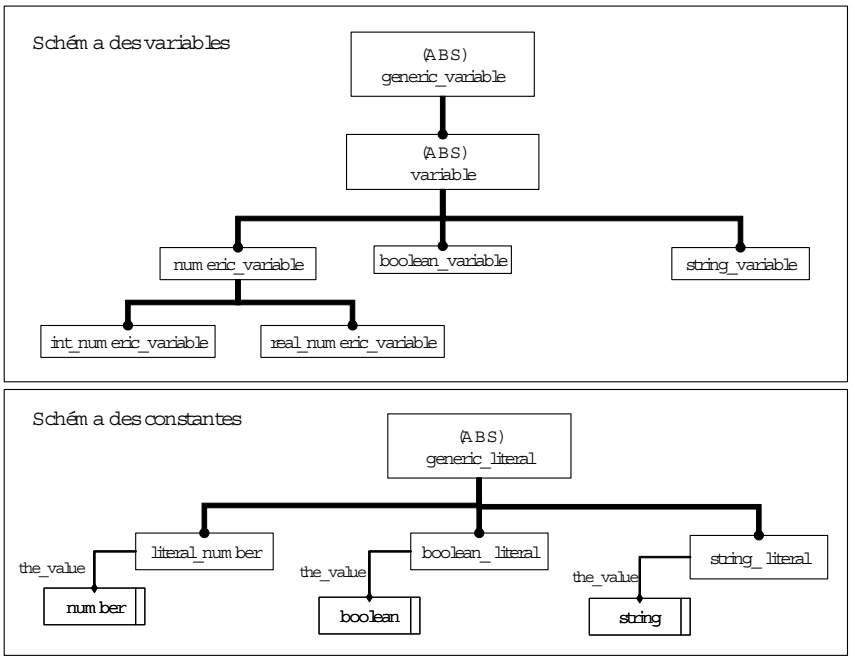

FIG. 6.7 – Représentation des variables et constantes dans le modèle des expressions ISO 13584-20 [5]

Dans une expression générique, l'entité **simple_generic_expression** représente soit une variable générique (*generic_variable*), soit une constante générique (*generic_literal*).

[20] L'arité d'une expression est le nombre d'arguments (variables, constantes, eux-mêmes expressions) sur lesquels elle porte. On dira que l'expression est *unaire* si son arité est égale à 1, *binaire* si son arité est égale à 2 et *multi-aires* si son arité est plus à 2.

Les domaines de variables/constantes sont représentés par un typage fort effectué par le soustypage de l'entité variable/constantes (par exemple, *numeric_variable*/*literal_number*, *boolean_variable*/*boolean_literal*, *String_variable*/*String_literal*, etc.) (voir la figure 6.7). Le type de données de chaque expression (i.e. variable) est vérifié lors de l'échange des expressions.

Expression. Pour le cadre de cette thèse, nous nous intéressons au schéma des expressions (voir la figure 6.8). L'entité **expression** de ce schéma est en fait un sous-type de l'entité *generic_expression* qui est limité au domaine numérique, logique, ou de chaîne de caractères. Le modèle des expressions spécifie un ensemble très riche d'opérateurs.

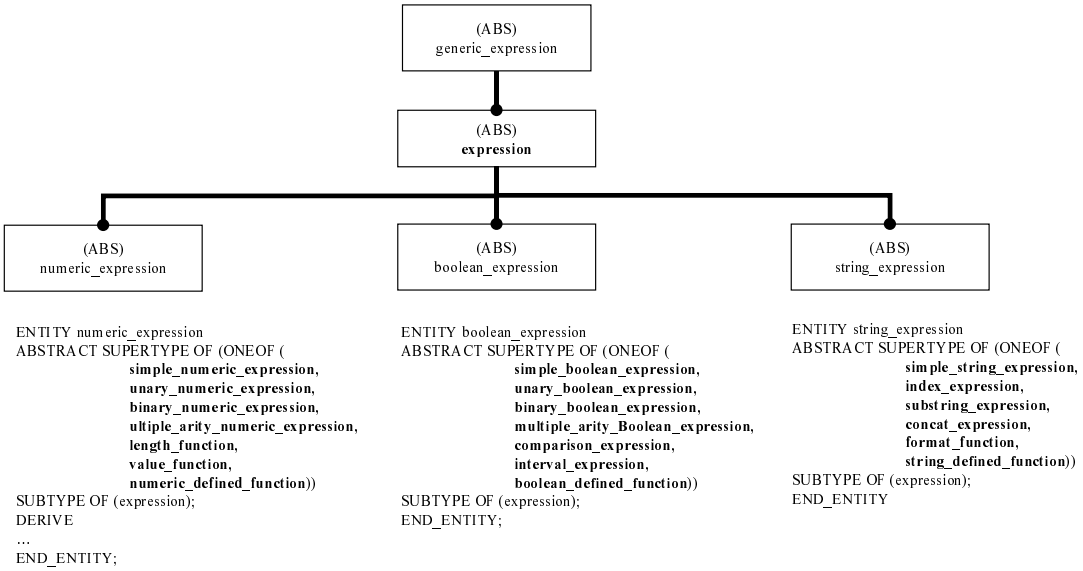

FIG. 6.8 – modèle des expressions ISO 13584-20 [5]

6.5.3 Association des variables dans les expressions à leurs valeurs

Dans le modèle de représentation des expressions, chaque variable est attachée à une sémantique particulière dans le modèle général qui définit l'interprétation qui doit en être faite (associer une variable à une propriété d'une classe, par exemple).

Les variables sont représentées dans le modèle des expressions comme étant des soustypes des expressions. Chaque variable a trois aspects différents [5] :

1. une représentation syntaxique qui est un symbole utilisé pour construire une expression,
2. un type de données qui définit le domaine de valeurs de la variable,
3. une sémantique qui définit sa signification et permet de récupérer sa valeur lors d'une évaluation.

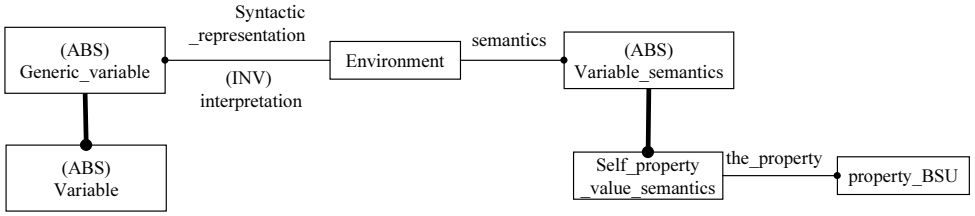

FIG. 6.9 – Association sémantique entre une variable et la sémantique qui y est attachée

Une variable particulière est représentée comme une instance de l'entité variable. Chaque instance est associée à un identificateur (représenté par '#' suivi d'un numéro dans le modèle d'échange EXPRESS, par exemple) qui constitue le symbole représentant la variable dans une expression.

Pour déterminer la signification d'une variable dans le modèle d'échange, chaque variable doit être associée à une sémantique particulière. Cette sémantique est représentée par l'entité abstraite **variable_semantics** qui doit être sous-typée à des sémantiques particulières selon l'utilisation de la variable (associer une variable à une propriété, par exemple). Cette association permet de doter la variable d'une valeur au moment de l'évaluation d'une expression.

Dans notre modèle, la relation sémantique est assurée par une entité d'association intermédiaire : il s'agit de l'entité **environnement** dans la figure 6.9.

La section suivante présente le schéma que nous modélisons pour représenter les correspondances entre deux ontologies.

6.5.4 Le schéma *General_Mapping*

Nous proposons maintenant un schéma du "mapping" général. Il permet de représenter les correspondances identifiées dans la section 2 sous forme des instances d'un modèle.

Le schéma *General_Mapping* est présenté dans la figure 6.10. Ce schéma représente les trois entités principales suivantes :

1. l'entité *General_Mapping* modélise une correspondance entre une classe source ($ClassSource$) et une classe cible ($ClassTarget$). Cette entité représente une relation entre classes de type $1-1$ (source-cible).
2. l'entité $filter$ est associée avec la *General_Mapping* à travers l'attribut $ClassMap$. Elle permet d'identifier si une instance de la classe source peut être intégrée dans la classe cible :
   – si une instance de la classe source peut être projetée sur la classe cible, cette instance doit satisfaire la condition présentée par l'entité $Boolean\_Expression$. Cette dernière est associée à la $filter$ à travers l'attribut *condition*.
   Notons que pour les instances de la classe source satisfaisant la condition de filtre, la classe cible est la classe la plus minimale dans l'ontologie cible sur laquelle ces

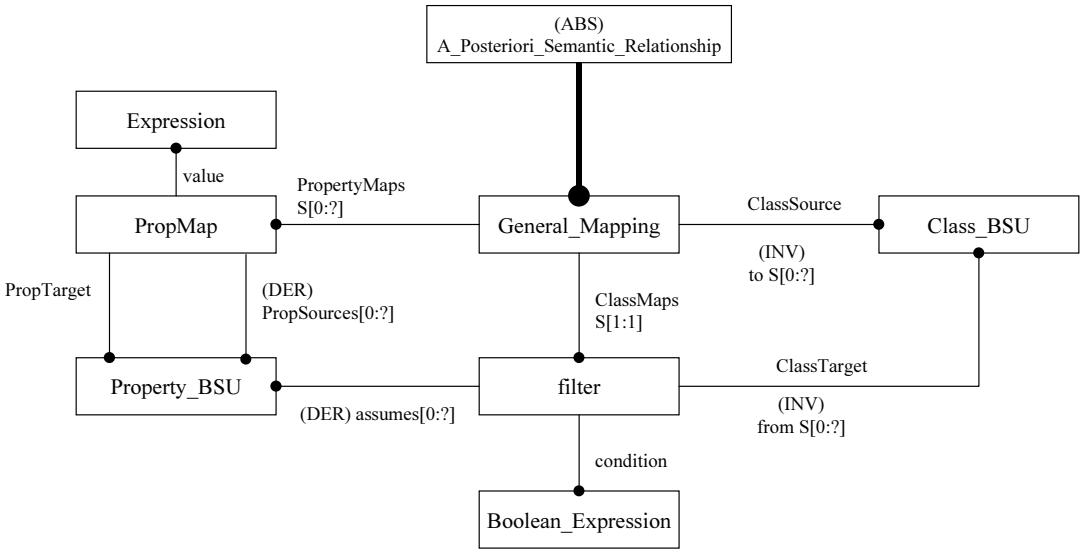

FIG. 6.10 – Relation General_Mapping

instances peuvent être projetées.

- l'entité *Boolean_Expression* ci-dessus modélise une expression booléenne qui est appliquée sur des propriétés caractérisant les instances de la classe source (attribut *assumes*).

3. l'entité *PropMap* représente une relation entre une propriété cible (*PropTarget*) et l'ensemble propriétés sources (*PropSources*) (une relation entre propriétés de type $1 : n$). Cette relation est interprétée par une expression (entité *Expression*) qui permet de calculer la valeur de la propriété cible en fonction des propriétés sources.

Remarquons qu'une classe source peut être mise en correspondance avec n classes cibles à travers n *general_mapping*. La composition de ces n correspondances présente une relation de type $1 - n$. Inversement, plusieurs classes sources peuvent être mappées à une classe cible. Cela présente une relation de type $n - 1$. Afin de respecter l'hypothèse $H1$ dans la section 5.3, la condition suivante doit être satisfaite :

- $\exists map_1, map_2 \in general\_mapping : map_1.ClassSource = map_2.ClassSource$, et
- $\forall x \in map_1.ClassSource (= map_2.ClassSource)$:
  $\neg(\bar{x}$ [21] $\in map_1.ClassMap.ClassTarget \wedge \bar{x} \in map_2.ClassMap.ClassTarget)$.

Exemple 14 *La figure 6.11 illustre un exemple du format d'échange (fichier physique) d'une instance PropMAP qui représente la correspondance "nom entier"* $\rightarrow^{1:2}\{$*nom, prénom*$\}$ *présentée dans la figure 6.2. L'instance PropMAP considérée (#90) lie la propriété source (#25) et l'expression de concaténation (#91). La dernière est appliquée sur les deux variables (#93,#100) qui sont représentées sémantiquement par deux propriétés sémantiques (#94, #101) dont les propriétés associées sont les deux propriétés sources : #58, #62.*

[21] $\bar{x}$: la projection de l'instance x dans l'ontologie cible.

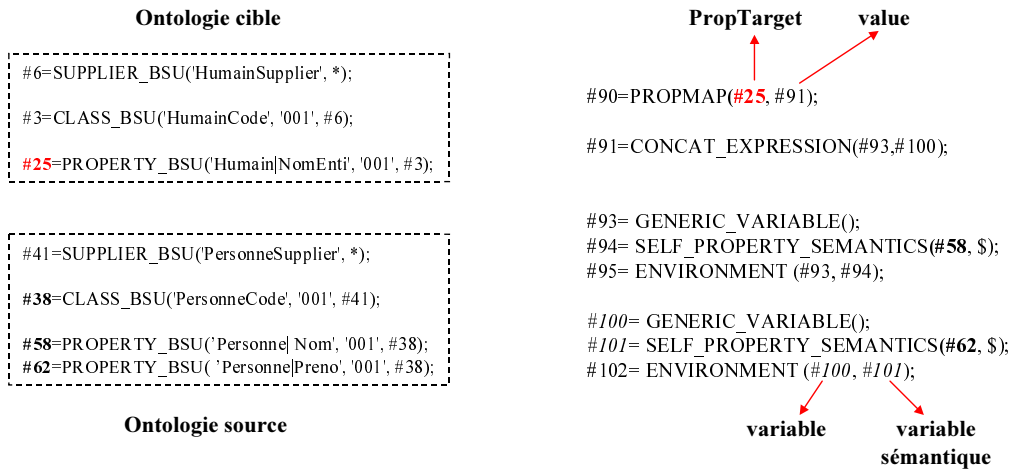

FIG. 6.11 – exemple du format d'échange d'une instance de l'entité *PropMAP*.

Dans la section suivante, nous allons présenter notre implémentation concernant l'intégration des BDBOs par *réification des correspondances entre ontologies*.

6.6 Mise en oeuvre

Notre modèle des relations a été implémenté dans le langage *EXPRESS*. Puis il a été compilé par *ECCO*.

En exploitant ce modèle, une application permettant d'une part de définir (manuellement) les correspondances entre deux ontologies, d'autre part d'intégrer des BDBOs référençant ces ontologies est en cours. L'implémentation de cette application consiste en deux modules séparés :

1. l'interface utilisateur qui devrait permettre (i) de parcourir les deux ontologies à articuler, (ii) de saisir les classes correspondantes et les expressions sur leurs propriétés, et les prédicats de filtre éventuellement (iii) de vérifier la correction syntaxique des expressions saisies, et (iv) de sauvegarder la correspondance sous forme d'un modèle. Ce module est implémenté dans JAVA en utilisant le modèle des relations à travers les APIs générées par ECCO. La figure 6.12 représente l'interface graphique implémentée.
2. l'intégration des BDBOs par réification des correspondances entre ontologies qui est implémentée selon l'algorithme présenté dans la section 4. Ce module, validé dans l'environnement ECCO, sera détaillé dans cette section.

Nous allons d'abord présenter l'approche que nous avons utilisée pour l'évaluation des expressions qui est considérée comme la partie plus importante du module d'intégration, et ensuite quelques fonctions fondamentales dans notre implémentation.

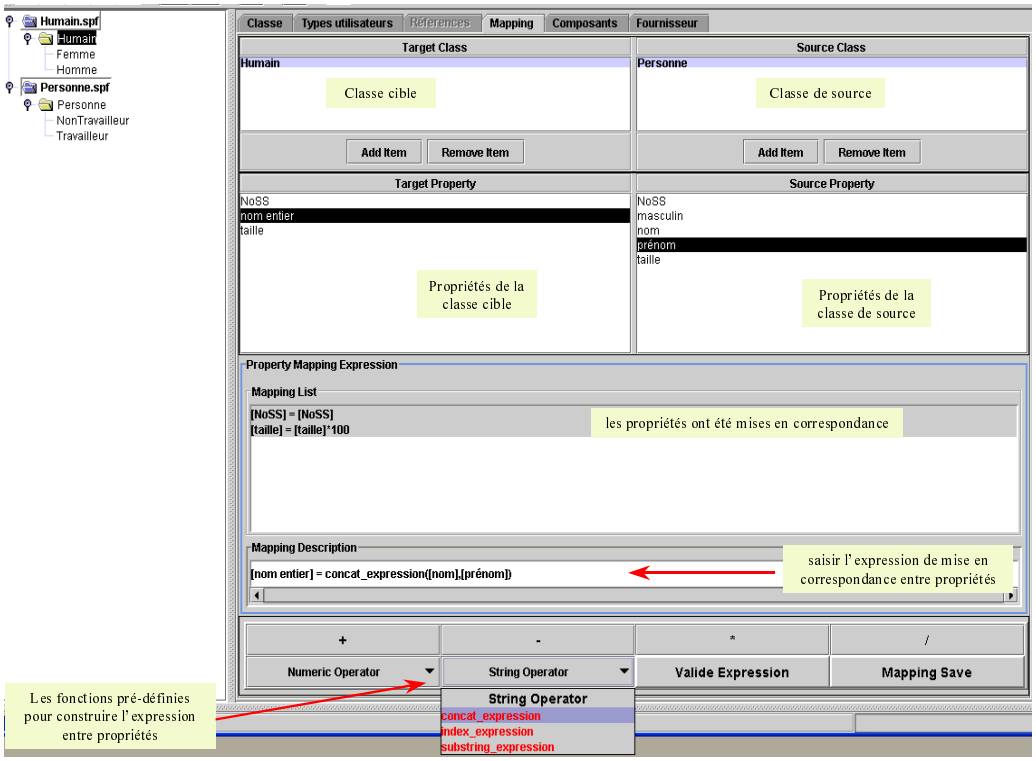

FIG. 6.12 – Interface graphique pour définir la correspondance entre deux ontologies

6.6.1 Evaluation des expressions

Pour réaliser l'évaluation d'une expression, trois approches sont possibles [32] :

Auto évaluation des expressions. Cette approche est basée sur la programmation événementielle avec le langage EXPRESS. Elle consiste à embarquer dans chaque opérateur d'expression sa propre fonction d'évaluation. Cela peut se réaliser en complétant les entités représentant des expressions par des attributs dérivés.

Évaluation par programmes compilables générés. La deuxième approche d'évaluation d'expressions méta-programmées est fondée sur la génération de programmes compilables représentant une traduction des méta programmes, représentant la connaissance procédurale, dans un langage compilable (Ada, C++, Java, ...) ou interprété (PL/SQL, JavaScript, ...). Le(s) programme(s) généré(s) seront ensuite compilé(s) et/ou exécuté(s).

Évaluation générique des expressions. L'évaluation générique est une des approches possibles pour l'évaluation des méta-modèles. Elle est basée sur l'utilisation d'un seul programme générique pouvant évaluer n'importe quel type d'expression. Ce programme peut être écrit dans n'importe quel langage de programmation pouvant accéder à une base de données EXPRESS (à des instances de méta-modèles modélisés en EXPRESS).

Cet accès est réalisé soit directement (cas d'un langage utilisant les notations EXPRESS : EXPRESS-C par exemple), soit à travers une API (Application Programming Interface) (cas d'un langage différent d'EXPRESS : C++, JAVA, ...).

```
FUNCTION evaluation_expression (expr : expression, ins: dic_class_instance ): simple_value;
LOCAL
        type_of_ex : SET OF STRING := TYPEOF(ex);
        res : simple_value;
END_LOCAL;

IF ( (expression_schema+'.GENERIC_LITERAL') IN type_of_ex ) THEN
    RETURN (expr.the_value);
END_IF;

IF ( (expression_schema+'.VARIABLE') IN type_of_ex ) THEN
    res:=get_value_of_variable (expr, ins);
END_IF;

IF ( (expression_schema +'.UNARY_GENERIC_EXPRESSION') IN type_of_ex ) THEN
    res := get_value_of_binary_generic_expression (expr,ins);
END_IF;
IF ( (expression_schema +'.BINARY_GENERIC_EXPRESSION') IN type_of_ex ) THEN
    res := get_value_of_unary_generic_expression (expr ,ins);
END_IF;

IF ( (expression_schema +'.MULTI_ARITY_GEN_EXPRESSION') IN type_of_ex ) THEN
    res := get_value_multi_arity_gen_expression (expr,ins);
END_IF;

RETURN(res);
END_FUNCTION;
```

↓

```
(****** MULTI_ARITY_GEN_EXPRESSION ********)
FUNCTION get_value_multi_arity_gen_expression  expr : expression, ins: dic_class_instance ) :  simple_value;
LOCAL
        type_of_ex : SET OF STRING  := TYPEOF(ex);
        res : simple_value;
END_LOCAL;
…
IF ( (expr_schema+'.PLUS_EXPRESSION') IN type_of_ex ) THEN
    res := expr.operands[1];
    REPEAT i:=2 TO SIZEOF(expr.operands);
        res := res * evaluation_expression(expr.operands[i]);
    END_REPEAT;
END_IF;
…
RETURN(res);
END_FUNCTION;
```

FIG. 6.13 – exemple d'évaluation d'une expression par une fonction générique

Ce programme générique parcourt l'arbre représentant les expressions en ordre post-fixé. L'évaluation commence par les feuilles et remonte jusqu'à la racine. Ce programme est exécuté d'une manière récursive. Le programme d'évaluation peut être une fonction, par exemple, constituée de plusieurs blocs permettant d'évaluer chacun un type précis d'expression (*div_expression*,*sin_function*,*concat_expression*, ...).

Notre implémentation dans cette thèse suit la troisième approche en utilisant EXPRESS-C. La figure 6.13 donne un exemple d'une partie d'une fonction générique permettant d'évaluer la *plus_expression*. Les opérandes d'une expression peuvent être elles-mêmes des expressions (un arbre). La fonction générique est donc appelée d'une manière récursive jusqu'au renvoi final des résultats de l'expression à évaluer. Cette fonction reçoit en paramètre d'entrée une expression et une instance de classe sur laquelle l'expression est évaluée (cette instance fournit les valeurs pour les variables figurant dans l'expression). Elle renvoie une valeur générique, de type *simple_value*, qui est le résultat de l'évaluation de cette expression. Le type *simple_value* est une union de types de base : numériques, de chaîne de caractères, et booléennes.

Nous allons présenter dans la partie suivante quelques fonctions que nous avons effec-

tuées sous EXPRESS-C.

6.6.2 Quelques fonctions implémentées

L'algorithme d'intégration des BDBOs par réification des correspondances entre ontologies que nous présentons dans ce chapitre a été validé dans l'environnement ECCO.

```
• All_GeneralMapping := récupérer tous les instances de type General_Mapping;
• REPEAT FOREACH map IN All_GeneralMapping;
   /* identifier la classe de source et celle cible */
   csource := map.ClassSource ;
   ccible := map.filter.ClassTarget;
  /* calculer la population et le schéma de la classe de source */
   popSource := get_Pop(csource);
   schSource  := get_Sch(csource);
  /* vérifier si ccible est une classe de base ⇒ si non: créer une table des instances dont le schéma Sch = ˇ pour ccible */
  IF NOT EXISTS (ccible.referenced_by[1]) THEN
     init_table (ccible);
  END_IF;
  /* mettre à jour le schéma des instances (SchCible) de la classe ccible avec les propriétés ¸ Applic(ccible),
              mais qui ne sont pas dans SchCible et peuvent être valuées par les propriétés dans schSource */
  update_Schema_with_valuableProp (ccible, schSource, sub);
  /* intégrer les instances ¸ popSource, qui satisfaient la condition de filtre*/
  REPEAT FOREACH  ins  IN  popSource;
     IF ( condition_checking (ins, map.filter.condition) ) THEN
        /*  projeter l 'instance ins dans la classe cible */
         projection_ClassInstance (ins, ccible, map);
     END_IF;
  END_REPEAT;
• END_RPEAT;
```

FIG. 6.14 – Algorithme d'intégration par réification des correspondances entre ontologies

Nous présentons, dans la figure 6.14, l'algorithme d'implémentation pour l'intégration des instances lorsque leurs classes (la classe de source et celle cible) sont articulées par $General\_Mapping$.

Nous citons ci-dessous quelques fonctions importantes qui ont été implémentées :

- $get\_Pop(c_1)$ permettant de calculer la population de la classe c_1 dans la source qui comporte les populations des classes subsumées. Cette population est la population *virtuelle* de c_1 (notons que la population virtuelle est celle matérialisée si la classe est une classe de base et n'a aucune sous-classe.).
- $get\_Sch(c_1)$ permettant de calculer le schéma de la classe c_1 dans la source. Ce calcul est implémenté selon l'équation 4.1 du chapitre 4. Nous voulons ici assurer que toutes propriétés figurant dans le schéma de la classe de source sont valuées.
- $evaluation\_expression(expr, ins)$ permettant d'évaluer d'une façon générique l'expression $expr$ avec les valeurs des variables qui sont celles des propriétés, associées à ces variables, fournie à l'instance ins. Cette fonction a été abordée dans la partie

```
Procedure insert_ClassInstance_withExpression (ins : dic_class_instance, tab: class_extension, map: general_mapping);
…
/* insérer dans la table tab une nouvelle instance dont les propriétés sont à «null», cette nouvelle instance est l'image de
l'instance ins dans la table tab*/
new_ins := insert new ClassInstance(tab) ;
Image(ins) := new_ins;
/* calculer la valeur des propriétés de new_ins*/
REPEAT FOREACH p IN get_schema(tab) ;
  /* récupérer l'expression correspondant à p: s'il n'existe pas l'expression: pmap = null */
  pmap := get_prop_mapping(p,map);
  IF EXISTS (pmap) THEN
    /* vérifier si p peut être valuée par les propriétés dans ins */
    IF (is_able_to_be_valued(map, ValuedProp(ins)) THEN
      /* évaluer l'expression correspondant p */
      val := evaluation_expression( pmap.value, ins);
      /* si p n'est pas une association: insérer la nouvelle valeur val pour la propriété p de l'instance new_ins*/
      IF NOT (val = NULL)  THEN
          update_value(new_ins,p,val);
      ELSE
      /* si p est  une association alors:
        - si la valeur de p (get_value(p,ins)) existe dans Image alors: insérer l'image de la valeur de p pour new_ins,
        - si non: insérer p et la valeur de p dans une relation (  P_s x I_s), nommée AssociationList. */
        IF EXIST ( Image(get_value(p,ins) ) THEN
              update_value(new_ins,p, Image(get_value(p,ins)));
        ELSE
              AssociationList (p):= get_value(p,ins);
             /* cette relation et la relation Image seront utilisées dans la fin du processus de l'intégration
                                  pour remplacer la valeur NULL de toute propriété qui est une association
                                                          par l'image de AssociationList (p) dans Image */
        END_IF;
      END_IF;
    END_IF;
  END_IF;
…
END_REPEAT;
….
End_Procedure;
```

FIG. 6.15 – Algorithme pour projeter une instance source dans une table cible

précédente (voir la figure 6.13). Notons que si une propriété de l'instance *ins* est une association, cette fonction rend la valeur $NULL$.

- $get\_value\_of\_variable(var, ins)$ permettant de récupérer la valeur d'une propriété de l'instance *ins*. Cette propriété représente également la sémantique de la variable *var*.
- $update\_Schema\_with\_valuableProp(ccible, schSource, map)$ permettant de mettre à jour le schéma de la table des instances réelles [22] de la classe *ccible*. Cette table est complétée par les propriétés dans $Applic(ccible)$, qui ne sont pas dans $Sch(ccible)$. Supposons que p est une nouvelle propriété à ajouter dans $Sch(ccible)$, p doit satisfaire les conditions suivantes :
  1. $p \in Applic(ccible)$,
  2. $p \notin Sch(ccible)$,
  3. $\exists pmap \in \{map.PropertyMaps\}$ pour que :
     $(pmap.PropTarget = p) \wedge (pmap.PropSources \subset schSource)$.
     Cette condition est vérifiée par la fonction $is\_able\_to\_be\_valued(map, schSource)$. Elle permet de vérifier si la valeur d'une propriété quelconque p de la classe cible (*ccible*) peut être évaluable en utilisant la correspondance entre ontologies (*map*) et les informations disponibles sur la source (*schSource*).
- $insert\_ClassInstance(ins, tab, map)$ permettant de projeter l'instance *ins* dans la table des instances *tab*, avec la relation *map* représentant une correspondance entre la classe de source de l'instance *ins* et la classe de base de la table *tab*. Une partie de l'implémentation de cette "Procedure" est présentée dans la figure 6.15.

6.7 Conclusion

Dans ce chapitre, nous avons défini la problématique de l'intégration de données par réification des correspondances entre ontologies. Dans cette approche, l'articulation entre ontologie locale et partagée n'est pas définie directement dans l'ontologie locale, mais elle est définie à l'extérieur, sous forme d'un modèle contenant des opérateurs eux-même réifiés. Ceci soit parce que l'ontologie n'était pas connue *a priori* par les sources, soit parce que la source souhaite s'articuler avec plusieurs ontologies. L'intégration consiste donc à définir le modèle d'articulation des ontologies, puis à intégrer les données.

Dans le contexte d'étude de cette thèse, les ontologies sont supposées être basées sur le même modèle d'ontologie. D'autre part, nous ne visons pas ici à la découverte automatique des relations sémantiques entre l'ontologie cible et celle de source. Nous nous concentrons plutôt sur la représentation des relations et les mécanismes d'intégration.

Nous avons d'abord discuté les correspondances entre classes et les correspondances entre propriétés. Des exemples étudiés, il résulte que, dans le cas le plus général, les cardi-

[22] les instances dont la classe de base est *ccible*

nalités des relations entre classes sont de type $n : m$. Il en est de même des correspondances entre propriétés. Nous avons alors remarqué que les deux relations de type $n : m$ pouvaient en fait se décomposer en relations de type $n : 1$ ou $1 : m$. Concernant les relations entre classes, une instance d'une classe source n'est représentée qu'une fois dans l'univers cible. Nous avons donc proposé de représenter les relations entre classes sous forme d'une relation entre chaque classe source et l'ensemble des classes cibles qui peuvent lui correspondre. La migration des données de l'ontologie source vers l'ontologie cible n'étant possible que lorsque l'on peut définir les opérateurs algébriques pour exprimer les correspondances. Nous avons proposé de définir par des prédicats portant sur les instances de la classe source les conditions d'appartenance à chacune des classes cibles. De même, une propriété cible ne se calculant qu'une fois, nous avons proposé d'associer à chaque propriété cible une fonction de calcul portant sur l'ensemble des propriétés sources qui peuvent contribuer à sa valeur. Cela nous a permis alors de formaliser la correspondance entre source et cible à l'aide d'un triplet constituée (1) d'une fonction qui fait correspondre à chaque classe source l'ensemble de ses classes cibles potentielles, (2) un ensemble de prédicats d'appartenance à chaque classe cible pour les instances de chaque classe source et (3) d'un ensemble de fonctions calculant chaque propriété cible pour chaque classe source. Nous avons montré que ce modèle permettant alors de calculer formellement toute l'information exprimée dans l'ontologie source qu'il est possible d'exprimer dans l'ontologie cible, c'est ce que nous avons appelé la *projection*.

Un des objectifs de cette représentation étant de pouvoir échanger les correspondances en tant que modèle, le problème s'est alors posé à la fois de réifier les différents éléments du modèle de correspondance, et en particulier les fonctions et opérateurs. Nous avons alors proposé de reprendre un modèle de réification d'expression EXPRESS déjà proposé dans la littérature, et nous avons construit sur cette base un modèle de correspondance entre ontologies qui nous semble susceptible d'exprimer (sous réserve éventuellement d'ajouter des opérateurs algébriques non pris en compte) tout type de "mapping" entre ontologies, ou tout au moins tout ce que nous avons pu imaginer dans notre domaine d'application.

Dans les domaines où il est possible que plusieurs ontologies concurrentielles apparaissent, l'approche que nous proposons ici permet alors de publier des données par rapport à une ontologie particulière, tout en leur associant des modèles de correspondances ("mapping") vers l'ensemble des ontologies du domaine. Tout utilisateur de ces données quelle que soit la ou les ontologies de domaine sur lesquelle(s) il s'appuie pourra alors intégrer les données automatiquement simplement en sélectionnant les correspondances qu'il veut appliquer.

Chapitre 7

Conclusion et Perspectives

Ce chapitre conclut nos travaux sur l'intégration automatique des bases de données à base ontologique avec une approche *a priori*. Nous y récapitulons notre contribution et présentons quelques perspectives.

7.1 Conclusion

L'intégration de sources de données, hétérogènes, autonomes et évolutives pose à la fois des problèmes structurels et des problèmes sémantiques. Les travaux concernant l'hétérogénéité structurelle sont relativement anciens et ont abouti à diverses approches permettant de la traiter dans le contexte des fédérations de bases de données ou des multi-bases de données. L'hétérogénéité sémantique, par contre, reste la plus importante difficulté. Plusieurs systèmes récents d'intégration de sources de données hétérogènes utilisent les ontologies afin de résoudre les conflits sémantiques. Les principales limitations de ces systèmes sont soit leur absence d'automatisation en l'absence d'une ontologie partagée, soit l'absence d'autonomie (ou la faible autonomie) des sources dans le cas où une ontologie partagée est utilisée, et, dans tous les cas, l'absence de prise en compte des besoins d'évolution asynchrone tant des différentes sources de données que des ontologies.

Dans ce travail, nous avons proposé des approches d'intégration à base ontologique de type matérialisé (entrepôt de données) permettant (1) une intégration automatique de sources de données, (2) une grande autonomie des sources et (3) une évolution asynchrone des schémas et des ontologies.

Les différentes contributions de ce travail sont les suivantes.

7.1.1 Proposition d'une nouvelle classification pour les systèmes d'intégration

Pour mieux comprendre les différentes approches et systèmes d'intégration, nous avons présenté un état de l'art. Cette étude nous a amené à proposer une nouvelle classifica-

tion des systèmes d'intégration, selon trois critères orthogonaux : (1) la représentation de données intégrées (approche médiateur ou entrepôt), (2) le sens de mise en correspondance entre schéma global et schéma local (GaV ou LaV) et (3) la nature du processus d'intégration (manuelle, semi automatique et automatique).

Nous avons de plus discuté les approches GaV et LaV. La plupart des études considèrent que l'approche LaV est flexible à l'ajout de nouvelles sources, tandis que l'approche GaV ne passe pas à l'échelle. En fait, cela dépend de l'hypothèse faite concernant les concepts contenus dans le schéma global, et les concepts existants (auxquels le système d'intégration doit donner accès) dans la nouvelle source. Si l'on fait les hypothèses usuelles dans le contexte LaV, à savoir : (1) une nouvelle source ne doit pas modifier le schéma global, (2) elle ne contient aucun nouveau concept, ou bien le schéma global ne représentera que partiellement la nouvelle source, alors l'effet d'adjonction d'une nouvelle source d'une approche GaV est tout à fait *similaire* à l'approche LaV. Elle passe même à l'échelle probablement beaucoup mieux du point de vue de la réponse aux requêtes.

7.1.2 Approche d'intégration par articulation a priori d'ontologies

Nous avons présenté une approche d'intégration à base ontologique. Cette approche suppose que chaque source de données possède sa propre ontologie (Base de données à base ontologique ou BDBO) et que l'ontologie locale s'articule a priori avec une ou des ontologie(s) partagée(s). Notre approche est matérialisée et le résultat de l'intégration est lui même une BDBO. Afin de garder l'autonomie de chaque source, cette dernière peut définir sa propre hiérarchie de classes, et, si besoin, rajouter les propriétés qui n'existent pas dans l'ontologie partagée.

Dans cette approche, l'intégration de BDBOs devient une loi de composition interne. Trois opérateurs algébriques d'intégration sont proposés : *FragmentOnto*, *ExtendOnto*, et *ProjOnto*. Dans FragmentOnto, on suppose que les ontologies locales des bases de données sont directement extraites de l'ontologie partagée (chaque ontologie locale est un fragment de l'ontologie partagée. C'est l'hypothèse souvent faite dans les systèmes d'intégration avec une ontologie partagée). Dans ProjOnto, chaque source définit sa propre ontologie. Par contre l'ontologie locale référence l'ontologie partagée en respectant la condition SSCR (Smallest Subsuming Class Reference). Dans ce scénario, on souhaite néanmoins intégrer les instances de chaque source comme des instances de l'ontologie partagée. Dans le dernier scénario, chaque ontologie locale est définie comme dans le scénario ProjOnto, mais l'on souhaite enrichir automatiquement l'ontologie partagée. Ensuite toutes les instances de données sont intégrées, sans aucune modification, au sein du système intégré.

Pour chaque scénario, nous avons d'abord décrit l'opérateur algébrique qui lui correspond, son algorithme d'intégration et enfin ses applications réelles possibles. L'ensemble de ces propositions a été validé sur des exemples issus du commerce électronique professionnel (B2B) et de l'intégration des catalogues de composants industriels (projet PLIB).

7.1.3 Gestion de l'évolution asynchrone des sources

Dans le contexte d'une intégration à base ontologique, les évolutions concernent à la fois les ontologies (locales et partagées), les schémas et les données. Lorsque les ontologies évoluent de façon incohérente entre les différentes sources, l'intégration automatique devient *impossible*. Pour résoudre ce problème, nous avons proposé d'encadrer les évolutions d'ontologies autorisées par le *principe de continuité ontologique*. Ce principe stipule qu'une ontologie ne peut infirmer un axiome qui se trouvait vérifié dans une version antérieure de l'ontologie. Ce principe nous a alors permis de proposer un mécanisme, dit *de version flottante*, qui permet de ne conserver dans l'entrepôt qu'une seule version de chaque classe et de chaque propriété, appelées versions courantes (en fait, la plus grande version connue). Ceci permet de consolider entre les versions courantes toutes les relations ayant existé entre les différentes versions des différents concepts. Nous avons ensuite discuté la possibilité d'historisation des instances figurant dans les différentes sources et ré-introduites lors de chaque rafraîchissement de l'entrepôt. Afin d'identifier les instances ontologiques bien qu'elles puissent, au cours du temps, être décrites par des ensembles différents de propriétés, nous avons proposé la notion de *clé sémantique*. Il s'agit d'un sous-ensemble de propriétés applicables d'une classe qui sont suffisantes pour en identifier les instances tout au long de l'évolution. Cette notion nous a alors permis de proposer une méthode d'historisation des instances qui évite toute duplication : chaque instance n'est représentée qu'une seule fois, elle consolide toutes les propriétés qui ont existé au cours du temps et deux attributs identifient les versions de classe pour lesquelles l'instance est ou a été valide (instance *multi-versionnée*).

Notre modèle été validé sous ECCO en considérant plusieurs ontologies de domaines, pour lesquelles un ensemble de sources a été défini. Des exemples de tailles significatives ont pu ainsi être traités.

7.1.4 Intégration par réification des correspondances entre ontologies

Il existe des cas où notre approche *a priori* ne peut s'appliquer. Soit qu'une ontologie partagée n'existe pas au moment où certaines sources sont créées. Soit qu'il existe plusieurs ontologies qui se chevauchent. Soit que les correspondances entre ontologies locales et partagées soient plus complexes que celles supportées dans la méthode d'articulation *a priori*. Nous avons proposé, dans ce cas une approche d'intégration. Cette approche consiste à représenter sous forme de modèle, au sein de chaque BDBO les correspondances entre son ontologie et une ou plusieurs ontologies partagées et à réifier, au sein de ces modèles, les opérateurs algébriques de transformation. Dans le cas le plus général, les correspondances entre ontologies, aussi bien au niveau classes qu'au niveau propriétés, les correspondances de type $n : m$ nécessitant des opérateurs algébriques de composition de classes et de compositions de valeurs de propriétés. Nous avons d'abord montré que

ces relations $n : m$ pouvaient s'exprimer simplement sous forme d'ensemble de relations $1 : m$. Puis nous avons proposé un méthode de méta-modélisation avec réification des opérateurs, pour représenter chaque correspondance $1 : n$. Cette approche permet alors de représenter de façon neutre, indépendante de tout langage ou environnement particulier, des correspondances algébriques complexes entre classes et/ou entre propriétés, tout en laissant la possibilité d'intégration automatique des différentes BDBOs. Nous avons montré que cette intégration automatique restait possible dans une approche entrepôt de données.

7.2 Perspectives

De nombreuses perspectives tant à caractère théorique que pratique peuvent être envisagées. Dans cette section nous présentons succinctement celles qui nous paraissent être les plus intéressantes.

7.2.1 de PLIB vers d'autres modèles d'ontologies

Dans cette thèse, le modèle d'ontologie PLIB a joué le rôle de pivot pour tous les algorithmes d'intégration. Il serait intéressant d'explorer d'autres modèles d'ontologies pour valider nos approches d'intégration. D'autres formats d'échange de BDBO que le format d'échange d'un schéma EXPRESS "*fichier physique*" [ISO10303-21 :1994], seraient également à étudier tout particulièrement des formats de type XML plus adaptés aux Web sémantique.

7.2.2 Utilisation simultanée d'ontologies en différents langages

Des modèles d'ontologies assez différents existent actuellement. Il serait intéressant d'explorer comment différents modèles d'ontologies peuvent coopérer dans une perspective d'intégration. En particulier nos approches sont toutes plutôt basées sur des calculs algorithmiques et des transformations algébriques. Il serait intéressant d'étudier comment ces méthodes peuvent coopérer avec des approches fondées sur la programmation logique pour aborder de nouvelles gammes de problèmes d'intégration de base de données.

7.2.3 Intégration à base ontologique dans une optique de médiation

Dans cette thèse, nous avons proposé des approches d'intégration dans une architecture d'entrepôt. Il serait intéressant d'adapter nos solutions dans une architecture de médiation, en mettant en évidence les différentes composantes du médiateur, le traitement de requêtes (réécriture de requêtes et localisation de sources pertinentes, etc.), les techniques d'optimisation de requêtes, et le passage à l'échelle. Tout particulièrement, la méthode

d'intégration par réification des correspondances entre ontologies semble se prêter de façon intéressante à une approche de type médiateur.

Table des figures

Liste des tableaux

Bibliographie

[1] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active xml : Peer-to-peer data and web services integration. *Proceedings of the International Conference on Very Large Databases*, pages 1087–1090, 2002.

[2] A. Aho and J. Ullman. Principles of compiler design. In *Addition Wesley Publishing Company*, 1977.

[3] Y. Aït-Ameur, F. Besnard, P. Girard, G. Pierra, and J. C. Potier. Formal Specification and Metaprogramming in the EXPRESS Language. In *International Conference on Software Engineering and Knowledge Engineering*, pages 181–189, 1995.

[4] Y. Ait-Ameur, G. Pierra, and E. Sardet. Using the express language for metaprogramming. In *Proceeding of the The 3rd International Conference of EXPRESS User Group EUG'95*, 1995.

[5] Y. Ait-Ameur and H. Wiedmer. *Logical resource : Logical model of expressions.* ISO-IS 13584-20. ISO Genève, 1998.

[6] Y. Aklouf, G. Pierra, Y. A. Ameur, and H. Drias. Plib ontology : A mature solution for products characterization in b2b electronic commerce. *Int. J. IT Standards and Standardization Res.*, 3(2) :66–81, 2005.

[7] Y. Arens and C. A. Knoblock. Sims : Retrieving and integrating information from multiple sources. *Proceedings of the International Conference on Management of Data (SIGMOD'1993)*, pages 562–563, May 1993.

[8] B. Bebel, J. Eder, C. Koncilia, T. Morzy, and R. Wrembel. Creation and management of versions in multiversion data warehouse. In *SAC '04 : Proceedings of the 2004 ACM symposium on Applied computing*, pages 717–723, New York, NY, USA, 2004. ACM Press.

[9] L. Bellatreche, D. Nguyen-Xuan, G. Pierra, and H. Dehainsala. Contribution of ontology-based data modeling to automatic integration of electronic catalogues within engineering databases. *To appear in : Computers in Industry Journal*, 2006.

[10] L. Bellatreche, G. Pierra, D. Nguyen-Xuan, and H. Dehainsala. Intégration de sources de données autonomes par articulation a priori d'ontologies. In *Ictes du XXIIème Congrès INFORSID, Biarritz, France*, pages 283–298, 2004.

[11] L. Bellatreche, G. Pierra, D. Nguyen-Xuan, H. Dehainsala, and Y. Ait-Ameur. An a priori approach for automatic integration of heterogeneous and autonomous databases. *International Conference on Database and Expert Systems Applications (DEXA'04)*, (475-485), September 2004.

[12] D. Beneventano and S. Bergamaschi. The momis methodology for integrating heterogeneous data sources. In *IFIP World Computer Congress, Toulouse, France.*, August 2004.

[13] D. Beneventano, S. Bergamaschi, S. Castano, A. Corni, R. Guidetti, G. Malvezzi, M. Melchiori, and M. Vincini. Information integration : The MOMIS project demonstration. In *The VLDB Journal*, pages 611–614, 2000.

[14] S. Bergamaschi, S. Castano, S. D. C. di Vimercati, S. Montanari, and M. Vincini. Exploiting schema knowledge for the integration of heterogeneous sorces. In *Convegno Nazionale Sistemi di Basi di Dati Evolute (SEBD98), Ancona, Italy*, pages 103–120, 1998.

[15] M. S. Bernardo Magnini, Luciano Serafini. Linguistic based matching of local ontologies. In *Proceedings of Meaning Negotiation Workshop at AAAI*, 2002.

[16] T. Berners-Lee, J.Hendler, and O. Lassila. The semantic web. *Scientific American*, pages 36–43, Mai 2001.

[17] P. Bernstein. Applying model management to classical meta data problems. *in Proceedings of the 2003 CIDR Conference*, 2003.

[18] P. Bernstein, L. M. Haas, M. Jarke, E. Rahm, and G. Wiederhold. Panel : Is generic metadata management feasible ? *vldb*, pages 660–662, 2000.

[19] P. Bernstein, A. Y. Havely, and R. A. Pottinger. A vision of managament of complex models. *in SIGMOD Record*, 29(4) :55–63, 2000.

[20] M. Body, M. Miquel, Y. Bédard, and A. Tchounikine. A multidimensional and multiversion structure for olap applications. In *DOLAP '02 : Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, pages 1–6, New York, NY, USA, 2002. ACM Press.

[21] Y. Breitbart, A. Silberschatz, and G. R. Thompson. Reliable transaction management in a multidatabase system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 215–224, 1990.

[22] A. Bullig, T. Schnadhorst, and W. Wilkes. Mapping of product dictionaries and corresponding catalog data. In *Proceedings of the 10th ISPE International Conference on Concurrent Engineering (CE'2003)*, 2003.

[23] J. Charlet, B. Bachimont, and R. Troncy. Ontologies pour le web sémantique. *La revue I3 : Information - Interaction - Intelligence*, Vol5 n°1, 2005.

[24] S. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The tsimmis project : Integration of heterogeneous information sources. *Proceedings of the 10th Meeting of the Information Processing Society of Japan*, pages 7–18, Mars 1994.

[25] P. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1) :9–36, 1976.

[26] S. Chen, B. Liu, and E. A. Rundensteiner. Multiversion-based view maintenance over distributed data sources. *ACM Transactions on Database Systems*, 4(29) :675–709, December 2004.

[27] P. Coad and Yourdon. *Object-oriented analysis.* Prentice-Hall, Englewood Cliffs, 1991.

[28] S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker : Ontology based access to distributed and semi-structured information. In *DS-8*, pages 351–369, 1999.

[29] H. Dehainsala, S. Jean, D. Nguyen-Xuan, and G. Pierra. Ingénierie dirigée par les modèles en express : un exemple d'application. In *1ére journée d'Ingénierie dirigée par les modèles*, pages 155–167, 2005.

[30] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to map ontologies on the semantic web. In *Proceedings of WWW2002*, May 2002.

[31] A. Doan, J. Madhavan, P. Domingos, and A. Y. Halevy. Ontology matching : A machine learning approach. *Handbook on Ontologies*, pages 385–404, 2004.

[32] M. EL-Hadj-Mimoune. *Contribution à la modélisation explicite et à la représenta-*

tion des données de composants industriels : application au modèle PLIB. PhD thesis, Université de Poitiers, 2004.

[33] J. Euzenat, J. Barrasa, P. Bouquet, R. Dieng, M. Ehrig, M. Hauswirth, M. Jarrar, R. Lara, D. Maynard, A. Napoli, G. Stamou, H. Stuckenschmidt, P. Shvaiko, S. Tessaris, S. van Acker, I. Zaihrayeu, and T. L. Bach. D2.2.3 : State of the art on ontology alignment. *Technical report, NoE Knowledge Web project delivable*, 2004.

[34] D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, and P. Patel-Schneider. Oil : An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, pages 38–44, 2001.

[35] F. François Goasdoué, V. Lattès, and M. C. Rousset. The use of carin language and algorithms for information integration : The picsel system. *International Journal of Cooperative Information Systems (IJCIS)*, 9(4) :383–401, December 2000.

[36] M. R. Genesereth, A. M. Keller, and O. M. Duschka. Infomaster : an information integration system. In *ACM SIGMOD International Conference on Management of Data*, pages 539–542, 1997.

[37] C. Goh, S. Bressan, E. Madnick, and M. D. Siegel. Context interchange : New features and formalisms for the intelligent integration of information. *ACM Transactions on Information Systems*, 17(3) :270–293, 1999.

[38] G.-L. G. Gomez. *Construction automatisée de l'ontologie de systèmes médiateurs. Application à des systèmes intégrant des services standards accessibles via le Web.* PhD thesis, Université Paris XI Orsay, 2005.

[39] T. Gruber. A translation approach to portable ontology specification. *Knowledge Acquisition*, 5(2) :199–220, 1995.

[40] N. Guarino and C. A. Welty. A formal ontology of properties. In *Knowledge Acquisition, Modeling and Management*, pages 97–112, 2000.

[41] N. Guarino and C. A. Welty. Ontological analysis of taxonomic relationships. *in Proceedings of 19th International Conference on Conceptual Modeling (ER'00)*, pages 210–224, October 2000.

[42] N. Guarino and C. A. Welty. Evaluating ontological decisions with ontoclean. *Communications of the ACM*, 45(2) :61–65, 2002.

[43] M.-S. Hacid and C. Reynaud. L'intégration de sources de données. *La revue I3 : Information - Interaction - Intelligence*, Vol5 n°1, 2005.

[44] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The stanford data warehousing project. *IEEE Quarterly Bulletin on Data Engineering ; Special Issue on Materialized Views and Data Warehousing*, 18(2) :41–48, 1995.

[45] J. Heflin. *Towards the Semantic Web : Knowledge Representation in a Dynamic, Distributed Environment.* PhD thesis, University of Maryland, College Park., 2001.

[46] J. Heflin. Owl web ontology language use cases and requirements. *http ://www.w3.org/TR/webont-req/*, 2004.

[47] J. Heflin and J. Hendler. Dynamic ontologies on the web. *American Association for Artificial Intelligence*, pages 443–449, 2000.

[48] M. Jarke and Y. Vassiliou. Data warehouse quality design : A review of the dwq project. In *Invited Paper, 2nd Conference on Information Quality. Massachusetts Institute of Technology, Cambridge*, 1997.

[49] S. Jean. Langage d'exploitation de base de données ontologiques. *Mémoire pour l'obtention du DEA T3IA, Université de Poiters*, juin 2004.

[50] Y. Kalfoglou and M. Schorlemmer. Ontology mapping : the state of the art. *Knowl. Eng. Rev.*, 18(1) :1–31, 2003.

[51] N. Karam, S. Benbernou, L. Debrauwer, M.-S. Hacid, and M. Schneider. Les logiques de description pour le tri sémantique de documents sur le web. *Ingénierie des Systèmes d'Information*, 10(2) :69–89, 2005.

[52] M. Kifer, G. Lausen, and J. Wu. Logical foundations of objectoriented and frame-based languages. *Journal of the ACM*, pages 741–843, 1995.

[53] M. Klein. *Change Management for Distributed Ontologies*. PhD thesis, Amsterdam University, 2004.

[54] W. J. Labio, Y. Zhuge, J. L. Wiener, H. Gupta, H. García-Molina, and J. Widom. The WHIPS prototype for data warehouse creation and maintenance. In *Proceedings of SIGMOD*, pages 557–559, 1997.

[55] B. T. Le, R. Dieng-Kuntz, and F. Gandon. On ontology matching problems - for building a corporate semantic web in a multi-communities organization. In *6e Conference Internationale en Systeme d'Information d'Entreprise (ICEIS)*, pages 236–243, 2004.

[56] J. Leukel, V. Schmitz, and F.-D. Dorloff. B2b e-procurement beyond mro ? In *Proceedings of the 6th International Conference on Electronic Commerce Research (ICECR-6)*, pages 493–500, October 23-26 2003.

[57] A. Y. Levy, A. Rajaraman, and J. J. Ordille. The world wide web as a collection of views : Query processing in the information manifold. *Proceedings of the International Workshop on Materialized Views : Techniques and Applications (VIEW'1996)*, pages 43–55, June 1996.

[58] N. Lumineau. *Organisation et Localisation de Données Hétérogènes et Répar-ties sur un Réseau Pair-à-Pair*. PhD thesis, Université Pierre et Marie Curie - Paris VI, 2005.

[59] N. Lumineau, A. Doucet, and S. Gançarski. Thematic schema building for mediation-based peer-to-peer architecture. *Electronic Notes in Theoretical Computer Science*, 150(2) :21–36, 2006.

[60] J. Madhavan, P. Bernstein, P. Domingos, and A. Halevy. Representing and reasoning about mappings between domain models. *in Proceedings of the 18th National Conference on Artificial Intelligence (AAAI'02)*, pages 80–86, 2002.

[61] F. Manola, E. Miller, W3C, and B. McBride. Rdf primer. *http ://www.w3.org/TR/2004/REC-rdf-primer-20040210/*, Juin 2005.

[62] D. L. McGuinness and F. Harmelen. Owl web ontology language overview. *W3C Recommendation*, 10 February 2004.

[63] E. Mena, V. Kashyap, A. P. Sheth, and A. Illarramendi. OBSERVER : An approach for query processing in global information systems based on interoperation across pre-existing ontologies. In *Conference on Cooperative Information Systems*, pages 14–25, 1996.

[64] P. Merle, C. Gransart, and J.-M. Geib. Corba-script and corba web : A generic object-oriented dynamic environment upon corba. In *In Proceedings of TOOLS Europe 96, Paris, France*, pages 97–112, 1996.

[65] A. Miller. Wordnet : A lexical database for english. *Communications of the ACM*, 38(11) :39–41, November 1995.

[66] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *VLDB '93 : Proceedings of the 19th International Conference on Very Large Data Bases*, pages 120–133, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[67] M. Minsky. Matter, mind and models. *International Federation of Information Processing Congress*, 1 :45–49, 1965.

[68] M. Minsky. A framework for representing knowledge. *in Winston PE, ed. The Psychology of computer vision. New-York : McGraw-Hill*, 1975.

[69] P. Mitra, G. Wiederhold, and M. Kersten. A graph-oriented model for articulation of ontology interdependencies. *Lecture Notes in Computer Science*, 1777 :86+, 2000.

[70] A. Napoli. Une brève introduction aux logiques de descriptions. *Cours de la Logique de Description*, 2005.

[71] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, pages 36–56, 1991.

[72] N. Noy and M. Klein. Ontology evolution : Not the same as schema evolution. *Knowledge and Information Systems*, 5, 2003.

[73] N. Noy and M. Musen. The prompt suite : Interactive tools for ontology merging and mapping. *Technical report, SMI, Stanford University*, 2002.

[74] N. F. Noy and M. A. Musen. Ontology versioning in an ontology management framework. *IEEE Intelligent Systems*, 19(4) :6–13, 2004.

[75] OntoBroker®. How to write f-logic programs. In *A Tutorial for the Language F-Logic*. Copyrighted by Ontoprise GmbH, November 2004.

[76] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *11th Conference on Data Engineering*, pages 251–260. IEEE Computer Society, 1995.

[77] C. Parent and S. Spaccapietra. Intégration de bases de données :panorama des problèmes et des approches. *Ingénierie des systèmes d'information*, 4(3), 1996.

[78] G. Pierra. Bibliothèque neutre de composants standard pour la cao : le projet européen cad/lib. *Revue internationale de CFAO et d'Infographie*, 4(2) :35–53, 1989.

[79] G. Pierra. A multiple perspective object oriented model for engineering design. *in New Advances in Computer Aided Design & Computer Graphics*, pages 368–373, 1993.

[80] G. Pierra. Un modèle formel d'ontologie pour l'ingénierie, le commerce électronique et le web sémantique : Le modèle de dictionnaire sémantique plib. *Journées Scientifique WEBSEMANTIQUE, Paris*, 2002.

[81] G. Pierra. Context-explication in conceptual ontologies : The plib approach. *Proceedings of CE'2003, Special track on Data Integration in Engineering, Madeira, Portugal, edited by R. Jardim-Gonçalves and J. Cha and A. Steiger-Garçao*, pages 243–254, July 2003.

[82] G. Pierra. Context-explication in conceptual ontologies : Plib ontologies and their use for industrial data. *to appear in Journal of Advanced Manufacturing Systems (JAMS)*, 2006.

[83] G. Pierra, Y. Ait-Ameur, and E. Sardet. *Logical Model For Parts Libraries*. ISO-DIS 13584-24. ISO Genève, 594 pages, 1998.

[84] G. Pierra, H. Dehainsala, Y. Aït-Ameur, and L. Bellatreche. Base de données à base ontologique : principe et mise en oeuvre. *Ingénierie des systèmes d'information*, 2005.

[85] A. D. Preece, K. ying Hui, W. A. Gray, P. Marti, T. J. M. Bench-Capon, D. M. Jones, and Z. Cui. The KRAFT architecture for knowledge fusion and transformation. *Knowledge Based Systems*, 13(2-3) :113–120, 2000.

[86] D. Price. Standard Data Access Interface. *ISO-CD 10303-22*, 1995.

[87] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. In *Proceedings of the International Conference on Very Large Databases*, 2001.

[88] C. Reynaud and G. Giraldo. An application of the mediator approach to services over the web. *Special track "Data Integration in Engineering, Concurrent Engineering (CE'2003)*, pages 209–216, July 2003.

[89] J. F. Roddick. Dynamically changing schemas within database models. *Aust. Comput. J.*, 23(3) :105–109, 1991.

[90] J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7) :383–393, 1995.

[91] D. C. Rogozan. Gestion de l'évolution d'une ontologie : méthodes et outils pour un référencement sémantique évolutif fondé sur une analyse des changements entre versions de l'ontologie. *Rapport de recherche en informatique cognitive*, 2005.

[92] E. A. Rundensteiner, A. Koeller, and X. Zhang. Maintaining data warehouses over changing information sources. *Commun. ACM*, 43(6) :57–62, 2000.

[93] E. Sardet. *Intégration des approches modélisation conceptuelle et structuration documentaire pour la saisie, la représentation, l'échange et l'exploitation d'informations. Application aux catalogues de composants industriels.* PhD thesis, Université de Poitiers, 1999.

[94] D. Schenck and P. Wilson. Information modelling the express way. *Oxford University Press*, 1994.

[95] E. Sciore. Versioning and configuration management in an object-oriented data model. *The VLDB Journal*, 3(1) :77–106, 1994.

[96] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3) :183–236, 1990.

[97] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. In *Journal on Data Semantics IV*, pages 146–171. Lecture Notes in Computer Science, December 2005.

[98] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *J. Data Semantics IV*, pages 146–171, 2005.

[99] G. Staub and M. Maier. Ecco tool kit - an environnement for the evaluation of express models and the development of step based it applications. *User Manual*, 1997.

[100] L. Stojanovic. *Methods and Tools for Ontology Evolution.* PhD thesis, Karlsruhe University, 2004.

[101] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. In *EKAW*, pages 285–300, 2002.

[102] N. Stojanovic and L. Stojanovic. Usage-oriented evolution of ontology-based knowledge management systems. In *CoopIS/DOA/ODBASE*, pages 1186–1204, 2002.

[103] S. Suwanmanee, D. Benslimane, P.-A. Champin, and P. Thiran. Wrapping and integrating heterogeneous databases with OWL. In *ICIES*, 2005.

[104] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis, and S. Skiadopoulos. A generic and customizable framework for the design of etl scenarios. *Information Systems*, 30(7) :492–525, 2005.

[105] T. Vögele, H. S., and S. G. Buster- an information broker for the semantic web. *Knstliche Intelligenz*, 3 :31–34, July 2003.

[106] P. R. S. Visser, M. Beer, T. Bench-Capon, B. M. Diaz, and M. J. R. Shave. Resolving ontological heteroge-

neity in the kraft project. *10th International Conference on Database and Expert Systems Applications (DEXA'99)*, pages 668–677, September 1999.

[107] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-based integration of information - a survey of existing approaches. *Proceedings of the International Workshop on Ontologies and Information Sharing*, pages 108–117, August 2001.

[108] H.-C. Wei and E. Ramez. Study and comparison of schema versioning and database conversion techniques for bitemporal databases. *Sixth International Workshop, TIME-99 Proceedings*, pages 88–98, 1999.

[109] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The strobe algorithme for multisource warehouse consistency. In *PDIS Conference, Miami Beach-Florida*, pages 146–157, 1996.

Résumé

Les principales limitations des systèmes d'intégration de données existants sont soit leur absence d'automatisation en l'absence d'une ontologie partagée, soit l'absence d'autonomie des sources dans le cas où une ontologie partagée est utilisée, et, dans tous les cas l'absence de prise en compte des besoins d'évolution asynchrone tant des différentes sources de données que des ontologies. Dans cette thèse, nous proposons d'abord une approche d'intégration permettant d'assurer une intégration entièrement automatique de sources de données, tout en laissant à chacune des sources une autonomie significative tant au niveau de sa structure qu'au niveau de son évolution. Cette approche suppose que chaque source contienne à la fois sa propre ontologie et les relations sémantiques qui l'articulent a priori avec une ou des ontologie(s) partagée(s). Nous proposons ensuite un modèle qui permet une gestion automatique de l'évolution asynchrone de notre système d'intégration à base ontologique. L'hypothèse fondamentale autour de ce modèle, appelée principe de continuité ontologique, stipule qu'une évolution d'une ontologie ne peut infirmer un axiome antérieurement vrai. Nous proposons enfin dans la dernière partie de cette thèse, une méthode d'intégration par réification des correspondance entre ontologies. Cette approche permet de traiter de façon neutre, indépendante de tout langage ou environnement des correspondances arbitrairement complexes entre ontologies. L'ensemble de ces propositions a été validé sur des exemples issus du commerce électronique professionnel (B2B) et de l'intégration des catalogues de composants industriels (projet PLIB).

Mots-clés: Système d'intégration, Modélisation à base ontologique, Autonomie, Evolution asynchrone, Articulation d'ontologies, PLIB.