Le modèle DOAN (DOcument ANnotation Model) Modélisation de l'information complexe appliquée à la plateforme Arisem Kaliwatch Server

Nicolas Dessaigne

UNIVERSITÉ DE NANTES

École Polytechnique

ÉCOLE DOCTORALE

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX

2005

Thèse de Doctorat de l'Université de Nantes

Spécialité : INFORMATIQUE

Présentée et soutenue publiquement par

Nicolas DESSAIGNE

le 22 décembre 2005

au Laboratoire d'Informatique de Nantes-Atlantique, Université de Nantes

Le modèle DOAN (DOcument ANnotation Model) Modélisation de l'information complexe appliquée à la plateforme Arisem Kaliwatch Server

Jury

Président	:	Pr. Anne DOUCET	LIP6, Paris
Rapporteurs	:	Claude CHRISMENT, Pr.	IRIT, Toulouse
		Jean-Marie PINON, Pr.	LIRIS, Lyon
Examinateurs	:	Noureddine MOUADDIB, Pr.	LINA, Nantes

Directeur de thèse : Pr. José MARTINEZ

Laboratoire : LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE. 2, rue de la Houssinière, F-44322 NANTES CEDEX 3

N° ED 0366-248

LE MODÈLE DOAN
(DOCUMENT ANNOTATION MODEL)
MODÉLISATION DE L'INFORMATION COMPLEXE APPLIQUÉE À LA PLATEFORME ARISEM KALIWATCH SERVER

The DOAN Model
(DOcument ANnotation Model)
Complex Information Modelling applied to the Arisem Kaliwatch Server platform

Nicolas DESSAIGNE

favet neptunus eunti

Université de Nantes

Nicolas DESSAIGNE

Le modèle DOAN (DOcument ANnotation Model) Modélisation de l'information complexe appliquée à la plateforme Arisem Kaliwatch Server

xix+192 p.

Ce document a été préparé avec LaTeX2$_\varepsilon$ et la classe `these-LINA` version 0.92 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe `these-LINA` est disponible à l'adresse :

`http://www.sciences.univ-nantes.fr/info/Login/`

Impression : Thèse.tex – 8/3/2006 – 19:41

Révision pour la classe : $Id: these-LINA.cls,v 1.3 2000/11/19 18:30:42 fred Exp

Résumé

Nous présentons dans cette thèse le modèle DOAN (*DOcument ANnotation Model*), destiné à répondre aux besoins de modélisation de la société Arisem.

Arisem est éditeur de logiciels dans le domaine de la gestion des connaissances. La plateforme que l'entreprise propose s'inscrit dans le cycle collecte / analyse / diffusion de l'information. À partir de données de nature hétérogène et d'origines diverses (ex. : Internet, intranet, base de données), elle procède à différentes analyses (ex. : classement automatique, extraction de concepts émergents), afin de fournir des informations synthétiques et utiles à l'utilisateur. Partant de cette problématique, nous avons identifié trois besoins principaux pour le modèle : *expressivité*, *flexibilité* et *performances*.

Dans le cadre de cette thèse, nous avons développé un modèle basé sur le paradigme d'agrégation de facettes, qui permet aux concepteurs de décrire des données complexes, hétérogènes et évolutives. Au-delà de la simple notion de document, il rend possible la représentation d'objets métiers, comme par exemple des annotations ou des arbres de catégorisation. Complété par un système de types riches et par la capacité d'exprimer des contraintes entre facettes, ce modèle nous permet de répondre aux besoins d'expressivité et de flexibilité.

Nous proposons d'autre part un algorithme permettant de traduire les éléments du modèle DOAN en une implémentation relationnelle. Une fois le modèle instancié, les accès en modification sont contrôlés à l'aide de procédures stockées afin de garantir la consistance des données. Les accès en consultations sont en revanche effectués directement à l'aide de requêtes SQL. Les concepteurs peuvent ainsi faire des requêtes à la fois complexes et performantes, tirant parti au maximum des possibilités du système de gestion de bases de données. Cette approche permet une montée en charge importante et répond aux besoins de performances.

Mots-clés : Modélisation d'information, traduction relationnelle, gestion des connaissances.

Abstract

This thesis introduces the DOAN model (*DOcument ANnotation Model*), which aims at answering the modelling needs of the company Arisem.

Arisem is a software vendor acting on the knowledge management market. It offers a platform that processes data during the collection, analysis and dissemination steps of the information flow. As it works on heterogeneous data coming from various sources (e.g., internet, intranet, databases), it performs several analyses (e.g., automatic classification, extraction of emergent concepts), in order to provide synthetic information to the user. From this problematic, we have identified three main needs for the model: *expressivity*, *flexibility* and *performance*.

Within the context of this thesis, we have developed a model based on the facet aggregation paradigm. It enables designers to describe complex, heterogeneous and evolving data. Beyond the simple notion of document, it permits the representation of business objects, such as annotations or classification trees. Supplemented by a rich type system and the ability to express constraints between facets, this model enables us to answer the needs of expressiveness and flexibility.

On the other hand, we propose an algorithm able to translate DOAN elements into a relational implementation. Once the model is instantiated, update accesses are controlled by using stored procedures in order to ensure data consistency. Read accesses are in return performed directly by using SQL queries. Designers are thus able to use complex queries without sacrificing performance, exploiting database management systems functionalities. This approach scales well and answers the needs of performance.

Keywords: Information modelling, relational translation, knowledge management.

Remerciements

Je voudrais exprimer ma plus sincère gratitude au Professeur José Martinez qui a su diriger cette thèse avec rigueur et patience, et au Professeur Noureddine Mouaddib qui m'a incité à faire cette thèse et aidé dans mes premières démarches.

Je suis profondément reconnaissant aux Professeurs Claude Chrisment et Jean-Marie Pinon de m'avoir fait l'honneur d'évaluer ce manuscrit de thèse. Je tiens aussi à remercier le Professeur Anne Doucet d'avoir accepté de présider mon jury de thèse.

Un grand merci à la société Arisem pour m'avoir accueilli durant toute la durée de ce travail. Je suis particulièrement reconnaissant à Olivier Spinelli avec qui j'ai eu un grand plaisir à travailler et qui fut une source d'inspiration inépuisable. Je remercie également Alain Garnier pour ses suggestions utiles et son support constant. Merci à tous mes autres collègues pour leur disponibilité et leurs encouragements.

Lors de mes séjours réguliers à Nantes, j'ai eu le plaisir de découvrir un laboratoire agréable et accueillant. Je remercie notamment l'association des jeunes chercheurs *Login* pour m'avoir permis de m'intégrer rapidement. J'adresse aussi mes plus sincères remerciements à Anthony, Marina et la petite Alexane Mottier pour l'amitié qu'ils m'ont constamment témoignée lorsqu'ils m'accueillaient sous leur toit.

Je tiens à remercier pour leurs corrections et conseils tous ceux qui ont relu ce manuscrit. En particulier, merci à Anne Gibert pour son aide précieuse et ses encouragements durant toute la rédaction, à Audrey Liese dont le regard critique et le soutien furent inestimables, et à Laurent Dosdat dont les remarques étaient toujours des plus pertinentes.

Je remercie enfin tous les gens que je n'ai pas cités, amis, famille, ou chercheurs, et qui ont contribué, d'une façon ou d'une autre, à la réussite de cette aventure.

Avant-propos

Cette thèse s'est déroulée dans le cadre d'un contrat CIFRE avec la société *Arisem*[1], éditeur de logiciels dans le domaine de la *gestion des connaissances*. Au-delà de l'apport scientifique de ce travail, le contexte industriel et particulièrement « nouvelle économie » fut riche en enseignements quant à la gestion d'une entreprise, tant dans un environnement euphorique que dans une situation de crise.

En effet, la thèse a débuté en 2001 alors que, ce que l'on a ensuite nommé la « bulle » de la nouvelle économie, était à son apogée. Quelques mois plus tard, son éclatement a mis en difficulté de nombreux éditeurs dont Arisem. Après avoir été une *start-up* très en vue[2], l'entreprise a subi de plein fouet les conséquences de son positionnement et de la baisse des investissements des grands comptes. Après un premier rachat par un investisseur indépendant en mars 2002, elle a réussi à survivre mais a vu ses ambitions fortement réduites. Deux ans plus tard, la société était de nouveau en difficulté et c'est le groupe Thales[3] qui s'est porté acquéreur cette fois-ci. La puissance financière de son nouvel actionnaire permet aujourd'hui à Arisem d'envisager l'avenir avec plus de sérénité.

Ces difficultés et rebondissements dans la vie de la société ont bouleversé la stratégie qui a dû s'adapter aux contraintes économiques. Elles ont eu une influence non négligeable sur cette thèse. Tout d'abord, la dimension fortement multimédia des premières orientations fut rapidement abandonnée au profit d'un recentrage sur le modèle lui-même. D'autre part, l'idée initiale de remplacer intégralement le modèle précédent fut abandonnée pour l'approche plus pragmatique d'intégration progressive des résultats.

Cette expérience m'a aussi donné l'occasion de travailler sur des sujets connexes à cette étude qui ne sont pas détaillés dans mon mémoire. J'ai ainsi eu l'occasion de participer activement au projet K-Mining [32], module d'extraction de connaissances à partir de corpus, qui fut un succès commercial d'Arisem. Le projet de recherche AB-STRAT [34], dont l'objectif est la création de résumés à partir de listes de diffusion, fut également l'occasion de mettre en pratique quelques résultats. Enfin, mon rôle de doctorant dans la société m'a donné l'opportunité de la représenter dans des événements liés au Web sémantique [33, 45].

[1] www.arisem.com

[2] Arisem a été lauréat du concours « e-business 2001 » en avril ; son logiciel OpenPortal4U a été élu « Meilleur logiciel de l'année 2001 » au salon IDT/Net en mai ; et la société a remporté le prix de « l'Entreprise Croissance et Emploi » en juin lors de la 3ème édition de « Tremplin Entreprises », organisée par le Sénat en partenariat avec l'ESSEC.

[3] www.thalesgroup.com

Sommaire

PARTIE I

Introduction et analyse des besoins

CHAPITRE 1

Introduction

Cette thèse s'est déroulée dans le cadre d'un contrat CIFRE avec la société Arisem, éditeur de logiciels. La solution proposée par Arisem est destinée au marché de la gestion de la connaissance*, plus communément appelé* KM (Knowledge Management).

L'approche que nous avons suivie au cours de ce travail a été de proposer un modèle en vue de répondre à la problématique métier proposée par la société. Après avoir évoqué cette problématique, nous donnerons dans cette introduction un aperçu des résultats obtenus. Nous présenterons ensuite la structure de la thèse afin d'en guider sa lecture.

1.1 Problématique

Alors que nous étions confrontés il y a quelques années à une problématique de disponibilité de l'information, nous sommes aujourd'hui submergés par un volume de données supérieur à nos propres capacités d'analyse. Des documents sont produits massivement et distribués dans des formats électroniques variés. Ils contiennent de plus en plus d'informations, structurées ou semi-structurées, textuelles ou multimédias (ex. : auteurs, dates, mots-clés, résumés, images, etc.). La gestion de ces connaissances est ainsi devenue un enjeu majeur pour les entreprises.

Afin de gérer cette vaste quantité de données et de pouvoir l'exploiter, des solutions de gestion des connaissances sont apparues. Arisem en particulier propose une plateforme logicielle capable de lire des documents (HTML, PDF, Office, etc.) et d'en extraire une information sémantique. Elle propose plusieurs actions sur les documents : analyse plein texte, analyse sémantique, extraction d'informations structurées, analyse de corpus, classement automatique, annotations par l'utilisateur, etc. Un document peut être analysé à la fois par des outils autonomes et par des outils en ligne, à la fois par des outils automatiques et par des humains, à la fois par Arisem et par ses intégrateurs ou clients.

Dans le cadre de cette problématique de gestion des connaissances, nous pouvons identifier trois principaux besoins :

- Une importante expressivité afin de pouvoir représenter des structures complexes ;
- Une importante flexibilité afin de pouvoir gérer des données hétérogènes, les analyser de multiples façons, et supporter leurs évolutions ;
- Le support de fortes montées en charge afin de pouvoir traiter des volumes de documents très importants.

Face à ces besoins, le modèle de données actuel de la solution éditée par Arisem a atteint ses limites. Et si les modèles que l'on trouve aujourd'hui dans la littérature répondent généralement à une partie de ces besoins, aucun ne donne une réponse satisfaisante à leur ensemble. Cette constatation a motivé le travail dont cette thèse est l'aboutissement.

1.2 Résultats obtenus

Les besoins liés à la problématique ont conduit à la création du modèle DOAN (DOcument ANnotation Model). S'inspirant de différentes familles de modèles existants (modèle Entité-Association et ses extensions, modèle à rôles, modèles à frames), il introduit la notion de « facette » pour représenter l'information.

Alors qu'une entité agrège plusieurs facettes pour représenter les données la constituant, des contraintes entre les facettes permettent d'exprimer les lois du domaine de l'application. Ce mécanisme permet de simuler en partie le fonctionnement d'un système à objets sans en supporter les rigidités.

Le description des facettes bénéficie d'un système de type riche, proposant notamment des structures complexes très utiles aux concepteurs d'applications de gestion des connaissances (ex. : listes, arbres, graphes).

L'évolutivité du modèle est quant à elle assurée par des opérations d'attachement et de détachement de facettes. Le type même d'un objet peut ainsi évoluer au cours du temps.

Si le modèle répond aux besoins d'expressivité et de flexibilité, nous nous appuyons sur son implémentation pour répondre aux besoins de performances. Pour cela, nous avons retenu les Systèmes de Gestion de Bases de Données (SGBD) relationnels comme cible prioritaire. Leur maturité, leur large adoption dans l'industrie, ainsi que leur tenue en charge, en adéquation avec la quantité très importante de données à traiter, en font aujourd'hui le meilleur candidat pour l'implémentation. Toutefois, la généricité du modèle rend possible une implémentation différente, par exemple au-dessus d'un SGBD XML ou à objets.

Pour réaliser cette implémentation, nous avons développé un algorithme transformant récursivement les types complexes de DOAN en relations d'un SGBD. Afin de s'assurer de la consistance des données, les accès en modification se font grâce à des « méthodes » (implémentées *via* des procédures stockées) dont les effets sont maîtrisés. Ensuite, le système s'en remet au langage d'interrogation, en l'occurrence SQL, pour ce qui est des accès en lecture. C'est la meilleure façon d'obtenir des performances élevées, les requêtes pouvant être réellement ensemblistes.

Les tests de performances effectués montrent que la demande initiale de tenue en charge est respectée.

Ces contributions ont fait l'objet d'une présentation à EJC 2005 [35][1].

1.3 Plan de la thèse

Ce document est divisé en quatre parties. La première, dont cette introduction constitue le chapitre initial, apporte les éléments contextuels nécessaires à la compréhension de l'ensemble de la thèse. Le chapitre 2 détaille notamment la problématique, et conclut sur les besoins qui ont conduit à ce travail.

La seconde partie est consacrée à l'état de l'art des modèles d'information. Avant de les détailler, nous utilisons le chapitre 3 pour introduire les principaux concepts de modélisation. Nous avons fait le choix de consacrer un chapitre entier à ce récapitulatif afin de permettre une meilleure compréhension des apports de chaque proposition et finalement de les comparer. Les chapitres suivants sont l'occasion d'étudier chaque famille de modèle :

- Les modèles Entité-Association (E/A) et E/A étendus (y compris UML, ORM et RDF) dans le chapitre 4 ;
- Les modèles à base de *rôles* dans le chapitre 5 ;

[1] *15th European-Japanese Conference on Information Modelling and Knowledge Bases.*

- Les modèles à base de *frames* (issus de l'IA en général), incluant les logiques de descriptions et les modèles à base d'« ontologies », dans le chapitre 6.

Le chapitre 7 conclut cette partie avec une comparaison des approches entre elles et avec les besoins exprimés en première partie.

La troisième partie de ce mémoire contient nos principales propositions. Le chapitre 8 y détaille le modèle DOAN (*DOcument ANnotation Model*). Nous y introduisons en particulier ses principaux concepts, les contraintes qu'il permet d'exprimer et son système de types. Nous montrons aussi comment il permet de répondre aux besoins d'expressivité et de flexibilité exprimés au chapitre 2. Le chapitre suivant concerne l'implémentation de DOAN. Après avoir justifié le choix du relationnel, nous y détaillons la transformation du modèle DOAN dans le modèle relationnel. Le chapitre 10 nous permet ensuite de montrer comment l'implémentation choisie répond aux besoins de performances et de support de la charge.

Enfin, en dernière partie, nous concluons en récapitulant les contributions de la thèse et en les mettant en perspective.

À la suite du corps de la thèse, un ensemble de trois annexes se focalise sur certains aspects techniques de l'implémentation. Le cycle de vie des entités et notamment les opérations d'attachement et de détachement sont présentées en annexe A. L'annexe B, quant à elle, décrit un mécanisme que peut utiliser un concepteur ou un programmeur pour interagir avec les événements du système et adapter le noyau à ses besoins. Enfin, l'annexe C introduit une approche permettant de sécuriser les applications basées sur notre proposition.

CHAPITRE 2

Problématique

Cette thèse s'intéresse au domaine de la gestion des connaissances, et plus particulièrement aux problèmes de modélisation d'applications d'intelligence économique. La société Arisem nous a fournit les cas concrets permettant de valider la problématique et les besoins induits que nous présentons dans ce chapitre.

Après avoir introduit le domaine, nous nous intéresserons aux différentes étapes du flux de l'information dans une solution d'intelligence économique. La section 2.3 nous permettra ensuite d'étudier les deux niveaux d'information que sont les documents et les objets métier. Nous observerons ensuite les contraintes de sécurité et de performances. La conclusion nous donnera enfin l'occasion de résumer les besoins liés à la problématique qui ont inspiré ces travaux.

2.1 La gestion des connaissances

La *gestion des connaissances*, plus communément appelée *KM* pour *Knowledge Management*, est un domaine en plein essor dans l'industrie. Sa définition la plus répendue est celle de Malhotra [66] :

Définition 1. *La gestion des connaissances répond aux problèmes critiques d'adaptation, de survie et de compétence d'une organisation face aux changements discontinus d'environnements. En particulier, elle englobe les processus qui cherchent à mettre en synergie les capacités à traiter les données des technologies de l'information, et les capacités à créer et innover des êtres humains.*[1]

Après une ère de pionniers et d'expériences pilotes, puis une prise de conscience progressive de l'importance du KM par les décideurs, nous sommes à l'aube de l'ère de la maturité. L'enquête réalisée par Knowings[2] en 2003 sur la vision des dirigeants en matière de KM [63] montre la grande variété des domaines auxquels la gestion des connaissances s'applique. L'amélioration de la productivité est la priorité et passe d'abord par la préservation et la meilleure exploitation du capital de « connaissances métier » existant dans l'entreprise. Nous retrouvons ainsi le KM dans des thèmes tels que l'exploitation des retours d'expériences, la capitalisation et le partage des savoirs, mais aussi la veille concurrentielle et technologique. Capitaliser, diffuser, collaborer, rechercher, tels sont les principaux besoins des entreprises en matière de KM.

Alors que certains outils sont encore peu utilisés (ex. : la gestion des compétences), d'autres sont devenus des standards (ex. : les outils de veille ou la gestion électronique de documents). La solution

[1] Définition originelle : *« Knowledge Management caters to the critical issues of organizational adaption, survival and competence in face of increasingly discontinuous environmental change. Essentially, it embodies organizational processes that seek synergistic combination of data and information processing capacity of information technologies, and the creative and innovative capacity of human beings. »*

[2] Knowings est un éditeur de logiciels qui développe des solutions collaboratives de gestion de contenu. Il réalise chaque année une enquête auprès des dirigeants sur leur vision du KM.

Kaliwatch Server [4] répond par exemple à une importante partie du spectre de besoins couverts par le KM : *crawling* et surveillance de sources externes et internes, classification automatique, diffusion ciblée et personnalisée, recherche multilingue, travail collaboratif et recherche d'experts. Elle est basée sur une approche sémantique et multilingue exploitant une base de connaissances métiers [33, 45].

La gestion des connaissances étant un domaine très vaste, nous allons nous concentrer par la suite sur l'aspect *intelligence économique* qui est à l'origine des principaux besoins ayant conduit à cette thèse. Le Rapport du Commissariat au Plan a donné en 1994, dans le rapport Martre, une définition de l'intelligence économique qui sert de référence à l'industrie [26] :

Définition 2. *L'intelligence économique peut être définie comme l'ensemble des actions de recherche, de traitements et de diffusion (en vue de son exploitation) de l'information utile aux acteurs économiques.*

Plus récemment, le gouvernement français a montré son intérêt dans le domaine en commanditant le rapport Carayon [21], puis en prenant des mesures pour empêcher des technologies clés de passer sous contrôle étranger [44, 31].

2.2 Flux de l'information

Reprenant les actions de la définition 2, une application en intelligence économique se caractérise par un traitement de l'information passant par trois étapes principales : la collecte, l'analyse et la diffusion. Les figures 2.1 et 2.2 illustrent ce processus.

2.2.1 Collecte

En règle générale, un « veilleur » ou un analyste dispose de nombreuses sources d'informations qui sont externes (Internet, fils de presse, etc.) ou internes[3] (Intranet, bases de données, messagerie électronique, fichiers locaux, etc.). Cette masse d'information étant gigantesque, sa pertinence globale est extrêmement faible. La première étape dans le flux informationnel consiste donc à collecter dans cette masse l'information intéressante par rapport aux besoins.

Cela se fait généralement avec une méthodologie que l'on pourrait qualifier « d'opportunisme dirigé ». D'un côté, l'analyste s'intéresse de près à des sources précises et identifiées et, d'un autre côté, il s'appuie sur une collecte plus « générale » pour découvrir des informations nouvelles.

L'analyse sémantique permet de ne conserver que l'information correspondant aux besoins exprimés tout en exploitant au mieux le contexte métier de l'utilisateur (exprimé dans la base de connaissances). Une approche *cross-lingue* peut en outre permettre d'exprimer une recherche dans sa langue maternelle et d'obtenir des résultats pertinents dans toutes les langues supportées par la base de connaissances[4].

2.2.2 Analyse

Une fois la collecte terminée, nous disposons d'un sous-ensemble de l'information disponible qui répond à la demande de l'utilisateur[5]. Même s'il est considérablement réduit par rapport à toute l'informa-

[3] L'enquête Knowings [63] met en avant l'importance grandissante donnée aux sources internes à l'entreprise dans les démarches de KM.

[4] Les approches *multilingues* fonctionnent dans plusieurs langues, mais ne renvoient généralement des résultats que dans la langue de la requête. Les approches *cross-lingues* s'appuient sur un langage pivot « conceptuel », indépendant de la langue, pour traiter l'information. Les résultats ne sont donc pas limités à la langue de la requête.

[5] Ce sous-ensemble n'est pas nécessairement complet. Notamment, dès que l'on s'intéresse aux sources présentes dans la Toile, du fait des limitations inhérentes au *crawling* et de la masse considérable de pages disponibles, il est impossible de

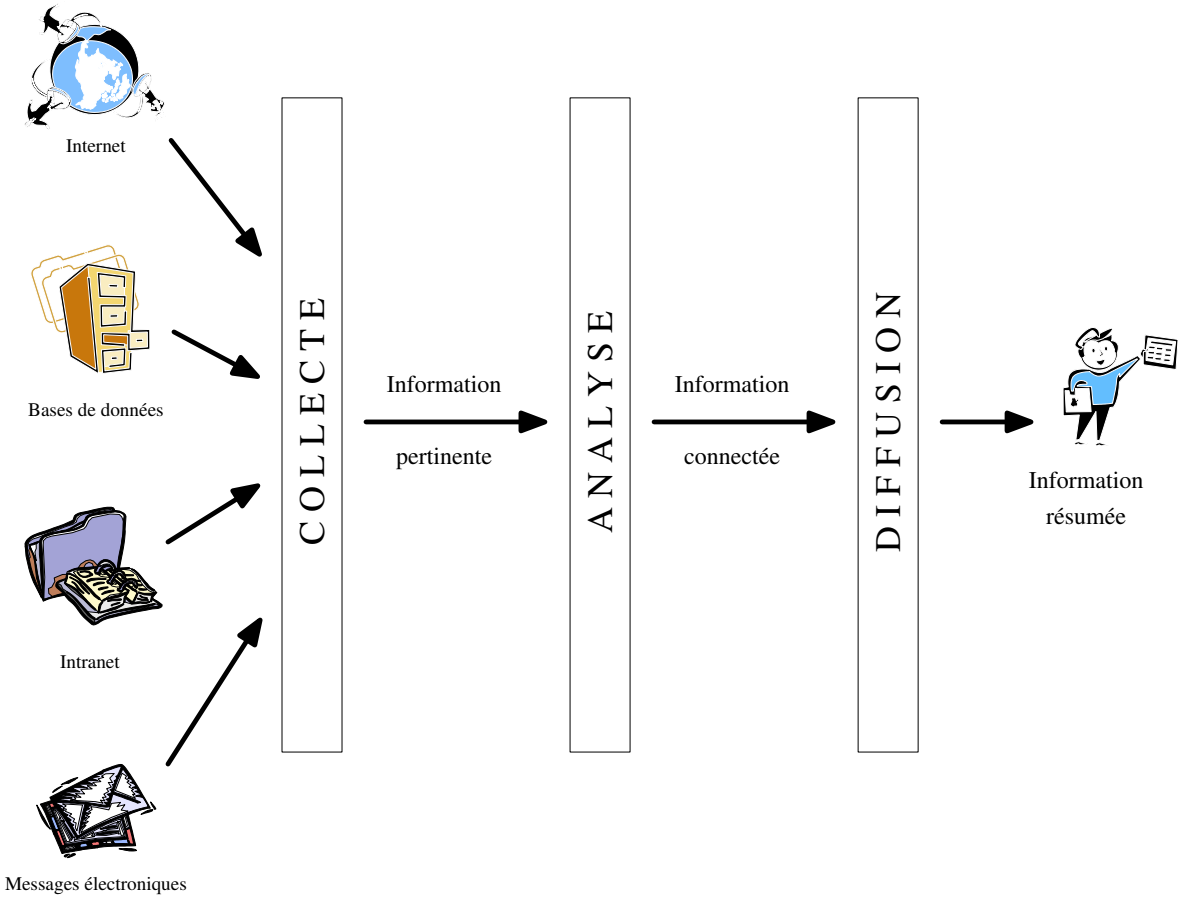

Figure 2.1 – Flux de l'information

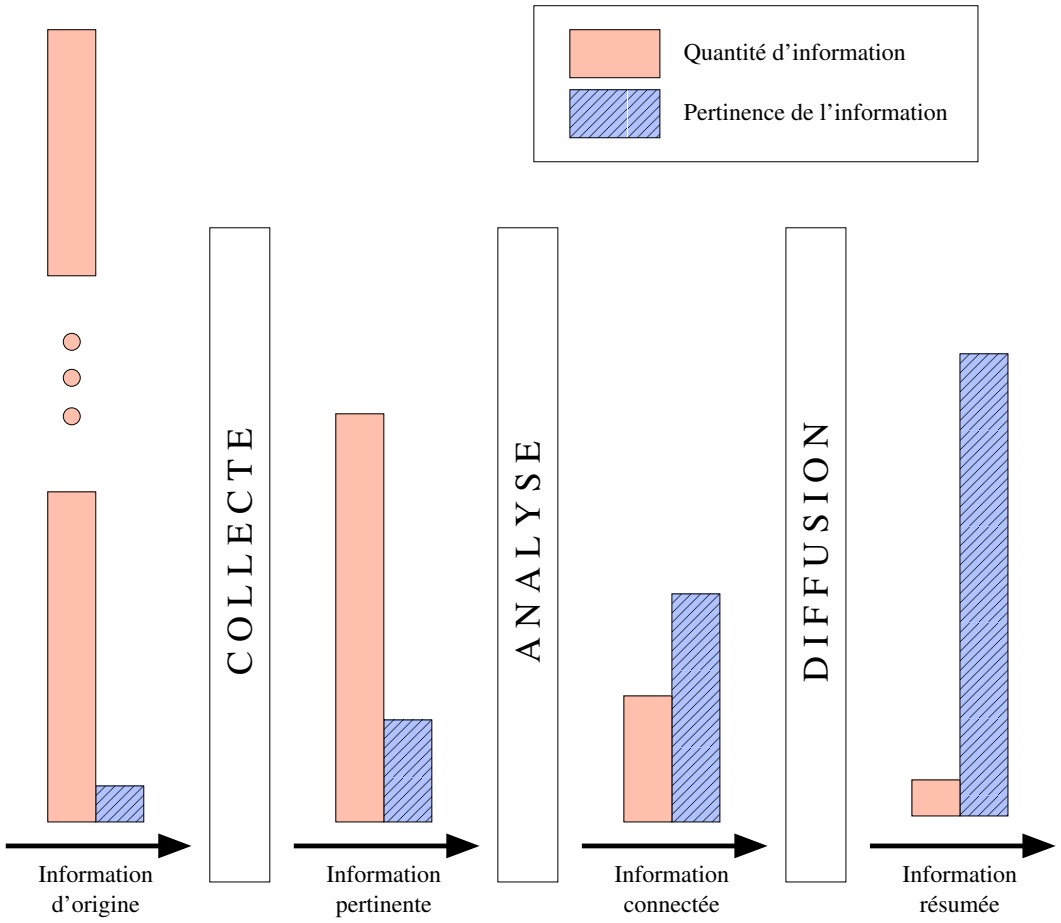

Figure 2.2 – Évolution de la pertinence de l'information en fonction du traitement

tion disponible, ce sous-ensemble représente cependant une masse d'information encore trop importante pour être traitée manuellement.

Pour aider l'utilisateur dans sa tâche, nous allons donc entrer dans une phase d'analyse plus fine du contenu des documents rapatriés. L'analyse des documents n'est cependant pas un processus simple et standard.

En effet, la nature et la structure des documents nécessitent de combiner différentes approches. L'analyse d'un texte peut, par exemple, s'attacher à associer des concepts aux mots qui sont lus, alors que l'analyse d'une image va se concentrer sur sa texture et ses couleurs ainsi que sur l'extraction de concepts. Alors que dans une lettre d'information, nous allons nous intéresser à chaque article indépendamment des autres, l'analyse d'une étude de marché doit pouvoir faire ressortir une thématique générale tout en fournissant une analyse fine de chacune de ses parties.

Plus important encore, les besoins de l'utilisateur changent en fonction de ses intérêts et du contexte et peut conduire à des analyses et résultats très variés. Le KM touche de nombreux domaines [63] dans lesquels les données n'ont pas toujours le même sens et sont exploitées différemment. Un utilisateur peut être intéressé non pas par un ou plusieurs documents à l'état brut, mais par leurs titres et dates, leurs résumés, les différences avec des versions plus anciennes, ou encore la liste des personnes ou organisations qui y sont citées. Ces informations sont autant de *points de vue* sur les documents.

Le principal outil proposé dans cette étape du flux documentaire est la classification automatique des documents dans une ontologie. En consultant seulement certains nœuds de classement, l'utilisateur peut accéder à l'information de façon contextuelle. Des croisements entre ontologies lui permettent de visualiser rapidement la répartition des documents, et il peut encore affiner sa recherche en effectuant des requêtes sémantiques sur les documents.

Un module d'analyse de corpus, par exemple *K-Mining* [32], complète généralement le classement. Il permet, pour un ensemble de documents, d'extraire les thèmes dominants et les entités nommées (personnes, entreprises, etc.). L'analyse des co-occurrences des termes extraits vient étoffer l'information disponible (cela va, par exemple, nous permettre de découvrir l'existence d'un accord de partenariat entre un concurrent et un fournisseur).

D'autres approches viennent compléter ces possibilités. Nous voulons en particulier être capable de :

- *Exploiter la structure des documents*. Dans le cas de lettres d'information (*newsletters*) par exemple (source marketing très fréquente), il faut découper les articles pour les traiter indépendamment (la source servant alors de contexte). Cette voie a notamment été explorée dans le cadre du projet AB-STRAT [34], dont le but est de réaliser des résumés (*digests*) de listes de diffusion.
- *Faire un suivi temporel des informations contenues dans une source* (ex. : statistiques de citations d'un concurrent dans la presse).
- *Détecter les anomalies*. En effet, elles sont souvent le résultat d'un événement qu'il est intéressant d'identifier. Prenons l'exemple du marché de la défense : l'augmentation importante à partir du 12 novembre 2004 du nombre de documents dans la presse citant Thales et EADS doit générer une alerte[6].

Le résultat de cette phase d'analyse doit permettre à l'utilisateur d'exploiter au mieux l'information découverte. Pour l'aider dans sa consultation et découverte des liens, l'utilisateur dispose d'une vision fortement « connectée » de l'information. En d'autres termes, d'un simple clic, il lui est possible d'aller d'une information extraite vers les éléments qui ont permis l'extraction (par exemple *via* une mise en valeur dans le(s) texte(s) d'origine), ou encore d'aller du nom d'un concurrent vers sa fiche complète, etc.

récupérer toutes les pages traitant d'un sujet particulier.

[6] Cette augmentation était due à une rumeur de rachat de Thalès par EADS

2.2.3 Diffusion

Une fois l'analyse terminée, l'utilisateur dispose d'une information fortement « connectée » qu'il peut aisément consulter (par navigation ou par recherche sémantique directe). Cependant, ce format n'est pas adéquat pour une diffusion (typiquement à l'attention d'une direction). Le but de cette troisième phase dans le flux informationnel est donc de produire une synthèse de l'information collectée et analysée sous un format simple à consulter (facile à imprimer en tant que lettre d'information notamment). Le projet AB-STRAT [34] explore notamment ces besoins.

Cette synthèse peut difficilement se faire de façon totalement automatique. Nous proposons donc ici d'apporter une aide à l'utilisateur lui permettant de sélectionner les informations qu'il veut mettre en valeur dans son résumé. La génération du rapport peut en revanche se faire de façon automatique en se basant sur un modèle spécifié.

2.3 Deux niveaux d'information

L'objectif premier d'une application d'intelligence économique n'est pas de récolter des documents, mais bien d'enrichir la connaissance d'une entreprise sur son domaine, et notamment sur ses concurrents. Cette connaissance peut bien entendu être apportée manuellement par les experts, mais l'application doit aussi tirer profit de l'importante masse de documents disponibles pour en extraire des informations utiles.

Alors que les documents représentent une information brute, nous utilisons des *objets métier* pour représenter la connaissance structurée du domaine. Ils correspondent aux personnes, entreprises, marchés, produits, etc., c'est-à-dire à tous les concepts du domaine, et sont au centre de « l'intelligence » du produit. En complément des descriptions des utilisateurs, l'analyse documentaire les enrichit avec de nouveaux points de vue donnant accès aux informations extraites.

Documents et objets métier se situent donc à deux niveaux complémentaires d'information. Dans le cadre d'une navigation sur le résultat d'une analyse par exemple, lier l'information documentaire à l'information métier est indispensable à une exploitation efficace.

2.3.1 Documents

Les documents constituent la plus grande partie de l'information « brute » sur laquelle nous allons nous baser pour découvrir ou générer l'information réellement utile. Ils forment la source primaire d'information. Comme nous l'avons vu en section 2.2.1, ils peuvent provenir de nombreuses sources différentes et sont dès lors de nature peu structurée. Ni leur production, ni leur diffusion n'est contrôlée. En conséquence, il est difficile de trouver un traitement homogène qui exploite l'ensemble des informations disponibles dans les documents.

Pour pallier à leur nature semi-structurée, la majorité des outils se contente de traiter les documents comme du texte brut. La plupart des informations structurelles ou d'un méta-niveau ne sont pas exploitées (seuls quelques méta-données et les liens HTTP sont généralement extraits).

Découpage et structure. Les lettres d'information constituent une source d'information très fréquente. Elles ont cependant la particularité d'être constituées d'une liste d'articles traitant de sujets ayant peu de rapports entre eux. Analyser une lettre d'information en tant que document brut n'apporte donc qu'une faible part de l'information réellement disponible. La solution consiste en un découpage des articles en autant de documents indépendants. La figure 2.3 illustre ce découpage avec la lettre du Journal du Net.

Chaque article est extrait de la lettre et peut être considéré comme un document à part entière.

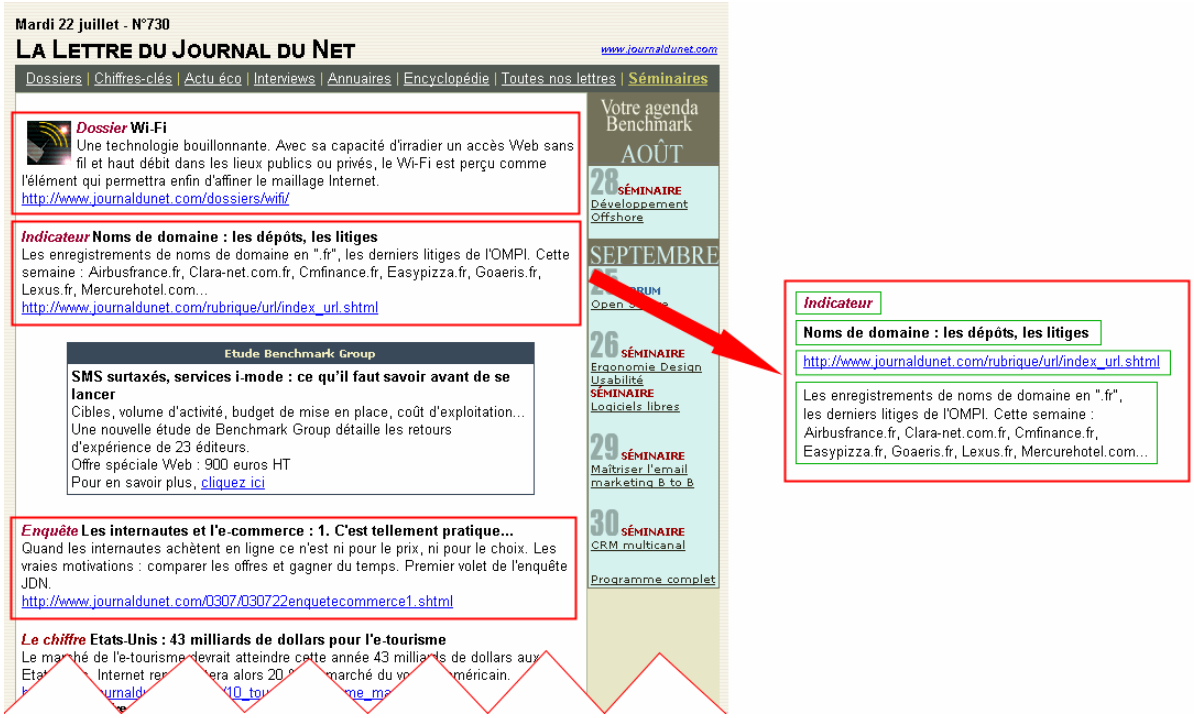

Figure 2.3 – Découpage d'une lettre d'information

Sur le même principe, dès lors qu'un minimum d'information structurelle est disponible dans un document, il est intéressant de l'exploiter pour identifier ses parties, chaque partie pouvant à son tour être découpée en sous-parties. Alors qu'un document pris dans son ensemble peut être très éloigné de la problématique d'un utilisateur, une de ses parties peut en revanche être très pertinente.

La découverte de la structure d'un document est d'autant plus efficace que sa source est connue et fournit des documents homogènes. La lettre du Journal du Net (comme la plupart des lettres d'information) en est un bon exemple : sa structure est toujours la même et nous permet de faire une découpe fine des articles qu'elle contient. Des outils spécialisés pour certaines sources pourront ainsi collaborer avec des outils moins efficaces mais travaillant dans un cadre plus général.

Méta-données et enrichissement. Comme nous l'avons vu, l'analyse du contenu d'un document ne peut pas s'appuyer sur une structure forte. À l'opposé, une simple indexation en texte intégral est fonctionnellement très limitée et pose des problèmes de pertinence. Nous allons donc nous intéresser à étendre la description des documents. En premier lieu, nous extrairons des informations comme leurs titres, auteurs, dates, etc. Mais, au-delà de ces méta-données traditionnelles, de nombreux enrichissements sont possibles.

Des utilisateurs pourront notamment ajouter des annotations ou une simple notation et ainsi enrichir le document avec de nouveaux points de vue. Ces nouvelles méta-données sont d'autant plus pertinentes qu'elles ont été saisies manuellement. Elles doivent donc être analysées elles aussi et participer au résultat de l'analyse au côté des documents.

Les résultats de l'analyse permettent eux aussi d'enrichir les documents. Ainsi, un nouveau point de vue « sémantique » leur est ajouté. Il sert, par exemple, de base à la création d'autres points de vue contextuels exhibant des extraits pertinents, c'est-à-dire les phrases du texte répondant le mieux à un

contexte (ou une requête).

Comme indiqué en section 2.2.2, d'autres outils (existants ou à venir) peuvent amener des enrichissements (ex. : la liste des personnes dont le nom apparaît dans le document). La figure 2.4 présente le processus général d'analyse, montrant comment différents outils peuvent collaborer pour enrichir un document avec différents points de vue.

Le modèle de données doit donc tenir compte de l'évolution des analyses et des outils disponibles. Imaginons que les auteurs d'une publication soient enregistrés dans une simple chaîne de caractères. Plus tard, de nouveaux outils sont capables de distinguer les différents auteurs, ainsi que leurs noms et prénoms. Plutôt que de modifier l'ancien point de vue (avec les bogues de mises-à-jour d'un logiciel que cela peut occasionner), nous préférerons en créer un nouveau, plus détaillé, qui déclenchera la création de l'ancien en renseignant sa valeur. Ainsi, anciennes et nouvelles applications peuvent évoluer progressivement et de façon transparente. En général, de nouveaux points de vue peuvent apparaître dès que de nouvelles analyses sont développées ou que de nouveaux besoins se font jour, en capitalisant éventuellement sur des points de vue existants.

Exploitation et diffusion. L'exploitation des documents ne s'arrête pas à leur analyse sémantique et à leur classement. Une fois un document analysé, l'utilisateur peut le consulter simplement, et surtout accéder aux éléments ayant permis de générer l'information à laquelle il s'intéresse. À partir de la vue lui donnant la liste des entreprises citées, il peut notamment accéder au texte originel ayant permis d'extraire cette information. Cette localisation est simplement faite *via* une mise en valeur (*highlighting*) du texte. Des liens vers l'intérieur du document sont donc nécessaires.

La consultation doit aussi être complétée par la diffusion pour atteindre toutes les strates de la décision. En plus de constituer la principale source d'information de l'analyse, les documents restent le moyen le plus utilisé pour diffuser le résultat des analyses et traitements effectués (cf. section 2.2.3). Cela nécessite de fournir aux utilisateurs des outils d'aide à la production. Les rapports ou résumés ainsi produits peuvent l'être dans différents formats en fonction des besoins (ex. : texte simple, HTML, PDF).

Plusieurs patrons peuvent être fournis pour aider à la génération. Nous pouvons, par exemple, obtenir un rapport suivant le plan suivant :

- quelques informations importantes au début (déterminées par le profil ou la requête à l'origine du rapport) ;
- un résumé des événements importants ;
- la liste des documents ayant permis de découvrir les informations résumées au-dessus (selon le format de génération, des liens entre informations synthétiques et documents sources peuvent être conservés).

2.3.2 « Objets »

L'analyse des documents collectés va nous permettre de découvrir de quel(s) sujet(s) traitent ces derniers. Mais s'il est intéressant de savoir qu'un concurrent est cité dans la presse, il est également très utile de pouvoir recouper cette information avec celles émanant d'autres sources (documentaires ou non), et notamment disponibles sur sa fiche.

Dans ce contexte, au-delà du terme qui le désigne, nous considérons le concurrent comme un objet métier, une entité qui va agréger toutes les informations disponibles sur lui. Consulter la fiche d'un concurrent pourra nous indiquer son dirigeant, son chiffre d'affaire, mais aussi les autres entreprises de même nature (concurrents potentiels), les acteurs avec qui il a passé des accords, l'évolution de sa

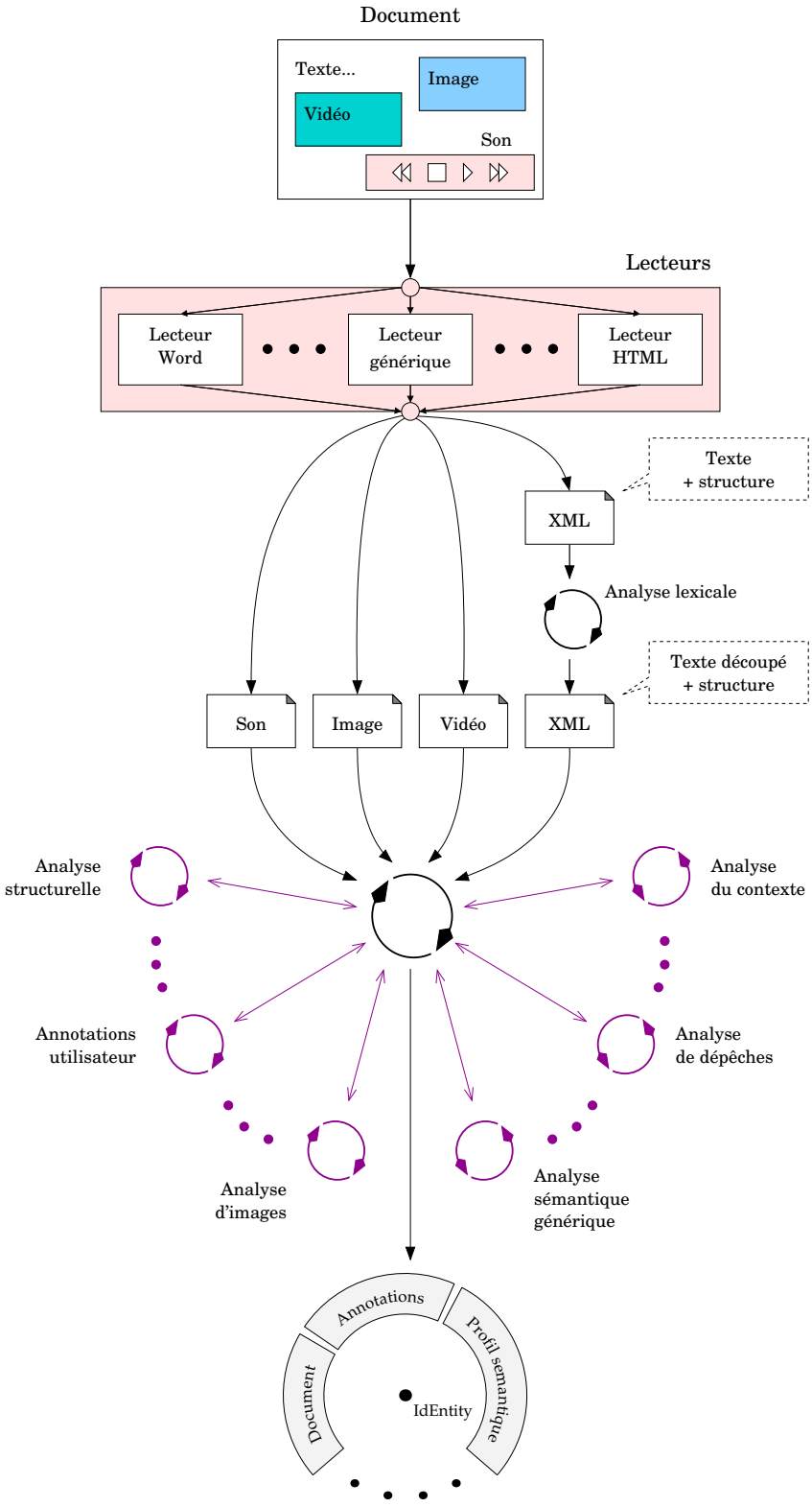

Figure 2.4 – Processus d'analyse documentaire

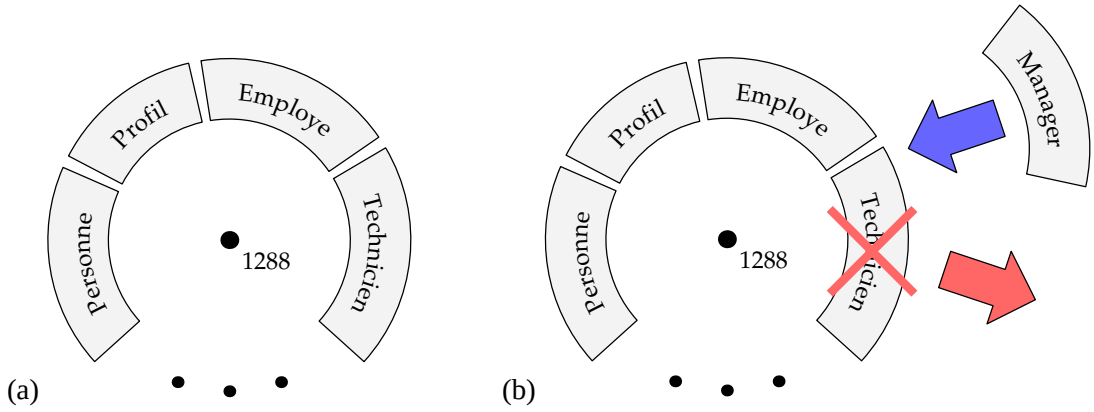

Figure 2.5 – L'entité Jacques Durand (ID 1288) avant (a) et après (b) sa promotion

présence dans la presse le mois dernier, etc. Sans oublier la liste de ses occurrences dans le *corpus* documentaire qui nous intéresse.

Pour simplifier la consultation, nous pouvons aussi l'observer selon une « dimension » parmi plusieurs. La dimension business, par exemple, peut nous montrer ses résultats et ses principaux clients , alors que la dimension concurrentielle se concentrera sur ses partenaires et concurrents, ainsi que sur les appels d'offres perdus ou gagnés face à eux. D'autres dimensions sont bien sûr envisageables (ex. : technologique, marketing, etc.).

Au-delà des concurrents et plus généralement des entreprises, il existe d'autres objets métier intéressants : personnes, produits, marques, etc. Les objets métier évoluent constamment et encore plus que pour les documents, ils bénéficient des enrichissements que peuvent leur apporter le système ou les utilisateurs. Étant plus visibles car moins nombreux que les documents, les enrichissements qui leur sont faits ont d'autant plus d'impact.

Imaginons qu'un utilisateur demande au système de collecter des documents concernant les organismes génétiquement modifiés, et que l'un de ces documents est la page personnelle de Jacques Durand, une personne déjà présente dans la base de connaissances (si elle ne l'est pas, une entité la représentant peut être créée dynamiquement). L'entité correspondante sera alors mise à jour avec un nouveau point de vue AUTEUR DE PAGE *Web* alors que le point de vue existant PROFIL sera mis à jour avec le concept OGM. Imaginons maintenant que cet auteur soit aussi le manager d'un important concurrent dans l'industrie agro-alimentaire. Sans avoir trouvé de contenu explicite décrivant la relation, nous avons maintenant un doute sur le fait que le concurrent soit un minimum intéressé par les organismes génétiquement modifiés. D'autres documents peuvent maintenant être observés dans ce contexte et confirmer ou infirmer notre hypothèse.

Ainsi, comme les documents, les objets métier peuvent évoluer en étant annotés avec de nouveaux points de vues, ou par le retrait d'anciens. Imaginons qu'avant de devenir manager, Jacques Durand était technicien. Son évolution est illustrée sur la figure 2.5 où elle se traduit par le retrait du point de vue TECHNICIEN et l'ajout du point de vue MANAGER.

2.4 Sécurité

Toute solution de traitement de l'information est amenée à travailler avec des documents et données confidentiels. Il est donc important de fournir une gestion des droits des utilisateurs.

Dans la version 3.7 de Kaliwatch Server par exemple, l'accès est sécurisé au niveau des sources pour les documents, et au niveau des objets eux-mêmes pour les objets métier. Bien qu'appropriée dans le cas général, cette granularité est toutefois insuffisante dans le cadre d'applications sensibles. Une sécurité au niveau des méta-données des documents comme des objets est nécessaire. Il est naturel, par exemple, que toutes les annotations ne soient pas publiques.

Il ressort de la variété des projets qu'il n'existe pas de compromis parfait entre simplicité et performance d'un côté, et sécurité et personnalisation de l'autre. Dans un cas, il pourra s'agir d'une application ne contenant pas de données sensibles et visant plusieurs milliers d'utilisateurs en consultation, alors que dans un autre, seuls quelques utilisateurs auront des accès différenciés et personnalisés à des informations confidentielles.

Chaque client ayant des besoins spécifiques, la politique de sécurité doit idéalement pouvoir être personnalisée pour chaque application.

2.5 Performances

L'hétérogénéité et la complexité des données à manipuler posent un vrai défi aux concepteurs d'applications en KM, qui doivent en plus prendre en compte l'émergence de nouveaux besoins. Ils ont besoin de modèles expressifs pour gérer cette information, mais sont confrontés à la relative inadéquation entre l'expressivité d'un modèle et ses performances dans un environnement de production[7].

En effet, une solution d'intelligence économique doit être capable de traiter d'importants volumes à la fois en terme de documents (plusieurs millions) et d'objets métier (organisations, produits, personnes, etc. mais surtout jusqu'à plusieurs milliers d'utilisateurs). Si un projet commence souvent par un pilote et quelques utilisateurs, son succès entraîne un déploiement à toute l'entreprise. Les performances, la robustesse et surtout les capacités à monter en charge sont donc des critères très importants pour ces applications. Pour répondre à ces besoins, les concepteurs se tournent généralement vers les systèmes de gestion de bases de données relationnels, standard bien établi qui offre robustesse et extensibilité en fonction de la charge, mais qui sont limités en terme d'expressivité.

Les concepteurs ont donc besoin d'un modèle qui réunit ces deux aspects antagonistes : expressivité et performances.

2.6 Conclusion et besoins induits

L'examen de la problématique nous permet de distinguer trois principales catégories de besoins pour le modèle de données sous-jacent :

- **Expressivité.** Afin de pouvoir générer et manipuler de la connaissance cohérente sur les entités du système (documents et objets métier), le modèle doit être suffisamment expressif pour pouvoir représenter des structures complexes et des contraintes.
- **Flexibilité.** Les entités sont souvent de nature hétérogène (notamment les documents), sont intrinsèquement composites et peuvent être observées sous de multiples points de vue, souvent liés entre eux. Qui plus est, les informations les concernant sont amenées à évoluer au cours de la durée de vie de l'application alors que de nouvelles connaissances sont acquises. Pour traduire ces aspects de l'information traitée, le modèle doit donc être *flexible* et *évolutif*.

[7] Dans [58], Karp montre notamment que les modèles à base de *frames* souffrent d'importantes limitations en terme de persistance et de passage à l'échelle.

- **Montée en charge.** Même si le nombre d'objets métier n'est pas susceptible de dépasser certaines limites dans le temps (une fois l'application déployée), le nombre de documents n'est pas limité. Le modèle doit donc supporter les changements d'échelle dus à un déploiement, puis une montée en charge progressive.

En plus de ces besoins principaux, la sécurité des accès aux données est souvent un critère important. Nous préférons cependant la considérer comme transversale au modèle et dépendante de l'implémentation. Les besoins en sécurité étant très variés, chaque application pourra appliquer une politique différente en fonction de son contexte.

Face à cette problématique, le modèle de données actuel de la solution éditée par Arisem [5] atteint aujourd'hui ses limites. Des évolutions successives ont permis de l'utiliser avec succès dans des environnements atteignant un million de documents et jusqu'à 5 000 utilisateurs (en consultation principalement) mais au détriment des fonctionnalités les plus complexes. Le principe d'agrégation de points de vue sur les objets permet une certaine flexibilité, mais l'absence de types de haut niveau en limite l'expressivité. L'utilisation d'une base de données relationnelle a rendu possible un bon niveau de performances mais l'implémentation *ad hoc* de nombreux aspects complique la maintenance et les évolutions. Le modèle utilisé n'a également qu'un seul espace de noms pour les objets, ce qui le rend difficile à réutiliser ponctuellement.

Ces limites nous ont conduits à proposer un nouveau modèle plus adapté aux besoins exprimés. Avant de détailler ses spécificités dans la partie III de ce document, faisons un tour d'horizon dans les chapitres suivants des principaux systèmes de modélisation de l'information disponibles aujourd'hui.

PARTIE II

État de l'art

La modélisation des données a toujours été au cœur de tout système de gestion de l'information [55]. En conséquence, elle a été étudiée par des communautés d'horizons différents. Si une convergence des concepts de modélisation se dessine avec le temps, tous les aspects du problème n'ont pas conduit à un accord parfait, en partie à cause de besoins applicatifs particuliers à chaque communauté.

Dans cette partie, nous allons passer en revue les principales propositions de modèles de données. Le domaine étant très vaste, il est difficile d'être exhaustif. Nous nous sommes donc tenus à décrire les principales familles de modèles.

Ces familles s'appuient toutes sur des notions de modélisation très proches. Pour mieux comprendre l'intérêt de chacune vis-à-vis des situations à modéliser, le premier chapitre de cette partie introduit les concepts généraux de modélisation de l'information. Les chapitres suivants nous permettront d'étudier les choix et extensions de chaque modèle :

- *Les modèles Entité-Association (E/A) et E/A étendus (y compris UML, ORM et RDF) dans le chapitre 4 ;*
- *Les modèles à base de* rôles *dans le chapitre 5 ;*
- *Les modèles à base de* frames *(issus de l'IA en général), incluant les logiques de descriptions et les modèles à base d'« ontologies », dans le chapitre 6.*

Enfin, le chapitre 7 nous permettra de les comparer aux besoins exprimés dans la partie précédente.

CHAPITRE 3

Modélisation de l'information complexe

La modélisation des données a toujours été au cœur de la problématique des systèmes d'information [55] et le sujet de nombreuses propositions.

Afin de mieux comprendre les similitudes existant entre chaque modèle et leurs différents apports, nous introduirons et passerons en revue dans ce chapitre des concepts de modélisation bien connus.

Nous commencerons en premier lieu, par introduire les notions fondamentales d'entité et d'association. Nous décrirons ensuite le concept de classe, puis les différents types de contraintes, et terminerons le chapitre par quelques notions complémentaires.

3.1 Entités et associations

Quand nous parlons d'information ou plus généralement de *connaissance*, nous utilisons un concept de niveau supérieur (méta). En effet, nous parlons de connaissance sur des « choses », quel que soit le nom que nous leur donnons : entités, sujets, ressources, objets, etc. Or, seules, ces choses n'expriment pas d'information de haut niveau. Ce sont les associations qui les lient qui constituent la principale source de connaissance.

Ainsi, au cœur de tout système de représentation de la connaissance se trouvent les concepts d'*entité* et d'*association*. Comme nous le verrons par la suite, ces deux concepts ont été introduits et étendus dans diverses propositions et par différentes communautés. Une convergence de concepts apparaît cependant, bien que tous les aspects du problème ne trouvent pas un accord parfait.

3.1.1 Entités

L'*entité* est la notion centrale à tout modèle d'information. Elle sert de représentation aussi bien aux objets de la vie réelle (ex. : l'employé Jacques Durand, le livre ODMG 3.0 ou encore la facture n°01512), qu'aux concepts plus abstraits (ex. : la couleur bleu, le domaine de l'astro-physique ou le marché des produits laitiers).

Selon les propositions, l'entité est aussi dénommée *objet*, *individu*, *ressource* ou *sujet*. Le terme *instance* est aussi utilisé mais a un sens plus précis comme nous allons le voir en introduisant la notion de classe.

Propriétés. Jusqu'ici, une entité n'est qu'un élément parmi d'autres dans l'ensemble des entités du système. Afin de l'exploiter, nous allons lui associer une description.

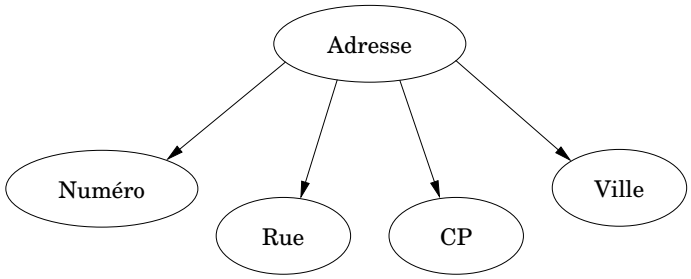

Figure 3.1 – Une propriété composite : adresse

Les informations permettant de décrire une entité sont appelées ses *propriétés* ou ses *attributs*. Dans l'exemple ci-dessous, l'entité représentant Jacques Durand a les propriétés `nom`, `sexe`, `âge` et `adresse`.

Jacques Durand est un homme de 28 ans habitant à Paris.

Ces propriétés n'ont pas les mêmes domaines. En effet, si le nom est une simple chaîne de caractères, l'âge est un entier et le sexe n'accepte que deux valeurs, homme ou femme. En général, les domaines des attributs sont spécifiés en utilisant des types de données de base disponibles dans la plupart des langages de description ou de programmation (entier, chaîne de caractères, etc.), avec parfois une restriction (ex. : un intervalle ou une énumération de valeurs). Toutefois, des types plus complexes peuvent être utilisés.

Les propriétés *composites* sont divisées en de plus petites parties qui représentent des sous-propriétés ayant un sens indépendant. Une adresse, par exemple, est composée d'un numéro, d'une rue, d'un code postal et d'une ville, comme illustrée sur la figure 3.1. Une sous-propriété peut à son tour être composite, formant ainsi une hiérarchie.

Une propriété peut aussi être *multivaluée*, c'est-à-dire, accepter plusieurs valeurs. Une personne peut par exemple avoir plusieurs prénoms. Le nombre de valeurs qu'une propriété accepte peut être restreint par des bornes.

Les propriétés composites et multivaluées peuvent être imbriquées pour former des *attributs complexes*. Choisissons arbitrairement de représenter les propriétés multivaluées entre accolades {} et les propriétés composites entre parenthèses (). Si une personne peut avoir plusieurs lieux d'habitation et pour chacun plusieurs numéros de téléphone, un attribut résidence peut être spécifié de la manière suivante :

```
{Résidence({NuméroTéléphone},
           Adresse(Numéro,
                   Rue,
                   CodePostal,
                   Ville))}
```

La figure 3.2 en donne une version graphique où les propriétés multivaluées sont indiquées par une ellipse double.

Enfin, tous les attributs d'une entité ne sont pas nécessairement stockés. Il est ainsi possible de s'appuyer sur les valeurs des propriétés d'une entité pour déterminer une nouvelle information. Comme le montre l'exemple ci-dessous, l'âge d'une personne peut être déduit à partir de sa date de naissance.

Jacques Durand est né le 12 septembre 1976 et a donc 28 ans.

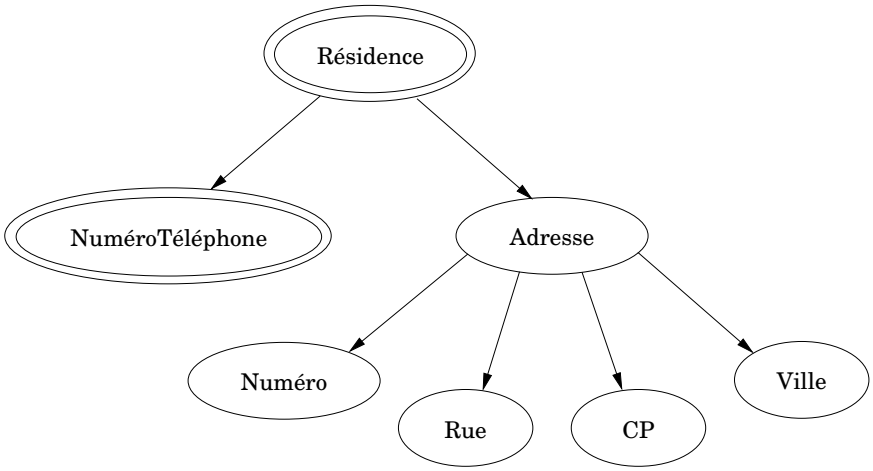

Figure 3.2 – Un attribut complexe : résidence (version graphique)

Notion d'identité. Afin de pouvoir accéder aux entités et travailler avec, nous avons besoin de pouvoir les distinguer les unes des autres. Quel que soit l'état du système, nous devons pouvoir identifier de façon *unique* chaque entité.

Dans le domaine des bases de données relationnelles, le concept de *clef* est utilisé. Une clef est un ensemble d'attributs dont les valeurs représentent une combinaison unique dans l'ensemble des combinaisons possibles. Imaginons une relation décrivant des personnes par exemple. Typiquement, le couple d'attributs $(nom, prénom)$ peut être utilisé comme clef de la relation et identifier chaque personne de manière unique. Dans une base de données plus volumineuse où plusieurs personnes peuvent partager le même nom (ex. : un annuaire téléphonique), il sera parfois nécessaire d'utiliser une clef plus grande ou encore de créer un attribut spécifique pour identifier les entités.

Si dans la réalité, il est toujours possible de distinguer différents objets, la représentation à un certain niveau d'abstraction dans un modèle peut déboucher sur deux entités différentes mais ayant le même état. L'exemple de vrais jumeaux illustre parfaitement cette contradiction : bien qu'ils aient exactement les mêmes attributs physiques, les jumeaux sont bien deux entités distinctes. Pour résoudre ce problème, de nombreux modèles s'appuient sur la notion d'*identifiant* unique, appelé communément *OID* (pour *Object IDentifier*). Dans un modèle utilisant des OIDs, l'existence d'une entité est indépendante de la valeur de ses attributs. L'*identité* (deux fois la même entité) est différente de l'*égalité* (entités avec les même valeurs d'attributs) [59].

Contrairement aux clefs, les OIDs ne sont pas modifiables une fois qu'ils ont été créés[1] et ne portent aucune information sur les entités (autre qu'un moyen de les identifier). Une conséquence importante de ce comportement est que quels que soient les changements d'état d'un système, d'une part un OID référence toujours la même entité, et, d'autre part, une entité a toujours le même OID. Une personne changeant de nom par exemple aura toujours le même identifiant.

Une dernière façon d'identifier des entités consiste en l'utilisation d'identifiants internes, ou *surrogates* [50]. Un *surrogate* est un représentant interne à un SGBD, affecté à la représentation d'une entité quand elle est entrée dans la base. C'est un concept d'implémentation, proche de la notion d'OID, mais qui reste interne au système, donc invisible à l'utilisateur.

[1] En pratique, les mises à jour de clefs sont rarement autorisées par les applications

	Clefs	**OIDs**	**Indentifiants internes (*surrogates*)**
1.	Concept de base de données	Concept de modélisation	Concept d'implémentation
2.	Peut porter de l'information	Ne porte pas d'information	Ne porte pas d'information
3.	Modifiable	Invariant	Invariant
4.	Unique dans chaque état d'une base (ou d'une relation)	Unique pour tous les états possibles du "monde"	Unique pour tous les états possibles d'un SGBD
5.	Problème de transfert d'information fréquent	Problème de transfert d'information rare	Problème de transfert d'information entre différents SGBDs
6.	Généralement affecté par un utilisateur	Affecté par un générateur d'OIDs	Affecté par le SGBD
7.	Visible à l'utilisateur	Visible à l'utilisateur	Invisible à l'utilisateur

Table 3.1 – Différences entre les clefs, les OIDs, et les identifiants internes

La table 3.1 reproduit le tableau récapitulatif, proposé par Wieringa et de Jonge [101], décrivant les différences entre ces trois notions.

3.1.2 Associations

En agrégeant des valeurs pour leurs propriétés, les entités permettent de représenter les concepts et objets dont nous avons besoin dans le système d'information. Elles ne sont cependant pas suffisantes pour représenter toute l'information. En effet, une part importante de celle-ci, souvent la plus pertinente, correspond aux *associations* qui lient les entités entre elles.

Chen introduit la notion d'association par l'exemple du mariage [25] (cf. figure 3.3). Deux entités représentant des personnes sont liées une-à-une par une association représentant le concept de mariage. Chaque entité participant à une association joue un *rôle* dans celle-ci (ici « mari » ou « femme »).

Arité. L'association présentée en figure 3.3 est *binaire*, mais si c'est le cas le plus fréquent, des exceptions existent.

L'exemple suivant présente une association ternaire, correspondant à la concurrence entre deux entreprises sur un marché. Il est illustré sur la figure 3.4.

Les entreprises TopAgro *et* Terroir.com *sont concurrentes sur le marché des fromages au lait cru.*

Une association peut avoir comme *arité* tout entier positif. Il est cependant rare de voir des associations d'une arité supérieure à trois. Certains modèles se limitent d'ailleurs aux associations binaires, plus simples à manipuler. Dans un tel cas, une association naturellement ternaire peut par exemple être

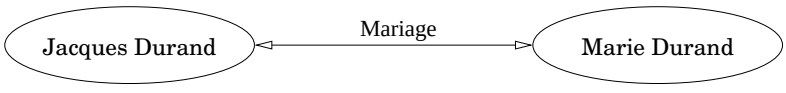

Figure 3.3 – Association représentant le mariage entre Jacques et Marie Durand

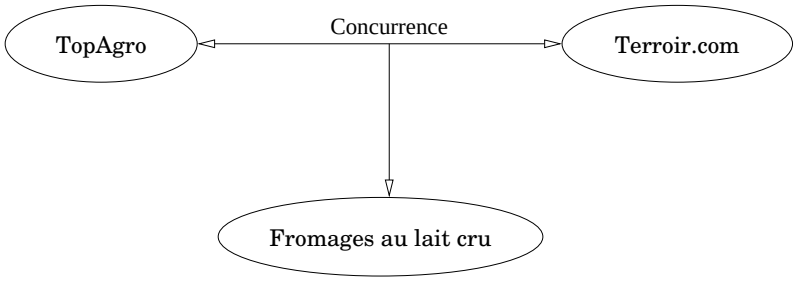

Figure 3.4 – Association ternaire représentant la concurrence entre deux entreprises sur un marché

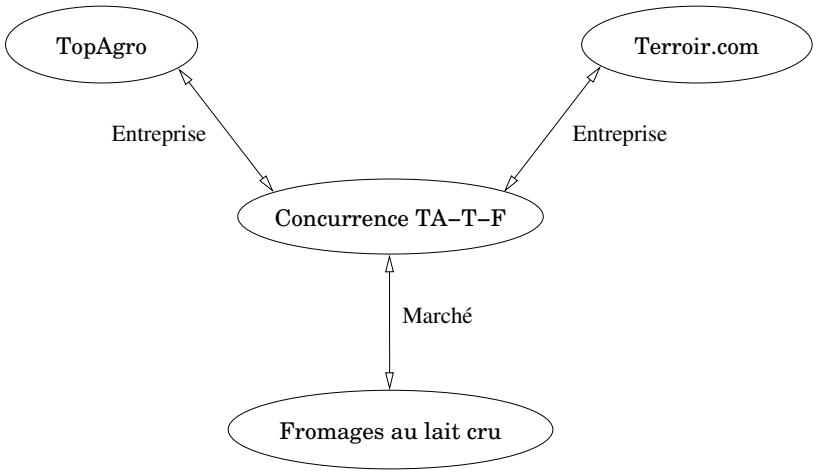

Figure 3.5 – Une réification possible de l'association ternaire de la figure 3.4

transformée en une entité et trois associations binaires comme le montre la figure 3.5. C'est une forme de *réification*, l'association devenant elle-même une entité.

Les associations *unaires* sont particulières car liées à une seule entité. La seule information qu'elles fournissent directement est donc leur existence ou non. De fait, la majorité des modèles les simulent avec de simples propriétés booléennes. Elles permettent cependant à ceux les exploitant (par exemple *Object Role Modeling* [51]), d'exprimer certaines contraintes plus simplement (cf. section 4.3.3).

Direction. Certaines associations doivent pouvoir être traversées dans les deux sens. Le mariage par exemple est une association *bidirectionnelle* : à partir de n'importe lequel des époux, nous pouvons traverser l'association pour obtenir l'autre. Dans d'autres cas, un « sens de lecture » naturel rend inutile la traversée inverse d'une association. Une association *unidirectionnelle* est alors suffisante. Dans un contexte documentaire par exemple, la liant un document à son contenu ne peut être traversée que dans ce sens[2].

Dans certains modèles (ex. : *Unified Modeling Language* [84], présenté en section 4.2), les associations binaires peuvent être remplacées par de simples attributs dont la valeur est l'entité à mettre en relation. Une telle association est alors naturellement unidirectionnelle et ne peut être traversée qu'à par-

[2]De nombreux outils documentaires permettent d'obtenir des documents à partir de leur contenu, mais ils utilisent alors un index de termes et non un lien inverse du contenu vers le document (la recherche se fait à partir d'un ensemble de mots-clés et non à partir du contenu dans sa totalité).

tir de l'entité ayant la propriété correspondante. Pour la rendre bidirectionnelle, une propriété *inverse* doit être présente sur l'entité liée.

3.2 Classes d'entités

Les concepts d'entité et d'association permettent de représenter des données dans un système d'information. Toutefois, leur manipulation directe est très complexe et fastidieuse. Par exemple, toutes les personnes ont un nom, mais attribuer une propriété `nom` à chaque entité représentant une personne est fastidieux. Il nous faut de plus assurer que toutes les entités représentant des personnes ont bien une propriété `nom`.

Pour simplifier leur traitement, nous allons donc regrouper les entités dans des ensembles en fonction de leur nature. L'ensemble DOCUMENT contiendra par exemple toutes les entités représentant des documents. Ces collections sont appelées *classes* d'entités et vont nous permettre de définir des conditions nécessaires que les entités devront remplir pour leur appartenir.

Chaque classe possède un type qui est la description abstraite partagée par l'ensemble de ses entités. Comme nous l'avons vu, toutes les personnes doivent avoir un nom. Pour traduire ce fait, il nous suffit de définir une propriété `nom` de type **string** sur le type de la classe d'entités PERSONNE. De façon plus générale, chaque type d'un classe d'entités définit l'ensemble minimum des propriétés que toutes les entités qu'elle regroupe doit avoir.

Opérations. Chaque classe d'entités nous donne ainsi accès à un ensemble d'objets partageant une définition commune. En s'appuyant sur cette base, nous pouvons étendre les classes avec des *opérations* (appelées aussi *méthodes* dans les approches à objets). Plus complexe qu'une simple propriété, une opération peut s'appuyer sur les valeurs des propriétés d'une entité pour effectuer des actions particulières, telles que l'impression ou l'analyse d'un document. En règle générale, les opérations sont implémentées dans un langage de programmation et ne sont limitées que par son expressivité.

Dans le cadre de nos besoins en modélisation, nous nous intéressons cependant davantage aux données et à leur manipulation qu'aux opérations qui sont du domaine de l'application[3].

Encapsulation. Une partie des propriétés et des opérations d'une entité sont souvent dédiées à l'implémentation et ne servent jamais à l'utilisateur. Il est donc inutile et même dangereux de les exposer, l'utilisateur pouvant être tenté de les utiliser, et au final, obtenir un comportement incohérent et non pérenne. À partir des fondements mathématiques de la théorie des types de données abstraits, le concept d'*encapsulation* a ainsi émergé dans les langages de programmation.

Dans cette approche, une entité possède une interface unique et au moins une implémentation. L'interface spécifie l'ensemble des opérations qui peuvent être effectuées sur elle[4] ; c'est sa seule partie visible. L'implémentation comprend à la fois les données et les méthodes de l'entité. Les données représentent son état, et les méthodes décrivent, dans un langage de programmation, l'implémentation de chaque opération.

En règle générale, l'interface et l'implémentation d'une entité sont définie par sa classe. L'interface n'est autre que le type abstrait de la classe. Cette séparation entre interface et implémentation permet de

[3] Comme nous le verrons en section 8.3.3, nous utilisons toutefois des opérations dans notre implémentation pour contrôler les mises à jour.

[4] La lecture des propriétés se fait alors à l'aide d'*observateurs*.

dissocier complètement l'utilisation de l'objet de sa structure interne. De cette façon, les programmes utilisant un objet sont protégés des changements dans son implémentation.

3.2.1 Instanciation

Comme nous l'avons vu, une classe d'entités regroupe toutes les entités de même nature et définit via son type un ensemble de critères (propriétés et opérations) que ces entités doivent avoir.

Deux approches existent pour associer une entité à une classe d'entités. La première consiste à simplement créer une entité avec toutes ses propriétés et ses éventuelles opérations, puis chercher à la classifier dans une classe existante (en vérifiant si elle remplit ses critères). La seconde approche prend une orientation différente en considérant que la définition d'une classe décrit un patron qui peut servir de cadre à la création de nouvelles entités. Ainsi, lors de sa construction, la nouvelle entité est directement classée dans la classe qui a permis de la créer. On dit alors que l'on *instancie* la classe et la nouvelle entité est appelée une *instance* de la classe. L'entité représentant Jacques Durand par exemple, est une instance de la classe PERSONNE.

Par généralisation, nous appelons lien d'*instanciation* le lien entre une entité et les classes auxquelles elle appartient. Une classe indique à la fois la définition des conditions nécessaires à toute entité pour lui appartenir, et la collection de toutes ses instances.

Soit E une classe d'entités et e une entité de cette classe. Nous avons alors : $e \in E$.

Instanciation multiple. Dans un souci de simplification, beaucoup de modèles limitent les liens d'instanciation à une association unique entre une instance et une classe (en dehors de la hiérarchie de spécialisation introduite en section 3.2.3). En pratique, chaque entité est créée comme instance d'une classe et n'en change jamais au cours de sa durée de vie. Cette approche est logique : il est ainsi incohérent qu'une entité représente à la fois une personne et un document.

Ce choix ne répond cependant pas à tous les besoins et, comme nous le verrons par la suite, d'autres modèles privilégient la flexibilité en autorisant l'*instanciation multiple*.

Dans l'exemple ci-dessous, l'entité représentant Jacques Durand « instancie » à la fois les classes EMPLOYÉ et ACTIONNAIRE, chacune ayant sa définition propre.

Jacques Durand est un employé dans le contexte entreprise,
mais est considéré comme un actionnaire dans le contexte boursier.

Nous avons donc :

$$jacquesDurand \in \text{EMPLOYÉ} \wedge jacquesDurand \in \text{ACTIONNAIRE}$$

La classe EMPLOYÉ peut, entre autres, spécifier un attribut `IdEmployé` alors que ACTIONNAIRE peut spécifier la participation de ses instances dans au moins une association de possession avec des actions (cardinalité `1..n`). Selon le contexte dans lequel se trouve l'application, les entités pourront ainsi être vues comme appartenant à des classes différentes.

3.2.2 Associations entre classes

De la même façon que les classes nous permettent de spécifier des propriétés pour leur instances, nous pouvons généraliser la notion d'association en la faisant porter sur les classes et non les entités elles-mêmes. Nous parlons alors d'*associations entre classes* (par opposition aux *associations entre entités*).

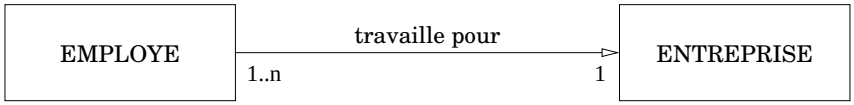

Figure 3.6 – Association entre les classes EMPLOYÉ et ENTREPRISE

Imaginons par exemple une association ⟨`travaille pour`⟩ entre les classes EMPLOYÉ et ENTREPRISE. Elle nous indique que chaque instance de EMPLOYÉ peut prendre part à une association avec une instance d'ENTREPRISE. Il nous est possible de connaître l'entreprise pour laquelle travaille chaque instance de EMPLOYÉ.

Cardinalité. Rien n'empêche cependant une entité d'instancier la classe EMPLOYÉ sans pour autant participer à une association avec un employeur. Pour exprimer cette contrainte, nous allons spécifier une *cardinalité* pour chaque rôle de l'association entre les classes. La cardinalité permet d'indiquer les nombres minimum et maximum de fois qu'une instance d'une classe peut participer à un rôle dans une association. Observons quelques exemples :

- `0..n` indique que l'association est optionnelle et que chaque instance n'est pas limitée en nombre de participations ;
- `1..1` indique que chaque instance doit participer à une et une seule association ;
- `1..n` indique que chaque instance doit participer à au moins une seule association ;
- `0..1` indique que chaque instance doit participer à au plus une seule association.

La contrainte de cardinalité minimum est parfois appelée une *contrainte de participation*. Quand le minimum est égal à zéro, la participation est dite *partielle*, alors que quand le minimum est égal ou supérieur à un, la participation est dite *totale*.

L'exemple des employés et entreprises peut alors être illustré par la figure 3.6. Nous considérons ici que chaque employé ne travaille que pour une et une seule entreprise[5] et qu'une entreprise a au moins un employé. La participation des classes EMPLOYÉ et ENTREPRISE dans l'association ⟨`travaille pour`⟩ est donc totale.

Réification. Comme nous venons de le voir, une association est définie sur plusieurs classes d'entités pour lesquelles nous pouvons contraindre la cardinalité. Toutefois, ce niveau d'abstraction n'est pas toujours suffisant. Il peut par exemple être intéressant d'attribuer une propriété à une association (ex. : la date d'un mariage). Le mécanisme de *réification* répond à ce besoin.[6]

Réifier une association revient à la transformer en une classe d'entités à part entière (cette nouvelle classe est parfois appelée *classe d'associations*). Chacun de ses rôles est remplacé par une nouvelle association vers la nouvelle classe. Devenue une classe, l'association réifiée peut bénéficier de nouvelles possibilités et, par exemple, définir des propriétés. Il est aussi possible de la spécialiser pour des cas plus précis comme nous allons le voir dans la section suivante.

3.2.3 Hiérarchie de classes

Afin de représenter son environnement, l'esprit humain classifie naturellement les entités qui le composent et il les organise entre elles. Cette classification prend généralement la forme d'une hiérarchie

[5] La cardinalité `1..1` est alors abrégée en un simple `1`.

[6] Certains modèles, tel le modèle Entité-Association [25], permettent aux associations d'avoir des propriétés.

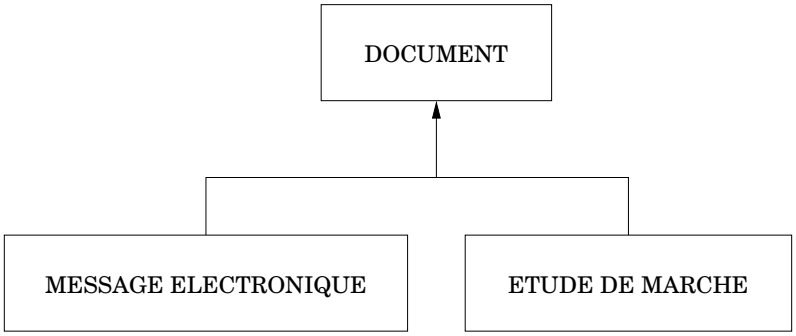

Figure 3.7 – Hiérarchie simplifiée des classes de documents

dans laquelle les concepts les plus généraux chapeautent les concepts les plus précis. Nous allons donc modéliser et organiser les classes d'entités de la même façon et créer des associations particulières *super-classe / sous-classe*, souvent appelées plus simplement *classe / sous-classe*[7]. La figure 3.7 illustre la hiérarchie de classes exprimée par l'exemple ci-dessus.

Les sous-classes d'une classe permettent de regrouper ses entités dans des sous-ensembles. Ainsi, les classes MESSAGE ÉLECTRONIQUE et ÉTUDE DE MARCHÉ sont des sous-classes de DOCUMENT. De même, les classes EMPLOYÉ, JOURNALISTE et ÉTUDIANT sont des sous-classes de PERSONNE.

Une entité membre d'une sous-classe est toujours membre de ses super-classes (ex. : tout message électronique est aussi un document). De fait, on dit qu'une sous-classe *hérite* de tous les attributs et associations de ses super-classes. Si S est une sous-classe de E, alors nous avons toujours $S \subseteq E$.

L'association classe / sous-classe est transitive et une sous-classe hérite donc de ses super-classes mais aussi des super-classes de celles-ci.

Spécialisation / généralisation. On dit que l'on *spécialise* une classe d'entités quand on définit un ensemble de sous-classes pour celle-ci. Inversement, on dit que l'on *généralise* des classes d'entités quand on définit une super-classe rassemblant leurs caractéristiques communes. Ainsi, une *spécialisation* $Z = \{S_1, S_2, \ldots, S_n\}$ est un ensemble de sous-classes qui ont la même super-classe G. Inversement, G est appelée la *généralisation* des sous-classes $\{S_1, S_2, \ldots, S_n\}$.

Une classe peut avoir plusieurs spécialisations.

PERSONNE, par exemple, peut avoir une spécialisation $Z_1 = \{\text{EMPLOYÉ}\}$ (une spécialisation peut n'avoir qu'une classe) et une autre $Z_2 = \{\text{HOMME}, \text{FEMME}\}$.

On dit que la spécialisation Z est *totale* si et seulement si $\bigcup_{i=1}^{n} S_i = G$ (G est alors appelée une classe *abstraite*). Dans le cas contraire, Z est dite *partielle*.

Dans l'exemple précédent, Z_2 est totale (une personne est forcément un homme ou une femme) alors que Z_1 est partielle (toutes les personnes ne sont pas des employés). L'exemple des documents de la figure 3.7 illustre lui aussi une spécialisation partielle. De nombreux documents ne sont ni des messages électroniques, ni des études de marché.

On notera que l'approche de modélisation employée a une influence sur la partialité des spécialisations. En regroupant les instances de plusieurs classes préexistantes, le mécanisme de généralisation conduit à une spécialisation totale. Alors qu'en distinguant parmi les instances d'une classe celles qui

[7] Cette association est aussi appelée couramment ISA (pour *is-a*, c'est-à-dire « est-un »). En effet, on dit « un employé *est une* personne », « un manager *est un* employé », etc.

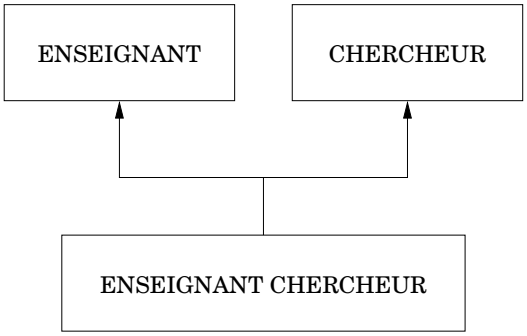

Figure 3.8 – Exemple d'héritage multiple

répondent à un critère particulier, le mécanisme de spécialisation conduit à une spécialisation partielle. Seul un sous-ensemble des instances est généralement classifié dans la(les) nouvelle(s) classe(s).

On dit que la spécialisation Z est *disjointe* si et seulement si $\forall(i,j) \in [1,n]^2 \mid i \neq j, S_i \cap S_j = \emptyset$. Dans l'exemple précédent, Z_2 est disjointe (une personne ne peut pas être à la fois un homme et une femme). Z_1 n'étant composée que d'une classe, elle est disjointe elle-aussi. En revanche, la spécialisation de la classe DOCUMENT illustrée par la figure 3.7 n'est pas disjointe. En effet, un message électronique peut tout à fait contenir une étude de marché.

Héritage multiple. Dans le cas général, une classe d'entités peut avoir plusieurs super-classes. Par exemple, la classe ENSEIGNANT CHERCHEUR hérite à la fois des classes ENSEIGNANT et CHERCHEUR (figure 3.8). Nous parlons alors d'*héritage multiple*.

L'héritage multiple pose cependant certaines difficultés en cas de conflit. Dans l'exemple précédent, à la fois ENSEIGNANT et CHERCHEUR peuvent définir une propriété ayant le même nom, mais pas la même sémantique. `Nombre d'étudiants` peut par exemple correspondre dans un cas au nombre d'étudiants suivant les cours de l'enseignant et, dans l'autre, au nombre d'étudiants encadrés par le chercheur dans le cadre de thèses ou de masters.

De nombreuses solutions ont été proposées pour résoudre ces conflits. Alors que certains modèles (ex. : Java [92] et C# [71]) interdisent simplement l'héritage multiple et se contentent d'une hiérarchie[8], d'autres vont beaucoup plus loin et permettent même au concepteur de préciser le mécanisme à utiliser (*Metaobject Protocol* [60]).

On notera d'autre part que *instanciation multiple* et *héritage multiple* sont deux concepts différents. Alors que l'instanciation concerne les entités, l'héritage s'adresse aux classes. Lorsqu'une entité instancie une classe héritant de plusieurs super-classes, on parle d'instanciation simple (bien que par spécialisation, elle instancie en fait toutes ces super-classes). L'instanciation multiple désigne le fait qu'une entité instancie *directement* deux classes complètement indépendantes (cf. section 3.2.1)[9].

Catégorisation. Un concepteur peut parfois avoir besoin de modéliser une association entre classes dont un des rôles peut être joué par deux classes indépendantes.

[8] Ces langages utilisent le concept d'*interface* et exploitent la notion de sous-typage plutôt que celle d'héritage multiple. Le sous-typage ne prend pas en compte l'héritage dit d'implémentation et les sous-classes ne peuvent donc pas hériter d'attributs inutiles.

[9] Certains modèles à *frames* (ex. : THEO [74]) ne distinguent pas les classes et les instances, et ne contiennent donc pas cette différence. Par exemple, une R19 étant une sous-*frame* d'une Renault, nous pouvons créer une sous-sous-*frame* « ma R19 » ou « les R19 rouges ».

L'exemple ci-dessous nous indique qu'aussi bien une personne qu'une entreprise peuvent être propriétaires d'une voiture.

Jacques Durand possède une Citroën C4 mais utilise parfois la voiture de fonction de son entreprise.

Une solution à ce problème consiste à travailler avec un sous-ensemble d'entités défini sur l'union des classes PERSONNE et ENTREPRISE. Ainsi, la *catégorisation* consiste à modéliser l'union de plusieurs classes. Une *catégorie* C est une classe définie comme un sous-ensemble de l'union de n super-classes $E_1, E_2, \ldots, E_n$ ($n > 1$). Nous avons donc : $C \subseteq E_1 \cup E_2 \cup \ldots \cup E_n$. Sur le même principe que pour les spécialisations, une catégorie C est dite *totale* si $C = E_1 \cup E_2 \cup \ldots \cup E_n$ et *partielle* dans le cas contraire.

Dans le cadre de notre exemple, nous pouvons donc créer une classe PROPRIÉTAIRE comme étant une catégorie, sous-ensemble de l'union des personnes et des entreprises :

$$\text{PROPRIÉTAIRE} \subseteq \text{PERSONNE} \cup \text{ENTREPRISE}$$

On notera que PROPRIÉTAIRE n'est pas totale. En effet, toutes les personnes et entreprises ne possèdent pas une voiture.

3.2.4 Méta-modélisation

Les classes d'entités et les associations entre elles nous permettent de modéliser les informations avec lesquelles vont travailler les applications. Un niveau d'abstraction supplémentaire consiste à décrire ces classes qui forment le modèle en les décrivant dans un *méta-modèle*. Deux approches sont alors envisageables :

1. *Méta-modèle séparé.* C'est l'approche la plus simple. Les éléments du modèle sont décrits avec un formalisme dédié[10] qui peut par exemple être graphique (ex. : le modèle Entité-Association [25]).
2. *Méta-modèle intégré au modèle.* Il n'y a pas de distinction claire entre modèle et méta-modèle. Dans ce cadre, les classes d'entités sont elles-mêmes des entités ayant pour classe la classe CLASSE et CLASSE est une entité ayant pour classe elle-même. Chaque nouveau modèle « embarque »[11] le méta-modèle et est construit sur celui-ci. C'est le choix fait par le *Resource Description Framework* (RDF) [62] par exemple.

Là où la première approche offre l'avantage de la simplicité, la seconde privilégie la flexibilité. Il est par exemple possible d'étendre la classe CLASSE pour spécialiser son comportement dans un contexte particulier. Associée à la possibilité d'instanciation multiple, n'importe quelle entité peut aussi devenir une classe d'entités elle-même, en instanciant simplement la classe CLASSE (ou une de ses sous-classes).

Une approche hybride peut aussi être employée : tout en conservant une distinction claire entre les deux niveaux d'abstraction, le méta-modèle est exprimé en utilisant le modèle lui-même (ex. : le méta-modèle de l'Unified Modeling Language (UML) utilise en partie UML [79]). À moins d'un besoin important en flexibilité, ce compromis est souvent suffisant et évite d'introduire un formalisme particulier. Il permet notamment d'utiliser la même syntaxe de requêtes pour interroger aussi bien modèle que méta-modèle.

[10] Plusieurs formalismes peuvent éventuellement être utilisés, par exemple, un formalisme graphique et un autre textuel.

[11] En fait, le modèle ne fait que référencer la description du méta-modèle qui est exprimée avec la même sémantique.

3.3 Contraintes

Les classes d'entités et les associations permettent au concepteur de modéliser tous les éléments de son domaine d'application. Cependant, une importante partie des règles du domaine ne peut être exprimée ainsi. Pour répondre à ce besoin et ainsi assurer la validité des données d'une application, nous utilisons des contraintes sur les valeurs et sur les classes.

3.3.1 Contraintes sur les valeurs

Comme nous l'avons vu en section 3.1.1 les propriétés ont un *type* contraignant les valeurs possibles. Ainsi, nous ne pouvons pas utiliser un entier pour renseigner la date de naissance d'une personne par exemple. Il est généralement possible de définir plus précisément le *domaine* de valeurs d'une propriété grâce à un intervalle ou une énumération de valeurs autorisées. Par exemple, le concepteur peut décider que la date de naissance d'un employé ne peut être antérieure à 1920 ou postérieure à 1990, et que son degré de connaissance dans un domaine doit être choisi parmi l'énumération `{néophyte, initié, expert}`.

Jusqu'à présent, nous avons toujours utilisé des types de base pour les propriétés (chaînes de caractères, entiers, dates, etc.). Toutefois, tout type est généralement utilisable, y compris les classes d'entités.

La notion d'auteur de document est un cas typique de ce besoin. Considérons l'exemple suivant :

L'auteur de ce message électronique est Jacques Durand.

Plutôt que d'utiliser une chaîne de caractères, il est plus intéressant d'avoir directement une entité PERSONNE comme valeur, ce qui nous permet ensuite d'accéder à d'éventuelles informations complémentaires (ex. : numéro de téléphone, adresse, etc.). En couplant cela avec une hiérarchie de classes, nous pouvons par exemple spécifier que la classe d'entités MESSAGE ÉLECTRONIQUE hérite de sa super-classe DOCUMENT une propriété `auteur`, qui a pour domaine la classe AUTEUR, sous-classe de PERSONNE regroupant les auteurs de documents.

Nous remarquerons qu'en réalité, définir une propriété sur une classe d'entités revient à créer une association unidirectionnelle (cf. section 3.1.2). Dans le cas précédent, nous avons indirectement créé une association unidirectionnelle entre les classes DOCUMENT et AUTEUR. Pour la rendre bidirectionnelle, il nous faut créer une propriété *inverse*, `documents`, sur AUTEUR.

Cette approche est utilisée par des modèles tels que *Unified Modeling Language* (UML) [84] (cf. section 4.2), mais ne fait pas l'unanimité. Le modèle Entité-Association par exemple (cf. section 4.1) n'autorise que les types de base pour les propriétés et force donc la création d'associations explicites.

Cardinalité. Il est courant que des documents soient rédigés par plusieurs auteurs. Inversement, un auteur n'écrit rarement qu'un seul texte. Bien qu'il soit possible de créer de nombreuses propriétés `auteur1`, `auteur2`, etc., cela devient fastidieux et totalement inapproprié dès que le nombre de valeurs maximum est trop important. Nous avons donc besoin de pouvoir utiliser un ensemble de valeurs pour une seule propriété.

C'est le rôle des propriétés multivaluées que nous avons introduites en section 3.1.1. Cet aspect peut être exprimé de façon précise grâce à des contraintes de cardinalité, en définissant des nombres minimum et maximum de valeurs. Une propriété mono-valuée a ses minimum et maximum tous deux égaux à un. Et de façon similaire aux associations, un minimum à zéro indique son caractère optionnel.

Types complexes. Plutôt que d'utiliser des propriétés multivaluées pour représenter les collections de valeurs, d'autres modèles ont fait le choix de généraliser cette notion grâce à des types complexes construits sur d'autres types (ex. : le modèle objet proposé par l'*Object Data Management Group* [22]). En général, plusieurs types sont disponibles : ensemble (**set**), famille (**bag**), liste (**list**), etc.

Ainsi, plutôt que de définir une propriété multivaluée `auteurs` de type AUTEUR, nous lui préférons dans cette approche une propriété `auteurs` de type **list**(AUTEUR)[12]. Des contraintes de cardinalité peuvent aussi compléter la définition (ici, nombres minimum et maximum d'auteurs).

Une collection est un type à part entière. Il est donc possible de définir un autre type qui s'en sert et ainsi d'imbriquer des collections. Imaginons par exemple que nous devions modéliser un texte sous la forme de paragraphes, eux-mêmes composés de phrases. Nous pourrons alors définir un attribut :

$$texte : \textbf{list}(paragraphe : \textbf{list}(phrase : \textbf{string}))$$

3.3.2 Contraintes sur les classes d'entités

Nous avons déjà défini en section 3.2.2 les notions de cardinalité et de participation dans le cadre des associations. Nous allons nous intéresser aux autres contraintes possibles sur les classes d'entités.

Disjonction. La contrainte la plus courante entre classes est la *disjonction*. Quand il existe une contrainte de disjonction entre deux classes (appelée aussi contrainte d'*exclusion*), une entité ne peut les instancier tous les deux.

Cette contrainte est souvent utilisée pour assurer la disjonction entre deux sous-classes d'une même classe. Prenons l'exemple suivant :

Jacques Durand ne peut être à la fois un employé et un prestataire.

Une contrainte d'exclusion est définie entre les classes EMPLOYÉ et PRESTATAIRE, tous deux sous-classes de PERSONNE, pour éviter qu'une personne ne puisse instancier les deux simultanément. Cette contrainte nous permet donc de spécifier des spécialisations totales (cf. section 3.2.3).

Il n'est pas nécessaire de déclarer des contraintes d'exclusion entre toutes les classes disjointes d'un modèle. En effet, elles sont héritées naturellement des super-classes. Une entité étant automatiquement instance de toutes les super-classes de sa classe, elle ne pourra instancier en même temps une classe en exclusion avec l'une d'elles. Ainsi il suffit de définir des contraintes d'exclusion entre les classes « racines » de la hiérarchie pour en faire profiter toutes les sous-classes.

Certains langages à objets, tels que Java [92] ou C# [71], n'ont pas besoin de cette contrainte. En effet, n'autorisant ni héritage multiple ni instanciation multiple, les classes sont naturellement en disjonction avec toutes celles qui ne sont ni super-classes ni sous-classes à quelque degré que ce soit.

Contraintes ensemblistes sur les rôles. En dehors de la hiérarchie de classes, une autre catégorie de contraintes concerne les rôles joués dans les associations.

Le directeur d'une société doit toujours en être un employé.

[12] Nous choisissons une liste plutôt qu'un ensemble pour conserver l'ordre des auteurs.

Cette simple assertion implique une contrainte d'*inclusion* entre le rôle `directeur` de l'association ⟨`dirige`⟩ et le rôle `employé` de l'association ⟨`travaille pour`⟩ : pour qu'une entité participe à une association ⟨`dirige`⟩, elle doit aussi participer à une association ⟨`travaille pour`⟩ avec la même entité entreprise.

Des contraintes de *disjonction* et d'*égalité* sont également possibles entre les rôles.

Contraintes générales. Certains modèles vont encore plus loin et permettent d'exprimer des contraintes sous la forme d'assertions logiques de premier ordre. C'est notamment le cas des logiques de description [20].

Nous pouvons par exemple restreindre les entités pouvant jouer le rôle `expert` de l'association ⟨`surveille marché`⟩ en n'autorisant que les employés qui ont déjà rédigé au moins deux études de marché. Cet ensemble peut s'exprimer sous la forme (formalisme de [20]) :

$$\text{EMPLOYÉ} \sqcap \exists^{\geq 2} \texttt{auteur}.\text{ÉTUDEMARCHÉ}$$

c'est-à-dire l'intersection entre l'ensemble des employés et l'ensemble des entités connectées à au moins deux instances de ÉTUDEMARCHÉ *via* le rôle `auteur`.

3.4 Autres notions

Avant d'étudier plus en détails quelques modèles, observons succinctement quelques notions supplémentaires « atypiques ».

Raisonnement. Bien que ne faisant pas partie des éléments du modèle en tant que tel, certaines propositions incluent des mécanismes de raisonnement. C'est notamment le cas pour certains modèles à bases de *frames* et toutes les logiques de description [19].

Le principal mécanisme de raisonnement est la *classification*. On dit que la classe A *subsume* la classe B si le concept que A représente est plus général que le concept que B représente. Toutes les entités de B sont aussi des entités de A, et B représente donc un sous-ensemble de A : $B \subset A$. La classe AUTEUR par exemple, subsume la classe JOURNALISTE, puisque tous les journalistes sont des auteurs.

Les logiques de description calculent automatiquement si une classe en subsume une autre. Ce calcul de subsomption est la base de la classification. Il permet la création automatique d'une hiérarchie de classes sans que l'utilisateur n'ait à déclarer explicitement toutes les spécialisations.

Distribution des propriétés. Certains modèles ouverts permettent de définir des entités ou des classes à plusieurs endroits différents. Le *Resource Description Framework* (RDF) [62] par exemple, permet de distribuer les propriétés des entités dans des fichiers différents. La description de chaque entité, ou ressource, correspond à l'agrégation de toutes les propriétés la concernant dans toutes les définitions accessibles. RDF n'utilise pas d'OID mais un identifiant explicite pour référencer les entités. Ainsi, il est possible d'étendre toute entité définie ailleurs en complétant simplement sa définition.

Le fonctionnement est identique pour les classes d'entités. RDF *Schema* [16] utilise RDF pour définir les classes. Il est donc possible de les étendre de la même façon en leur adjoignant de nouvelles propriétés. Cette possibilité permet notamment d'enrichir un schéma de base pour répondre à des besoins particuliers à une application.

Les identifiants des classes et entités étant explicites, il est peut probable que différents concepteurs choisissent les mêmes noms. Dans le cadre des ontologies, le *Web Ontology Language* (OWL) [86] introduit ainsi des propriétés permettant de mettre en correspondance des schémas différents.

Accès contextuel. Dans son entreprise, Jacques Durand est vu comme un employé avec ses qualifications et ses tâches. Dans le contexte boursier, il est vu comme un actionnaire. Son emploi n'est alors pas pertinent et n'a pas besoin d'être connu. Inversement, l'entreprise n'a pas besoin de connaître les investissements qu'il a faits sur son plan d'épargne actions.

Bien qu'étant la même personne et donc la même entité, Jacques Durand peut être vu de plusieurs façons différentes selon le contexte. ACTIONNAIRE et EMPLOYÉ sont des *points de vue* qu'une application peut exploiter en fonction de ses besoins.

La notion de *vue* est fournie par les bases de données relationnelles mais il ne s'agit que d'une façon de présenter des données. Les modèles à base de rôles vont plus loin avec la notion de *rôle* [28] en lui adjoignant des contraintes (voir le chapitre 5 pour plus de détails).

Évolution du modèle. La plupart des modèles de données sont statiques et n'évoluent pas dans le temps. Les classes d'entités sont définies une fois pour toutes et les entités, une fois créées, ne peuvent changer de classe. Comme nous le verrons par la suite, des exceptions existent cependant. RDF permet ainsi d'étendre la définition de classes existantes, alors que l'acquisition et la perte de types est centrale aux modèles à base de rôles.

3.5 Conclusion

Ce chapitre nous a permis d'introduire les principaux concepts de modélisation de l'information. Nous pouvons les répartir arbitrairement dans deux catégories qui reflètent les premiers besoins introduits en section 2.6 :

Expressivité :	Classes d'entités
	Types complexes *dont* Structures complexes
	Recouvrement instances et classes
	Spécialisation / généralisation
	Héritage multiple
	Catégorisation
	Associations
	Associations n-aires
	Contraintes
	Contraintes en logique de 1^{er} ordre
	Raisonnement / Inférence
Flexibilité :	Multi instanciation
	Acquisition / perte de type
	Accès contextuel
	Évolution du schéma
	Fusion de schémas

Ajoutons à cette liste deux concepts correspondants au troisième besoin exprimé :

Performances : Persistance
Passage à l'échelle

Dans les chapitres suivants, ce canevas va nous permettre de situer chaque modèle. Nous l'utiliserons de nouveau dans le chapitre 7 pour les comparer entre eux et avec les besoins de exprimés dans le chapitre précédent. La table 7.1 en particulier (page 70), regroupe l'ensemble des modèles dans un récapitulatif.

CHAPITRE 4

Famille Entité-Association (E/A)

Le modèle E/A est l'une des premières tentatives de représentation de la connaissance d'un point de vue sémantique. Cependant, comme les autres modèles de données de ce type, il a été originairement introduit comme un outil de conception de schémas. La simplicité de sa représentation graphique permet de raisonner avec une sémantique de haut-niveau lors de la phase de conception, avant de transposer pour implémentation le schéma obtenu dans un modèle traditionnel, typiquement le modèle relationnel [55].

*Nous allons présenter dans ce chapitre les principaux modèles de la famille E/A : le modèle E/A initial et son extension le modèle E/A Étendu, UML (*Unified Modeling Language*), ORM (*Object-Role Modeling*), et enfin RDF (*Resource Description Framework*).*

4.1 Modèle Entité-Association (E/A)

4.1.1 La proposition initiale

Le modèle E/A a été introduit pour la première fois par Peter Chen en 1976 [25]. Il utilise les deux concepts d'*entité* et d'*association* (cf. section 3.1). Les ensembles d'entités (appelés classes d'entités) sont décrits par un nom et un ensemble fixé d'attributs : $(E, \{a_1 : T_1, \ldots, a_n : T_n\})$ avec $n \geq 0$ (habituellement $n > 0$). Formellement, ces attributs sont autant de fonctions de cet ensemble nommé d'entités vers les domaines des attributs : $\{a_1 : E \to T_1, \ldots, a_n : E \to T_n\}$.

Généralement, il existe une « clef naturelle », c'est-à-dire un sous-ensemble des attributs $\{a_{i_1}, \ldots, a_{i_m}\}$ avec $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$, qui est nécessairement unique pour toute entité donnée : $E_1 \leftrightarrow T_{i_1} \times \ldots \times T_{i_m}$. Une classe d'entités sans *clef* est appelé *classe d'entités faible*.

Les associations entre classes sont également décrites par un nom et un ensemble d'attributs, et au moins deux classes d'entités qui ne sont pas nécessairement distinctes : $(R, \{a_1 : T_1, \ldots, a_n : T_n\}, \langle E_1, \ldots, E_p \rangle_{p \in \mathbb{N}})$ avec $n \geq 0$ et $p \geq 2$ (habituellement $n = 0$ et $p = 2$). On notera que les classes d'entités sont ordonnées car une même classe peut jouer plusieurs rôles dans une association (ex. : « père » et « enfant » dans un lien généalogique, ou « mari » et « femme » dans un lien marital). Chaque rôle est aussi contraint par une cardinalité (un ou plusieurs).

Cette définition va plus loin que celle introduite en section 3.2.2 en introduisant des attributs. Formellement, les attributs d'une association sont autant de fonctions du produit cartésien des entités : $\{a_1 : E_1 \times \ldots \times E_p \to T_1, \ldots, a_n : E_1 \times \ldots \times E_p \to T_n\}$.

Un schéma E/A est un ensemble de classes d'entités et d'associations sur ces classes. Comme illustré par la figure 4.1, la notation originale de Chen utilise des rectangles pour les classes d'entités, des losanges pour les associations et des ellipses pour les attributs. Les attributs soulignés (`IdEmp` et `SIREN`)

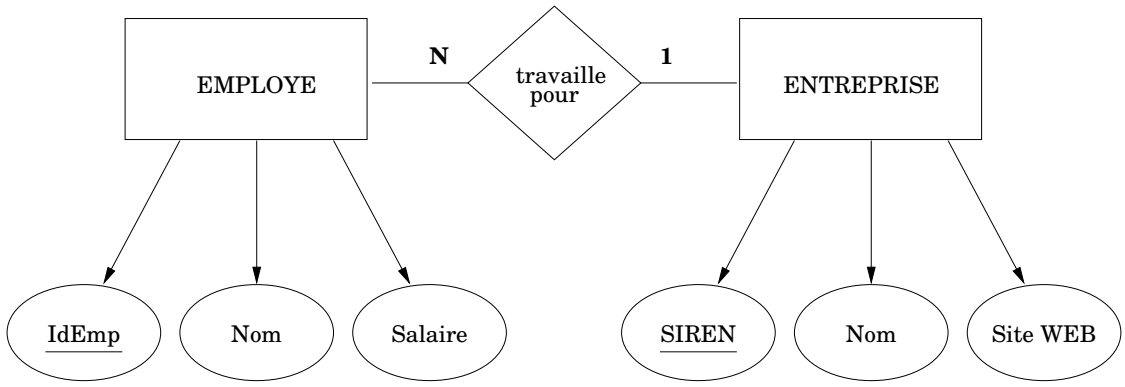

Figure 4.1 – Schéma E/A simplifié représentant la relation entre une entreprise et ses employés

représentent les clés des classes d'entités. Le « N » et le « 1 » indiquent que l'association est plusieurs-à-un : chaque employé travaille pour au plus une entreprise, mais plusieurs employés peuvent travailler pour la même entreprise.

La figure 4.2 montre le cas plus complexe d'une association ternaire ($p = 3$). Elle illustre la relation de concurrence existant entre des entreprises sur un marché. L'association elle-même a un attribut (`agressivité`), et la classe d'entités ENTREPRISE intervient deux fois dans l'association.

Les associations ternaires (ou de degré plus important) sont plus complexes à modéliser que les associations binaires (certains outils n'autorisent que les associations binaires). Il est possible de les remplacer par trois associations binaires, mais ces dernières ne sont généralement pas équivalentes. Une meilleure solution consiste en leur réification en une classe d'entités à part entière. C'est ce qui est fait dans la figure 4.3 pour la association de concurrence de l'exemple précédent.

Le modèle E/A est d'abord un outil de conception de schémas qui sont généralement traduits ensuite dans le modèle relationnel. Le modèle E/A ne pouvant représenter facilement toutes les dépendances fonctionnelles, le résultat obtenu est (légèrement) inférieur à une modélisation directe avec des dépendances fonctionnelles [67].

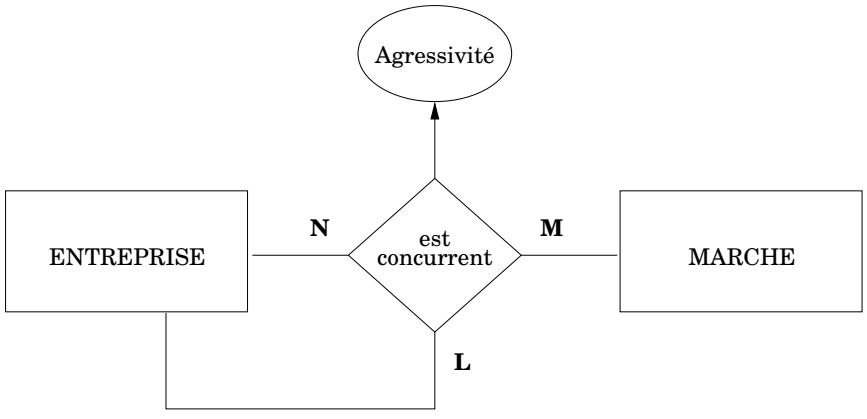

Figure 4.2 – Association ternaire représentant des entreprises concurrentes sur un marché

4.1.2 Vers le modèle Entité-Association Étendu

Les concepts de modélisation E/A présentés étaient à l'origine suffisants pour représenter beaucoup de systèmes d'information traditionnels. Cependant, les applications devenant plus complexes, les concepteurs ont eu besoin d'outils représentant les propriétés des données et leurs contraintes de façon plus précise. Ceci a conduit au développement de concepts de modélisation sémantiques qui furent incorporés progressivement dans les modèles conceptuels.

Ainsi, la proposition initiale de Chen fut ensuite étendue par de nombreux travaux [55] qui ont amené au modèle Entité-Association Étendu ou *Enhanced Entity-Relationship model* (EER[1]).

En extension au modèle E/A initial, nous y retrouvons les résultats de nombreuses propositions, tels que les attributs composites ou multivalués [38], et les contraintes structurelles sur les associations [99]. Le modèle EER introduit aussi explicitement le concept de classe (une entité est une instance de classe), les notions de spécialisation, de généralisation et de catégorisation [38, 95], ainsi que les contraintes utilisables sur ces notions.

Attributs complexes. Si les domaines des attributs sont généralement spécifiés en utilisant les types de données de base disponibles dans la plupart des langages (entier, chaîne de caractères, etc.), des types plus complexes peuvent être utilisés. Il s'agit des attributs *composites* et *multivalués* que l'on a introduits en section 3.1.1. La figure 3.2 illustrait l'imbrication de ces notions pour former un *attribut complexe*.

Contrainte de participation. Cette contrainte peut être appliquée à une association pour définir si l'existence d'une entité est liée à une autre. La participation d'une classe d'entités dans une association peut ainsi être *totale* ou *partielle*. La participation de la classe EMPLOYÉ dans l'association de la figure 4.1 est totale : un employé ne peut pas exister s'il ne travaille pas pour une entreprise. En revanche, dans le cadre d'une association un-à-un « dirige » entre les classes d'entités EMPLOYÉ et ENTREPRISE, elle sera partielle : seulement un sous-ensemble des entités de classe EMPLOYÉ dirige des entreprises.

En complément de la contrainte de cardinalité du modèle E/A, cette contrainte permet d'exprimer des contraintes de cardinalité plus complexes, telles qu'introduites en section 3.2.2.

Super-classes, sous-classes et héritage. Le principal apport du modèle EER est la notion d'héritage *via* les associations super-classes et sous-classes.

Cette notion est détaillée en section 3.2.3. Notons cependant que dans le modèle EER, l'appartenance d'une entité à une sous-classe peut être demandée explicitement par le concepteur, mais aussi spécifiée par un prédicat. On dit dans ce cas qu'une sous-classe S de E est *définie par un prédicat* p sur les attributs de E : $S = E[p]$. En prenant l'exemple de la classe PERSONNE, nous pouvons définir HOMME $=$ PERSONNE$[genre = mâle]$. Une spécialisation entière peut aussi être *définie par un attribut* si un prédicat portant sur le même attribut définit chaque sous-classe membre de la spécialisation.

Catégorisation. La *catégorisation* consiste à modéliser l'union de plusieurs classes. Comme les notions de spécialisation et de généralisation, elle est détaillée en section 3.2.3. Examinons donc directement comment nous pouvons illustrer graphiquement ces notions.

[1] Le sigle EER est aussi utilisé pour *Extended Entity-Relationship*

Un exemple. La figure 4.3 présente le schéma EER du modèle réduit et simplifié d'une application d'intelligence économique. Comme plusieurs notations existent, nous avons choisi de nous baser sur celle utilisée par Elmasri et Navathe dans *Fondamentals of Database Systems* [39]. Elle est résumé dans la figure 4.4

Les classes sont représentées par des rectangles. Observons la classe PERSONNE ; son attribut `adresse` nous permet d'illustrer la notion d'*attribut complexe*. Une adresse est en effet composée de plusieurs informations qui sont représentées comme dépendantes de l'attribut principal (nous nous limitons dans cet exemple au numéro, à la rue, au code postal et à la ville). Nous considérons également qu'une personne peut avoir plusieurs adresses et représentons ce fait en doublant l'ellipse de l'attribut pour indiquer qu'il est multivalué.

Par ailleurs, nous avons identifié deux *spécialisations* de la classe PERSONNE. D'une part, nous distinguons les employés des étudiants. Une même personne peut à la fois être un EMPLOYÉ et un ÉTUDIANT (par exemple un doctorant en contrat Cifre). Nous utilisons alors un « o » dans un cercle pour indiquer que ces classes ne sont pas disjointes[2]. Certains systèmes n'autorisant pas une entité à appartenir à deux classes distinctes, une solution est de créer une sous-classe héritant à la fois de EMPLOYÉ et ÉTUDIANT. D'autre part, nous isolons dans une sous-classe différente les personnes qui sont des auteurs. Cette spécialisation n'a qu'une seule classe.

Le lien *classe / sous-classe* est représenté dans le schéma par un signe d'inclusion ($\subset$) sur le trait.

Comme cela avait été illustré sur la figure 4.1, il existe une association entre les classes EMPLOYÉ et ENTREPRISE. Cependant, à la différence du précédent schéma, le trait partant de EMPLOYÉ est ici doublé. Cette notation indique que la participation de cette classe dans l'association ⟨`travaille pour`⟩ est *totale* : un employé ne peut exister dans l'application s'il ne travaille pas pour une entreprise. En revanche, seule une partie des employés ont une fonction de direction ; la participation de la classe dans l'association ⟨`dirige`⟩ n'est donc que *partielle*.

Les entreprises sont détenues par des actionnaires, mais ceux-ci peuvent aussi bien être des individus que d'autres sociétés. Nous représentons donc la classe ACTIONNAIRE comme une *catégorie*, sous-ensemble de l'union des classes PERSONNE et ENTREPRISE. Un « u » dans un cercle est utilisé comme notation de l'union.

Les entreprises entre elles sont aussi concurrentes sur un marché. Nous avions choisi de représenter ce fait par une association ternaire dans la figure 4.2. Nous avons pris une autre optique dans la figure 4.3 en réifiant la notion de concurrence dans une classe. N'ayant pas de clef, la classe CONCURRENCE est une *classe d'entités faible*. Pour pouvoir distinguer ses instances entre elles, nous utilisons tout ou partie de ses associations alors appelées *associations identifiantes*. La participation de la classe dans ses associations identifiantes est forcément totale. Dans le schéma EER, aussi bien les classes d'entités faibles que les associations identifiantes sont indiquées par le doublement du rectangle ou losange les entourant.

Un marché peut être décrit par une étude dédiée. Cette nouvelle entité étant un document, nous considérons que ÉTUDE DE MARCHÉ est une sous-classe de DOCUMENT.

Les documents sont fournis par une source. Considérant que la notion de source est « abstraite », nous imposons à toutes ses entités d'être aussi instances d'une sous-classe. La spécialisation {SOURCE WEB, SOURCE INTERNE, SOURCE MAIL, SOURCE BD} est donc *totale* et, comme pour les participations, une double ligne est utilisée comme notation. De plus, les collections d'entités correspondant à chaque type de source sont *disjointes* : une source ne peut par exemple pas être externe et interne simultanément. Cette disjonction est indiquée dans le schéma par un « d » dans le cercle liant les classes de la spécialisation.

[2] le « o » correspond en anglais au terme *overlapping*, ce qui indique un recouvrement des ensembles.

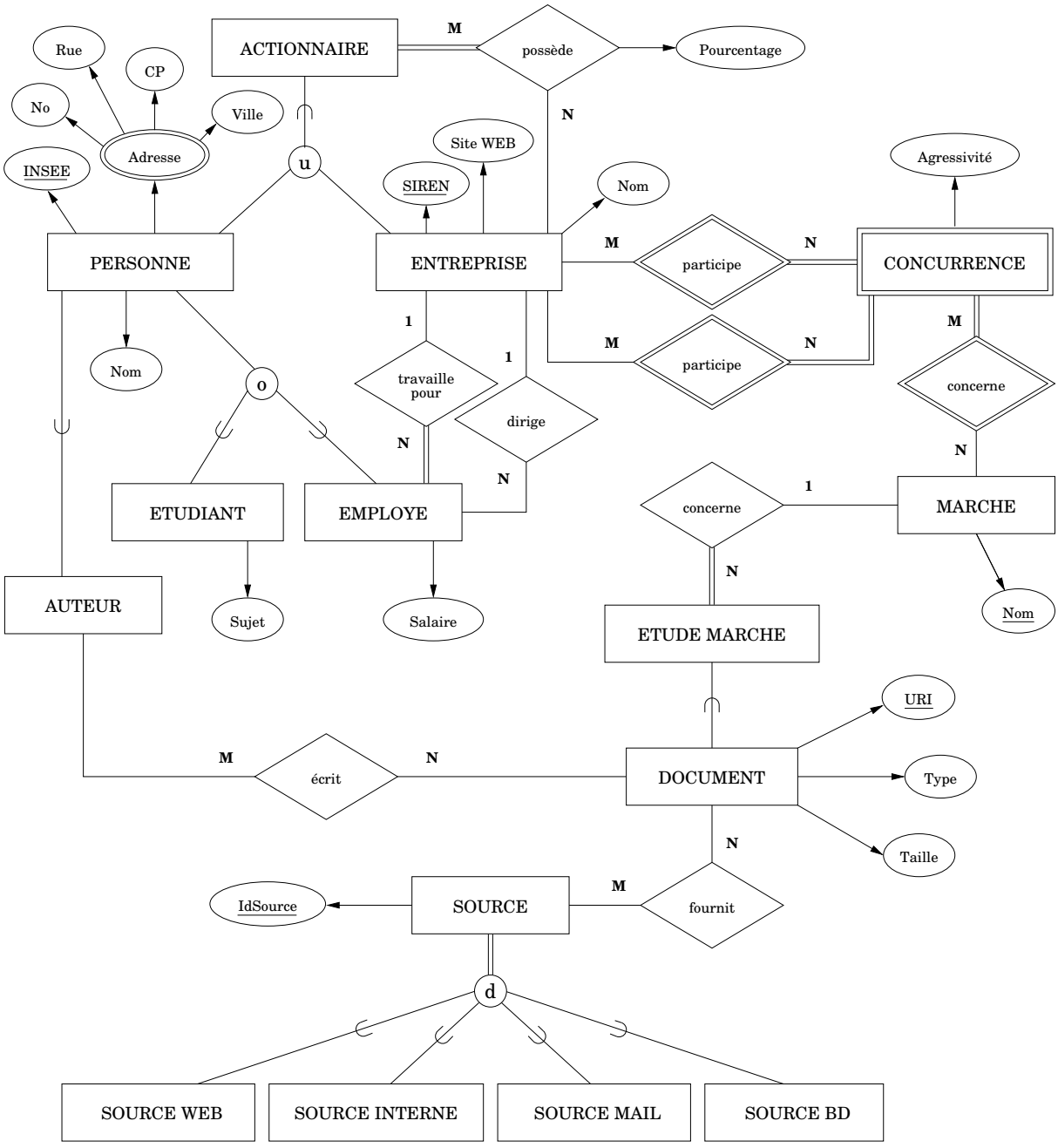

Figure 4.3 – Un schéma EER

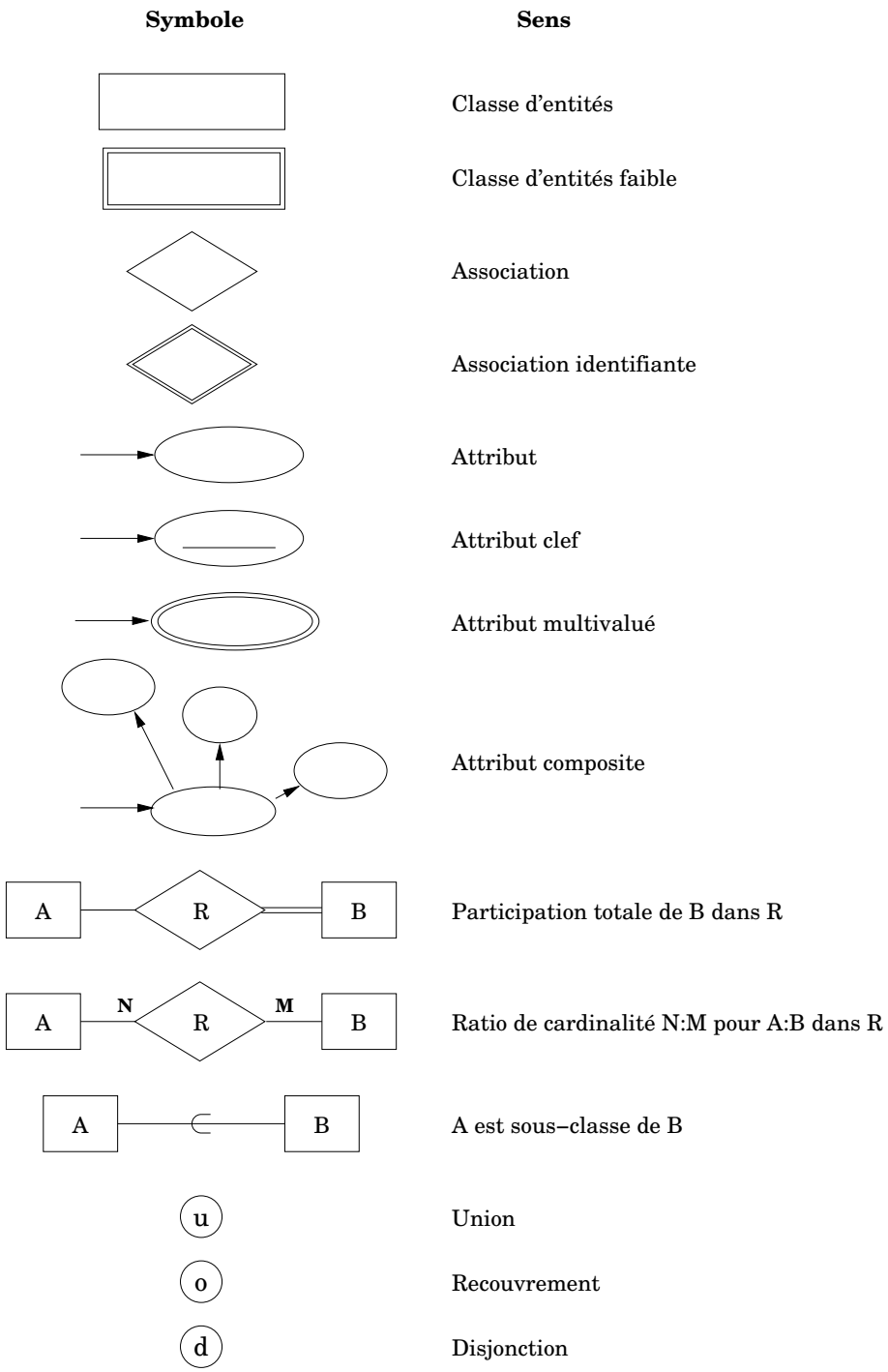

Figure 4.4 – Résumé de la notation des schémas EER

4.1.3 Conclusion et limites

Le modèle E/A a été l'un des premiers à introduire les notions fondamentales d'*entité* et d'*association*. En tant que pionnier, il souffre de limitations importantes quant à la modélisation de domaines complexes (il ne fournit pas la notion de spécialisation par exemple). Sa version étendue (modèle EER) a cependant permis de répondre à beaucoup de ces limitations en introduisant des concepts de plus haut niveau. Il est notamment l'un des seuls à proposer la notion de catégorisation.

Aussi bien E/A que EER sont tout d'abord des outils de conception de schémas. Ils fournissent ainsi une sémantique de haut niveau lors de la phase de conception, avant de transposer le schéma obtenu dans un modèle traditionnel, typiquement le modèle relationnel. Cette particularité leur permet de bénéficier de tous les avantages des systèmes de gestion de bases de données, notamment en terme de performances et de montée à l'échelle. En revanche, ces schémas sont statiques et une fois instanciés et en production, il est impossible de les faire évoluer. Ils ne supportent aucun des concepts que nous avons regroupés dans la catégorie *flexibilité* en section 3.5.

4.2 *Unified Modeling Language* (UML)

La combinaison de trois modèles à objets, la méthode OMT de Rumbaugh [83], la méthode de Jacobson [57] et la méthode de Booch [12], a conduit à la création de UML (*Unified Modeling Language*) [84]. Ce dernier est aujourd'hui largement accepté et utilisé, et a été adopté par l'OMG (*Object Management Group*) comme standard pour l'analyse et la conception à objets [78].

UML propose de nombreuses méthodes et diagrammes différents, incluant les diagrammes d'état, les diagrammes de séquence, les diagrammes de cas (*use-cases*), etc. Nous ne nous intéressons ici qu'aux diagrammes de classes.

On ne parle généralement pas de modélisation de données au sujet d'UML, mais de modélisation à base d'objets. Au lieu de classes d'entités, il se base sur des classes d'objets et est davantage utilisé pour la conception d'applications que pour la création de modèles de données. Toutefois, un examen plus approfondi de ses concepts montre qu'il est très proche des modèles E/A.

4.2.1 Classes

En plus d'un nom et d'un ensemble d'attributs, les classes UML sont décrites par un ensemble d'*opérations* : $(C, \{a_1 : T_1, \ldots, a_n : T_n\}, \{(o_1, S_1), \ldots, (o_p, S_p)\})$ avec $n \geq 0$ et $p \geq 0$. Chaque opération o_i est associée à une signature S_i qui consiste en un ensemble de paramètres d'entrée et un paramètre optionnel de sortie : $S_i = \{in_1^i : T_1^i, \ldots, in_j^i : T_j^i[, out^i : T_{out}^i]\}$ avec $j \geq 0$.

Contrairement au modèle E/A traditionnel, les types des attributs et des paramètres des opérations peuvent aussi bien être des types de base que des noms de classe. Un attribut dont le type est une classe représente implicitement une association unidirectionnelle entre la classe dans laquelle il est défini et celle qu'il référence. Il est en revanche impossible de définir des attributs complexes (cf. section 3.1.1). Il faut alors réifier le domaine complexe de l'attribut en une classe à part entière et utiliser cette dernière comme son type.

Nous avons repris une partie de l'exemple donné pour le modèle EER dans la figure 4.5. Comme nous pouvons le voir, la notation utilisée pour les classes sépare le nom, les attributs et les opérations dans trois parties distinctes d'un rectangle. Afin d'éviter de surcharger le schéma, nous avons choisi de ne pas représenter les types des attributs et les signatures des opérations, et opté pour une représentation simplifiée de certaines classes.

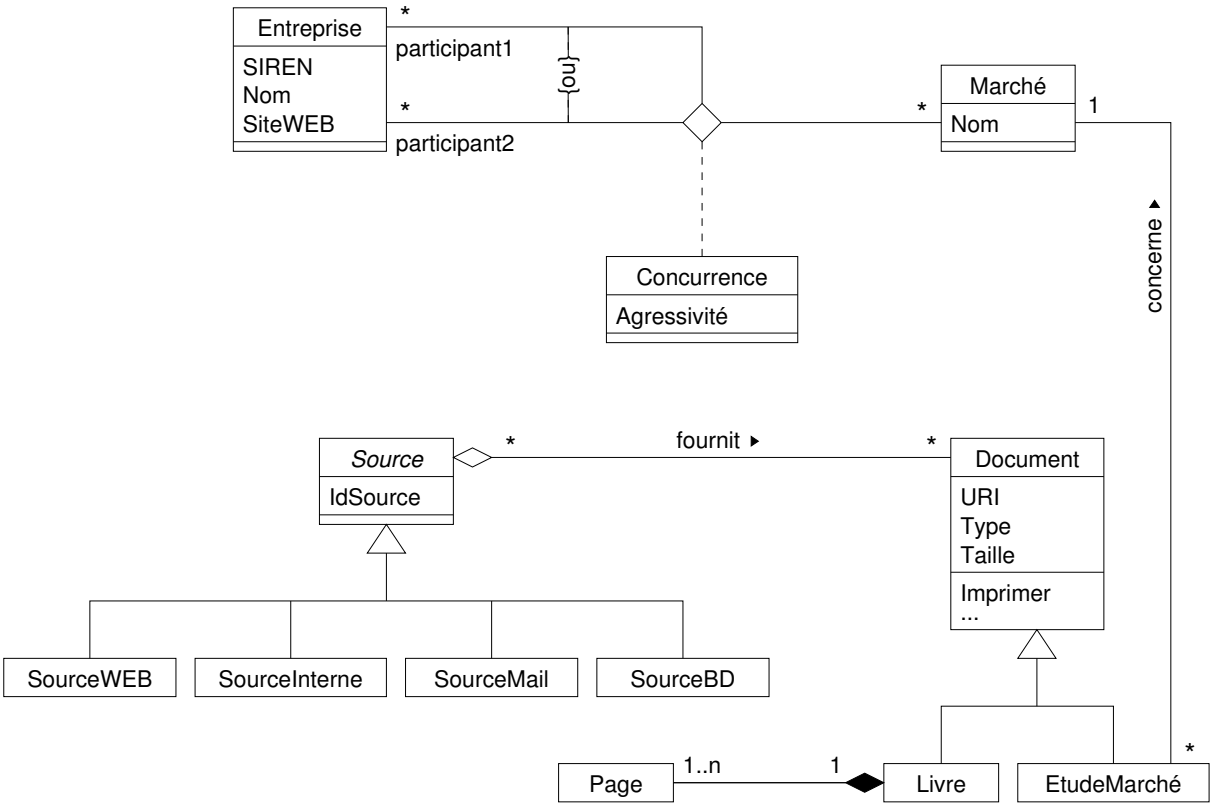

Figure 4.5 – Un schéma UML

Les liens de spécialisation / généralisation sont simplement représentés en UML par des flèches, comme nous pouvons le voir par exemple entre la classe DOCUMENT et ses deux sous-classes, ETUDE-MARCHÉ et LIVRE. En plus des propriétés statiques de leurs super-classes, les sous-classes héritent de leurs comportements, c'est-à-dire leurs opérations.

Si l'ensemble des spécialisations d'une classe est total (au sens E/A), alors la classe est appelée *abstraite*. Un objet ne peut instancier cette classe directement ; il doit instancier une de ses sous-classes. Nous distinguons une classe abstraite d'une classe normale en écrivant son nom en italique (cf. classe SOURCE dans le schéma) ou en utilisant le stéréotype «`abstract`» devant le nom de la classe.

4.2.2 Associations

Les associations UML sont décrites par un nom (optionnel) et par un ensemble de classes qui ne sont pas nécessairement distinctes : $(R, \langle C_1, \ldots, C_n \rangle_{n \in \mathbb{N}})$ (généralement $n \geq 2$). Elles n'ont pas d'attributs, mais peuvent être réifiées dans des classes d'associations. Comme pour les associations du modèle E/A, l'ordre des classes est important, la même pouvant jouer plusieurs rôles. Dans le cas où chaque rôle est explicité par un nom, nous pouvons aussi définir les associations par : $(R, \{r_1 : C_1, \ldots, r_n : C_n\})$

La cardinalité de chaque rôle est contrainte et peut être précisée avec une grande finesse. Elle peut prendre comme valeur :

- n : exactement n ($n \geq 0$)
- $n..m$: de n à m ($n \geq 0$, $m > 0$ et $m > n$)
- $*$: plusieurs, c'est-à-dire 0 ou plus (équivalent à $0..n$)

- $n..*$: n ou plus ($n \geq 0$)

La navigabilité des associations peut être restreinte, les rendant alors *unidirectionnelles*. Il est aussi possible de *qualifier* une association, ce qui permet entre autres de simuler les associations identifiantes du modèle E/A. Une telle association ne lie les instances de la classe cible qu'à une sous-partie des instances de la classe d'origine.

Dans la figure 4.5, la classe CONCURRENCE est une classe d'associations qui représente l'association ternaire entre deux entreprises et un marché. Les rôles respectifs des entreprises sont représentés dans le schéma (`participant1` et `participant2`) et une contrainte supplémentaire indique qu'une même instance de la classe ENTREPRISE ne peut participer à une association dans les deux rôles à la fois : une entreprise ne peut être concurrente avec elle-même.

4.2.3 Agrégation et composition

Les agrégations et compositions sont deux types particuliers d'associations. Une *agrégation* est une association qui exprime un couplage fort et une relation de subordination entre deux classes. Une *composition* est une agrégation forte, c'est-à-dire que le cycle de vie des instances agrégées est dépendant de leur agrégat. Si ce dernier est détruit (ou copié), ses composants le sont aussi. Au contraire des agrégations, les compositions sont exclusives : une instance de composant ne peut être liée qu'à un seul agrégat.

Ces deux types d'association sont illustrés dans la figure 4.5. Nous y retrouvons, d'une part, une association d'agrégation entre les sources et leurs documents (ceux-ci peuvent provenir de plusieurs sources différentes ou même ne pas être liés à une source) et, d'autre part, il existe une association de composition entre un livre et ses pages. Le schéma nous indique que toute page appartient à un livre et un seul, et qu'un livre a au moins une page. La notation UML utilise un losange creux pour indiquer une agrégation et un losange plein pour indiquer une composition.

4.2.4 Autre notions

UML propose aussi la notion de *paquetage* (*package*) qui permet d'encapsuler des éléments d'un modèle dans un élément logique de plus forte granularité. Nous retrouvons également la notion d'*interface*, applicable aussi bien aux classes qu'aux paquetages, et qui caractérise un comportement par un ensemble d'opérations. Un paquetage ou une classe *réalisant* une interface doit supporter toutes les opérations définies par celle-ci. Plusieurs notions supplémentaires sont introduites par UML (ex. : la visibilité des attributs), mais ne sont pas détaillées ici car elles ne relèvent pas de la modélisation d'information.

4.2.5 Conclusion et limites

Hormis l'absence de catégorisation, UML propose une expressivité équivalente à EER. Comme ce dernier, c'est avant tout un outil de conception et il manque de flexibilité une fois le modèle instancié.

Toutefois, il répond à des besoins plus larges que la modélisation de l'information et s'intéresse à toute la chaîne de création d'une application. À ce titre, et contrairement au modèle E/A étendu, il permet de modéliser des aspects applicatifs (ex: définition d'opérations). À contrario, il est moins adapté à une implémentation relationnelle.

4.3 *Object-Role Modeling* (ORM)

Les principes fondateurs de ORM ont été proposés en 1976 par Falkenberg sous le nom de *object-role model* [41]. Ils ont ensuite été étendus par de nombreuses propositions jusqu'à aboutir à ce qu'on regroupe aujourd'hui sous le terme de ORM ou *Object-Role Modeling* [51, 52]. ORM est aussi parfois appelé NIAM pour *Natural language* (ou *Nijssen*) *Information Analysis Method* [76].

ORM représente l'information sous la forme d'*objets* (entités ou valeurs) qui jouent des *rôles* dans des associations. Contrairement à la majorité des modèles, la notion d'attribut n'existe pas : ils sont remplacés par des associations avec des valeurs.

ORM est parfois qualifié d'*approche orientée-faits*. En effet, le processus de conception de schémas conceptuels (CSDP pour *Conceptual Schema Design Procedure*) proposé commence par transformer des exemples de l'information à modéliser en faits élémentaires. Ceux-ci permettent ensuite de créer des types de faits qui correspondent à des associations. Le modèle final est obtenu après plusieurs autres étapes permettant d'ajouter les contraintes et de vérifier la validité du modèle.

Une autre particularité d'ORM est l'accent mis sur la verbalisation. Tous les éléments du modèle peuvent être transformés en des phrases simples en langage naturel, ce qui permet une meilleure communication entre le concepteur et son interlocuteur.

4.3.1 Entités et valeurs

Les *classes d'entités* n'ont pas d'attributs dans le modèle ORM mais ont au moins un *schéma de référence*, qui indique comment faire correspondre chaque instance du type d'entités à des valeurs *via* des prédicats : $(E, \{\{p_1^1 : T_1^1, \ldots, p_n^1 : T_n^1\}, \ldots, \{p_1^q : T_1^q, \ldots, p_m^q : T_m^q\}\})$ avec $n \geq 1$ et $q \geq 1$ (généralement, $q = 1$ et $n = 1$). La notion de *clef* du modèle E/A peut être traduite par un schéma de référence dans le modèle ORM.

Les attributs au sens E/A étant remplacés par des associations, le modèle ORM introduit aussi la notion de *types de valeurs* pour les représenter. Un type de valeur n'est composé que d'un nom et d'un type de base : (V, T).

Les notions de type d'entités et type de valeurs sont regroupées dans la notion plus générale de type d'objets.

Comme illustré par la figure 4.6, les types d'entités sont représentés par des ellipses pleines et les types de valeurs par des ellipses en pointillés. Lorsque le schéma de référence est simple, il peut être indiqué directement en représentant le mode de référence entre parenthèses sous le nom du type d'entités.

Les liens de spécialisation / généralisation sont représentés par de simple flèches allant du sous-type vers le super-type. Comme nous pouvons le voir sur le schéma, des contraintes peuvent aussi être représentées. La spécialisation du type SOURCE est ainsi une partition : les sous-types sont exclusifs et leur union est égale à leur super-type.

4.3.2 Associations

Les associations sont décrites par un nom (optionnel), un ensemble de rôles (obligatoires ou non) et un ensemble de contraintes internes d'unicité : $(R, \langle O_1, \ldots, O_n \rangle, \langle \langle O_{i_1}, \ldots, O_{j_1} \rangle, \ldots, \langle O_{i_p}, \ldots, O_{j_p} \rangle \rangle)$ avec $n \geq 1$, $p \geq 1$ et $\forall x \in [1, p], (i_x, j_x) \in [1, n]^2$ et $i_x \leq j_x$. De plus, une association ne peut exister si elle ne fait pas intervenir au moins un type d'entités.

Comme l'illustre la figure 4.6, les associations sont représentées sous la forme de rectangles composés d'une case par rôle, et chaque rôle est relié à son type d'objets par une ligne. Le nombre de rôle d'une association correspond à son *arité*.

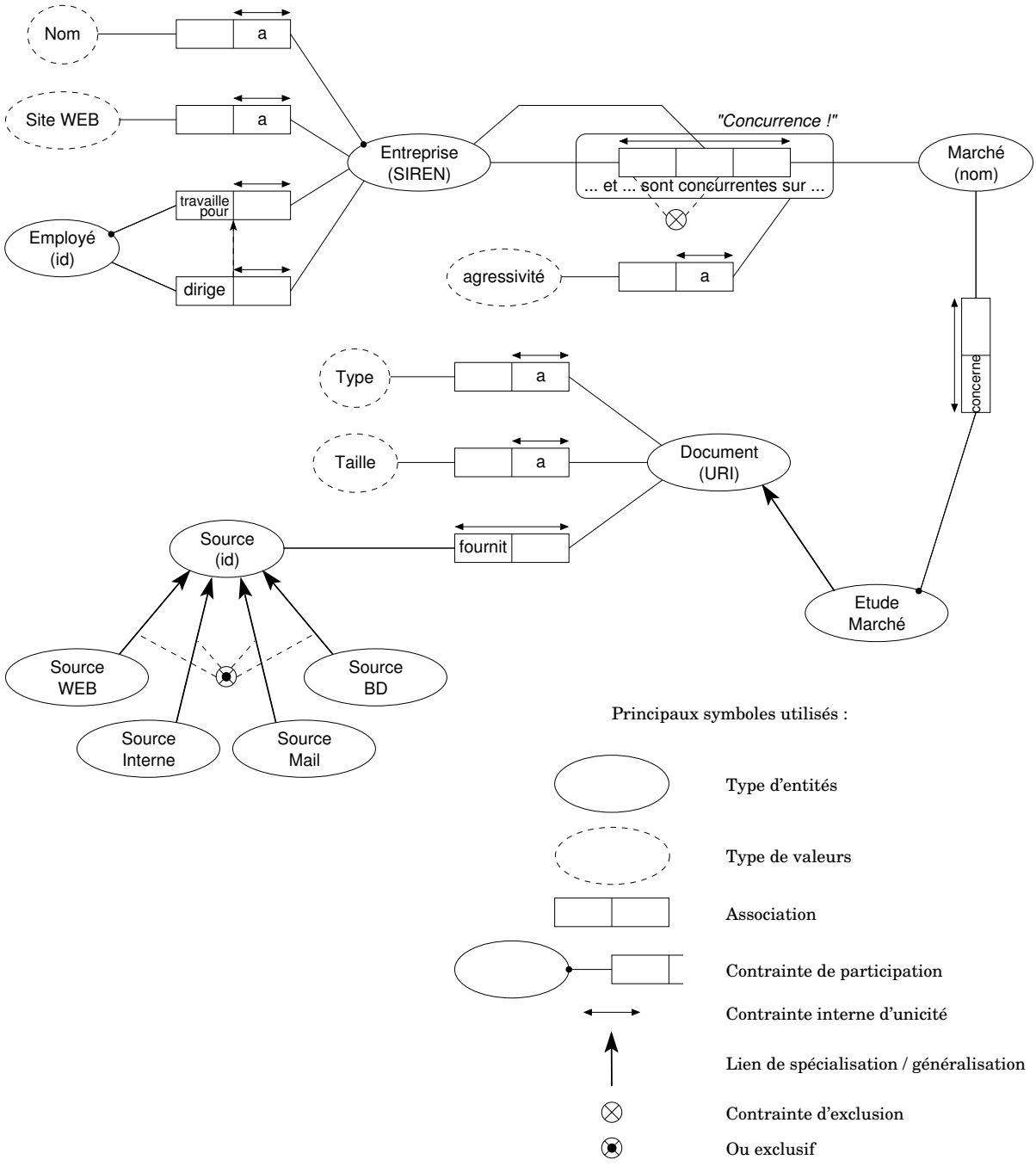

Figure 4.6 – Un schéma ORM

Pour indiquer qu'un rôle est obligatoire pour un type d'entités, un point noir est ajouté. Dans ce cas toute instance de ce type doit forcément jouer ce rôle. Une entreprise, par exemple, a forcément un nom, mais l'existence d'un site sur la Toile est optionnelle.

La désignation des associations est particulière. Dans le cas simple, son nom est inscrit dans ou à côté de la case correspondant à son premier rôle. D'autres lectures peuvent cependant être fournies, par exemple une phrase avec un « trou » pour chaque rôle : « *. . . et . . . sont concurrentes sur . . .* ».

Il n'existe pas de contraintes de cardinalité telles que nous les avons introduites en section 3.2.2. Elles sont ici remplacées par des *contraintes internes d'unicité*. Une simple flèche au dessus d'un ou plusieurs rôles indique qu'il ne peut pas exister deux instances d'association utilisant le même rôle ou la même combinaison de rôles. Une contrainte de cardinalité $1 : n$ est ainsi transformée en une contrainte interne d'unicité sur le rôle dont la cardinalité est multiple : le même objet ne peut être présent deux fois dans l'association. Pour une cardinalité $n : m$, la contrainte interne d'unicité est sur le couple de rôle, alors que pour une cardinalité $1 : 1$, nous aurons une contrainte sur chacun des rôles. Ce mécanisme est plus expressif dès lors que les associations ont une arité supérieure à deux.

Comme le montre l'association de concurrence du schéma, la réification est possible et elle est représentée par un cadre arrondi englobant le rectangle. Le nom du nouveau type est alors indiqué entre guillemets. Dans l'exemple représenté, le point d'exclamation indique que le type d'entités est indépendant : des instances de ce type peuvent exister sans prendre part à un fait.

4.3.3 Autres particularités

ORM a d'autres particularités par rapport aux modèles E/A et UML. La première d'entre elles étant l'arité des associations. En effet, il est possible en ORM de créer des associations unaires. Dans les autres modèles, cela peut être simulé par des attributs booléens, ou de façon moins directe par des sous-types. L'utilisation d'associations unaires offre cependant l'avantage de pouvoir facilement représenter des contraintes d'inclusion sur le schéma [54]. Nous pouvons par exemple indiquer que tous les fumeurs sont des personnes susceptibles d'avoir un cancer en créant une contrainte d'inclusion entre deux associations unaires ⟨`fume`⟩ et ⟨`est à risque`⟩.

D'autres contraintes plus complexes, intraduisibles graphiquement dans les modèles E/A et UML, peuvent l'être en ORM (ex. : contraintes d'inclusion jointe) [54].

Une limite que n'ont pas les autres modèles est posée pour la réification : il est impossible de réifier des associations $1 : n$ (contrainte interne d'unicité sur un seul rôle). En effet, il est toujours possible « d'aplatir » une telle réification sans perte d'information sous la forme de plusieurs associations et de contraintes d'inclusion. Autoriser la réification d'associations $1 : n$ ne se justifie que dans des cas très particuliers [53].

Pour compléter le modèle correspondant à un schéma, ORM permet de représenter une partie de la population. Les valeurs sont alors indiquées dans des *tables de faits* qui contiennent une colonne par rôle des associations auxquelles elles correspondent.

4.3.4 Conclusion et limites

ORM met l'accent sur les associations et les rôles qu'y jouent les objets. Il est plus adapté que UML à une modélisation de données (l'implémentation des schémas en relationnel est aisée), mais lui laisse l'avantage dans le domaine applicatif.

On notera qu'il est globalement plus expressif que E/A et UML, mais qu'en contrepartie, les schémas créés sont plus volumineux.

Comme les modèles précédents, ORM est un outil de conception de schémas et ne leur permet pas d'évoluer une fois instanciés.

4.4 *Resource Description Framework* (RDF)

L'émergence d'Internet, et plus généralement du concept de ressource, a créé des besoins spécifiques de modèles sémantiques, basés sur XML. En particulier, RDF [62] (*Resource Description Framework*), XTM [80] (*XML Topic Maps*), et MPEG-7 [30] sont apparus ces dernières années. Bien qu'ils viennent d'origines différentes (description de ressources pour RDF, gestion de documents pour XTM, description de contenus multimédias pour MPEG-7), ils ont tous le même but : représenter de la connaissance de haut niveau portant sur des ressources. Nous n'allons nous intéresser ici qu'au modèle le plus répandu : RDF.

Comme indiqué dans le *Cambridge Communiqué*, RDF est un membre de la famille de modélisation Entité-Association[3] [93]. Cependant, RDF seul n'est qu'un langage généraliste pour représenter de l'information sur Internet. *RDF Schema* [16] a ainsi été développé pour définir des modèles qui vont servir de vocabulaire dans des applications précises. Contrairement aux modèles étudiés jusqu'ici, il n'existe pas de notation graphique pour définir un schéma RDF. Nous pouvons cependant le représenter sous forme de graphe orienté et étiqueté, comme nous le verrons dans la figure 4.8.

4.4.1 Schéma RDF

Observons les différents éléments participant à un schéma RDF.

- $T = \{T_i\}$
  T est un ensemble regroupant les types de base (string, integer, URI...) et les classes définies par l'application (Auteur, Livre...). Les types de base sont regroupés dans une classe particulière `rdfs:Literal` (leurs valeurs sont appelées des littéraux).
- $R = \{R_i : \bigwedge_{k=0}^{m}{}_T\, T^1_{i_k} \times \bigwedge_{k=0}^{n}{}_T\, T^2_{i_k} | \forall k \in [0,m], T^1_{i_k} \in T \wedge \forall k \in [0,n], T^2_{i_k} \in T\}$
  R est un ensemble d'associations binaires entre les types. Généralement, nous avons $m = 1$ et $n = 1$. Dans le cas contraire, une ressource participant à une association doit être instance de tous les types définis pour son rôle. Ces associations binaires sont appelées *propriétés* en RDF/S (`rdf:Property`). Le premier rôle est appelé le *domaine* de la propriété (`rdfs:domain`) et le second son *image* (`rdfs:range`).
  Notons $domain(R_i) = \bigwedge_{k=0}^{m}{}_T\, T^1_{i_k}$ et $range(R_i) = \bigwedge_{k=0}^{n}{}_T\, T^2_{i_k}$.
- $T_i \prec_T T_j$ (inclusion)
  $\prec_T$ est une relation d'ordre partiel entre les types. C'est une association particulière qui indique que toute instance de T_i est aussi instance de T_j. Elle est notée `rdfs:subClassOf` en RDF/S.
  Par extension, nous définissons $\bigwedge_{k=0}^{m}{}_T\, T^1_{i_k} \prec_T \bigwedge_{k=0}^{n}{}_T\, T^2_{i_k} \Leftrightarrow \forall (k,l) \in [0,m] \times [0,n], T^1_{i_k} \prec_T T^2_{i_l}$.
- $R_i \prec_R R_j$ (inclusion)
  $\prec_R$ est une relation d'ordre partiel entre les associations binaires, vérifiant $domain(R_i) \prec_T domain(R_j)$ et $range(R_i) \prec_T range(R_j)$. Elle est notée `rdfs:subPropertyOf` en RDF/S.

[3] "*... RDF is a member of the Entity-Relationship modelling family ...*"

- $N : T \cup R \rightarrow A^*$ telle que $N^{-1} : A^* \not\rightarrow T \cup R$
  N définit une désignation non ambiguë des types et des relations binaires, qui utilise des URI (Universal Resource Identifier) [10].

4.4.2 Instance RDF

Observons maintenant les instances des ensembles introduits.

- $I_T = I_{T_1} \cup \ldots \cup I_{T_{|T|}}$ avec $I_{T_j} = \{v \in T_j\}$
  I_T est l'ensemble de toutes les instances des types de l'application, vérifiant $I_{T_j} \subseteq I_{T_k} \Leftrightarrow T_j \prec_T T_k$. Une ressource peut être instance de plusieurs classes sans relation entre elles : $\forall (j,k) \in [1,|T|]^2, I_{T_j} \cap I_{T_k}$ n'est pas nécessairement vide, aussi bien pour les instances de classes utilisateurs (identifiées par des URI) que pour les littéraux.
- $I_R = I_{R_1} \cup \ldots \cup I_{R_{|R|}}$ avec $I_{R_j} = \{v \in R_j\}$
  I_R est l'ensemble de toutes les instances d'associations, vérifiant $I_{R_j} \subseteq I_{R_k} \Leftrightarrow R_j \prec_R R_k$.
  Un triplet de I_R est appelé un *statement* en RDF, par exemple ("http://www.artchive.com/x.jpeg", "author", "Pablo Picasso"), que nous pouvons aussi traduire par « la propriété `author` de la ressource `http://www.artchive.com/x.jpeg` a pour valeur `Pablo Picasso` ».

RDF en tant que graphe Ainsi, RDF est formalisable par le septuplet suivant :

$$(T, R, \prec_T, \prec_R, N, I_T, I_R)$$

Nous pouvons conceptuellement simplifier le modèle en posant :

$$G = (T \cup I_T, \prec_T \cup \prec_R \cup R \cup I_R, N)$$

soit un graphe muni de labels sur *certains* nœuds et arcs.

Comme nous pouvons le voir, nous retrouvons avec RDF/S la plupart des notions que nous avons introduites précédemment. Quelques particularités existent cependant. Tout d'abord, T n'inclut pas uniquement des classes d'entités définis par l'utilisateur, mais aussi un type spécial pour les littéraux (qui peut être spécialisé en plusieurs sous-types). Nous pouvons le rapprocher de l'ensemble des types d'objets d'ORM qui regroupent les types d'entités et les types de valeurs.

Toute l'information sur les types et sur les liens les unissant est représentée par des associations binaires. Il n'existe ainsi pas de notion d'attribut ou d'association d'une arité différente de deux. Ces associations, appelées *propriétés*, peuvent s'appliquer sur plusieurs types. Dans les modèles vus précédemment, il était nécessaire de généraliser les types pour obtenir le même comportement.

Une association particulière ($\prec_T$) permet de définir des spécialisations / généralisations entre types. Une association portant sur un type est aussi disponible pour tous ses sous-types.

RDF/S introduit aussi une autre association ($\prec_R$) permettant de spécialiser / généraliser les propriétés, c'est-à-dire les associations elles-mêmes. Cela permet notamment de créer des contraintes d'inclusion entre des associations portant sur des types et leurs généralisations. Ce type de contraintes est difficilement exprimable dans les autres modèles de la famille E/A.

Par contre, les contraintes d'inclusion sur les rôles ne sont pas exprimables directement : nous ne pouvons pas spécifier que pour avoir une propriété, une instance doit d'abord en avoir une autre.

4.4.3 Un exemple

Observons ces quelques concepts sur un exemple simple (cf. figure 4.7). Nous noterons que le langage RDF est utilisé pour décrire les schémas RDF. Le méta-modèle est décrit dans le langage du modèle.

Afin d'éviter une syntaxe lourde faisant intervenir des URIs, nous définissons des entités dans la déclaration du type du document et nous utilisons des espaces de noms (*namespaces*) XML [15]. La figure 4.8 présente le même schéma RDF sous la forme d'un graphe. Pour la rendre plus lisible, nous avons utilisé la même syntaxe abrégée de la définition textuelle du schéma, sans pour autant y indiquer les espaces de noms utilisés.

Les mots-clés `rdfs:Class` et `rdf:Property` permettent de déclarer respectivement les types et les associations. Il ne s'agit que de marqueurs syntaxiques permettant d'exprimer de manière plus élégante l'association binaire `rdfs:type` (on utiliserait alors `rdf:resource` comme marqueur). Toutes les informations du schéma ne sont en effet constituées que d'instances d'associations binaires qui sont définies dans le schéma lui-même, dans le schéma définissant RDF/S [16], ou dans tout autre schéma RDF/S.

Lorsqu'elles ont le même domaine (premier rôle), les instances d'associations peuvent être regroupées autour de la valeur de leur domaine afin d'être plus lisibles. C'est la présentation que nous avons choisie dans la figure 4.7. Le nœud XML principal correspond au domaine, les titres de ses fils correspondent aux noms des propriétés alors que les valeurs des portées sont précisées en attribut ou en sous-nœud quand il s'agit d'un littéral.

On remarquera que la propriété `Dirige` est une sous-propriété de `TravaillePour` : toutes les instances de la première de ces associations sont aussi instances de la seconde. Un employé ne peut diriger une entreprise s'il ne travaille pas pour elle.

4.4.4 Conclusion et limites

RDF ne souffre pas des importantes limites de flexibilité des autres modèles de la famille E/A. Il est ainsi facile d'ajouter des éléments dans un schéma et de faire évoluer les instances en conséquence.

En revanche, son manque de contraintes le rend peu utilisable pour modéliser des domaines complexes, et rend notamment les évolutions du schéma et des instances très difficiles à contrôler.

L'implémentation de choix pour RDF étant le système de fichier, il ne supporte pas bien les montées en charge importantes. L'arrivée récente de bases de données dédiées à RDF (ex. : Sesame [17]) permet cependant de répondre en partie à cette limitation.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf  "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#"> ]>

<rdf:RDF
  xml:base="http://www.arisem.com/2004/07/exemple"
    xmlns:rdf="&rdf;"
    xmlns:rdfs="&rdfs;">

  <!-- Classes -->

  <rdfs:Class rdf:ID="Personne">
  </rdfs:Class>

  <rdfs:Class rdf:ID="Etudiant">
      <rdfs:subClassOf rdf:resource="#Personne"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="Employé">
      <rdfs:comment>Un employé ne peut exister que s'il travaille pour une
          entreprise.</rdfs:comment>
      <rdfs:subClassOf rdf:resource="#Personne"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="Entreprise">
  </rdfs:Class>

  <!-- Properties -->

  <rdf:Property rdf:ID="NomPersonne">
      <rdfs:label>Nom</rdfs:label>
      <rdfs:domain rdf:resource="#Personne"/>
      <rdfs:subPropertyOf rdf:resource="&rdfs;label"/>
  </rdf:Property>

  <rdf:Property rdf:ID="NomEntreprise">
      <rdfs:label>Nom</rdfs:label>
      <rdfs:domain rdf:resource="#Entreprise"/>
      <rdfs:subPropertyOf rdf:resource="&rdfs;label"/>
  </rdf:Property>

  <rdf:Property rdf:ID="Salaire">
      <rdfs:domain rdf:resource="#Employé"/>
      <rdfs:range rdf:resource="&rdfs;Literal"/>
  </rdf:Property>

  <rdf:Property rdf:ID="TravaillePour">
      <rdfs:label>Travaille pour</rdfs:label>
      <rdfs:domain rdf:resource="#Personne"/>
      <rdfs:range rdf:resource="#Entreprise"/>
  </rdf:Property>

  <rdf:Property rdf:ID="Dirige">
      <rdfs:subPropertyOf rdf:resource="#TravaillePour"/>
  </rdf:Property>

</rdf:RDF>
```

Figure 4.7 – Exemple de schéma RDF

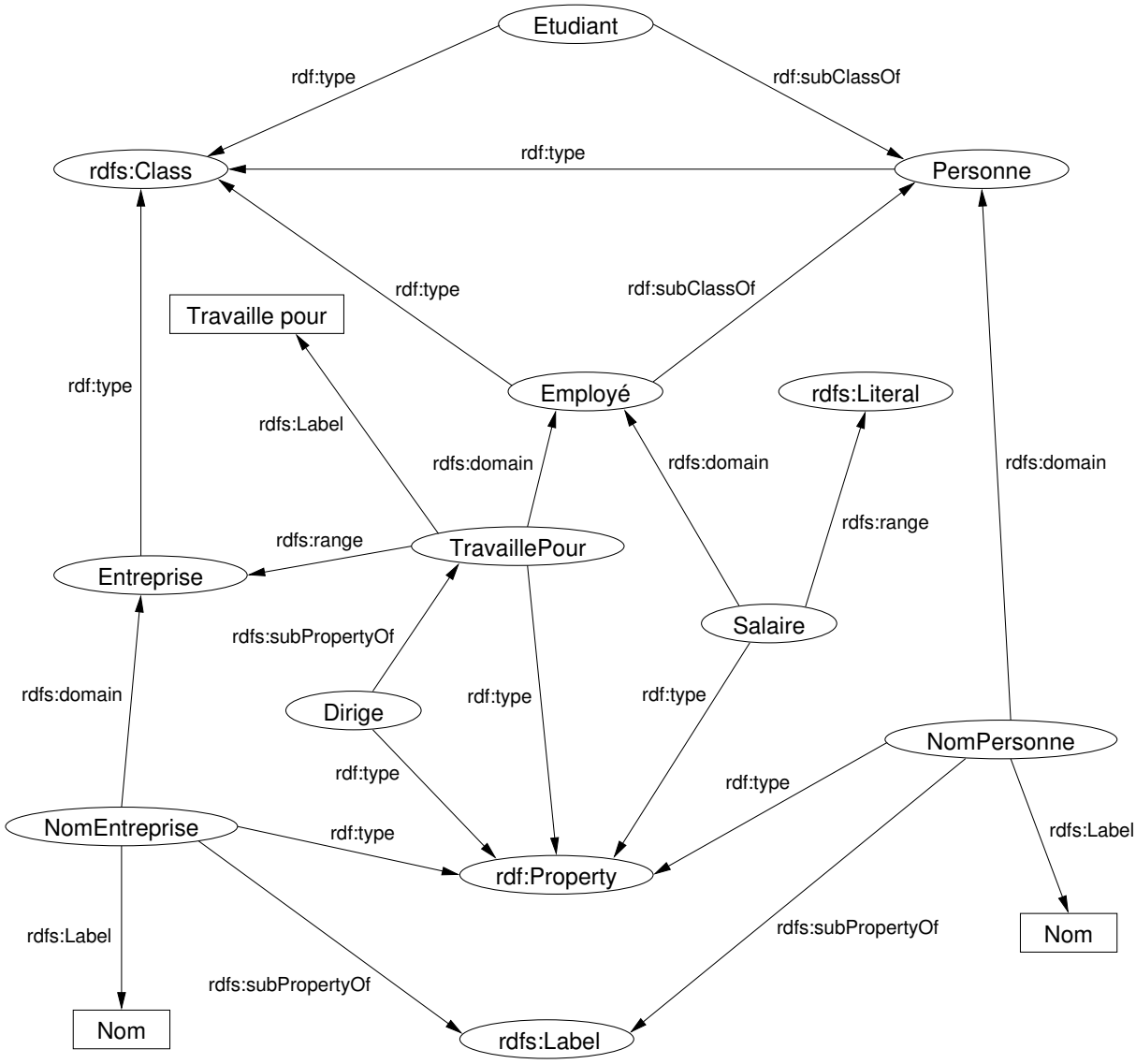

Figure 4.8 – Un schéma RDF sous forme de graphe

CHAPITRE 5

Modèles à base de rôles

Les modèles à objets font partie des extensions du modèle E/A (ex. : OMT, UML) qui incorporent du code dans les descriptions des classes d'entités, et qui accentuent l'importance de la relation d'héritage. Ils sont très adaptés à la modélisation d'applications, mais ne s'intéressent pas directement au stockage des données.

Cette limitation reçoit une réponse avec les Systèmes de Gestion de Bases de Données à Objets *(SGBDO), qui allient un modèle à objets avec des capacités de persistance. Les* rôles *sont ensuite venus les compléter pour répondre à des besoins de flexibilité et d'évolution.*

Après avoir introduit les systèmes à objets, ce chapitre détaille leur extension avec des rôles.

5.1 Modèles et bases de données à objets

5.1.1 Encapsulation

Alors que l'encapsulation n'était pas centrale dans les approches E/A, elle constitue le principal centre d'intérêt des chercheurs en modélisation d'applications, qui avaient besoin de créer des composants logiciels pouvant être échangés. À partir des fondements mathématiques de la théorie des types de données abstraits, ces recherches ont donné lieu à la programmation à objets.

Le concept d'encapsulation dans les langages de programmation vient donc des types de données abstraits. Comme nous l'avons vu en section 3.2, un objet a dans cette approche une interface, visible, et une ou plusieurs implémentations, cachées à l'utilisateur. Cette séparation entre interface et implémentation permet de séparer complètement l'utilisation de l'objet de sa structure interne. Ainsi, les programmes utilisant un objet sont protégés des changements dans l'implémentation.

5.1.2 Bases de données à objets

Les bases de données relationnelles ne supportent pas les formes non normalisées pour leurs relations. De nombreuses notions de modélisation introduites au chapitre 3 n'y sont pas disponibles.

Pour pallier à ce manque, une autre approche est née du rapprochement des communautés « bases de données » et « objet », et a engendré les Systèmes de Gestion de Bases de Données à Objets (SGBDO) [6, 8]. Ces systèmes combinent un modèle expressif à des capacités de traitement d'un important volume de données persistantes.

Contrairement aux modèles de type E/A ou ORM qui s'appuient sur la notion de valeur, les SGBDO s'appuient sur la notion d'identifiant interne ou OID (*Object IDentifier*). Dans un tel modèle, l'existence d'un objet est indépendante de la valeur de ses attributs. Comme nous l'avons vu, l'*identité* est différente de l'*égalité* [59] (cf. *notion d'identité* en section 3.1).

Outre la notion d'OID, les SGBDO s'approprient des notions venant du monde des objets :

- *Objets complexes.* Les instances peuvent être aussi simples que des entiers ou chaînes de caractères, mais aussi utiliser des constructeurs complexes récursivement (familles, tableaux, listes, etc.).
- *Encapsulation.* Les objets encapsulent à la fois les données et les programmes, et ne publient qu'un ensemble d'opérations possibles.
- *Hiérarchies de classes.* Les objets sont regroupés dans des ensembles appelés classes. Les classes définissent le comportement des objets qui les instancient et peuvent être organisées dans des hiérarchies de spécialisation / généralisation.
- *Polymorphisme, surcharge et assignation tardive (*late binding. Le polymorphisme permet à des opérations ayant des signatures différentes de partager le même nom. Une opération différente peut, par exemple, être définie pour chaque type de paramètre. La surcharge permet, quant à elle, de redéfinir une opération dans une sous-classe. Ainsi, il sera possible de spécialiser le comportement hérité d'une super-classe (l'opération la plus spécialisée sera appelée). La détermination de l'opération à exécuter est alors faite lors de l'exécution (assignation tardive).

Les fonctionnalités des SGBDO ont été standardisés par l'ODMG (*Object Data Management Group*) [22].

Si les SGBD à objets sont plus expressifs que les SGBD relationnels, tout comme les modèles à objets dont ils sont issus, ils manquent de flexibilité. S'inspirant des travaux sur le modèle relationnel, un système de *vues* sur objets a été proposé [1]. Ce mécanisme permet au concepteur de spécifier des attributs implicitement, au lieu de les stocker, et d'introduire des classes « virtuelles » dans la hiérarchie de classes. Ces nouvelles classes peuvent être définies soit par :

- *Spécialisation* : la classe ADULTE est, par exemple, définie comme contenant toutes les instances de PERSONNE dont l'âge est de 18 ans ou plus (la valeur de l'âge peut aussi être spécifiée en paramètre de la classe donnant lieu à une *classe paramétrée*) ;
- *Généralisation* : la classe ENTITÉ LÉGALE est définie comme incluant les classes PERSONNE et ENTREPRISE ;
- *Généralisation comportementale* : la classe IMPRIMABLE inclut toutes les classes ayant une opération `imprimer`.

Ces différents mécanismes sont cependant très limités quant à l'évolution d'un objet au cours de sa vie. Il est uniquement possible de le faire apparaître dans de nouvelles classes virtuelles si certains de ses attributs changent. Ce besoin a conduit à la création de la notion de *rôle*.

5.2 Extension des systèmes à objets avec des rôles

L'un des premiers systèmes à base de rôle était celui de Bachman et Daya [7] qui étendait l'expressivité d'un modèle réseau avec la notion de rôle. Par la suite, les propositions autour de la notion de rôle se sont orientées davantage vers les systèmes à objets et notamment les SGBD à objets.

Dans le modèle E/A et ses extensions (UML et ORM notamment), le terme *rôle* désignait uniquement un emplacement nommé pour une association. C'est un autre sens que nous lui attribuons ici.

5.2.1 La notion de rôle

La plupart des modèles à objets considèrent qu'un objet ne peut être instance que d'une classe à la fois (en dehors de la généralisation) et qu'il ne peut pas en changer. Ces hypothèses ne sont cependant pas adaptées à la modélisation de certaines situations dynamiques du monde réel. L'exemple le plus repris dans la littérature [100, 47, 64, 28] est celui d'un étudiant devenant un employé. Dans un modèle objet classique, les classes ÉTUDIANT et EMPLOYÉ ont la même super-classe PERSONNE, mais une instance

créée en tant qu'étudiant ne peut devenir ensuite un employé. Dans un modèle à base de rôles, ÉTUDIANT et EMPLOYÉ sont des rôles qu'une personne peut jouer parmi d'autres. Ils permettent de voir la personne sous un point de vue particulier.

Le concept de rôle répond donc à deux besoins particuliers apparaissant lorsque l'on veut modéliser des entités évolutives avec des modèles à objets :

- le changement dynamique de type,
- un accès dépendant du contexte (utilisation d'une perspective particulière).

Dahchour *et al.* ajoutent un troisième besoin à cette liste [28] : l'instanciation multiple du même type (ex. : un étudiant peut être enregistré dans deux universités, un chef de projet peut diriger plusieurs projets). Toutes les approches à base de rôles ne permettent cependant pas cette multi-instanciation.

Tout type n'est cependant pas considéré comme un rôle : indépendamment du rôle qu'une personne peut tenir à un moment donné (ex. : étudiant ou employé), elle restera toujours une personne. Guarino présente ainsi une distinction entre les types de rôles et les types naturels [49]. Pour qu'un concept soit un type de rôles, il faut que ses instances soient en relation avec d'autres instances et qu'elles puissent entrer ou sortir du domaine du concept sans perdre leur identité. Au contraire, les instances de types naturels ne peuvent pas perdre leur type sans perdre leur identité et il n'est pas requis qu'elles soient en relation avec d'autres instances. PERSONNE, par exemple, est un type naturel car ses instances seront toujours des personnes et que leur existence est indépendante de la présence d'associations. D'un autre côté, ÉTUDIANT est un type de rôle puisque pour être étudiant, une inscription dans une université est nécessaire et que la fin des études n'entraîne pas la perte d'identité. On notera que cette définition laisse de côté certains types comme ENFANT par exemple : une personne peut passer de enfant à adulte sans perdre son identité alors qu'aucune association n'est nécessaire pour son existence. Ces types sont alors considérés comme des *états* d'une instance.

Pour plus de simplicité, nous préférons la définition de classes d'objets et de classes de rôles donnée par Wieringa et de Jonge [100] :

- une instance d'une classe d'objets ne peut exister sans être instance de cette classe,
- il existe des états possibles du monde pour lesquelles une instance d'une classe de rôles peut exister, en tant qu'objet, sans être instance de cette classe de rôles.

Cette définition ne force pas un rôle à apparaître dans une association.

5.2.2 Identité d'objet et identité de rôle

Si toutes les approches considèrent qu'un objet conserve toujours son identité lorsqu'il acquiert ou perd des rôles, elles sont en désaccord quant à la notion d'identité de rôle (*RID* pour *Role IDentity*, à opposer à la notion de *OID* pour *Object IDentity*).

La littérature se divise ainsi entre les propositions qui refusent aux rôles une identité distincte de leur objet associé [2, 102, 89, 27] et celles qui les identifient indépendamment [100, 47, 64, 28]. Considérons, par exemple, que la personne Pierre devienne un étudiant puis un employé. Dans la première approche, Pierre est représenté par un seul OID, alors que dans la deuxième, il est représenté par un OID, celui de son premier état en tant que PERSONNE, et par deux RID indépendants qui correspondent à Pierre respectivement en tant qu'étudiant et en tant qu'employé.

La notion d'identité de rôle a été introduite dans [100] et est généralement couplée à une relation particulière « joué-par ». Cette association entre un objet et ses rôles permet d'identifier à partir d'un rôle quel est l'objet qui le joue. L'utilisation de RID a trois avantages directs :

- Elle facilite la multi-instanciation de classes de rôles. Pour indiquer que Pierre est inscrit dans deux universités différentes, il suffit d'instancier deux fois la classe de rôles ÉTUDIANT, chaque

instance ayant son RID propre et étant jouée par le même objet (un seul OID).

- La relation « joué-par » peut être facilement étendue pour qu'un rôle puisse aussi être joué par un rôle, créant ainsi une arborescence dont la racine est toujours un objet. Pierre en tant qu'employé peut par exemple jouer le rôle de CHEF DE PROJET.
- Différencier les OID des RID permet de compter séparément les objets et leurs rôles. Prenons l'exemple de la classe d'objets PERSONNE et de la classe de rôles PASSAGER et intéressons-nous au nombre de passagers prenant le métro en une journée. À chaque fois qu'une personne prend le métro, elle acquiert un nouveau rôle passager avec un RID unique. Ainsi, nous pouvons facilement différencier le nombre de passagers du nombre de personnes réelles ayant pris le métro.

Les rôles d'un objet concerne toujours le même objet qui apparaît dans différents contextes. Ainsi, d'autres propositions considèrent que les instances des classes de rôles ne doivent pas avoir d'identité distincte. La relation « joué-par » devient alors inutile. L'implémentation en est plus simple et potentiellement plus performante, mais la multi-instanciation de rôles et les rôles sur rôles sont alors beaucoup plus complexes à modéliser.

Albano *et al.* [2], ainsi que Coulondre et Libourel [27] n'autorisent simplement pas ces notions. Quant à Steimann [89], il permet la multi-instanciation mais avec une approche complexe, s'appuyant sur un modèle dynamique où chaque rôle n'existe que dans le cadre d'une association. Enfin, le modèle DOOR [102] (*Dynamic Object-Oriented database programming language with Roles*) évite l'utilisation de références sur les rôles en les identifiant par le nom de leur classe et leurs valeurs. Quand un objet n'a qu'un rôle d'un certain type, ses valeurs peuvent être omises (ex. : `pierre!Étudiant.id` désigne le numéro d'étudiant de Pierre). Dans le cas contraire, un critère booléen sur les valeurs est ajouté pour contrôler le résultat (ex. : `pierre!(Étudiant|université="Nantes").id`). DOOR permet aussi aux rôles de jouer eux-mêmes des rôles. Dans ce cas, l'identification d'un rôle part forcément de l'objet racine (ex. : `pierre!Employé!ChefDeProjet` désigne le rôle CHEF DE PROJET joué par le rôle EMPLOYÉ de Pierre). Il est impossible de référencer directement un rôle.

5.2.3 Hiérarchie de rôles

Comme les classes d'objets, les classes de rôles peuvent être organisées dans une hiérarchie de spécialisation. La classe de rôles DOCTORANT peut, par exemple, être vue comme une sous-classe de la classe de rôles ÉTUDIANT. Un objet ayant le rôle DOCTORANT a donc automatiquement le rôle ÉTUDIANT. De plus, comme pour les objets, chaque sous-classe hérite des propriétés et valeurs de sa super-classe.

Si les différentes approches convergent sur la notion de hiérarchie de classes de rôles, elles diffèrent par contre quant à la relation que doit avoir cette hiérarchie avec les classes d'objets (et leur hiérarchie).

Dans le modèle proposé par Albano *et al.* [2], chaque classe d'objets est associée à une hiérarchie de classes de rôles. Les modèles de Coulondre et Libourel [27], Dahchour *et al.* [28] et Gottlob *et al.* [47], vont plus loin : une sous-classe d'objets hérite de la hiérarchie de classes de rôles de sa super-classe. Cette approche a cependant une limite : les notions de classes d'objets et de classes de rôles, à rapprocher des types naturels ou de rôles, sont orthogonales. Par exemple, aussi bien des PERSONNEs que des ENTREPRISEs peuvent jouer les rôles d'ACTIONNAIRE ou de CLIENT. Ici, ces rôles doivent être dédoublés, ou une super-classe d'objets généralisant PERSONNE et ENTREPRISE doit être créée[1].

Pour éviter cet écueil, d'autres approches définissent des hiérarchies de classes d'objets et de rôles indépendantes. Dans ce cas, une relation doit lier les classes d'objets aux classes de rôles qu'elles peuvent jouer (ex. : *played-by* dans [100], *role-filler* dans [89]). DOOR [102] utilise une autre approche consistant

[1] Dans le cadre des modèles E/A la notion de catégorie a été introduite pour répondre à ce besoin [38].

à qualifier les rôles : la définition d'un rôle inclut la liste des classes (d'objets et de rôles), dont les instances peuvent le jouer.

Il existe deux types possibles d'héritage entre une classe et ses sous-classes : la *spécialisation* ou la *délégation*. L'héritage par spécialisation est l'héritage traditionnel entre classes : une sous-classe (d'objets ou de rôles) peut être utilisée partout où sa super-classe est demandée et hérite de toutes ses propriétés. L'héritage par délégation fait le pont entre les objets et leurs rôles. Si un rôle est interrogé pour une propriété qu'il n'a pas, il peut déléguer la question à l'objet qui le joue. Albano *et al.* [2] suivent une approche particulière du fait qu'ils ne distinguent pas l'implémentation des classes d'objets de celles de rôles : seul l'héritage par spécialisation est utilisé et un objet n'est manipulé que *via* ses rôles.

5.2.4 Contraintes sur les rôles

Si l'utilisation d'une hiérarchie de classes de rôles permet d'exprimer naturellement certaines contraintes (ex. : le fait que tout chef de projet est un employé est simplement exprimé par une relation de spécialisation entre les classes de rôles respectives), d'autres sont plus difficiles à spécifier. Aucune des approches étudiées ne permet par exemple d'indiquer simplement qu'il est impossible à un objet de jouer en même temps les rôles EMPLOYÉ et CHÔMEUR.

Quelques approches [47, 91, 89, 28] proposent d'autre part de définir des règles de transition qui permettent de définir les séquences permises d'acquisition ou de perte de rôle.

5.3 Conclusion et limites

Au même titre que les modèles de la famille E/A, les modèles à base de rôles offre une bonne expressivité. En particulier, ceux qui sont construits sur des SGBDO permettent l'utilisation des structures complexes proposées par l'ODMG.

Considérant la flexibilité, les modèles à base de rôles offrent également une extension intéressante sur les modèles à objets traditionnels : ils permettent de changer dynamiquement le type d'une instance. Cependant, seuls les rôles joués par un objet peuvent évoluer, sa classe de base est, elle, immuable. La notion de rôle permet également à ces modèles d'offrir un accès contextuel aux entités.

Dans les implémentations, les classes de rôles sont souvent représentées par des interfaces ou des classes abstraites (héritage multiple). Les implémentations s'appuyant sur des bases de données sont rares. Celles qui le sont apparaissent comme les moins expressives (hiérarchies de classes de rôles liées à une classe d'objets, pas de contraintes, pas de RID, etc.), et sont limitées par le manque d'adoption des SGBDO par l'industrie.

CHAPITRE 6

Modèles à base de frames

Les modèles de la famille E/A sont pour la plupart empreints d'une culture issue du monde des bases de données. En parallèle à ces travaux, la communauté de l'intelligence artificielle (IA) s'est concentrée sur des aspects plus dynamiques. Ces recherches ont d'abord été dominées par les systèmes à base de frames*, avant de se concentrer sur les logiques de description qui en héritent (DL pour* Description Logics*). Ces deux modèles sont l'objet des deux premières parties de ce chapitre. Nous le terminerons en étudiant succinctement le* Web Ontology Language *(OWL), récemment approuvé par le W3C [98], qui s'appuie à la fois sur les travaux portant sur les DL et sur le standard RDF.*

6.1 Systèmes à base de *frames*

Le concept de *frame* a été proposé par Martin Minsky au milieu des années 70 [73]. Il a ensuite donné lieu à la création de plusieurs dizaines de systèmes à base de *frames*, dont les principales familles sont basées sur SRL [43], KL-ONE [14] et *UNIT Package* [88]. Ces propositions, bien que basées sur des principes similaires, se sont appuyées sur des choix très différents et souvent incompatibles [58]. Cette hétérogénéité a motivé la spécification d'une API (*Application Programming Interface*) commune similaire à ODBC (*Open Data Base Connectivity*) pour le monde des bases de données relationnelles : OKBC (*Open Knowledge Base Connectivity*) [24].

La description détaillée des différences de chaque système n'étant pas pertinente dans le cadre de cette étude des modèles d'information, nous nous contenterons de décrire le modèle générique sur lequel se base OKBC.

6.1.1 *Frames, slots* et facettes

La notion de *frame* peut être vue comme une autre façon de nommer les entités. Cependant, comme dans le modèle RDF où les instances et les classes sont des « ressources », une *frame* peut aussi bien décrire une entité qu'un ensemble d'entités. Ces deux concepts ayant des caractéristiques différentes, nous distinguons deux catégories de *frames* : les instances (ou individus) et les classes. Appelons E l'ensemble des *frames* [1], I l'ensemble des instances et C l'ensemble des classes du système. Nous avons alors $E = I \cup C$.

Les données concernant les *frames* sont représentées par des *slots*. Chaque *frame* est ainsi associée à un ensemble de *slots* qui lui sont propres, appelés *own slots*. Chacun d'entre eux décrit soit une propriété

[1] Nous préférons utiliser E (pour entité) pour éviter toute confusion avec l'ensemble F des facettes que nous allons introduire plus loin.

d'un *frame*, soit une association binaire entre deux *frames*. Appelons S l'ensemble des *slots* du système, nous avons alors la fonction partielle :

$$\forall s \in S, s : E \not\rightarrow 2^{E \cup V} \tag{6.1}$$

où V représente l'ensemble des valeurs des types de base et $2^{E \cup V}$ représente l'ensemble des parties de l'ensemble $E \cup V$.

Les *slots* sont à leur tour décrits par des *facettes*. Chaque couple (*frame*, *own slot*) est ainsi associé à un ensemble de facettes qui lui sont propres, appelées *own facets*. Les facettes permettent de décrire les *slots* d'une *frame*. Nous pouvons les utiliser pour ajouter un commentaire ou une valeur par défaut, mais elles sont surtout utiles pour exprimer des contraintes. Une facette peut, par exemple, restreindre le type des valeurs d'un *slot* aux instances d'une classe précise (facette `:VALUE-TYPE` dans OKBC). Appelons F l'ensemble des facettes du système, nous avons alors la fonction partielle :

$$\forall f \in F, f : (S, E) \not\rightarrow 2^{E \cup V} \tag{6.2}$$

On notera que les *slots* et facettes peuvent être réifiés en tant que *frames*, ce qui permet de décrire les propriétés des relations qu'elles représentent.

Les systèmes à base de *frames* ont aussi la particularité de permettre l'attachement aux *slots* de procédures, appelées *dæmons*, pour déclencher des actions quand on y accède. Des *dæmons* peuvent notamment surveiller les événements d'ajout, de suppression, de modification de la valeur d'un slot, ou encore la création d'une frame avec ce slot. Ce mécanisme permet de réagir aux changements d'état du système et de l'adapter à des besoins particuliers.

6.1.2 Classes

Une classe est un ensemble de *frames* qui sont appelées ses instances. Une *frame* peut être instance de plusieurs classes. Une classe étant un type particulier de *frame*, nous en déduisons qu'elle peut elle-même être instance d'une classe (dans ce cas, $I \cap C \neq \emptyset$). Appelons c^I l'ensemble des instances de la classe c.

Comme tous les modèles étudiés jusqu'ici, ceux à base de *frames* permettent de faire des spécialisations / généralisations et proposent dans ce but les notions de *super-classe* et de *sous-classe*. Si l'on note $\prec$ la relation de spécialisation, nous avons alors : $c_1 \prec c_2 \Leftrightarrow c_1^I \subseteq c_2^I$.

Outre les *own slots* et leurs *own facets*, une classe peut recevoir des *template slots* et leurs *template facets*. Les *template slots* d'une classe permettent de décrire les *own slots* que toutes ses instances doivent avoir, ainsi que les *template slots* que toutes ses sous-classes doivent avoir. Les *template slots* permettent ainsi de représenter les propriétés de types que nous avons introduites en section 3.2.

Les équations 6.1 et 6.2 sont valables pour les deux types de *slots* et facettes. Pour les différencier, nous nous proposons d'utiliser les assertions suivantes :

$\forall(s,e,v) \in (S,E,E \cup V)$	$own(s,e,v)$	$\Leftrightarrow$	s est un *own-slot* de la *frame* e et a pour valeur v
$\forall(f,s,e,v) \in (F,S,E,E \cup V)$	$own(f,s,e,v)$	$\Leftrightarrow$	f est une *own-facet* du *own-slot* s pour la *frame* e et a pour valeur v
$\forall(s,c,v) \in (S,C,E \cup V)$	$template(s,c,v)$	$\Leftrightarrow$	s est un *template-slot* de la classe c et a pour valeur v
$\forall(f,s,c,v) \in (F,S,E,E \cup V)$	$template(f,s,c,v)$	$\Leftrightarrow$	f est une *template-facet* du *template-slot* s pour la classe c et a pour valeur v

L'instanciation et l'héritage se traduisent alors par :

$$\begin{aligned}
template(s,c,v) &\Rightarrow \forall i \in c^I, own(s,i,v) \\
template(f,s,c,v) &\Rightarrow \forall i \in c^I, own(f,s,i,v) \\
c_1 \prec c_2 \wedge template(s,c_2,v) &\Rightarrow template(s,c_1,v) \\
c_1 \prec c_2 \wedge template(f,s,c_2,v) &\Rightarrow template(f,s,c_1,v)
\end{aligned}$$

Il n'est pas nécessaire pour un *slot* d'avoir une valeur pour exister : dès qu'une facette a une valeur pour un couple (*slot, frame*), nous considérons que la *frame* possède le *slot*. Cette particularité permet notamment de définir des *templates slots* au niveau des classes tout en permettant aux instances d'avoir des valeurs différentes. Nous retrouvons ici l'héritage classique de propriétés :

$$\begin{aligned}
\exists v_1 \in E \cup V | template(s,c,v_1) \\
\vee \exists f \in F, \exists v_2 \in E \cup V | template(f,s,c,v_2) &\Rightarrow template\text{-}exists(s,c) \\
\exists v_1 \in E \cup V | own(s,c,v_1) \\
\vee \exists f \in F, \exists v_2 \in E \cup V | own(f,s,c,v_2) &\Rightarrow own\text{-}exists(s,c) \\
template\text{-}exists(s,c) &\Rightarrow \forall i \in c^I, own\text{-}exists(s,i) \\
c_1 \prec c_2 \wedge template\text{-}exists(s,c_2) &\Rightarrow template\text{-}exists(s,c_1)
\end{aligned}$$

6.1.3 Conditions suffisantes et nécessaires

Contrairement aux modèles de la famille E/A pour lesquelles l'appartenance d'une entité à une classe est déclarative, les systèmes à base de *frames* peuvent s'appuyer sur des *conditions*. Nous distinguons dans ce cadre deux types de classes dans OKBC : les classes primitives et non-primitives.

Les valeurs des *template slots* et facettes associées à une classe non-primitive sont considérées comme spécifiant des conditions *nécessaires* et *suffisantes* pour qu'une *frame* soit instance de cette classe. La classe DOCUMENT, en particulier, pourrait être une classe non-primitive définie par un auteur et un contenu. Savoir qu'une entité a un auteur et un contenu suffit pour conclure qu'il s'agit d'un document.

En revanche, les valeurs des *template slots* et des facettes associées à une classe primitive, sont considérées comme spécifiant seulement des conditions *nécessaires* à l'appartenance à la classe. Typiquement, les classes représentant des « choses naturelles » sont des classes primitives. Une base de connaissances peut spécifier beaucoup de propriétés d'une personne par exemple (nom, adresse, etc.), mais ne contiendra généralement pas de conditions suffisantes pour conclure qu'une entité est une personne.

6.1.4 Du général au particulier

OKBC a été créé non pas comme un système à base de *frames* à part entière, mais comme une interface d'accès commune aux systèmes à base de *frames* existants ou à venir. Ainsi, le modèle décrit ci-dessus se veut suffisamment général et flexible pour être capable de représenter différents formalismes.

Chaque système à base de *frame* a ses particularités et restreindra généralement les opérations fournies par OKBC. Protégé-2000 [77], par exemple, utilise une démarche plus proche des modèles traditionnels dans le cadre d'un système dédié à l'acquisition de connaissances. Ainsi, une part de la généricité du modèle de OKBC est sacrifiée au profit d'une plus grande simplicité d'usage : une *frame* doit instancier une classe et une seule, un *own slot* est toujours dérivé d'un *template slot* (il est impossible d'attacher un *slot* directement à une instance), etc.

6.1.5 Conclusion et limites

Si les modèles à base de rôles héritent d'une partie des concepts de la famille E/A, les modèles à bases de *frames* ont, quant à eux, été développés dans un cadre beaucoup plus indépendant.

Leur expressivité est très bonne. Ils n'offrent pas de structures complexes, mais ils permettent en revanche de définir un type par de simples conditions d'appartenance. Les contraintes y sont également plus riches, autorisant une modélisation plus fine du domaine.

D'autre part, provenant de la communauté *intelligence artificielle*, l'approche *frames* s'est plus concentrée sur les aspects dynamiques et d'inférence que sur la capacité de traitement de données applicatives. Cela a permis à ces systèles de disposer d'une flexibilité plus importante que les modèles de la famille E/A. En revanche, s'ils peuvent aisément modéliser des domaines et leurs contraintes, ils ne sont pas adaptés à la modélisation et au traitement d'importants volumes de données (ex. : bases documentaires). En effet, ils travaillent généralement uniquement en mémoire vive et sauvegardent leurs données ponctuellement sous forme d'un fichier. Ils sont donc limités très rapidement par la quantité de mémoire disponible.

6.2 Logiques de description

Les logiques de description, appelées communément DL pour *Description Logics*, ont été introduites comme un effort visant à unifier et à donner une base logique formelle à divers systèmes à base de *frames*, ainsi qu'aux réseaux sémantiques, aux représentations à objets, aux modèles sémantiques, etc. [20, 18]. Elles sont basées sur la logique des prédicats qu'elles étendent avec des constructeurs ayant une sémantique de plus haut niveau.

Les premières propositions concrètes sont arrivées avec le système KL-ONE [14] à la fin des années 70. Par rapport aux systèmes à base de *frames* plus classiques[2], les logiques de description limitent leur expressivité pour permettre des mécanismes d'inférence valides, complets et traçables. Des propositions plus récentes sont cependant capables de conserver une expressivité importante sans pour autant sacrifier l'efficacité et la complétude du raisonnement [68].

Le modèle de connaissance des logiques de descriptions peut se résumer à une restriction du modèle plus générique de OKBC. Les DL décrivent la connaissance au moyen de *concepts* et de *rôles* qui correspondent respectivement à des prédicats unaires et binaires. Dans le formalisme des *frames*, les concepts correspondent à des classes et les rôles à des *template slots*.

[2] Karp considère les logiques de description et KL-ONE en particulier comme un système à base de *frames* [58].

Les concepts sont définis par des constructeurs utilisant principalement des opérateurs booléens sur les concepts et des quantifications sur les rôles. D'autres constructeurs peuvent inclure des restrictions de nombre sur les rôles ou encore des constructeurs sur les rôles eux-mêmes comme l'intersection, la concaténation ou l'inverse. Ces assertions logiques permettent de définir non seulement des conditions nécessaires qu'un objet instanciant un concept doit remplir, mais aussi des conditions suffisantes d'appartenance. Par exemple, le concept DIPLÔMÉ peut être défini par l'assertion $\texttt{Personne} \sqcap \exists \texttt{diplôme}$, c'est-à-dire l'ensemble des objets instances du concept PERSONNE qui sont connectés par le rôle `diplôme` à au moins un objet.

L'utilisation de la logique des prédicats permet aux logiques de description de classifier automatiquement les objets et de construire une hiérarchie de classes (cf. *raisonnement* en section 3.4).

Les logiques de description souffrent généralement des mêmes limites que les systèmes à base de *frames* quant à leur support de la charge. KL-ONE, par exemple, utilise le langage LISP [87] pour modéliser l'information. Une base de connaissances correspond à un ensemble d'expressions LISP qui doivent être exécutées au lancement de l'application (la classification doit être refaite à chaque fois). Cette approche, appropriée pour de petites bases, n'est plus du tout adaptée lorsqu'il s'agit de traiter d'importants volumes d'information.

6.3 *Web Ontology Language* (OWL)

OWL, bien qu'arrivé récemment avec l'avènement du *Web* sémantique, s'appuie sur des travaux plus anciens et notamment sur les systèmes à base de *frames* et sur les logiques de description.

L'exigence d'une intégration des données sur la Toile a conduit à la création du langage d'ontologies du DARPA *Agent Markup Language* : le DAML-ONT [90]. Il se base sur RDF et RDF *Schema*, ce qui permet l'interopérabilité avec l'importante base de contenus déjà existante sur Internet. Parallèlement, une autre initiative, elle aussi basée sur les langages de la Toile, a été à l'origine du *Ontology Inference Layer* (OIL) [42] qui est une vraie logique de description. Les buts de ces langages étant très similaires, le meilleur de chacun fut associé dans un nouveau langage appelé DAML+OIL [70]. C'est ce langage, en incorporant les enseignements tirés de son utilisation, qui a servi de base à OWL [86].

6.3.1 Ontologies

Comme son nom l'indique, le but premier de OWL est de décrire des ontologies. Comme le décrit Tom Gruber dans son interview du bulletin AIS SIGSEMIS [65], les ontologies sont avant tout des traités permettant de communiquer[3]. Dans le contexte de la modélisation de la connaissance, il définit plus précisément une ontologie comme étant une spécification d'une conceptualisation (*« a specification of a conceptualization »*) [48]. Cette définition donne lieu à beaucoup de variations dans son interprétation. McGuinness [69] propose un spectre de définitions des ontologies en fonction de l'expressivité de leurs spécifications. Nous le reproduisons dans la figure 6.1.

L'une des notions les plus simples correspondant à une ontologie est celle de vocabulaire. Les catalogues en sont un exemple : ils fournissent une interprétation sans ambiguïté des termes en leur associant un identificateur unique. Les glossaires vont un peu plus loin en associant des définitions aux termes (ces interprétations sont éventuellement ambiguës, mais peuvent aussi être combinées avec des identificateurs). Les thésaurus sont les premiers à introduire des relations entre les termes, typiquement les synonymes et parfois les hyponymes.

[3] *« Every ontology is a treaty – a social agreement – among people with some common motive in sharing. »*

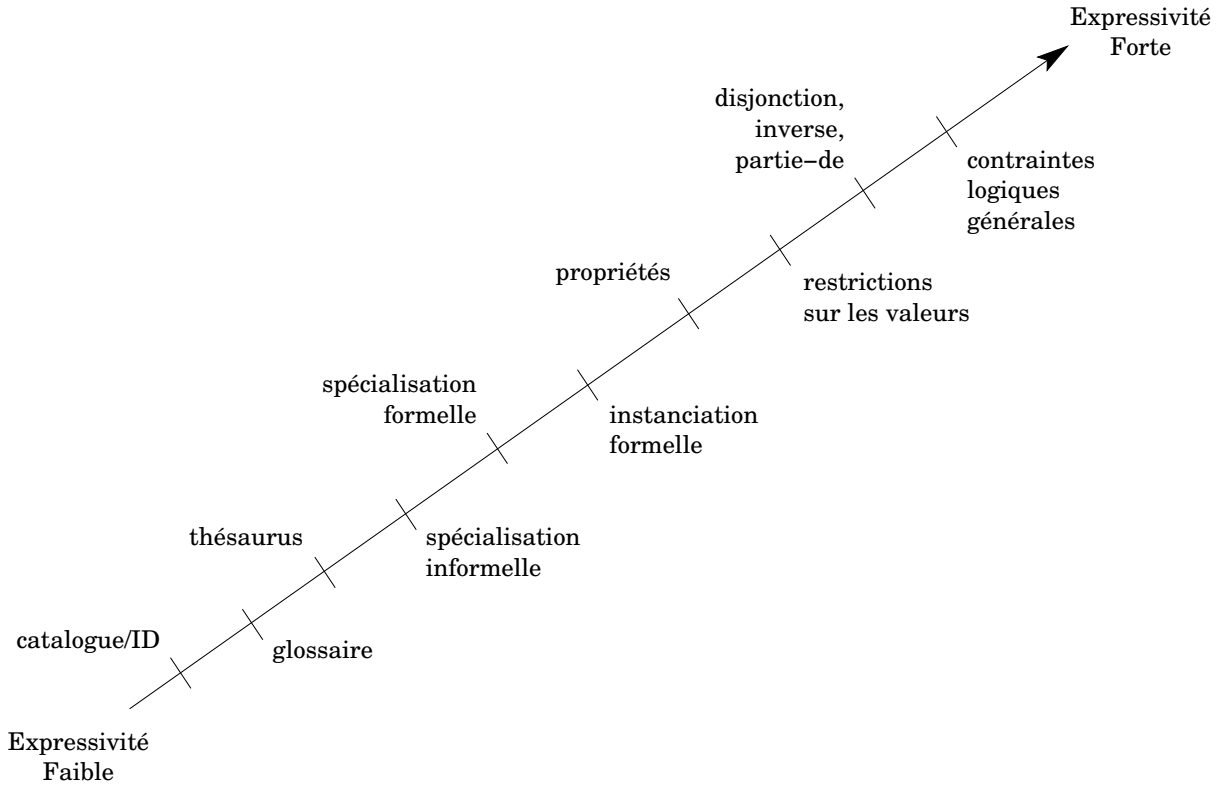

Figure 6.1 – Spectre de définitions des ontologies

Des hiérarchies informelles sont disponibles sur la Toile. Yahoo par exemple fournit une hiérarchie explicite sous la forme d'un arbre de catégories, mais la relation de spécialisation utilisée n'est pas toujours stricte : il peut arriver qu'une instance d'une sous-classe ne soit pas instance d'une super-classe (la rubrique VÉHICULE contient par exemple une sous-rubrique CONSEIL qui ne correspond pas à une vraie spécialisation). Les hiérarchies de spécialisation / généralisation formelles vont plus loin en forçant toute sous-classe d'une classe C à être aussi sous-classe de toutes les super-classes de C, ce qui est nécessaire pour exploiter la notion d'héritage. La relation d'instanciation formelle implique en plus que toute instance d'une classe C soit aussi instance de toutes les super-classes de C.

Le point suivant dans le spectre introduit la notion de propriété qui est particulièrement intéressante dans le cadre de l'héritage. L'introduction de contraintes sur les valeurs de ces propriétés permet d'être plus expressif. C'est dans cette catégorie que se situent la plupart des modèles étudiés jusqu'ici. Les contraintes pourront ensuite être étendues jusqu'à permettre l'utilisation arbitraire de toute assertion décrite en logique du premier ordre (cf. *contraintes générales* en section 3.3.2).

6.3.2 Les trois langages OWL

Pour répondre à un important spectre de besoins, OWL fournit trois sous-langages de plus en plus expressifs :

- *OWL Lite* supporte la création d'une hiérarchie avec des contraintes simples (il est par exemple impossible de définir pour les valeurs une cardinalité différente de 0, 1 ou plusieurs).
- *OWL DL*, appelé ainsi pour sa correspondance avec les logiques de description, est beaucoup plus expressif tout en conservant la décidabilité et complétude des raisonnements. Ainsi, quelques restrictions doivent être imposées (par exemple, une classe ne peut être instance d'une autre classe).
- *OWL Full* est le plus expressif des trois, mais ne garantit pas la calculabilité des raisonnements. Une classe pourra notamment être considérée comme une instance (comme le permet RDF *Schema*). En revanche, il est peu probable qu'un outil de raisonnement soit capable un jour de supporter toutes les caractéristiques de OWL *Full* [86].

Sur le spectre des définitions de la figure 6.1, ces trois sous-langages se situent progressivement parmi les plus expressifs, l'expressivité de OWL *Full* lui permettant de se confondre avec le point le plus extrême.

6.3.3 Conclusion et limites

Comme constaté, OWL est à la fois très expressif et très flexible. Il permet des opérations pour lesquelles les autres modèles sont souvent peu adaptés. Par exemple, OWL dispose de propriétés (`equivalentClass`, `equivalentProperty`, `sameAs`, etc.) facilitant la mise en correspondance ou la fusion de schémas, et plus globalement la diffusion des ontologies.

Pour autant, son approche clairement dédiée à la conception d'ontologies le limite quelque peu dans le monde documentaire. Les ressources manipulées n'ont par exemple pas d'identifiant unique : deux noms peuvent représenter la même entité (d'autres assertions permettent au moteur d'inférence de déterminer l'égalité des deux ressources). Dans les bases documentaires, la notion de clef est souvent utilisée pour différencier les entités. Celle-ci peut être simulée en utilisant la propriété `inverseFunctional` de OWL mais cette dernière ne s'applique qu'à une propriété et ne permet pas de définir des clefs sur plusieurs attributs. Qui plus est, en OWL DL, elle ne peut s'appliquer qu'aux propriétés dont la portée est de type objet. OWL *Full* est nécessaire pour l'utiliser avec des types de bases.

D'autre part, OWL souffre des mêmes problèmes que RDF (cf. section 4.4.4) en terme de support de la charge et est ainsi peu adapté à la modélisation d'importants volumes d'informations.

CHAPITRE 7

Récapitulatif et conclusion sur l'état de l'art

Après avoir étudié les principaux modèles d'information dans les chapitres 4, 5 et 6, nous allons maintenant résumer leurs avantages et inconvénients face aux besoins exprimés au chapitre 2.

Pour cela, nous allons tout d'abord faire un récapitulatif de ces modèles en relation avec les notions introduites au chapitre 3. Nous poursuivrons ensuite par une analyse comparative, avant de conclure sur les modèles existants.

7.1 Rappel des principaux modèles

Dans les chapitres précédents, nous avons passé en revue trois familles de modèles : le modèle entité-association (E/A) [25] et ses héritiers d'une part [55, 38, 99, 95, 39], les modèles à bases de rôles [7, 100, 47, 64, 28, 49, 2, 102, 89, 27] ensuite, et enfin les modèles à base de frames [73, 43, 14, 88, 58, 24, 77, 20, 18, 14, 68]. Cette classification est cependant en partie arbitraire. En effet, certains modèles sont difficiles à placer exclusivement dans une catégorie. UML (*Unified Modeling Language*) [84] par exemple est à la croisée des chemins avec les modèles à base d'objets dont les rôles sont une extension. OWL (*Web Ontology Language*) [86] est basé à la fois sur les logiques de description et RDF (*Resource Description Framework*) [62, 16], ce dernier faisant partie des modèles E/A.

Nous avons aussi pu observer que les notions introduites au chapitre 3 sont en grande partie partagées, parfois sous des dénominations différentes, par l'ensemble des modèles étudiés. La table 7.1 présente un récapitulatif des principales notions réparties arbitrairement parmi les besoins d'expressivité, de flexibilité, et de performances, exprimés au chapitre 2. Le cas des modèles à base de rôles, des modèles à bases de *frames* et des logiques des descriptions est particulier. Ces dénominations agrègent en effet de nombreuses propositions différentes qui ont parfois leurs spécificités. Dans ce récapitulatif, nous caractérisons ces regroupements dans leur ensemble même si certaines propositions divergent.

Quand un concept de modélisation n'est supporté qu'en partie, ou quand seulement une part (non négligeable) des modèles d'un groupe le supporte, cela est indiqué avec un carré blanc.

7.1.1 Précisions sur le récapitulatif

Apportons quelques précisions à la table 7.1 :

- Les *structures complexes* correspondent à des types structurés tels que les listes ou les familles, mais aussi à des types à base de graphes comme les arbres. Les modèles à base de rôles construits

Expressivité

	E/A	EER	UML	ORM	RDF	Rôles	Frames	DL	OWL	Besoin
Classes d'entités	√	√	√	√	√	√	√	√	√	√
Types complexes	×	√	√	√	√	√	√	√	√	√
dont Structures complexes	×	×	×	×	×	□	×	×	×	√
Recouvrement instances et classes	×	×	×	×	√	×	√	×	□	×
Spécialisation / généralisation	×	√	√	√	√	√	√	√	√	√
Héritage multiple	×	√	√	√	√	√	√	√	√	√
Catégorisation	×	√	×	×	×	×	√	√	√	×
Associations	√	√	√	√	√	√	√	√	√	√
dont n-aires	√	√	√	√	×	□	×	×	×	×
Contraintes	√	√	√	√	√	√	√	√	√	√
dont logique de 1^{er} ordre	×	×	×	×	×	×	□	√	√	×
Raisonnement / Inférence	×	×	×	×	×	×	□	√	√	×

Flexibilité

	E/A	EER	UML	ORM	RDF	Rôles	Frames	DL	OWL	Besoin
Multi instanciation	×	×	×	×	√	□	√	√	√	√
Acquisition / perte de type	×	×	×	×	√	□	□	□	√	√
Accès contextuel	×	×	×	×	×	√	×	×	×	√
Évolution du schéma	×	×	×	×	√	□	×	×	√	√
Fusion de schémas	×	×	×	×	□	×	×	×	√	×

Performances

	E/A	EER	UML	ORM	RDF	Rôles	Frames	DL	OWL	Besoin
Persistance	√	√	√	√	√	√	√	√	√	√
Passage à l'échelle	√	√	□	√	□	□	×	×	□	√

Table 7.1 – Récapitulatif des modèles d'information étudiés

sur des systèmes de gestion de bases de données à objets (SGBDO) (ex. : [27]), supportent une partie de ces types (*dictionary*, *set*, *bag*, *list*, etc.).

- La notion de *recouvrement entre instances et classes* est abordée en section 3.2.4. Un modèle la supportant permet à ses classes d'être elles-mêmes des instances d'autres classes. Dans le cadre d'OWL, seulement le langage OWL *Full* [86] offre cette possibilité.
- La *catégorisation*, introduite en section 3.2.3, est disponible statiquement dans le modèle E/A étendu (EER), mais dynamiquement dans les systèmes à base de *frames* et leur extensions que sont les logiques de description (DL) et OWL. Les catégories sont en effet représentées dans ces systèmes par des conditions nécessaires et suffisantes d'appartenance.
  On remarquera d'autre part que si la catégorisation n'est pas supportée par RDF et les modèles à base de rôles, ces derniers répondent cependant en partie au besoin ayant conduit à cette notion. Reprenons l'exemple des propriétaires de voitures de la section 3.2.3. Dans le premier cas, la multi-instanciation de RDF nous permet de créer une classe PROPRIÉTAIRE indépendante des classes PERSONNE et ENTREPRISE. Dans le second, nous pouvons utiliser un rôle dédié pour les représenter.
- Dans le cadre des modèles à base de rôles, la *multi-instanciation* n'est possible que pour les rôles. Comme dans les modèles à objets, il est impossible d'instancier plusieurs classes.
- Comme pour l'instanciation multiple, le *changement de type* n'est disponible dans les modèles à base de rôles que pour les rôles. Une fois créé, un objet ne peut changer de classe.
  Les systèmes à base de frames et les logiques de description, quant à eux, ne permettent pas de changement explicite de type. Cependant, l'utilisation de conditions d'appartenance pour des classes permet aux instances d'acquérir et de perdre dynamiquement des classes quand leurs données sont modifiées. Bénéficiant de son héritage RDF, OWL se distingue et permet le changement explicite de type (par ajout et suppression de propriétés).
- Les systèmes supportant l'*évolution du schéma* permettent aux schémas d'évoluer une fois instanciés. Dans le cas des rôles, certains modèles (ceux basés sur un SGBDO notamment [2, 27]) permettent de définir de nouveaux rôles et de les attribuer ensuite à des objets préexistants.
- Alors que OWL fournit des propriétés dédiées à la *fusion de schémas* (cf. section 6.3.3), il n'existe pas de notion explicite en RDF. Le regroupement de plusieurs fichiers « compatibles » est cependant possible.
- Il est possible de faire persister les données dans tous les modèles étudiés. Cependant, cette persistance n'est pas toujours centrale au modèle et ne correspond parfois qu'à un déchargement de la mémoire dans un fichier (cas de beaucoup de systèmes à base de *frames*, par ex. : Protégé-2000 [77]), ou encore à la sauvegarde des instructions LISP [87] ayant permis de construire la base en mémoire (cas de la logique de description KL-ONE [14]). Seuls les modèles E/A, EER et ORM (*Object-Role Modeling*) [51, 52] ont été conçus avec un objectif d'instanciation dans un système de gestion de bases de données relationnelles (SGBDR). Certains systèmes à base de rôles sont, eux, construits sur un SGBDO [2, 27].
- La gestion de la persistance vue ci-dessus a une influence directe sur la capacité des modèles à supporter les montées en charge. Le *passage à l'échelle* est ainsi possible dans les modèles construits sur un SGBDR, et pour ceux des modèles à base de rôles construits sur un SGBDO. UML étant surtout dédié à la modélisation d'application, il est rare que ses données soient sauvegardées dans une base de données, mais cela est cependant possible.
  RDF et OWL utilisent généralement le système de fichiers pour gérer la persistance, ce qui ne permet pas les montées en charge importantes. Cependant, l'utilisation de SGBD émergeants dédiés à RDF peut permettre d'obtenir de meilleures performances (ex. : Sesame [17]).

La colonne *Besoin* dans la table 7.1 précise les besoins exprimés en section 2.6.

- Les *structures complexes*, peu présentes dans les modèles traditionnels, sont particulièrement utiles à la modélisation d'applications d'intelligence économique. S'il est possible de les simuler avec des types plus simples, les arbres, et plus généralement les graphes, sont par exemple très utiles à la modélisation de hiérarchies.
- La possibilité pour une instance d'être aussi une classe pour d'autres instances améliore l'expressivité d'un modèle, mais a un coût important en complexité induite. N'ayant pas identifié ce besoin particulier, le support du *recouvrement entre instances et classes* n'est pas requis.
- La *catégorisation* en tant que telle n'est pas nécessaire. En revanche, un *accès contextuel* aux entités doit être disponible.
- Les *associations n-aires* pouvant être simulées par des associations binaires, elles ne nous paraissent pas nécessaires. Pour plus de simplicité, nous nous permettons donc de n'utiliser que des associations binaires.
- Notre but est de modéliser des applications traitant d'importants volumes de données. Si l'efficacité du système est primordiale, l'*inférence* de nouvelles données n'est pas centrale. Les capacités de *raisonnement* ne doivent donc pas contraindre la conception de notre modèle. Pour la même raison, il ne nous paraît pas nécessaire de s'imposer le support de contraintes complexes comme celles exprimées en *logique du 1^{er} ordre*.

7.2 Analyse

7.2.1 Les principaux modèles E/A

Du fait de son âge, le modèle E/A manque de nombreux concepts de modélisation apparus ensuite dans ses extensions. Parmi celles-ci nous pouvons remarquer que EER, UML et ORM ont des caractéristiques très proches. Bien que dédiées à des utilisations différentes, leur *expressivité* est similaire et n'est pas très éloignée des besoins exprimés.

Un rapide regard aux concepts composant la partie *flexibilité* nous montre en revanche leurs limites. Ils se concentrent sur les aspects structurels des applications et ont ainsi tendance à être très statiques. Une fois que le schéma a été créé, il est très difficile de le faire évoluer, les répercussions sur les instances étant trop complexes. Dans ces modèles, le couplage des instances avec leurs *uniques* classes est aussi très fort. Une fois créées, les instances ne peuvent guère évoluer que par la mise à jour des valeurs de leurs propriétés.

En contrepartie de leur peu de flexibilité, les performances de ces modèles de la famille E/A profitent de leur instanciation statique. L'implémentation relationnelle leur est particulièrement profitable dans ce domaine.

7.2.2 RDF, un modèle à part

Le modèle RDF fait figure d'étranger dans la famille E/A. S'il en fait bien partie [93], son héritage XML et « Internet » lui offre une flexibilité dont les autres membres de la famille E/A ne disposent pas. Il est simple par exemple d'ajouter des éléments dans un schéma.

En revanche, son manque de contraintes le rend peu utilisable pour modéliser des domaines complexes. L'intégrité des données doit en effet être contrôlée par l'application.

Par ailleurs, contrairement aux autres modèles E/A, l'implémentation traditionnelle de RDF se fait sous forme de fichiers. Ce choix offre là encore de la flexibilité. La base de connaissances est ainsi facile

à échanger et de simples ajouts/suppressions de fichiers suffisent parfois à faire des modifications par ailleurs complexes dans d'autres modèles.

Cependant, le choix du système de fichier empêche RDF de supporter d'importantes montées en charge. Pour qu'une requête soit efficace, il faut en effet examiner tous les fichiers composant la base. Le mieux étant encore de les conserver dans une structure mémoire pour un accès rapide. L'arrivée récente de bases de données dédiées à RDF (ex. : Sesame [17]) permet de répondre en partie à cette limitation.

7.2.3 Les rôles, un concept prometteur

Les modèles à base de rôles offrent eux aussi une expressivité intéressante. Certains d'entre eux (ex. : [27]) fournissent même des structures complexes (listes, applications (*map*), etc.) facilitant la modélisation d'applications en intelligence économique.

Leur flexibilité, même si elle n'est pas au niveau de certains autres modèles (OWL en particulier), bénéficie de la notion de *rôle*, centrale à l'approche. Ainsi, ce sont les seuls modèles étudiés qui offrent un accès contextuel aux entités. Seules les propriétés du rôle accédé sont disponibles à l'application. Les propriétés non pertinentes dans le contexte ne sont pas exposées.

Au cours de sa « vie », une entité peut acquérir de nouveaux rôles et en perdre d'anciens. Cela lui permet de changer de type, ou du moins de proposer de nouveaux contextes pour y accéder, mais seulement dans une certaine mesure. En effet, la classe avec laquelle l'instance a été créée est, quant à elle, immuable. De plus, les rôles disponibles pour une entité sont restreints à ceux permis par sa classe.

Parmi les modèles à base de rôles, ceux qui sont basés sur des SGBDO [2, 27] nous intéressent tout particulièrement. Le système de gestion de bases de données sous-jacent permet en effet le traitement d'importants volumes de données persistantes. Ces modèles souffrent néanmoins de la relative inadéquation des SGBDO avec les besoins des entreprises, et de façon plus pragmatique de leur faible pénétration du marché.

7.2.4 *Frames* et logiques de description

Provenant de la communauté intelligence artificielle, les systèmes à base de *frames* et les logiques de description s'intéressent à des aspects plus dynamiques de la modélisation, et particulièrement à la possibilité d'inférer des faits nouveaux d'après des données existantes. Plutôt que décrire les classes avec des définitions statiques, ils privilégient généralement l'utilisation de conditions d'appartenances nécessaires et/ou suffisantes (cf. section 6.1.3). La classification permet, en outre, dans les logiques de description d'inférer automatiquement une hiérarchie de classe.

Très expressifs, ils permettent de décrire des contraintes complexes (ex. : logique du 1^{er} ordre) indisponibles dans les précédents modèles abordés. La flexibilité est également importante dans ces modèles. Il est ainsi possible de définir une nouvelle classe grâce à des conditions d'appartenance, pour qu'elle soit automatiquement peuplée par les instances y répondant, ces dernières pouvant naturellement instancier plusieurs classes.

Cependant, ces capacités ont un impact négatif sur les performances. Si les modèles à base de *frames* et les logiques de description sont bien adaptés à la modélisation de petits domaines complexes, et au raisonnement sur ces modèles, ils montrent leurs limites dès que les volumes d'information sont importants.

7.2.5 OWL, le meilleur de deux mondes

OWL, en capitalisant à la fois sur RDF et sur les logiques de description (*via* DAML-ONT [90], OIL [42] et ensuite DAML+OIL [70]), hérite du meilleur de ces approches. Comme le montre la table 7.1, il s'agit du modèle supportant le plus grand nombre de concepts d'expressivité et de flexibilité introduits. Son adoption grandissante pour modéliser des ontologies et les partager semble donc assurée.

Cependant, s'il hérite du meilleur, il ne peut éviter toutes les limites de ses ancêtres. OWL n'est ainsi pas adapté dès lors que les volumes de données à traiter peuvent devenir importants.

7.3 Conclusion

Même si aucun des modèles étudiés ne correspond exactement aux besoins exprimés dans le chapitre 2, nous pouvons tirer les enseignements suivants de leur examen.

Tout d'abord, les modèles qui ont été créés comme des outils de conception de schémas en vue de leur implémentation dans un SGBD (E/A, EER, ORM), sont logiquement ceux qui supportent le mieux les montées en charge. En n'accordant pas la priorité aux problématiques de flexibilité et d'évolution, la traduction dans le modèle relationnel reste simple et compréhensible. La plus grande flexibilité des autres catégories de modèles est acquise au détriment des performances. Les volumes de données à traiter étant particulièrement importants (cf. chapitre 2), la traduction dans un modèle relationnel est à envisager.

Le concept de type est toujours central, mais la notion de classe utilisée pour le représenter par la plupart des modèles est trop contraignante. Les rôles, plus flexibles et permettant de contextualiser l'accès aux entités, sont une voix d'étude prometteuse et en adéquation avec le modèle utilisé jusqu'alors par la solution Arisem [5]. Des fondements ODMG de certains modèles à base de rôles, nous pouvons aussi retenir l'utilisation de structures complexes.

Les possibilités de raisonnement étant absentes de nos priorités, l'approche des logiques de description n'est pas adaptée. De plus, la définition d'une classe à l'aide de conditions d'appartenance semble difficilement compatible avec une approche base de données et grand volume d'information. Le concept de *dæmon* utilisé par les systèmes à base de *frames* peut en revanche être intéressant à exploiter (cf. section 6.1.1).

RDF, et plus encore OWL, offrent de nombreux avantages dont en particulier leur importante adoption en qualité de standards de la toile. S'ils ne sont pas appropriés pour modéliser d'importants volumes de données, ils peuvent être très utiles si nous les cantonnons dans l'un de leur rôles principaux : permettre l'interopérabilité de systèmes hétérogènes. Au même titre que d'autres modèles de même catégorie qui n'ont pas été abordés dans cette étude (ex. : XML Topic Maps [80], MPEG-7 [30]), RDF et OWL peuvent en effet servir de format d'importation / exportation de données.

PARTIE III

Proposition

La première partie nous à permis d'exposer les besoins en modélisation liés à la problématique, c'est-à-dire expressivité, flexibilité *et* performances.

Dans la seconde partie, nous avons étudié les principaux modèles disponibles et en avons souligné les limites.

Cette troisième partie nous permet de décrire notre proposition et de montrer en quoi elle répond mieux aux besoins exprimés. Le chapitre 8 détaille le modèle DOAN et précise comment il répond aux besoins d'expressivité et de flexibilité. Dans le chapitre 9, nous présentons son implémentation relationnelle, et notamment l'algorithme de traduction des types de DOAN dans le modèle relationnel. Enfin, le chapitre 10 nous permet de montrer que cette implémentation répond au besoin de performances.

CHAPITRE 8

DOAN : Un modèle flexible

*Dans ce chapitre, nous introduisons les concepts du modèle DOAN (*DOcument ANnotation Model*). Le principal but poursuivi dans leur élaboration est de répondre aux besoins d'*expressivité *et de* flexibilité *exprimés dans le chapitre 2.*

Comme nous l'avons vu dans le chapitre 2, chaque processus de l'application génère des annotations pour les composants documentaires dont il est responsable. Les annotations ne sont pas de simples commentaires, mais généralement des ensembles d'informations structurées, qui correspondent au résultat du processus. Les processus peuvent travailler indépendamment ou, au contraire, coopérer pour enrichir les annotations. Ils travaillent sur diverses parties des documents originaux ou de leurs annotations existantes.

Nous avons donc besoin de trouver un juste équilibre entre le gain obtenu par la création d'informations structurées et la flexibilité inhérente nécessaire aux processus pour travailler ensemble. Les facettes (section 8.1) et contraintes (section 8.2) répondent à ce dernier besoin, alors qu'un système de types strict et riche (section 8.3) répond au premier. Après avoir introduit ces différents aspects du modèle, nous terminerons ce chapitre en les mettant en relation avec les besoins exprimés.

8.1 Concepts fondateurs

8.1.1 Une classe d'entités unique

Introduisons tout d'abord l'ensemble E des entités du système. Cet ensemble est virtuellement infini et ne requiert aucun attribut prédéfini.

$$E = \{e_1, e_2, \ldots\} \tag{8.1}$$

Les entités composant E peuvent être associées à tout objet du monde réel, par exemple des documents ou des employés. N'ayant pas d'attributs, elles peuvent être comparées à de simples identifiants, qui ne portent intrinsèquement aucune information sur les objets qu'ils représentent.

E est l'*unique* classe d'entités du système et toutes les entités lui appartiennent.

8.1.2 Facettes

Les entités du système ne portant aucune donnée en elles-mêmes, nous devons leur associer des informations sur ce qu'elles représentent. C'est le rôle des *facettes*.

L'ensemble F des facettes du système est défini par :

$$F = N_F \rightarrow T \tag{8.2}$$

où N_F est l'ensemble des noms de facettes et T est un type générique que nous décrirons ensuite plus précisément.

Chaque *extension de facette* doit être interprétée comme une *fonction partielle* de l'ensemble des entités vers un ensemble de valeurs $\mathcal{T}$ d'un type donné de T[1] :

$$\forall f \in N_F, f : E \not\rightarrow \mathcal{T} \tag{8.3}$$

Les fonctions de facettes doivent être partielles car toutes les entités du système ne sont pas censées fournir tous les points de vue. Il n'y a aucun sens, par exemple, à ce qu'un document soit observé *via* une facette EMPLOYÉ.

La notion de facette est ainsi la réponse du modèle à ce que nous avons appelé « point de vue » dans le chapitre 2. En offrant un accès contextuel aux entités, elle est comparable à celle de « rôle » dans les systèmes à base de rôles introduits au chapitre 5. Elle est cependant plus flexible car son utilisation n'est pas contrainte par une classe.

Ruben Prieto-Diaz a introduit une notion similaire de facette dans le but de faire des classifications de composants logiciels [81]. Dans son approche, les facettes peuvent être considérées comme les dimensions d'un espace de classification cartésien, la valeur d'une facette étant la position du composant dans cette dimension.

8.1.3 Schéma

Un *schéma* S est défini comme un couple formé par un ensemble d'entités et un ensemble de facettes.

$$S = \langle E, F \rangle \tag{8.4}$$

avec au minimum une facette :

$$F \neq \emptyset \tag{8.5}$$

8.1.4 Extension de schéma

Une *extension de schéma* $\mathcal{S}$ d'un schéma S est un sous-ensemble arbitraire des entités et extensions de facettes. Formellement, une extension de schéma est définie comme :

$$\mathcal{S} = \langle \mathcal{E} \subset E, \mathcal{F} \subset 2^{N_F \times \mathcal{E} \not\rightarrow \mathcal{T}} \rangle \tag{8.6}$$

où $2^{N_F \times \mathcal{E} \not\rightarrow \mathcal{T}}$ représente l'ensemble des parties de l'ensemble des extensions de facettes et tel que :

$$\forall e \in \mathcal{E}, \exists (f, t) \in N_F \times \mathcal{T} : (f, e, t) \in \mathcal{F} \tag{8.7}$$

i.e., pour chaque entité, au moins une facette est définie dessus (dans le cas contraire, une entité ne serait qu'un simple identifiant inutile).

Afin de présenter un exemple dans le contexte d'une application, reprenons les besoins exprimés dans le chapitre 2. Les applications d'intelligence économique envisagées manipulent plusieurs sortes d'entités (e.g., personnes, documents, etc.) *via* différents points de vue (e.g., auteurs, managers, études de marché, etc.).

Reprenons dans la figure 8.1 l'exemple que nous avions introduit dans la problématique (figure 2.5).

[1] Nous utilisons des lettres capitales pour le modèle et des lettres calligraphiques pour les valeurs correspondantes.

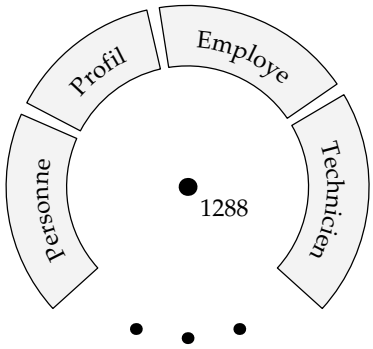

Figure 8.1 – L'entité Jacques Durand (ID 1288) supportant plusieurs facettes

Comme nous l'avons vu précédemment, les points de vue introduits dans la problématique sont représentés par des facettes dans le modèle. Dans ce cadre, nous pouvons ainsi définir $\mathcal{S}$ comme constitué de $\mathcal{E}$ et $\mathcal{F}$, tels que $\mathcal{E} = \{\ldots, ID1287, ID1288, ID1289, \ldots\}$, et $\mathcal{F}$ contienne les extensions des facettes décrivant les points de vue sur les personnes, les documents, etc. :

$$\begin{array}{l}\{\ldots,\\ \quad (Personne, \text{ID1288}, (\text{“Jacques”}, \text{“Durand”})),\\ \quad (Technicien, \text{ID1288}, -),\\ \quad \ldots\}\end{array} \tag{8.8}$$

8.2 Évolutions et Contraintes

Le modèle met l'accent sur la notion de *produit cartésien*, c'est-à-dire que toute entité du « monde réel » peut être associée à, ou définie par, un nombre *illimité* de facettes. Ainsi, pendant sa durée de vie, une entité peut évoluer très facilement en y associant ou en y supprimant des valeurs de facettes. Ces opérations se traduisent par le changement des extensions des facettes respectives dans $\mathcal{F}$ et éventuellement par l'ajout ou la suppression d'une entité dans $\mathcal{E}$.

Lorsque nous associons une valeur de facette à une entité, nous disons que nous *attachons* la facette à l'entité. Inversement, lorsque nous supprimons une valeur de facette d'une entité, nous disons que nous *détachons* la facette de l'entité. Nous parlerons aussi d'opérations respectivement de *plug* et d'*unplug* de facettes.

Ces opérations sont à la source de la flexibilité du système. Reprenons à nouveau l'exemple de la problématique dans la figure 8.2. En attachant ou détachant des facettes successivement, une entité peut évoluer aisément au cours de sa durée de vie : Jacques Durand, après avoir été technicien quelques années évolue naturellement vers un poste à responsabilités. Toutefois, un minimum de contrôle est nécessaire puisqu'il existe dans le monde réel des contraintes qui peuvent limiter la flexibilité.

8.2.1 Facettes principales

Tout d'abord, toutes les facettes ne peuvent être utilisées comme la première facette d'une entité. Nous pouvons, par exemple, imposer qu'il n'y a aucun sens à créer juste une PERSONNE, cette entité doit au moins représenter autre chose, comme un AUTEUR ou un EMPLOYÉ. Dans ces circonstances, la facette PERSONNE ne peut pas servir à la création d'une nouvelle instance *ex nihilo*. Comme nous

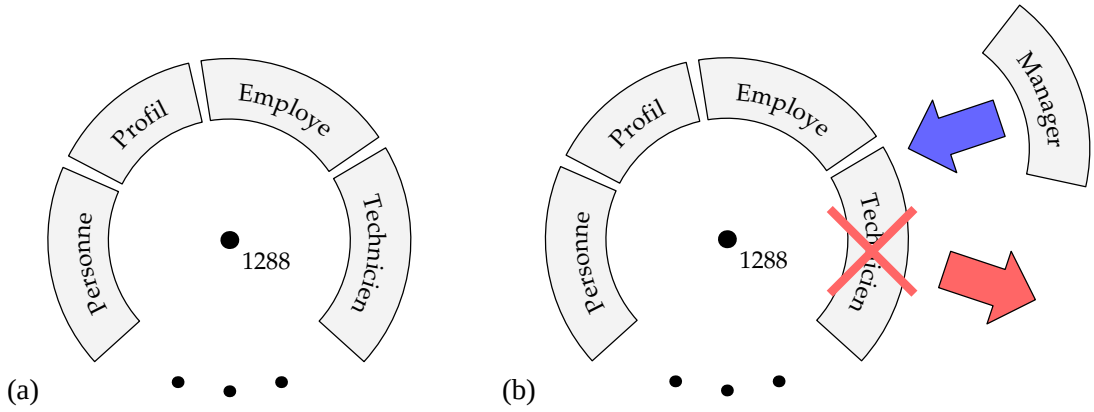

Figure 8.2 – L'entité Jacques Durand (ID 1288) avant (a) et après (b) sa promotion

le verrons dans la prochaine section, elle doit être impliquée par AUTEUR ou EMPLOYÉ, qui sont des facettes autorisées pour la création de nouvelles entités.

Nous définissons donc un sous-ensemble non vide de noms de *facettes principales* (ou *main facets*) M_F :

$$\emptyset \subset M_F \subseteq N_F \tag{8.9}$$

En remplacement de l'équation (8.7), l'existence d'une entité est maintenant subordonnée au support d'au moins une facette principale :

$$\forall e \in \mathcal{E}, \exists (f,t) \in M_F \times \mathcal{T} : (f,e,t) \in \mathcal{F} \tag{8.10}$$

En conséquence, si la dernière facette principale supportée par une entité est détachée, alors cette entité et les facettes restantes sont automatiquement supprimées (voir la section A.2 de l'annexe sur le cycle de vie pour un exemple d'implémentation).

La facette principale agit comme une classe dans des paradigmes plus classiques, au moins lors de la création d'instances. Cependant, ce couplage entre une instance et sa « classe » est faible. Au contraire des modèles à objets, il est en effet possible à une entité de supporter plusieurs facettes principales en même temps. Par exemple, une personne peut être créée comme un EMPLOYÉ ; ensuite, elle peut devenir simultanément un EMPLOYÉ et un AUTEUR de documentation ; finalement, elle restera un AUTEUR en quittant la société. La mutabilité du modèle est intrinsèquement importante. En comparaison, les rôles [27] sont restreints à ceux définis sur la classe de leur instance.

8.2.2 Contraintes sur les facettes

En plus de la notion de facette principale, d'autres contraintes nous permettent de contrôler la flexibilité. La catégorie la plus simple de ces contraintes concerne les valeurs des facettes (cf. section 3.3.1). Il s'agit simplement de limiter les valeurs d'une facette en en contraignant le domaine grâce, par exemple, à un intervalle ou une énumération de valeurs autorisées.

Par contraste, les contraintes sur les facettes sont plus complexes et nous permettent d'imposer des règles sur les mécanismes d'attachement et de détachement de facettes. Il en existe trois types : exclusion, implication et implication légère (soft implication).

Exclusion. L'*exclusion* est une contrainte binaire et bi-directionnelle. Elle interdit à toute instance de supporter simultanément les deux facettes spécifiées. Formellement, nous avons la définition suivante :

$$f_1 \text{ rejects } f_2, f_2 \text{ rejects } f_1 \Rightarrow \mathbf{dom}(f_1) \cap \mathbf{dom}(f_2) = \emptyset \tag{8.11}$$

où nous utilisons une notation simplifiée pour décrire le domaine d'une extension de la facette f dans l'extension de schéma $\mathcal{S}$ (l'indice est supprimé quand il n'y a pas d'ambiguïté) :

$$\mathbf{dom}_{\langle \mathcal{E}, \mathcal{F} \rangle}(f) = \{e \in \mathcal{E} | \exists t \in \mathcal{T} : (f, e, t) \in \mathcal{F}\} \tag{8.12}$$

Si nous tentons d'attacher une facette à une entité supportant déjà une autre facette en exclusion avec la nouvelle, l'opération est interrompue et une erreur est retournée.

Implication. La contrainte d'*implication* nous aide à nous assurer que toutes les facettes qui « travaillent » ensemble sont présentes. Par exemple, attacher la facette EMPLOYÉ *implique* que la facette PERSONNE est déjà présente sur l'instance. Ainsi, si PERSONNE n'est pas encore supportée par l'entité, elle est attachée automatiquement avant EMPLOYÉ. Pour la même raison, il est impossible de détacher PERSONNE si EMPLOYÉ est présente. Formellement, nous notons :

$$f_1 \text{ implies } f_2 \Rightarrow \mathbf{dom}(f_1) \subseteq \mathbf{dom}(f_2) \tag{8.13}$$

Ainsi, les implications sont transitives :

$$f_1 \text{ implies } f_2 \wedge f_2 \text{ implies } f_3 \Rightarrow f_1 \text{ implies } f_3 \tag{8.14}$$

Les implications sont maintenues sous la forme d'un graphe acyclique orienté. L'opération d'attachement des facettes impliquées est réalisée récursivement avant l'attachement de la nouvelle.

Implication légère. L'*implication légère*, que nous appelons aussi *soft implication* ou *softplug*, est similaire à l'implication lors de l'opération d'attachement mais ne déclenche aucune action durant le détachement. Par exemple, un EMPLOYÉ est généralement vu comme un UTILISATEUR du système, mais l'application peut fonctionner avec la facette EMPLOYÉ seule. En conséquence, la facette UTILISATEUR peut être détachée plus tard.

Il n'y a aucune restriction sur la structure du graphe d'implications légères. En particulier, il est possible de combiner une liste d'implications et une seule implication légère pour obtenir un comportement similaire à des implications croisées.

Ces trois contraintes sont exclusives par paires :

$$\begin{aligned} f_1 \text{ implies } f_2 &\Leftrightarrow \neg f_1 \text{ rejects } f_2 \\ f_1 \text{ implies } f_2 &\Leftrightarrow \neg f_1 \text{ softplugs } f_2 \\ f_1 \text{ rejects } f_2 &\Leftrightarrow \neg f_1 \text{ softplugs } f_2 \end{aligned} \tag{8.15}$$

L'évolution des contraintes est elle aussi contrôlée. La suppression d'une contrainte n'a aucune conséquence sur l'extension de schéma ; elle est donc toujours acceptée. En revanche, l'ajout d'une exclusion déclenche le contrôle des entités existantes. Une erreur est renvoyée si les données présentes et le modèle existant n'obéissent pas à la nouvelle contrainte. Pour l'implication, soit toutes les facettes requises sont attachées avec des valeurs par défaut, soit une erreur est retournée si la nouvelle implication

n'est pas admissible (au niveau de la structure du graphe acyclique orienté, ou à cause de contre-exemples dans les données).

D'autre part, l'opération de détachement (*unplug*) est sujette à un choix de conception : les facettes impliquées doivent-elles être détachées automatiquement avec les facettes qui les impliquent ? Nous avons décidé de donner le choix au concepteur entre une approche conservatrice où les facettes impliquées sont laissées telles quelles, et une approche plus agressive où les facettes qui ont été attachées automatiquement suite à une contrainte d'implication sont détachées elles aussi (voir l'option `CascadingUnplug`, dont l'influence est détaillée en section A.2 de l'annexe sur le cycle de vie).

8.3 Système de types riche pour les facettes

Nous avons décrit jusqu'ici un modèle capable de gérer des informations évolutives générées par des processus variés. Ces informations correspondent à autant de points de vue sur les documents et les objets métier du système. L'objet de cette section est de fournir un emplacement pour les données concrètes, c'est-à-dire les valeurs de facettes, $\mathcal{T}$. Comme nous allons le voir, le système de types que nous utilisons pour décrire les facettes, T, est très proche de la proposition de l'*Object Data Management Group* (ODMG) [22]. Une approche de typage stricte facilite une utilisation automatique des données collectées et analysées, alors qu'un ensemble riche de types fournit l'expressivité nécessaire à leur description.

8.3.1 Différents types

Nous pouvons classer les types de données que nous utilisons en quatre catégories en fonction de leur arité : types atomiques, unaires, binaires et n-aires.

Types atomiques. Les types atomiques sont les types de base du modèle : booléen (**boolean**), entier (**integer**), réel (**real**), caractère (**char**) et chaîne de caractères (**string**). À cette liste de base, nous ajoutons le type **date**. Il peut être facilement simulé en utilisant une composition de types de base, mais nous préférons en faire un type atomique à part entière pour faciliter son utilisation. C'est en effet un type fréquemment utilisé dans les applications de traitement documentaire.

Tous les modèles ont généralement une liste de types de base permettant de construire des types de plus haut niveau. Souvent, ces types sont proches de l'implémentation. Par exemple, le langage C [56] décompose les entiers en **short**, **int** et **long**, et les réels en **float**, **double** et **long double**, en fonction des valeurs maximales désirées. Tandis qu'en SQL [29], l'utilisation de **numeric** permet d'indiquer la précision désirée.

Dans le cadre de notre modèle, nous préférons nous abstraire de l'implémentation effective et utiliser des types plus génériques. Lors de la traduction dans un modèle particulier (cf. chapitre 9), nous pourrons ainsi exploiter au mieux les types disponibles. Les contraintes ayant été exprimées par le concepteur (par exemple un intervalle de valeurs possibles pour des entiers) nous permettront, de plus, d'affiner notre choix.

Types unaires. Les types unaires représentent tous des collections d'éléments ayant un type donné. Aux côtés des types « classiques » que sont ensemble (**set**), liste (**list**) et famille (**bag**), nous introduisons des types basés sur des graphes : arbre (**tree**), graphe (**graph**) et en particulier graphe acyclique orienté (**dag**).

Ces derniers sont en effet très utiles aux applications d'intelligence économique qui nous intéressent pour représenter notamment des hiérarchies, des organigrammes ou encore des ontologies. Ils pourraient être simulés (ex. : **tree**$(T) \equiv$ **map**(T, T)), mais le fait de les représenter à part entière va nous permettre de répondre à plusieurs questions d'implémentation importantes pour obtenir de bonnes performances (cf. chapitre 9).

Ces collections nous permettent aussi de représenter les attributs multivalués évoqués en section 3.1.1. La distinction entre les types ensemble, liste et famille nous permet de plus de préciser si les valeurs doivent être distinctes ou ordonnées. Des contraintes de cardinalité peuvent ensuite préciser les nombres minimum et maximum de valeurs.

Types binaires. Nous introduisons deux types binaires qui imitent le modèle relationnel et prennent en compte les dépendances fonctionnelles : l'application (**map**) et la bijection (**bijection**).

Types n-aires. Les types n-aires du modèle sont aussi au nombre de deux. Le n-uplet (**tuple**) nous permet de décrire des compositions de types hétérogènes et donc de représenter les attributs composites introduits en section 3.1.1. La relation (**relation**), qui imite elle aussi le modèle relationnel, n'est qu'un raccourci pour un ensemble de n-uplets (**relation**$(T_1, T_2, \ldots, T_n) =$ **set**(**tuple**$(T_1, T_2, \ldots, T_n)$)).

L'ensemble de ces types permet au concepteur de décrire de façon précise les facettes composant son modèle. Ainsi, avec ce système et d'après la définition (8.2), l'exemple (8.8) est défini par :

$$\textbf{facet}\ Personne\ \textbf{is tuple}(prénom : \textbf{string}, nom : \textbf{string})$$

Pour plus de simplicité, nous pouvons considérer la notion de facette comme un type à part entière et utiliser une syntaxe proche de l'ODMG pour écrire alors :

$$Personne : \textbf{facet}(\textbf{tuple}(prénom : \textbf{string}, nom : \textbf{string}))$$

La composition et l'imbrication de types permet de décrire des facettes plus complexes, comme l'illustre la facette suivante :

$$\begin{aligned} Document : \textbf{facet}(\textbf{tuple}(&titre : \textbf{string},\\ &date : \textbf{date},\\ &auteurs : \textbf{list}(auteur : \textbf{tuple}(prénom : \textbf{string},\\ &\qquad nom : \textbf{string})),\\ &contenu : \textbf{list}(\textbf{string}),\\ &annotations : \textbf{set}(annotation : \textbf{tuple}(auteur : \textbf{tuple}(prénom : \textbf{string},\\ &\qquad nom : \textbf{string}),\\ &\qquad contenu : \textbf{list}(\textbf{string}))))) \end{aligned}$$

Nous donnons ici la description de données récoltées sur des documents et leurs auteurs. Plusieurs types complexes sont utilisés dans cet exemple qui montre aussi comment ils peuvent être arbitrairement imbriqués : une facette contenant un n-uplet, dont un attribut est un ensemble de n-uplets, dont un attribut est une liste de chaînes de caractères.

8.3.2 Références

Comme nous l'avons introduit dans le chapitre 2, l'information que nous devons traiter est fortement « connectée ». Cela se traduit par la présence de liens entre les entités *via* leurs facettes : un DOCUMENT est, par exemple, écrit par un ou plusieurs AUTEUR(s), un EMPLOYÉ travaille dans une ENTREPRISE, etc.

Parmi les différentes possibilités de représenter les associations que nous avons étudiées en section 3.1.2, nous avons porté notre choix sur la plus simple : des références unidirectionnelles entre facettes uniquement. Nous introduisons donc le nouveau type **ref**, limité aux facettes. Le type **ref**$(Personne)$ par exemple, correspond au type des références à des entités supportant la facette PERSONNE. La restriction des références aux facettes nous permet, de plus, d'utiliser le nom de la facette f comme raccourci pour le type **ref**(f). On a ainsi pour l'exemple précédent : **ref**$(Personne) \equiv Personne$.

Ce choix nous permet d'utiliser les références comme un type unaire directement disponible pour définir les facettes (sur le même principe que UML, cf. section 4.2). Par contre, si le concepteur veut modéliser une association ayant une arité supérieure à deux, il est nécessaire de la réifier en une facette dont le type sera un n-uplet de références.

Cette approche par réification nous permet de simuler les associations simplement. Elle offre en outre l'avantage d'autoriser la création de facettes sur des instances d'associations et même sur des associations entre associations. Par exemple, une instance de l'association MARIAGE (entre deux PERSONNES, un HOMME et une FEMME) est CÉLÉBRÉ dans une église. L'association CÉLÉBRATION lie donc l'association MARIAGE à la facette ÉGLISE. Qui plus est, d'autres facettes peuvent aussi être attachées : DATE, TÉMOINS, MUSIQUES JOUÉES, etc.

Reprenons l'exemple de la facette DOCUMENT que nous venons d'introduire au-dessus. Plutôt que d'utiliser de simples chaînes de caractères pour en décrire les auteurs, les concepteurs feront en général le choix de les modéliser. Imaginons que nous disposions d'une facette PERSONNE et d'une facette plus précise AUTEUR l'impliquant. Si les auteurs de documents sont forcément identifiés comme des instances de AUTEUR, mais que toute personne peut écrire une annotation, la facette DOCUMENT devient alors :

$$
\begin{aligned}
Document : \textbf{facet}(\textbf{tuple}(&titre : \textbf{string},\\
&date : \textbf{date},\\
&auteurs : \textbf{list}(auteur : \text{AUTEUR}),\\
&contenu : \textbf{list}(\textbf{string}),\\
&annotations : \textbf{set}(annotation : \textbf{tuple}(auteur : \text{PERSONNE}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad contenu : \textbf{list}(\textbf{string})))))
\end{aligned}
$$

8.3.3 Comparaison ODMG

Les types que nous venons d'introduire sont très proches du modèle ODMG [22]. Nous pouvons en observer une comparaison dans la table 8.1.

L'introduction dans les sections précédentes du type de données **facet** (avec ses opérations associées – création, attachement, détachement – et ses contraintes) est la principale différence avec le modèle ODMG.

Une autre différence importante est que nous fournissons des types de données abstraits et non des classes avec des méthodes définies par l'utilisateur. Tous les types de DOAN viennent avec les opérations standard permettant de manipuler leurs valeurs (création, destruction, insertion, suppression et modification). Des points d'insertion sont aussi fournis pour permettre le déclenchement d'une ou plusieurs

ODMG	commun	DOAN
array, bitstring	integer, boolean, string… tuple/struct list, set, bag, map/dictionary ref	relation, bijection tree, dag, graph facet

Table 8.1 – Types ODMG et DOAN

actions utilisateur quand une opération standard est exécutée, sur le même principe que certains modèles à base de frames [14, 88] (cf. annexe B).

Une troisième différence est relative à l'ensemble des types standards utilisés. Tout d'abord, nous introduisons des types de données qui imitent le modèle relationnel, c'est-à-dire la relation, la bijection et l'application (cette dernière est aussi disponible en ODMG *via* le type **dictionary**). Ensuite, les types à base de graphe sont utiles pour les applications d'intelligence économique envisagées.

Les types spécifiques à ODMG peuvent être simulés : **array**$(T) \equiv$ **map**(**integer**, T) et **bitstring**$(T) \equiv$ **set**(T). La différence se situe au niveau de l'implémentation et non de la spécification, particulièrement pour **bitstring**.

Enfin, les références (type **ref** dans ODMG) sont mono-directionnelles dans notre modèle alors qu'elles sont bi-directionnelles dans le modèle ODMG.

8.4 Méta-modèle

Le système de types que nous venons d'introduire nous permet de décrire les facettes composant un schéma. Notre but est maintenant de modéliser ces descriptions dans un méta-modèle.

Pour ce faire, nous devons modéliser les types disponibles. La table 8.2 en donne la liste. Pour éviter toute confusion avec le modèle, nous les préfixons par le mot « Meta ».

Le méta-type **MetaType** représente l'union de tous les types du système (à l'exclusion de **MetaFacet**). Il correspond au type générique T que nous avons introduit dans l'équation 8.2.

Le méta-type **MetaFacet** est différent des autres. Il ne fait pas partie du méta-type **MetaType**. Une facette ne peut être utilisée comme type d'une autre facette : seules les références permettent de lier des facettes entre elles. En réalité, nous introduisons **MetaFacet** pour tenir le même rôle que l'ensemble

MetaType **is union**(BaseType, Metatuple, MetaMap, MetaUnion, MetaSet, MetaRef…)
BaseType **is set**(string)
Metatuple, MetaRelation **is map**(attribute_name: **string**, attribute_type: MetaType)
MetaMap, MetaBijection **is tuple**(domain: MetaType, range: MetaType)
MetaSet, MetaBag **is tuple**(type: MetaType)
MetaList, MetaTree, MetaDAG, MetaGraph **is tuple**(type: MetaType)
MetaRef **is tuple**(facet: MetaFacet)
MetaFacet **is tuple**(name: **string**)

Table 8.2 – Types du méta-modèle

Facets: **map**((facet: MetaFacet, type: MetaType))
MainFacets: **set**(facet: MetaFacet)
[MainFacets $\subseteq$ Facets]
Facets_Rejections: **set**(**tuple**(facet1: MetaFacet, facet2: MetaFacet))
[symetrical relationship]
Facets_Implications: **dag**(MetaFacet)
Facets_SoftPlugs: **graph**(facet: MetaFacet, softPluggedFacet: MetaFacet)
[Facets_Implications $\cap$ Facets_Rejections = $\emptyset$,
Facets_Implications $\cap$ Facets_SoftPlugs = $\emptyset$,
Facets_Rejections $\cap$ Facets_SoftPlugs = $\emptyset$]

Table 8.3 – Instances du méta-modèle

N_F des noms de facettes dans l'équation 8.2. La correspondance des noms de facette avec leur type est réalisée directement dans les instances du méta-modèle, qui sont décrites dans la table 8.3.

Le méta-modèle reste très simple. Nous n'avons en effet besoin de méta-modéliser que les facettes (*Facets*) et leurs contraintes. Les facettes principales (*MainFacets*) ne sont qu'un sous-ensemble des facettes du modèle. Les exclusions (*Facets_Rejections*) sont représentées par un ensemble de couples alors que les implications (*Facets_Implications*) et implications légères (*Facets_SoftPlugs*) exploitent les types à base de graphes introduits dans le modèle. Ces dernières contraintes binaires sont exclusives par paires.

À l'exclusion de l'union, que nous introduisons spécialement pour le méta-modèle, les types du modèle lui-même suffisent à décrire le méta-modèle. L'implémentation du méta-modèle pourra donc se baser en grande partie sur celle du modèle.

Le modèle pourra aussi nous servir à décrire des concepts apparentés et nécessaires à de nombreuses applications, telles que les utilisateurs et les groupes, les autorisations, etc. (cf. annexe C).

8.5 Réponse aux besoins

Reprenons dans la table 8.4 le récapitulatif étudié dans la table 7.1. Comme nous pouvons l'observer, le modèle DOAN répond parfaitement aux besoins exprimés d'expressivité et de flexibilité.

S'il n'utilise pas le concept de classe, la notion de facette lui permet d'aisément typer les entités. Le système de types utilisé permet non seulement de représenter des objets complexes, mais va plus loin que les autres modèles en mettant à disposition des structures complexes particulièrement utiles pour représenter les entités des applications envisagées (ex. : documents, objets métiers, annotations, arbres de classement, hiérarchies d'utilisateurs, etc.).

La spécialisation, si elle n'est pas disponible en tant que telle, est suppléée par l'utilisation de contraintes entre les facettes. Rappelons que notre modèle a pour but la modélisation des données et non la modélisation de l'application en tant que telle. Aussi le code n'est-il pas central à notre approche et dans ce cadre, l'utilisation de contraintes d'implication est plus flexible que l'héritage des modèles à objets par exemple. Toutes facettes pouvant arbitrairement en impliquer une ou plusieurs autre(s) (tant que les contraintes existantes sont respectées), la héritage multiple est aussi disponible.

À l'exception des contraintes entre les facettes d'un schéma, il n'existe aucune restriction sur la nature et le nombre de facettes que peut supporter une entité, d'où le support de l'instanciation multiple.

Expressivité

	E/A	EER	UML	ORM	RDF	Rôles	Frames	DL	OWL	Besoin	DOAN
Classes d'entités	√	√	√	√	√	√	√	√	√	√	√
Types complexes	×	√	√	√	√	√	√	√	√	√	√
dont Structures complexes	×	×	×	×	×	□	×	×	×	√	√
Recouvrement instances et classes	×	×	×	×	√	×	√	×	□	×	×
Spécialisation / généralisation	×	√	√	√	√	√	√	√	√	√	√
Héritage multiple	×	√	√	√	√	√	√	√	√	√	√
Catégorisation	×	√	×	×	×	×	√	√	√	×	×
Associations	√	√	√	√	√	√	√	√	√	√	√
dont n-aires	√	√	√	√	×	□	×	×	×	×	×
Contraintes	√	√	√	√	√	√	√	√	√	√	√
dont logique de 1^{er} ordre	×	×	×	×	×	×	□	√	√	×	×
Raisonnement / Inférence	×	×	×	×	×	×	□	√	√	×	×

Flexibilité

	E/A	EER	UML	ORM	RDF	Rôles	Frames	DL	OWL	Besoin	DOAN
Multi instanciation	×	×	×	×	√	□	√	√	√	√	√
Acquisition / perte de type	×	×	×	×	√	□	□	□	√	√	√
Accès contextuel	×	×	×	×	×	√	×	×	×	√	√
Évolution du schéma	×	×	×	×	√	□	×	×	√	√	√
Fusion de schémas	×	×	×	×	□	×	×	×	√	×	×

Performances

	E/A	EER	UML	ORM	RDF	Rôles	Frames	DL	OWL	Besoin	DOAN
Persistance	√	√	√	√	√	√	√	√	√	√	?
Passage à l'échelle	√	√	□	√	□	□	×	×	□	√	?

Table 8.4 – Comparatif entre DOAN et les modèles étudiés

Grâce aux opérations d'attachement et de détachement, une entité peut, de plus, acquérir des nouveaux types et en perdre des anciens au cours de son existence.

L'encapsulation n'est pas supportée par notre modèle, mais en contre-partie, les facettes offrent un accès contextualisé aux entités. Sur le même principe que les modèles à base de rôles, l'application n'accède qu'aux informations concernant ses besoins. En revanche, contrairement à ces modèles, le support des facettes n'est pas limité par la classe d'origine de l'entité. En évitant l'utilisation de deux concepts liés (classes et rôles), les facettes de DOAN offrent un accès à la fois plus simple et plus flexible aux entités.

D'autre part, alors qu'un schéma est déjà instancié, il est possible de le faire évoluer en lui ajoutant une nouvelle définition de facette. Cette dernière pourra ensuite être attachée à des entités pré-existantes. La modification de facettes existantes est cependant beaucoup plus complexe, du fait des nombreux impacts sur les entités la supportant. Le système de types permettant de décrire des structures très complexes, la mise à jour des instances en fonction des modifications d'une facette nécessitera potentiellement une intervention manuelle.

Les performances et notamment le support de la charge ne sont, en revanche, pas abordés directement dans le modèle. Répondre à ces besoins est le rôle de l'implémentation qui est détaillée au chapitre 9.

Notre approche, si elle diffère des modèles étudiés auparavant, permet de simuler leurs principaux concepts. Observons cela d'abord avec le premier modèle que nous avons étudié, le modèle entité / association, puis avec un modèle à objets de plus haut niveau.

8.5.1 Simulation d'un schéma E/A

Considérons le schéma E/A général :

- $(E_1, \{a_{1,1} : T_{1,1}, a_{1,2} : T_{1,2}, \ldots, a_{1,n_1} : T_{1,n_1}\})$
- $(E_2, \{a_{2,1} : T_{2,1}, a_{2,2} : T_{2,2}, \ldots, a_{2,n_2} : T_{2,n_2}\})$
- $\ldots$
- $(R_1, \{a'_{1,1} : T'_{1,1}, a'_{1,2} : T'_{1,2}, \ldots, a'_{1,n'_1} : T'_{1,n'_1}\}, \langle E_{1,1}, \ldots, E_{1,p_1}\rangle)$
- $(R_2, \{a'_{2,1} : T'_{2,1}, a'_{2,2} : T'_{2,2}, \ldots, a'_{2,n'_2} : T'_{2,n'_2}\}, \langle E_{2,1}, \ldots, E_{2,p_2}\rangle)$
- $\ldots$

Sa traduction en un ensemble de facettes est simple et directe :

- $e_1 : \begin{array}{lcl} E & \rightarrow & T_{1,1} \times T_{1,2} \times \ldots \times T_{1,n_1} \\ e & \mapsto & (a_{1,1}, a_{1,2}, \ldots, a_{1,n_1}) \end{array}$
- $e_2 : \begin{array}{lcl} E & \rightarrow & T_{2,1} \times T_{2,2} \times \ldots \times T_{2,n_2} \\ e & \mapsto & (a_{2,1}, a_{2,2}, \ldots, a_{2,n_2}) \end{array}$
- $\ldots$
- $r_1 : \begin{array}{lcl} E & \rightarrow & T'_{1,1} \times T'_{1,2} \times \ldots \times T'_{1,n'_1} \times f_{1,1} \times \ldots \times f_{1,p_1} \\ e & \mapsto & (a'_{1,1}, a'_{1,2}, \ldots, a'_{1,n'_1} \times e_{1,1} \times \ldots \times e_{1,p_1}) \end{array}$
- $r_2 : \begin{array}{lcl} E & \rightarrow & T'_{2,1} \times T'_{2,2} \times \ldots \times T'_{2,n'_2} \times f_{2,1} \times \ldots \times f_{2,p_2} \\ e & \mapsto & (a'_{2,1}, a'_{2,2}, \ldots, a'_{2,n'_2} \times e_{2,1} \times \ldots \times e_{2,p_2}) \end{array}$
- $\ldots$

où, comme nous l'avons vu en section 8.3.2, f_x est un raccourci syntaxique pour le type référence **ref**(f_x).

Pour faciliter la traduction d'un schéma E/A en un modèle de données relationnel, il existe souvent des restrictions. De plus, comme le modèle E/A ne peut pas représenter facilement toutes les dépendances

fonctionnelles, le résultat d'une telle transformation en relationnel est légèrement inférieur à une modélisation directe incluant les dépendances fonctionnelles [67]. Les modèles E/A étendus sont cependant apparus ensuite pour faciliter la description de schémas complexes.

Le modèle DOAN permet la description d'attributs avec des types non atomiques, certains d'entre eux incorporant des dépendances fonctionnelles (**map**, **bijection**, **tree**, **dag**, **list**(T_1) $\equiv \{0, 1, \ldots\} \rightarrow T_1$, and **bag**($T_2$) $\equiv T_2 \rightarrow \mathbb{N}^*$). Bien que les associations du modèle E/A ne soient pas directement supportées, elles peuvent être simulées par des facettes (cf. section 8.3.2).

8.5.2 Simulation d'un modèle à objets

Les modèles à objets font partie des extensions du modèle E/A (ex. : OMT, UML) qui incorporent du code dans la description des classes d'entités, et mettent l'accent sur le concept d'héritage. Notre modèle étant conçu pour décrire et annoter des documents, prendre en compte le code n'est pas une question majeure pour nous.

Héritage d'implémentation. Concentrons-nous sur les concepts de *classe* et d'*héritage* (d'implémentation) [94]. Une classe est décrite par un nom et un ensemble d'attributs : $(C, \{a_1 : T_1, \ldots, a_n : T_n\})$ avec $n \geq 0$ (en général $n > 0$ s'il ne s'agit pas d'une classe abstraite racine). Ces attributs sont autant de fonctions des instances de la classe vers les domaines des attributs : $\{a_1 : C \rightarrow T_1, \ldots, a_n : C \rightarrow T_n\}$. Les classes n'ont pas besoin d'une clef puisque les instances sont différenciées grâce à la notion d'identifiant d'objet unique (OID) (cf. *notion d'identité* en section 3.1.1).

Pour décrire la notion de sous-classe, chaque description de classe est étendue avec un ensemble (éventuellement vide) de super-classes : $(C, \{C^1, C^2, \ldots, C^m\}, \{a_1, \ldots, a_n\})$. Le graphe de spécialisation induit doit être un treillis (un graphe acyclique orienté avec des restrictions supplémentaires), mais il est souvent réduit à un arbre. En substance, la définition concrète d'une classe consiste en l'union, récursive, des attributs de ses super-classes avec ses propres attributs, c'est-à-dire que les attributs d'une classe C contiennent, au moins :

$$\{a_1^1, \ldots, a_{n_1}^1, a_1^2, \ldots, a_{n_2}^2, \ldots, a_1^m, \ldots, a_{n_m}^m, a_1, \ldots, a_n\}$$

Une instance appartient à *une* classe, et *indirectement* à toutes ses super-classes.

Simulation d'un arbre de spécialisation. D'une façon similaire à la simulation E/A, chaque définition de classe C peut être traduite dans une facette correspondante :

$$f : \begin{array}{ccc} E & \rightarrow & T_1 \times \ldots \times T_n \\ e & \mapsto & (a_1, \ldots, a_n) \end{array}$$

Des contraintes doivent ensuite être employées. Tout d'abord, chaque lien d'héritage entre une classe C et sa super-classe (unique) C^1 est traduit par une contrainte d'implication : f implies f^1.

En deuxième lieu, si C et C' sont des sous-classes de C^1, alors une contrainte d'exclusion doit être créée : f rejects f'. Avec m sous-classes directes, cela nous conduit à créer $\frac{m(m+1)}{2}$ contraintes de ce type. Cependant, il n'est pas nécessaire d'étendre l'exclusion à toutes les classes descendantes. En effet, lors de la tentative d'attachement d'une nouvelle facette à une entité, les implications récursives vont conduire au besoin de créer des facettes incompatibles, de fait interdisant l'attachement de la facette originale et de ses facettes impliquées.

Enfin, alors que les classes abstraites sont traduites en facettes, les classes non-abstraites sont traduites en *facettes principales*.

On remarquera que cette traduction est comparable à l'une des alternatives d'implémentation de schémas à objets dans des bases de données relationnelles [83].

Contrairement à la plupart des modèles à objets, mais d'une façon analogue à RDF, une entité dans notre proposition peut être associée à plusieurs facettes (classes) qui sont soit compatibles (pas de réjection), soit non-apparentées. Par exemple, une image provenant d'une page HTML peut être associée à une facette générale IMAGE décrivant sa taille, son format, etc., mais pas à la facette AUDIO qui est incompatible (puisque toutes deux « héritent » de la facette MULTIMÉDIA). Elle pourra cependant être associée à une facette non-apparentée PEINTURE et éventuellement à une facette ŒUVRE D'ART. Ainsi, notre modèle permet la *multi-instanciation*, qui est plus flexible que la modélisation à base de rôle.

8.6 Conclusion

Ce chapitre nous a permis de présenter le modèle DOAN et l'ensemble de ses éléments. DOAN est construit autour de la notion de *facette*. S'inspirant des concepts de classe et de rôle, les facettes offrent une meilleure flexibilité. En effet les entités peuvent en supporter un nombre illimité et aisément en changer au cours de leur existence.

En outre, tout en offrant un accès contextuel aux entités les supportant, les facettes détiennent leurs données et diffèrent en cela de la notion de vue.

Enfin, un ensemble de contraintes permet au concepteur d'exprimer les particularités du domaine à représenter. Plus souples que les notions de classe et de spécialisation, ces contraintes permettent cependant de les simuler. Complétées par un système de types riche, elles sont la source de l'expressivité du modèle.

Ces caractéristiques permettent à DOAN de répondre aux besoins d'expressivité et de flexibilité exprimés dans la problématique.

Nous répondons au troisième besoin, celui de performances, grâce à une implémentation relationnelle. Dans le chapitre suivant, nous introduirons ainsi l'algorithme de traduction des types de DOAN dans le modèle relationnel, tandis dans le chapitre 10, nous observerons plus précisément les performances et la réponse à la charge.

CHAPITRE 9

Implémentation

Comme nous l'avons vu au chapitre 2, les solution d'intelligence économique doivent traiter d'importantes quantités de données hétérogènes. Pour gérer cette information, ses concepteurs ont besoin d'un modèle expressif et flexible. Les approches disponibles répondant, en partie, à ces besoins sont cependant très limitées dès qu'elles sont confrontées à des montées en charges notables (cf. chapitre 7). Elles ne sont généralement pas construites sur une base de données, et quand c'est le cas, elles souffrent d'une pénalité supplémentaire due à la traduction à la volée entre leurs modèles natifs et les tables d'un système de gestion de bases de données relationnelles (SGBDR). De plus, elles ne disposent pas forcément de langage de requêtes et, si c'est le cas, l'optimisation des requêtes SQL correspondantes est difficile. Une requête ensembliste peut ainsi se retrouver décomposée en plusieurs requêtes unitaires. En conséquence, pour répondre aux besoins de montées en charge et de performances, les concepteurs d'application s'appuient souvent directement *sur des SGBDR, très présents sur le marché.*

Étant donnés ces deux besoins contradictoires, nous avons choisi de combiner les avantages d'un modèle flexible et expressif, le modèle DOAN, avec une implémentation relationnelle. Nous bénéficions ainsi d'un modèle riche, mettant à disposition de nombreux types de haut niveau, et d'un moyen performant d'en interroger les données. L'exécution de requêtes SQL est en effet très efficace, particulièrement quand toute l'information relative à une entité n'est pas requise par l'application (en évitant notamment des jointures coûteuses).

Après avoir exposé les raisons de notre choix des SGBD relationnels, nous allons dans ce chapitre détailler la traduction des types du modèle DOAN dans le modèle relationnel. Nous terminerons par une description succincte du noyau servant à l'implémentation.

9.1 Choix du modèle relationnel

Il est possible de traduire le modèle DOAN dans de nombreux modèles existants. Parmi eux, le modèle à objets permet sans doute la transformation (*mapping*) la plus directe. En effet, comme nous l'avons vu en section 8.3.3, les types que nous utilisons sont très proches de la proposition de l'*Object Data Management Group* (ODMG) [22]. La plupart d'entre eux trouvent leur équivalent en objet comme le montre la table 8.1 en page 85.

XML est également une cible intéressante. Son utilisation est très répandue et il existe désormais des bases de données dédiées à son traitement [13, 40]. Les avancées dans la standardisation de X-Query [11] promettent en outre d'intéressantes possibilités de requêtage.

Notre choix s'est cependant orienté vers le modèle relationnel. Plusieurs raisons nous y ont menés :

- Tout d'abord, les SGBDR sont le résultat de dizaines d'années de recherche et sont aujourd'hui un vrai standard industriel mature. Ils répondent efficacement aux besoins de traitement des volumes

importants d'informations traités par les applications Arisem (des millions de documents et des milliers d'utilisateurs). Ils satisfont, de plus, les futurs besoins de montée en charge.

- Les SGBDR dominent, par ailleurs, le marché du stockage de l'information. Il est ainsi très probable que les intégrateurs des applications Arisem soient déjà formés à leur utilisation, et les clients à leur maintenance. Ce choix permet aussi de limiter les coûts de licences pour le client qui pourra, au moins dans un premier temps, héberger la base de données applicative sur un serveur mutualisé.
- D'autre part, la consultation des données bénéficie grandement de l'efficacité des requêtes SQL dès lors que l'on travaille avec des ensembles. L'utilisation de lectures par lots permet d'obtenir en un laps de temps très court des informations sur de très nombreux enregistrements. En outre, quand toutes les informations d'une entité donnée ne sont pas requises par l'application, ces dernières peuvent être omises de la requête, évitant ainsi des jointures potentiellement coûteuses.
- Enfin, Arisem bénéficie dans ce domaine d'une large expérience acquise avec le modèle précédemment utilisé [5], qui était lui aussi implémenté dans une base de données relationnelle.

Comparés aux SGBD à objets ou encore aux SGBD XML, le choix des SGBD relationnels répond donc davantage à un souci de performances qu'à un besoin d'apports fonctionnels. L'expérience montre que les SGBD à objets ou XML ne sont pas aujourd'hui en adéquation avec les besoins des entreprises. De plus, si le modèle relationnel ne permet pas une transposition évidente depuis les types de données de notre modèle, il est cependant très adapté à sa logique d'agrégation de facettes. Les données correspondant à une entité se retrouvent ainsi distribuées sur autant de relations que de facettes supportées[1].

Des transformations dans d'autres modèles, en particulier objet et XML, sont cependant envisagées à l'avenir, en fonction de l'évolution de ces technologies. Le modèle, indépendamment de son implémentation, continuera à diriger le système d'information.

Nous avons choisi de clairement distinguer les accès en modification des données de ceux en pure consultation. Ainsi, le modèle fournit un choix de « méthodes » sous la forme d'une API (*Application Programming Interface*) pour peupler et mettre à jour la base, alors que la consultation se fait avec le langage du modèle d'implémentation choisi (OQL, *X-Query* ou SQL). Cette approche est similaire à la proposition de l'ODMG où les requêtes peuvent désencapsuler des objets [22].

L'utilisation d'une API explicitant les manipulations possibles permet d'assurer la consistance des données, tout en simplifiant leur mise à jour : il est plus aisé de faire appel à une procédure stockée (dans le cas de notre implémentation relationnelle) que de modifier directement les champs d'une ou plusieurs tables. La création de ces opérations étant de plus standardisée, ce choix nous permet d'activer à la demande un ensemble d'actions supplémentaires pour chaque opération (ex. : sécurité, journalisation, etc.).

En utilisant un SGBDR, les requêtes de consultation se font directement en SQL. Les intégrateurs sont ainsi libres de tirer parti au maximum des possibilités offertes par le langage et peuvent faire des requêtes à la fois complexes et performantes. Les simples consultations étant généralement beaucoup plus fréquentes que les modifications, cet « allègement » peut avoir un impact important sur les temps de réponse.

Afin de prendre en compte des bonnes pratiques de consultation, il est cependant conseillé d'utiliser un langage de macros pour simplifier l'écriture de requêtes. La gestion de la sécurité par exemple peut devenir beaucoup plus simple à réaliser et être moins sujette aux oublis dès que des macros sont utilisées (cf. annexe C).

[1] En réalité, une facette pourra être représentée par plusieurs relations en fonction de sa complexité.

$Document$: **facet**(**tuple**($titre$: **string**,
$date$: **date**,
$auteurs$: **list**($auteur$: AUTEUR),
$contenu$: **list**(**string**),
$annotations$: **set**($annotation$: **tuple**($auteur$: PERSONNE),
$contenu$: **list**(**string**)))))

Figure 9.1 – Définition du type *Document* avec une syntaxe proche de l'ODMG

9.2 Traduction dans le modèle relationnel

Une fois le modèle relationnel choisi comme cible d'implémentation, il nous faut y traduire tous les types de données du modèle.

Dans le but d'avoir une compréhension d'ensemble des résultats générés par l'algorithme de traduction, commençons par observer un exemple. Dans la figure 9.1, nous reprenons la description, donnée en section 8.3.1, d'informations collectées au sujet de documents et de leurs auteurs. Plusieurs types de données complexes, imbriqués arbitrairement, sont utilisés dans cet exemple.

Dans le monde relationnel, à cause de l'exigence de forme normalisée, cette définition claire doit être traduite dans un formalisme plus complexe. Plus précisément, nous désirons obtenir un résultat similaire aux relations de la figure 9.2 (ainsi que les procédures stockées associées pour gérer les opérations de mise à jour, ex. : `Document_annotations_contenu_insert(...)`).

9.2.1 Aperçu de l'algorithme de traduction

Afin de clarifier l'algorithme de traduction, nous l'avons segmenté en trois étapes :

1. Une transformation récursive de définitions de types complexes (similaire à l'ODMG) en une représentation proche du modèle relationnel ;
2. Une seconde transformation de la représentation proche du relationnel en une représentation vraiment relationnelle, durant laquelle des choix d'implémentation de haut niveau doivent être faits ;
3. Une dernière transformation dans une variante relationnelle, prenant en compte les derniers choix d'implémentation et les spécificités du SGBDR choisi.

Cette séparation en étapes successives nous permet d'isoler les choix à faire. La première étape est indépendante de l'implémentation et permet « d'aplatir » les collections, c'est-à-dire de supprimer leurs éventuelles imbrications. Les choix d'implémentations nécessaires à la traduction n'arrivent qu'en deuxième étape. On y choisira, par exemple, la meilleure implémentation pour un graphe en fonction des types utilisés ou des indications du concepteur. Et enfin, l'isolation de la traduction effective en SQL dans une dernière étape dédiée facilite le portage sur des SGBDR spécifiques.

Le concepteur peut configurer l'algorithme de traduction associant des indications (*hints*) aux types qu'il utilise. Il peut notamment indiquer s'il a besoin ou non de générer une fermeture pour les arbres ou les graphes acycliques orientés.

Cette démarche est similaire à l'approche MDA (*Model Driven Architecture*) [75] qui sépare la logique d'application de la plateforme d'implémentation. Des techniques de *mapping* standardisées permettent de transformer un modèle indépendant de la plate-forme (PIM, *Platform-Independent Model*) en un modèle spécifique à une plate-forme (PSM, *Platform-Specific Model*), et finalement en une implémentation (plusieurs niveaux de PIM et de PSM sont possibles).

Document	
id	**Entity**
titre	**varchar**
date	**datetime**
auteursID	**surrogate**
contenuID	**surrogate**
annotationsID	**surrogate**
primary key (id)	

Document_auteurs		
auteursID	**surrogate**	
index	**integer**	**check(index $\geq$ 0)**
auteur	**AuteurRef**	
primary key (auteursID, index)		

Document_contenu		
contenuID	**surrogate**	
index	**integer**	**check(index $\geq$ 0)**
contenuMember	**varchar**	
primary key (contentID, index)		

Document_annotations	
annotationsID	**surrogate**
auteur	**PersonneRef**
contenuID	**surrogate**
primary key (annotationsID, auteur, contenuID)	

Document_annotations_contenu		
contenuID	**surrogate**	
index	**integer**	**check(index $\geq$ 0)**
contenuMember	**varchar**	
primary key (contenuID, index)		

Figure 9.2 – Contrepartie relationnelle de la définition de la figure 9.1

Avant de détailler chacune de ces transformations, passons en revue quelques aspects nécessaires à la bonne compréhension de l'algorithme.

9.2.2 Notions préléminaires

Les types du modèle. Nous avons introduit en section 8.3 le système de types du modèle DOAN. Rappelons-les brièvement ici :

Définition 3. *Les types de données du modèle peuvent être classés de la façon suivante :*

- *types atomiques : boolean, integer, real, char, string et date (plus surrogate et reference) ;*
- *types unaires : set, list, bag, tree, dag, et graph (ex. : list(string)) ;*
- *types binaires : map et bijection (ex. : map(integer, string)) ;*
- *types n-aires : tuple et relation (ex. : relation(name: string, age: integer, address: tuple(...))).*

Nous distinguons également les regroupements suivants :

- *collections : types unaires et binaires, relation ;*
- *fonctions : bag, tree, list, bijection, map.*

Le type **surrogate** est particulier car il n'est pas disponible durant la conception. Il est formellement un produit dérivé du processus de transformation et est utilisé dans le modèle relationnel final pour définir des clés primaires et externes. En pratique, il correspond à un simple alias vers **integer**.

Désignation des attributs. Lors de sa traduction dans le modèle relationnel, une facette peut être transformée en de multiples relations. Au cours de la traduction, nous pouvons donc être amenés à créer de nouveaux attributs qu'il faudra nommer en fonction de la facette originale.

La désignation de ces nouveaux attributs est réalisé en fonction d'une pile de préfixes qui contient les noms des attributs étant à l'origine de la création. Un caractère spécial, '_' (blanc souligné), est utilisé comme séparateur. Par exemple, si une facette AUTEUR contient des publications, elles pourront être décrites, après traduction en relationnel, dans un nouvel attribut nommé $auteur\_publications$.

De plus, lorsqu'il définit une facette, le concepteur n'est pas obligé de donner un nom à chaque sous-type la composant. Pour éviter toute ambiguïté et être en adéquation avec le modèle relationnel, le résultat de la traduction ne contiendra que des attributs correctement nommés. Il est donc nécessaire, lors des transformations, d'attribuer des noms aux types qui n'en n'ont pas, en particulier lors de la création d'attributs supplémentaires à partir du type original.

Quand il est nécessaire, le choix de ces noms dépend du contexte de création et du type du nouvel attribut. Prenons le cas d'une collection dont nous voulons extraire le domaine dans un nouvel attribut « délégué » :

$$name : \textbf{collection}(T)$$

Le domaine de la collection n'étant pas nommé, on doit tout d'abord lui attribuer un nom. On utilisera ici le suffixe « member » que l'on concaténera au nom de la collection : $nameMember$. Ainsi, en utilisant le nom original comme préfixe du nouvel attribut (cf. ci dessus), on obtiendra comme nouvel attribut :

$$name\_nameMember : \textbf{bijection}(nameMemberID : \textbf{surrogate}, nameMember : T)$$

Si cette approche permet d'attribuer systématiquement un nom valide à tous les attributs, il est conseillé au concepteur, pour plus de clarté, de nommer explicitement tous les types qu'il utilise.

Les collections imbriquées. Avant de détailler l'algorithme de traduction en relationnel, observons l'impact que peut avoir la complexité que nous autorisons dans la création des types. Le concepteur a en effet la possibilité d'imbriquer plusieurs collections dans une même définition. Il pourra par exemple créer un attribut *annuaire* comme étant un **set**(**list**(**string**)).

Nous sommes donc en présence de collections de collections, complexité que le modèle relationnel standard ne supporte pas. On notera toutefois que les extensions récentes du langage SQL permettent de telles définitions (cf. SQL:2003 [37]). Étant donné d'une part le faible support actuel de ce récent standard, et d'autre part la restriction des collections disponibles (array et multiset), nous préférons nous appuyer sur les concepts plus anciens et mieux supportés de SQL-92 [29] au niveau relationnel, tout en nous réservant la possibilité de tirer parti des fonctionnalités supplémentaires fournies par les SGBD spécifiques au niveau SQL.

Par conséquent, nous nous retrouvons confrontés à la nécessité de matérialiser les collections imbriquées. Pour cela, nous créons une table différente pour stocker les données de chaque collection et nous utilisons des *représentants* (**surrogate**s) pour faire le lien entre les différents niveaux d'imbrication.

Dans certaines situations, il est possible de se passer de la création d'une table dédiée à chaque collection. Toutefois, si les attributs générés alors par la traduction sont bien relationnels, il nous est impossible de créer des opérations, c'est-à-dire des procédures stockées, cohérentes. Prenons l'exemple d'une liste de familles : **list**(**bag**(a : **T**)). Si **T** est un type de base, nous pourrions utiliser l'application suivante comme traduction relationnelle :

$$\begin{array}{c}\textbf{map}(\textbf{tuple}(index : \textbf{natural},\\ a : \textbf{T}),\\ count : \textbf{natural})\end{array}$$

Cependant, nous serions alors confrontés à un conflit entre la liste et la famille dans la création des opérations. L'opération d'ajout d'un nouvel élément a (`add`) doit-elle l'ajouter à la fin de la liste ou à l'intérieur d'une famille pour un index précis ? De plus, s'il est possible de lever l'ambiguïté sur ces opérations au prix d'un écart à leurs conventions de nommage (en rendant leurs noms plus explicites), il nous est impossible de modéliser la présence d'une famille vide dans la liste. Ces raisons nous incitent à systématiquement créer un nouvel attribut pour les collections imbriquées dans d'autres collections. Une collection est donc toujours matérialisée dans une table dédiée.

Même si SQL:2003 permet de créer des collections avec des valeurs initiales[2], nous ne pouvons pas nous appuyer dessus car il n'est pas encore suffisamment répandu sur le marché. Ainsi, nous créons toujours des collections vides par défaut[3]. L'ajout d'élément se fera par l'utilisation des opérations générées conjointement aux tables (typiquement une opération `add`).

Deux cas peuvent se présenter selon que l'on travaille avec :

- une collection de premier niveau ;
- ou une collection imbriquée dans une autre.

Aucune difficulté particulière ne se présente dans le premier cas, la table contient directement les éléments de la collection. Le second est cependant plus complexe car il faut identifier la collection que l'on veut modifier. Nous nous servons alors d'un identifiant (de type **surrogate**) généré automatiquement lors de la création de l'enregistrement « contenant » la collection. Cela contraint cependant l'intégrateur à obtenir cet identifiant avant d'appeler l'opération désirée (cf. section 9.2.5.2).

[2] `INTEGER MULTISET(2,3,5,7)` par exemple crée un ensemble contenant quelques nombres premiers.

[3] Dans certains cas, ce comportement pourra être spécialisé en insérant systématiquement des valeurs, notamment en présence de contraintes sur la cardinalité.

9.2.3 Transformation en relationnel-proche

Attachons-nous maintenant à décrire toutes les étapes de la traduction relationnelle, à commencer par la transformation en une représentation proche du modèle relationnel. Selon la complexité du type de données à transformer, cette première étape peut être évitée ou au contraire se révéler complexe et fastidieuse pour le concepteur (ou le programmeur) s'il devait la réaliser manuellement. Son but est de transformer un attribut, dont le type n'est pas contraint, en un ou plusieurs attributs ayant tous un type proche du modèle relationnel.

Un type relationnel-proche consiste en au plus trois niveaux de différentes natures :

1. Une éventuelle collection (ex. : **set**, **map**, **list**) contenant ...
2. Un éventuel n-uplet (ou des n-uplets imbriqués) contenant ...
3. Un ou plusieurs attribut(s) atomique(s) (ex. : **integer**, **char**).

De manière plus formelle, la définition d'un *type relationnel-proche* peut être établie grâce aux définitions (récursives et croisées) suivantes.

Définition 4. *Un type est un* type relationnel-proche basique *si c'est :*

- *un type atomique ;*
- *un* n-uplet relationnel-proche.

Par extension, on dira d'un attribut qu'il est relationnel-proche basique *quand il est défini par un* type relationnel-proche basique.

Définition 5. *Un* n*-uplet est un* n-uplet relationnel-proche *si les tous ses attributs sont définis par des* types relationnels-proches basiques.

Définition 6. *Un type est un* type relationnel-proche *si c'est :*

- *un* type relationnel-proche basique *;*
- *une collection unaire, dont le domaine est* relationnel-proche basique *;*
- *une application ou une bijection, dont le domaine et la portée sont* relationnels-proches basiques *;*
- *une relation, dont tous les attributs sont* relationnels-proches basiques.

Par extension, on dira d'un attribut qu'il est relationnel-proche *quand il est défini par un* type relationnel-proche.

Par exemple, les types

$$\textbf{string}$$

ou

$$\textbf{tuple}(name : \textbf{string}, \; age : \textbf{integer})$$

ou même

$$\textbf{map}(employeeID : \textbf{integer}, \; tuple : (name : \textbf{string}), \; address : \textbf{tuple}(ZIP : \textbf{string}, \; city : \textbf{string}, \ldots)))$$

sont des types de plus en plus complexes mais restent des types relationnels-proches. À l'opposé, le type **set**(**list**(**string**))) n'est pas un relationnel-proche car il contient des collections imbriquées.

Ainsi, le but de cette première étape de la traduction dans le modèle relationnel est « d'aplatir » les collections dans les définitions. Dans la suite de cette section, nous allons détailler les transformations des différents types. Notons que les types atomiques sont déjà relationnels-proches : ils ne nécessitent donc pas d'être transformés.

9.2.3.1 Transformation des types unaires

Parmi les types unaires du modèle, on dispose des ensembles (**set**), familles (**bag**) et listes (**list**) d'une part, et de types à base de graphes d'autre part : arbres (**tree**), graphes (**graph**) et graphes acycliques orientés (**dag**). Rappelons que dans le cadre des transformations décrites ci-après, le domaine de ces types n'est pas relationnel-proche basique. En effet, si c'était le cas, ils seraient directement relationnels-proches et aucune transformation ne serait nécessaire.

Considérons tout d'abord les ensembles, familles et listes. N'étant pas relationnel-proche basique, le domaine de ces types unaires est soit une collection, soit un n-uplet contenant une collection. Si le premier cas nécessite la création d'un nouvel attribut, de simples réécritures sont suffisantes dans le cas de n-uplets. Leur résultat doit cependant être transformé récursivement à son tour.

Ensemble (set) de n-uplets. La notion d'ensemble se prête très bien au modèle relationnel dont le concept de base, la relation, n'est autre qu'un ensemble de n-uplets. Cette équivalence nous permet de remplacer directement le type **set** par une **relation** lorsque son domaine est un n-uplet :

$$\begin{array}{c} name : \textbf{set}([a :] \textbf{tuple}(a_1 : T_1, a_2 : T_2, \ldots, a_n : T_n)) \\ \Downarrow \\ name : \textbf{relation}(a_1 : T_1, a_2 : T_2, \ldots, a_n : T_n) \end{array} \tag{9.1}$$

Famille (bag) de n-uplets. Contrairement à l'ensemble (**set**), une famille (**bag**) n'a pas de contrainte d'unicité sur ses éléments. Il est possible d'utiliser une transformation similaire à celle de l'ensemble en évitant explicitement de contraindre l'unicité. Toutefois, nous préférons une autre transformation, particulièrement adaptée quand on s'intéresse au nombre d'occurrences des éléments :

$$name : \textbf{bag}([a :] \textbf{tuple}(\ldots)) \quad \Rightarrow \quad \begin{array}{r} name : \textbf{map}(a : \textbf{tuple}(\ldots), \\ count : \textbf{natural}) \end{array} \tag{9.2}$$

Liste (list) de n-uplets. Une liste est une collection dans laquelle les éléments sont ordonnés en se basant sur un index commençant à zéro. Pour traduire cela en relationnel-proche, nous avons choisi d'introduire explicitement l'index[4] :

$$name : \textbf{list}([a :] \textbf{tuple}(\ldots)) \quad \Rightarrow \quad \begin{array}{r} name : \textbf{map}(index : \textbf{natural}, \\ a : \textbf{tuple}(\ldots)) \end{array} \tag{9.3}$$

Collection de collections. Nous sommes ici dans le cas où le domaine du type unaire considéré est une collection. Pour rendre le type relationnel-proche, nous introduisons un représentant et créons une bijection entre ce dernier et la collection correspondante :

[4] Formellement, une liste est un graphe dans lequel chaque sommet, à l'exception du premier et du dernier, a un et seulement un prédécesseur, et un et seulement un successeur. La transformation que nous avons choisie est plus simple, plus efficace, et basée sur la propriété qu'ont les valeurs des rangs des sommets à être uniques dans une liste.

$$
\begin{array}{c}
name : \textbf{unaryCollection}([a :] \textbf{collection}(T)) \\
\Downarrow \\
\begin{array}{l}
name : \textbf{unaryCollection}(aID : \textbf{surrogate}) \\
nameValues : \textbf{bijection}(aID : \textbf{surrogate}, \\
\qquad\qquad\qquad\qquad a : \textbf{collection}(T))
\end{array}
\end{array}
\tag{9.4}
$$

La collection de représentants sera réécrite dans l'étape suivante. Quant à la bijection (qui n'est pas relationnelle-proche car elle contient une collection), elle est traitée en section 9.2.3.3.

Limitation pour les types à base de graphes. Afin d'aider le concepteur, nous avons introduit dans le modèle DOAN des types à base de graphes (notons qu'ils sont absents de la proposition de l'ODMG). Pour simplifier leur utilisation, nous choisissons cependant de les limiter à ceux qui sont relationnels-proches.

La manipulation des attributs ayant un de ces types nécessite de créer ou de détruire des liens entre leurs valeurs. En effet, la création d'un nœud dans un arbre implique, par exemple, la création d'un lien entre le nouveau nœud et son nœud parent.

Or, il n'y a pas de réécriture simple pour les types à base de graphe : arbres (**tree**), graphes (**graph**) et graphes acycliques orientés (**dag**). Quand leur domaine a une définition complexe, nous devons introduire un représentant pour les rendre relationnels-proches. Cela déclenche la création d'un second attribut qui représente le lien un-à-un du représentant avec l'élément d'origine.

En conséquence, si un programmeur veut créer un lien entre deux nœuds d'un graphe qui sont des collections de valeurs, il lui faut préciser toutes ces valeurs dans l'appel de l'opération pour identifier les nœuds. Si la génération d'une telle procédure stockée est possible, la complexité de sa signature est telle qu'elle en devient inutilisable, et ce, particulièrement si le domaine du graphe contient lui aussi des collections imbriquées.

Ainsi, bien qu'il soit possible de transformer les types à base de graphe en relationnel-proche (avec la transformation 9.4), la traduction complète ne pourra aboutir à des opérations utilisables (cf. section 9.2.5.2). Nous préférons donc interdire de telles définitions et forcer le concepteur à réifier explicitement leur domaine, et à utiliser une référence.

9.2.3.2 Transformation des n-uplets et relations

La transformation des n-uplets (**tuple**s) et des relations est similaire. Nous commençons tout d'abord par transformer chaque attribut présent dans le n-uplet (resp. relation) en relationnel-proche, en utilisant au besoin un **surrogate**. La convention de réécriture nous assure que le premier attribut généré peut être utilisé comme délégué si une transformation conduit à la création de plusieurs attributs. Ainsi, les premiers attributs de toutes les transformations résultantes sont utilisés en tant que délégués et regroupés dans un nouveau n-uplet (resp. relation). Le résultat de la transformation est donc l'ensemble composé de ce n-uplet (resp. relation) et des éventuels attributs supplémentaires générés lors des transformations des attributs.

Observons plus en détail les différents cas possibles selon le type des attributs à transformer :

- Type de base. Il n'y a besoin de rien transformer, l'attribut est déjà relationnel-proche.
- N-uplet. On utilise alors de façon récursive cette même transformation. Le n-uplet généré est relationnel-proche et peut être utilisé comme délégué de cet attribut dans le n-uplet ou la rela-

tion finale, tandis que les éventuels attributs supplémentaires sont ajoutés au même titre que ceux générés par la transformation principale.

- Collection (notons que les collections incluent les relations). Dans ce cas, on crée un représentant de la collection avant de la transformer elle-même.

$$\begin{array}{c} attName : \textbf{collection}(a : A) \\ \Downarrow \\ attNameID : \textbf{surrogate} \\ attName : \textbf{bijection}(attNameID : \textbf{surrogate}, \\ \qquad\qquad attName : \textbf{collection}(a : A)) \end{array} \tag{9.5}$$

Le premier attribut créé servira donc de représentant pour l'attribut dans le n-uplet (resp. relation) généré ensuite. Le deuxième attribut n'est qu'une réécriture, et est donc transformé à son tour. Le résultat de cette transformation sera ajouté au résultat global.

9.2.3.3 Transformation des bijections

Si la bijection n'est *a priori* presque jamais utilisée pour décrire des types, elle est en revanche très utile comme type intermédiaire dans les transformations (ex. : les différentes collections). Elle se traduit par un lien un-à-un entre deux attributs :

$$\begin{array}{l} name : \textbf{bijection}([a :]\ A, \\ \qquad\qquad [b :]\ B) \end{array}$$

Nous nous plaçons ici dans le cas où le domaine A et le codomaine B ne sont pas tous deux relationnels-proches basiques. En effet, la bijection serait alors relationnelle-proche et aucune transformation supplémentaire ne serait nécessaire.

Si le domaine ou le codomaine est une collection, nous dégradons la bijection en une application. Imaginons que le codomaine soit une collection (dans le cas contraire, le domaine et la portée de l'application sont simplement inversés), nous réécrivons alors la bijection comme suit :

$$\begin{array}{l} name : \textbf{bijection}(a : A, \\ \qquad\qquad b : collection(T)) \end{array} \quad \Rightarrow \quad \begin{array}{l} name : \textbf{map}(a : A, \\ \qquad\qquad b : collection(T)) \end{array} \tag{9.6}$$

En utilisant cette dégradation, nous perdons une *dépendance fonctionnelle* : les valeurs de la collection ne nous permettent pas de déterminer celles du domaine correspondant. Nous sommes ici confrontés au même problème que celui des types à bases de graphes évoqué précédemment : la sémantique des valeurs des collections rend la requête permettant d'obtenir le domaine de la bijection à partir de son codomaine potentiellement trop complexe pour être exprimable par un programmeur.

Il est cependant possible de s'assurer de l'unicité des valeurs de la collection (à l'exception des listes qui, par définition, peuvent contenir plusieurs fois le même élément). Pour ce faire, nous avons le choix entre utiliser un index unique sur la table représentant la collection (et récursivement sur ses éventuelles collections imbriquées) et vérifier cette unicité dans les procédures stockées représentant les opérations. Toutefois, la vérification de cette contrainte d'unicité est coûteuse et nous avons choisi de la désactiver par défaut. Le concepteur peut cependant la forcer au besoin.

Pour ces mêmes raisons de complexité sémantique, nous interdisons la définition de bijections entre des collections (dans le cas contraire, la dégradation en application est impossible) : soit son domaine,

soit son codomaine, doit être un type relationnel-proche basique. La bijection étant un type rarement utilisé pour la conception, et l'algorithme de traduction que nous décrivons dans ce chapitre ne générant que des bijections avec un domaine de type **surrogate**, cette limitation ne pose guère de problèmes.

Tout en respectant ces conditions imposées, le domaine et le codomaine d'une bijection peuvent être très complexes (ex. : un n-uplet contenant des n-uplet imbriquées en tant que domaine, et une collection de forte imbrication elle-aussi en tant que codomaine). Aussi, pour simplifier le résultat de la transformation, nous commençons par transformer son domaine et son codomaine en relationnel-proche. Le processus étant récursif, plusieurs attributs peuvent alors être créés. La convention choisie pour la réécriture de type nous assure cependant que le premier attribut retourné peut toujours être utilisé en tant que délégué du domaine ou codomaine (en utilisant un **surrogate**), comme par exemple dans la transformation (9.4).

Par conséquent, nous obtenons à la suite de la transformation d'une bijection :

- d'une part une bijection (dont le domaine et le codomaine sont relationnels-proches basiques) ou une application (dont le domaine est relationnel-proche basique),
- et d'autre part d'éventuels attributs dérivés qui sont relationnels-proches.

Dans le cas où une application est générée, il est nécessaire de compléter la transformation afin de supprimer le dernier niveau d'imbrication. Pour ce faire, on applique une nouvelle transformation à l'application, décrite ci-après.

9.2.3.4 Transformation des applications

L'application (**map**) est l'un des types les plus couramment utilisés. Il sert à la fois pour décrire des types utilisateurs et dans le cadre des transformations, particulièrement lorsque l'on utilise un représentant (**surrogate**) pour un type complexe.

$$name : \textbf{map}([a :] A, \\ [b :] B)$$

Pour les même raisons qui nous ont poussés à limiter le type du domaine et du codomaine des bijections, nous restreignons le type du domaine des applications à être de relationnel-proche basique. La sémantique des valeurs des collections est trop complexe pour les utiliser en tant que domaine.

En suivant le même principe que pour les bijections, nous commençons par simplifier le domaine et la portée de l'application en les transformant en relationnel-proche. Ces transformations récursives auront pour résultat un ou plusieurs attributs. Cependant, dans le cas où plusieurs attributs sont générés, la convention de réécriture choisie nous permet de toujours utiliser le premier comme délégué du domaine ou de la portée (utilisation d'un **surrogate**).

Nous obtenons donc une application dont le domaine et la portée sont relationnels proches et éventuellement un ensemble d'attributs relationnels-proches dérivés de ces premières transformations. Nous avons alors un ensemble de cas possibles qui dépend du type de la portée :

1. Type relationnel-proche basique. Aucune transformation supplémentaire n'est nécessaire. L'application est déjà relationnelle-proche.
2. Ensemble (**Set**).

$$name : \textbf{map}(a : A, \\ b : \textbf{set}([c :] C)) \quad \Rightarrow \quad name : \textbf{relation}(a : A, \\ c : C) \tag{9.7}$$

3. Graphe (**Graph**), graphe acyclique orienté (**Dag**) et arbre (**Tree**).

$$\begin{array}{l} name : \textbf{map}(a : A, \\ \qquad\qquad b : \textbf{collection}([c :] C)) \\ \qquad\qquad \Downarrow \\ name : \textbf{bijection}(a : A, \\ \qquad\qquad bID : \textbf{surrogate}) \\ name\_b : \textbf{collection}(\textbf{tuple}(bID : \textbf{surrogate}, \\ \qquad\qquad c : C)) \end{array} \tag{9.8}$$

La transformation la plus simple (**map**(**tuple**$(A, C), C)$) n'est pas toujours possible car elle ne fonctionne pas si la collection est vide.

De plus, dans le cas particulier où le domaine de la collection est un **surrogate**, la création de la bijection intermédiaire n'est pas nécessaire. En effet, les opérations correspondant à l'application n'ont pas besoin d'apparaître explicitement car elle a été générée comme résultat d'une transformation précédente. Nous pourrons donc en profiter pour simplifier les opérations résultant de la traduction de la collection (voir l'exemple donné en section 9.2.5.3). Nous obtenons alors :

$$\begin{array}{l} name : \textbf{map}(a : \textbf{surrogate}, \\ \qquad b : \textbf{collection}([c :] C)) \end{array} \quad \Rightarrow \quad \begin{array}{l} name : \textbf{collection}(\textbf{tuple}(a : \textbf{surrogate}, \\ \qquad\qquad c : C)) \end{array} \tag{9.9}$$

4. Relation (**Relation**).

$$\begin{array}{l} name : \textbf{map}(a : A, \\ \qquad\qquad b : \textbf{relation}(b_1 : T_1, b_2 : T_2, \ldots, b_n : T_n)) \\ \qquad\qquad \Downarrow \\ name : \textbf{relation}(a : A, b_1 : T_1, b_2 : T_2, \ldots, b_n : T_n) \end{array} \tag{9.10}$$

5. Application (**Map**).

$$\begin{array}{l} name : \textbf{map}(a : A, \\ \qquad b : \textbf{map}(c : C, \\ \qquad\qquad d : D)) \end{array} \quad \Rightarrow \quad \begin{array}{l} name : \textbf{map}(nameMapKey : \textbf{tuple}(a : A, \\ \qquad\qquad\qquad\qquad c : C), \\ \qquad\qquad\qquad d : D) \end{array} \tag{9.11}$$

6. Bijection (**Bijection**).

$$\begin{array}{l} name : \textbf{map}(a : A, \\ \qquad\qquad b : \textbf{bijection}(c : C, \\ \qquad\qquad\qquad\qquad d : D)) \\ \qquad\qquad \Downarrow \\ name : \textbf{map}(nameMapKey : \textbf{tuple}(a : A, \\ \qquad\qquad\qquad\qquad\qquad\qquad c : C), \\ \qquad\qquad\qquad d : D) \end{array} \tag{9.12}$$

La relation générée n'est pas en troisième forme normale de Boyce-Codd. En effet, la transformation fait perdre une dépendance fonctionnelle en dégradant la bijection en une application.

Il est cependant possible de conserver cette dépendance fonctionnelle au niveau SQL. En effet, une simple clef candidate sur (a, d) permet de simuler l'application inverse. Pour déclencher la création de cette clef, nous ajoutons simplement une indication (*hint*) à l'application générée.

Contrairement à la dégradation d'une bijection contenant une collection, la transformation en application peut se faire indifféremment en privilégiant **C** ou **D**. En effet, ayant transformé au préalable la portée, nous avons l'assurance que la bijection considérée est relationnelle-proche.

7. Famille (**Bag**).

$$\begin{array}{l} name : \textbf{map}(a : A, \\ \qquad\qquad b : \textbf{bag}([c :] C)) \end{array} \Rightarrow \begin{array}{r} name : \textbf{bag}(nameMember : \textbf{tuple}(a : A, \\ c : C)) \end{array} \tag{9.13}$$

8. Liste (**List**).

   Nous devons ici exposer la transformation standard de la liste car le principe de transformation utilisé pour la famille :
   - est trop peu intuitif : **list**(**map**(A, C)) ;
   - ne génère pas un type relationnel-proche.

   Nous obtenons donc :

$$\begin{array}{c} name : \textbf{map}(a : A, \\ \qquad\qquad b : \textbf{list}([c :] C)) \\ \Downarrow \\ name : \textbf{map}(nameMapKey : \textbf{tuple}(a : A, \\ \qquad\qquad\qquad\qquad index : \textbf{natural}), \\ \qquad\qquad c : C) \end{array} \tag{9.14}$$

9.2.3.5 Exemple de transformation en relationnel proche

Un premier exemple. Afin de mieux comprendre le mécanisme de transformation, déroulons-le sur l'exemple d'un arbre de classification de documents. Pour commencer, contentons-nous d'une version simpliste où chaque noeud est un simple ensemble de documents :

$$classtree : \textbf{tree}(node : \textbf{set}(doc : \text{DOCUMENT}))$$

La première étape est constituée de la transformation du type de plus haut niveau : le **tree**.

$$\begin{array}{c} classtree : \textbf{tree}(node : \textbf{set}(doc : \text{DOCUMENT})) \\ \Downarrow \\ \begin{array}{l} classtree : \textbf{tree}(nodeID : \textbf{surrogate}) \\ classtree\_node : \textbf{bijection}(nodeID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad node : \textbf{set}(doc : \text{DOCUMENT})) \end{array} \end{array} \tag{9.15}$$

Le second attribut généré n'est pas relationnel-proche, il ne s'agit que d'une réécriture qu'il va maintenant falloir transformer.

$$\begin{array}{l} classtree\_node : \textbf{bijection}(nodeID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad node : \textbf{set}(doc : \text{DOCUMENT})) \\ \Downarrow \\ classtree\_node : \textbf{map}(nodeID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad node : \textbf{set}(doc : \text{DOCUMENT})) \end{array} \tag{9.16}$$

Nous avons dégradé la **bijection** en **map** en utilisant de nouveau une réécriture. Il nous faut donc maintenant transformer cette nouvelle application.

$$\begin{array}{l} classtree\_node : \textbf{map}(nodeID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad node : \textbf{set}(doc : \text{DOCUMENT})) \\ \Downarrow \\ classtree\_node : \textbf{relation}(nodeID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad doc : \text{DOCUMENT}) \end{array} \tag{9.17}$$

Nous obtenons donc en résultat final :

$$\begin{array}{l} classtree : \textbf{tree}(node : \textbf{set}(doc : \text{DOCUMENT})) \\ \Downarrow \\ classtree : \textbf{tree}(nodeID : \textbf{surrogate}) \\ classtree\_node : \textbf{relation}(nodeID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad doc : \text{DOCUMENT}) \end{array} \tag{9.18}$$

Introduction d'un nouvel élément. Pour aller plus loin dans notre compréhension de l'algorithme de transformation, observons comment l'introduction d'un nouvel attribut peut influer sur son déroulement.

$$\begin{array}{l} classtree : \textbf{tree}(node : \textbf{tuple}(docs : \textbf{set}(doc : \text{DOCUMENT}), \\ \qquad\qquad\qquad\qquad query : \textbf{string})) \end{array}$$

Dans cet exemple, nous avons simplement ajouté une requête (*query*) à chaque noeud de l'arbre. Elle peut par exemple être utilisée par un processus de percolation des documents dans l'arbre afin de les classer dans le bon noeud.

La première étape est similaire à celle de l'exemple précédent. Elle est composée de la transformation du type de plus haut niveau : le **tree**.

$$\begin{array}{l} classtree : \textbf{tree}(node : \textbf{tuple}(docs : \textbf{set}(doc : \text{DOCUMENT}), \\ \qquad\qquad\qquad\qquad query : \textbf{string})) \\ \Downarrow \\ classtree : \textbf{tree}(nodeID : \textbf{surrogate}) \\ classtree\_node : \textbf{bijection}(nodeID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad node : \textbf{tuple}(docs : \textbf{set}(doc : \text{DOCUMENT}), \\ \qquad\qquad\qquad\qquad\qquad query : \textbf{string})) \end{array} \tag{9.19}$$

En revanche, la transformation de la **bijection** générée va complètement différer de l'exemple précédent. En effet, nous sommes ici en présence d'un codomaine qui n'est pas relationnel-proche et que l'on va donc commencer par transformer (cf. section 9.2.3.3).

Ce codomaine est lui-même un **tuple**. Aussi, comme nous l'avons vu en section 9.2.3.2, nous allons tout d'abord créer un délégué pour son premier attribut (le second étant un attribut simple, il n'est pas nécessaire de le transformer) :

$$\begin{array}{l} classtree\_node\_docs : \textbf{set}(doc : \text{DOCUMENT}) \\ \Downarrow \\ docsID : \textbf{surrogate} \\ classtree\_node\_docs : \textbf{bijection}(docsID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad docs : \textbf{set}(doc : \text{DOCUMENT})) \end{array} \tag{9.20}$$

Le premier attribut généré va nous permettre d'avoir un n-uplet relationnel-proche. Le second doit être ajouté au résultat de la transformation, mais ne peut cependant pas l'être en l'état. En effet, il n'est pas encore relationnel-proche.

$$\begin{array}{l} classtree\_node\_docs : \textbf{bijection}(docsID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad docs : \textbf{set}(doc : \text{DOCUMENT})) \\ \Downarrow \\ classtree\_node\_docs : \textbf{map}(docsID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad docs : \textbf{set}(doc : \text{DOCUMENT})) \end{array} \tag{9.21}$$

Après avoir dégradé la bijection en application, il nous faut encore la transformer :

$$\begin{array}{l} classtree\_node\_docs : \textbf{map}(docsID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad docs : \textbf{set}(doc : \text{DOCUMENT})) \\ \Downarrow \\ classtree\_node\_docs : \textbf{relation}(docsID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad doc : \text{DOCUMENT}) \end{array} \tag{9.22}$$

Cet attribut est maintenant relationnel-proche et ne nécessite plus de nouvelle transformation. Nous pouvons maintenant reprendre la transformation de la bijection initiale et la compléter en utilisant le délégué généré et ce nouvel attribut. Nous obtenons alors :

$$\begin{array}{l} classtree\_node : \textbf{bijection}(nodeID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad node : \textbf{tuple}(docs : \textbf{set}(doc : \text{DOCUMENT}), \\ \qquad\qquad\qquad\qquad\qquad\qquad query : \textbf{string})) \\ \Downarrow \\ classtree\_node : \textbf{bijection}(nodeID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad node : \textbf{tuple}(docsID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad\qquad\qquad query : \textbf{string})) \\ classtree\_node\_docs : \textbf{relation}(docsID : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad doc : \text{DOCUMENT}) \end{array} \tag{9.23}$$

La transformation est alors terminée. Nous obtenons le résultat final suivant :

$$
\begin{array}{l}
classtree : \textbf{tree}(node : \textbf{tuple}(docs : \textbf{set}(doc : \text{DOCUMENT}), \\
\qquad\qquad\qquad\qquad\qquad query : \textbf{string})) \\
\qquad\qquad\qquad\qquad\qquad \Downarrow \\
classtree : \textbf{tree}(nodeID : \textbf{surrogate}) \\
classtree\_node : \textbf{bijection}(nodeID : \textbf{surrogate}, \\
\qquad\qquad\qquad\qquad\qquad node : \textbf{tuple}(nodesID : \textbf{surrogate}, \\
\qquad\qquad\qquad\qquad\qquad\qquad query : \textbf{string})) \\
classtree\_node\_docs : \textbf{relation}(docsID : \textbf{surrogate}, \\
\qquad\qquad\qquad\qquad\qquad doc : \text{DOCUMENT})
\end{array}
\tag{9.24}
$$

Notons que dans un exemple réel, chaque noeud porterait beaucoup plus d'information (ex. : stratégie de percolation, similarité minimum, etc.). Il serait alors beaucoup plus simple de le réifier dès la conception :

$$classtree : \textbf{tree}(node : \text{CLASSNODE})$$

9.2.4 Transformation en relationnel pur

En sortie de la première étape, nous obtenons une liste d'attributs dont le type est relationnel-proche. La complexité inhérente à la richesse des types du modèle, principalement l'imbrication sans restriction de collections, a disparu. Nous allons maintenant pouvoir nous attacher à transformer ces attributs en véritables attributs relationnels.

Pour se rapprocher du monde relationnel, introduisons tout d'abord une définition *étendue* de ce qu'est un type relationnel.

Définition 7. *Un type est un* type relationnel basique *si c'est :*

- *un type atomique ;*
- *un n-uplet dont tous les attributs sont définis par un type atomique.*

Par extension, on dira d'un attribut qu'il est relationnel basique *quand il est défini par un* type relationnel basique.

Définition 8. *Un type est un* type relationnel *si c'est :*

- *un* type relationnel basique, *c'est-à-dire un singleton dans un univers ensembliste ;*
- *une relation dont tous les attributs sont* relationnels basiques *;*
- *une application ou une bijection d'un attribut / n-uplet* relationnel basique *vers un autre attribut / n-uplet* relationnel basique.

Par extension, on dira d'un attribut qu'il est relationnel *quand il est défini par un* type relationnel.

9.2.4.1 Des n-uplets plats

La définition 7 ci-dessus indique en d'autres mots que les n-uplets doivent être « plats » pour être relationnels basiques et donc relationnels. Transformer des n-uplets imbriqués en un seul n-uplet revient à simplement renommer ses attributs. Par exemple, de

$$
\begin{array}{l}
\textbf{tuple}(\ldots, \\
\qquad address : \textbf{tuple}(ZIP : \textbf{string}, \\
\qquad\qquad\qquad\qquad city : \textbf{string}) \\
\qquad \ldots)
\end{array}
$$

nous pouvons dériver

$$\begin{array}{l}\textbf{tuple}(\ldots,\\ \quad addressZIP : \textbf{string},\\ \quad addressCity : \textbf{string},\\ \quad \ldots)\end{array}$$

Ainsi, dans la suite des transformations en relationnel, nous considérerons tous les n-uplets comme relationnels basiques. En présence, de n-uplets imbriqués, leur transformation en un seul n-uplet « plat » est considérée comme automatique.

Nous ne traiterons donc pas de la transformation de l'application, de la bijection et de la relation. En effet, la seule action éventuellement nécessaire pour rendre ces types relationnels est « l'aplatissement » de leurs attributs de type n-**uplet**.

9.2.4.2 Transformation des collections

Nous devons transformer les collections non standards en collections relationnelles, c'est-à-dire **map**, **bijection** et **relation**. Étant relationnelles-proches, leur domaine est forcément relationnel-proche basique (cf. définition 6).

Ensemble (set). Dans le modèle relationnel, la notion d'ensemble est presque synonyme de celle de relation, sa transformation est donc directe :

$$\begin{array}{c} name : \textbf{set}([tupleName :] \textbf{tuple}(a_1 : T_1, a_2 : T_2, \ldots, a_n : T_n)) \\ \Downarrow \\ name : \textbf{relation}(a_1 : T_1, a_2 : T_2, \ldots, a_n : T_n) \end{array} \tag{9.25}$$

Dans le cas où le domaine de l'ensemble à transformer n'est qu'un attribut atomique, on considérera ce dernier équivalent à un n-uplet à une valeur :

$$name : \textbf{set}([a :] T) \quad \Rightarrow \quad name : \textbf{relation}(a : T) \tag{9.26}$$

Famille et Liste (bag et list). Nous utilisons ici des transformations similaires à celles que nous avons exposées dans le cas relationnel-proche.

$$name : \textbf{bag}([a :] T) \quad \Rightarrow \quad \begin{array}{l} name : \textbf{map}(a : T,\\ \qquad\qquad count : \textbf{integer}) \end{array} \tag{9.27}$$

$$name : \textbf{list}([a :] T) \quad \Rightarrow \quad \begin{array}{l} name : \textbf{map}(index : \textbf{integer},\\ \qquad\qquad a : T) \end{array} \tag{9.28}$$

Arbre (tree). Ce type est déjà plus complexe et deux solutions s'offrent à nous en fonction de son domaine. Nous pouvons utiliser :

1. Une seule relation quand la taille du domaine est petite.

$$name : \textbf{tree}([a :] T) \quad \Rightarrow \quad \begin{array}{l} name : \textbf{map}(a : T,\\ \qquad\qquad aFather : T) \end{array} \tag{9.29}$$

Nous devons alors autoriser la racine de l'arbre à avoir un père NULL.

2. Deux relations, dont une sert de représentante, dans le cas contraire.

$$name : \textbf{tree}([a :] T) \quad \Rightarrow \quad \begin{array}{l} name : \textbf{map}(aID : \textbf{surrogate}, \\ \qquad\qquad aFatherID : \textbf{surrogate}) \\ nameValues : \textbf{bijection}(aID : \textbf{surrogate}, \\ \qquad\qquad a : T) \end{array} \tag{9.30}$$

La racine de l'arbre peut ici être déterminée comme étant la valeur n'ayant pas de père. Il n'est pas nécessaire de créer un enregistrement dans la première relation dont le père est NULL.

Le choix de la solution la plus appropriée dépend de ses besoins en espace, qui eux-mêmes dépendent de la taille de l'arbre exprimée par n, le nombre de noeuds. L'espace requis par chaque solution est :

1. $2 \times sizeof(T) \times n$,
2. $(sizeof(T) + sizeof(\textbf{surrogate})) \times n + 2 \times sizeof(\textbf{surrogate}) \times (n-1)$

Ainsi,

$$\begin{aligned} (1) \geq (2) \quad &\Leftrightarrow \quad 2n \times sizeof(T) \geq n \times sizeof(T) + 3n \times sizeof(\textbf{surrogate}) - 2 \times sizeof(\textbf{surrogate}) \\ &\Leftrightarrow \quad n \times sizeof(T) \geq 3n \times sizeof(\textbf{surrogate}) - 2 \times sizeof(\textbf{surrogate}) \\ &\Rightarrow \quad sizeof(T) \geq 3 \times sizeof(\textbf{surrogate}) - \circ(1) \end{aligned}$$

Si par exemple $sizeof(\textbf{surrogate}) = 4$ et $T = \textbf{string}(5)$, c'est-à-dire $sizeof(T) = 6$, la solution (1) est privilégiée car elle consomme moins d'espace.

Pour faciliter certaines requêtes, il est souvent utile de matérialiser la fermeture de l'arbre dans une relation supplémentaire. Sa création est donc le comportement choisi par défaut. Le concepteur peut cependant décider de ne pas la créer.

La création de la fermeture est subordonnée aux choix précédents. On obtient ainsi selon le cas :

1. $$\begin{array}{l} nameClosure : \textbf{relation}(a : T, \\ \qquad\qquad aDescendant : T) \end{array}$$

2. $$\begin{array}{l} nameClosure : \textbf{relation}(aID : \textbf{surrogate}, \\ \qquad\qquad aDescendant : \textbf{surrogate}) \end{array}$$

L'espace requis par cette relation additionnelle est : $2 \times sizeof(T/\textbf{surrogate}) \times p$ où p est le nombre d'éléments dans la fermeture : $n - 1 \leq p \leq \frac{n^2-n}{2}$ $\left(= \sum_{i=1}^{n-1} i\right)$.

La valeur de p est très difficile à estimer. Prenons le cas extrême où $p \to n$. Alors,

$$\begin{aligned} (1) \geq (2) \quad &\Leftrightarrow \quad 3 \times sizeof(T) \geq 5 \times sizeof(\textbf{surrogate}) - \circ(1) \\ &\Leftrightarrow \quad sizeof(T) \geq \frac{5}{3} \times sizeof(\textbf{surrogate}) - \circ(1) \end{aligned}$$

Dans le cas plus acceptable où $p \to 3n$,

$$\begin{aligned} (1) \geq (2) \quad &\Leftrightarrow \quad 7 \times sizeof(T) \geq 9 \times sizeof(\textbf{surrogate}) - \circ(1) \\ &\Leftrightarrow \quad sizeof(T) \geq \frac{9}{7} \times sizeof(\textbf{surrogate}) - \circ(1) \end{aligned}$$

Dans le but de simplifier les relations générées lorsqu'une relation de fermeture est créée, et en l'absence d'indications sur les valeurs probables de n et p, nous choisissons la transformation (2) dès lors que $sizeof(T) > sizeof(\textbf{surrogate})$.

Graphe acyclique orienté (dag). La transformation des graphes acycliques orientés est très similaire à la précédente. De même que pour les arbres, deux solutions s'offrent à nous en fonction du domaine du **dag** :

1. Une seule relation quand la taille du domaine est petite.

$$name : \textbf{dag}([a :] T) \quad \Rightarrow \quad \begin{array}{r} name : \textbf{relation}(aPredecessor : T, \\ aSuccessor : T) \end{array} \tag{9.31}$$

   Cette première transformation se contente de matérialiser les arcs du **dag** dans une relation. Un sommet intervenant dans plusieurs arcs sera donc présent plusieurs fois dans la relation.

2. Deux relations, dont une sert de représentante, dans le cas contraire.

$$name : \textbf{dag}([a :] T) \quad \Rightarrow \quad \begin{array}{r} name : \textbf{relation}(aPredecessor : \textbf{surrogate}, \\ aSuccessor : \textbf{surrogate}) \\ nameValues : \textbf{bijection}(aID : \textbf{surrogate}, \\ a : T) \end{array} \tag{9.32}$$

   L'utilisation de délégués évite de répéter plusieurs fois les sommets dans la relation principale. Cette seconde transformation est donc plus appropriée dans le cas où le domaine du **dag** est volumineux, c'est-à-dire principalement quand il s'agit d'un n-uplets ou d'une – longue – chaîne de caractères.

Ici aussi, le choix de la solution la plus appropriée dépend de ses besoins en espace, qui eux-mêmes dépendent de la taille du **dag** exprimée par n, le nombre de sommets, et m, le nombre d'arcs :

1. $2 \times sizeof(T) \times m$,
2. $2 \times sizeof(\textbf{surrogate}) \times m + (sizeof(T) + sizeof(\textbf{surrogate})) \times n$

Le graphe étant acyclique, on a :

$$\frac{n}{2} \leq m \leq \frac{n^2 - n}{2} \quad \left(= \sum_{i=1}^{n-1} i \right)$$

Ainsi,

$$\begin{aligned} (1) \geq (2) \quad &\Leftrightarrow \quad 2m \times sizeof(T) \geq n \times sizeof(T) + (2m + n) \times sizeof(\textbf{surrogate}) \\ &\Leftrightarrow \quad sizeof(T) \geq k \times sizeof(\textbf{surrogate}) \end{aligned}$$

où

$$k = \frac{2m + n}{2m - n}$$

Observons les valeurs de k en fonction de la quantité d'arcs dans le **dag** :

- S'ils ne sont que peu nombreux ($m \to \frac{n}{2}$), alors $k \to \infty$ et (1) consomme toujours moins d'espace (dès lors que l'on est pas dans le cas où n est très petit).
- Si au contraire il existe de nombreux arcs ($m \to \frac{n^2-n}{2}$), alors $k \to \frac{n^2}{n^2-2n}$. Dans ce cas, $\forall n \in [3, \infty[, k \in]1, 3]$.

En l'absence d'indications sur les valeurs de n et m, nous prenons l'hypothèse que $n \simeq m$ (distribution proche d'un arbre). Dans ce cas :

$$(1) \geq (2) \Leftrightarrow sizeof(T) \geq 3 \times sizeof(\textbf{surrogate})$$

Si par exemple $sizeof(\textbf{surrogate}) = 4$ et $T = \textbf{string}(5)$, c'est-à-dire $sizeof(T) = 6$, la solution (1) est privilégiée car elle consomme moins d'espace.

REMARQUE : Nous considérons que les sommets n'ont pas d'existence propre en dehors du contexte du **dag**. Si c'était le cas, la seconde solution serait obligatoire pour être capable de les référencer. Toutefois, ce cas est traité au niveau déclaratif en utilisant des références.

Comme pour les arbres, la matérialisation de la fermeture du **dag** dans une relation supplémentaire peut faciliter de nombreuses requêtes. De plus, elle peut améliorer les opérations de mise à jour en permettant une détection rapide des cycles. Sa création est donc le comportement choisi par défaut.

La création de la fermeture est subordonnée aux choix précédents. On obtient ainsi :

1. $nameClosure$: **relation**($a : T$, $aDescendant : T$)

2. $nameClosure$: **relation**(aID : **surrogate**, $aDescendant$: **surrogate**)

L'espace requis par cette relation additionnelle est : $2 \times sizeof(T/\textbf{surrogate}) \times p$ où p est le nombre d'éléments dans la fermeture ($m \leq p \leq \frac{n^2-n}{2}$).

Ici aussi, la valeur de p est très difficile à estimer. Prenons le cas extrême où $p \rightarrow m$, avec l'hypothèse précédente $n \simeq m$ (cas impossible car la profondeur du **dag** est alors forcément supérieure à 1). Alors,

$$\begin{aligned}(1) \geq (2) &\Leftrightarrow 3 \times sizeof(T) \geq 5 \times sizeof(\textbf{surrogate}) \\ &\Leftrightarrow sizeof(T) \geq \frac{5}{3} \times sizeof(\textbf{surrogate})\end{aligned}$$

Dans le cas plus acceptable où $p \rightarrow 3m$,

$$\begin{aligned}(1) \geq (2) &\Leftrightarrow 7 \times sizeof(T) \geq 9 \times sizeof(\textbf{surrogate}) \\ &\Leftrightarrow sizeof(T) \geq \frac{9}{7} \times sizeof(\textbf{surrogate})\end{aligned}$$

Dans le but de simplifier les relations générées lorsqu'une relation de fermeture est créée, et en l'absence d'indications sur les valeurs probables de n, m, et p, nous choisissons la transformation (2) dès lors que $sizeof(T) > sizeof(\textbf{surrogate})$.

Comme les besoins en espace de la fermeture peuvent être très importants, le concepteur a la possibilité de ne pas la créer, et de se reposer alors sur un mécanisme, moins efficace, basé sur le rang des sommets pour s'assurer de la non cyclicité du **dag**.

Graphe (graph). Comme nous avons pu l'observer ci-dessus, les résultats de la transformation du **tree** et du **dag** sont très proches. Le cas du **graph**, non acyclique et non orienté, est lui aussi similaire. Nous ne nous attarderons donc pas sur les détails.

Ici aussi, deux solutions existent en fonction de la taille du domaine :

1. Une seule relation quand la taille du domaine est petite.

$$name : \textbf{graph}([a :] T) \quad \Rightarrow \quad \begin{array}{r} name : \textbf{relation}(aEdge1 : T, \\ aEdge2 : T) \end{array} \tag{9.33}$$

2. Deux relations, dont une sert de représentante, dans le cas contraire.

$$name : \textbf{graph}([a :] T) \quad \Rightarrow \quad \begin{array}{r} name : \textbf{relation}(aEdge1 : \textbf{surrogate}, \\ aEdge2 : \textbf{surrogate}) \\ nameValues : \textbf{bijection}(aID : \textbf{surrogate}, \\ a : T) \end{array} \tag{9.34}$$

Le graphe n'étant pas orienté, l'ordre des sommets dans la relation n'a pas d'importance. Pour faciliter les requêtes SQL, une vue est automatiquement créée et présente la liste de tous les sommets partageant un arc avec un sommet donné. Les opérations de mises à jour doivent, de leur côté, s'assurer de l'unicité des arcs dans la relation.

En suivant le même calcul que pour le **dag**, on obtient (avec n le nombre de sommets et m le nombre d'arcs) :

$$(1) \geq (2) \Leftrightarrow sizeof(T) \geq k \times sizeof(\textbf{surrogate})$$

où

$$k = \frac{2m + n}{2m - n}$$

En l'absence d'indications sur les valeurs de n et m, nous prenons l'hypothèse que $n \simeq m$ et dans ce cas :

$$(1) \geq (2) \Leftrightarrow sizeof(T) \geq 3 \times sizeof(\textbf{surrogate})$$

Contrairement aux graphes acycliques orientés et aux arbres, la matérialisation d'une fermeture est ici moins utile et n'est pas faite par défaut. Dans le cas où le concepteur souhaite cependant créer une relation contenant l'ensemble des sommets atteignables, nous obtenons les même conclusions que pour le **dag** : en l'absence d'indications, nous choisissons la transformation (2) dès lors que $sizeof(T) > sizeof(\textbf{surrogate})$.

Au besoin, la transformation de nouveaux types à base de graphes, par exemple un graphe orienté acceptant les cycles, suivrait le même principe.

9.2.4.3 Exemple de transformation en relationnel pur

La transformation en relationnel pur est beaucoup plus simple que celle en relationnel proche. En effet, il n'est plus possible pour des collections d'être imbriquées.

Illustrons cependant cette étape avec un exemple simple : un graphe acyclique orienté de droits. Ce graphe peut en particulier servir à organiser les droits d'accès au système comme l'illustre l'annexe C.

$$rigths : \textbf{dag}(right : \text{RIGHT})$$

Une application pouvant nécessiter de vérifier très fréquemment des droits d'accès, cette opération doit être très réactive. Nous choisissons donc de créer une fermeture pour le graphe afin de pouvoir vérifier rapidement si le droit d'un utilisateur implique le droit demandé par l'application.

Le domaine du graphe correspond à une référence sur une facette. Imaginons que :

$$sizeof(\text{RIGHT})^5 = sizeof(\textbf{surrogate})^6$$

Dans ce cas, nous avons la transformation suivante en relationnel pur :

$$\begin{gathered} rigths : \textbf{dag}(right : \text{RIGHT}) \\ \Downarrow \\ \begin{aligned} rigths : \textbf{relation}(&rigthPredecessor : \text{RIGHT}, \\ &rigthSuccessor : \text{RIGHT}) \\ rigthsClosure : \textbf{relation}(&rigth : \text{RIGHT}, \\ &rigthDescendant : \text{RIGHT}) \end{aligned} \end{gathered} \tag{9.35}$$

9.2.5 Transformation en SQL

Nous sommes maintenant en présence d'attributs qui sont tous relationnels. Dans le but de profiter des spécificités de chaque SGDBR, nous introduisons une dernière étape qui va générer les instructions SQL finales. Nous voulons créer deux sortes d'éléments dans la base de données : des *tables* pour représenter les attributs et des *procédures stockées* pour implémenter les opérations nécessaires aux mises à jour de leurs données[7].

Les scripts SQL résultant du processus de traduction sont créés grâce à un langage de patrons (ex. : FreeMarker [46] ou Velocity [3]). Chaque type d'attribut et chaque procédure stockée correspond à un patron dédié (parfois plus quand plusieurs implémentations coexistent, comme par exemple pour le **dag**). Ainsi, des SGBDR spécifiques peuvent être ciblés en fournissant l'ensemble des patrons correspondant à leurs spécificités, en particulier leur langage natif.

9.2.5.1 Tables

Comme indiqué dans la définition 8, les tables générées peuvent être des singletons, c'est-à-dire des variables globales, des applications, des bijections ou des relations.

Les relations sont manifestement directement transposables dans le modèle relationnel standard, comme implémenté par les SGBDR. Les applications sont également très faciles à transformer en tables. À la différence des relations, il suffit de faire porter la clef primaire uniquement sur leur domaine. Les bijections subissent presque la même transformation que les applications : nous leur ajoutons simplement une clef candidate sur leur codomaine.

Dans le cas d'un singleton, c'est-à-dire un attribut dont le type est atomique ou un n-uplet, nous insérons simplement une clef de type quelconque que nous restreignons à une valeur unique qui est aussi la valeur par défaut :

$$\begin{gathered} name : \textbf{tuple}(a_1 : T_1, a_2 : T_2, \ldots, a_n : T_n) \\ \Downarrow \\ \begin{aligned} name : \textbf{map}(&key : \textbf{integer} = 1, \\ &\textbf{tuple}(a_1 : T_1, a_2 : T_2, \ldots, a_n : T_n)) \end{aligned} \end{gathered} \tag{9.36}$$

[5] Par convention de notation (voir section 8.3.2), $\text{RIGHT} \equiv \textbf{ref}(\text{RIGHT})$.

[6] Cas général où représentants et références utilisent des entiers stockés sur 32 bits.

[7] Des *vues* peuvent aussi être générées pour simplifier certaines requêtes (voir par exemple la transformation des graphes en section 9.2.4.2).

`set`	Remplace les valeurs du singleton par celles spécifiées.
`setXXX`	Remplace la valeur de l'attribut `XXX` par la valeur spécifiée. Il existe une fonction `setXXX` pour chaque attribut contenu dans le singleton.

Table 9.1 – Procédures stockées : Singleton

`add`	Ajoute un nouvel élément dans l'ensemble ou la relation. Si l'élément est déjà présent dans la collection, une erreur est retournée.
`remove`	Supprime l'élément spécifié de l'ensemble ou de la relation. Si un tel élément n'est pas présent dans la collection, la procédure termine silencieusement.
`clear`	Supprime tous les éléments de l'ensemble ou de la relation.

Table 9.2 – Procédures stockées : Ensemble et Relation

Il nous reste un dernier choix à effectuer concernant l'implémentation des types atomiques. Nous utilisons alors les types les plus appropriés de chaque SGBDR. Afin de représenter des entiers de taille réduite, nous nous appuierons, par exemple, sur le type **smallint** avec SQL Server, alors que nous utiliserons le type **number(2,0)** avec ORACLE[8].

9.2.5.2 Procédures stockées

Comme nous venons de le voir, chaque attribut relationnel donne lieu à la création d'une table. Ce n'est cependant pas aussi direct dans le cas des procédures stockées. En effet, elles correspondent aux opérations applicables aux types de haut niveau (**list**, **bag**, **dag**, etc.) et non uniquement aux types relationnels (**relation**, **map**, **bijection** et singleton). De plus, comme nous le verrons avec l'exemple du **dag** en Section 9.2.5.3, certaines opérations peuvent agir sur plusieurs tables à la fois, c'est-à-dire sur plusieurs attributs relationnels.

Nous nous baserons donc sur les attributs relationnels-proches et non relationnels pour créer les procédures stockées correspondant à leurs opérations.

Désignation des procédures. Comme nous l'avons vu au début de la section 9.2, les attributs sont désignés en utilisant un système de préfixes. La désignation des procédures suit le même principe : leur nom est préfixé par le nom de l'attribut auquel elles correspondent. Par exemple, si la définition d'un auteur contient un ensemble de publications représenté par un attribut $auteur\_publications$, nous utiliserons ce nom comme préfixe des opérations s'y appliquant (ex. : $auteur\_publications\_add$, $auteur\_publications\_remove$, etc.).

Liste des procédures. Les tables 9.1 à 9.8 décrivent les opérations créées pour chaque type relationnel-proche. Le singleton correspond à un n-uplet ou un attribut atomique, c'est-à-dire un n-uplet ne contenant qu'un attribut.

Cas particulier des représentants (surrogates). Dans le cas où l'on travaille avec des collections de collections, les collections imbriquées sont extraites dans un attribut externe mais restent dépendantes

[8] Dans ORACLE, le type **smallint** existe en tant que synonyme, mais pas comme une implémentation alternative.

`set`	Remplace la valeur correspondant à un domaine donné par celle spécifiée. Si le domaine n'existe pas, il est créé.
`remove`	Supprime l'élément correspondant au domaine spécifié de l'application. Si l'application n'a pas de valeur pour ce domaine, la procédure termine silencieusement.
`clear`	Supprime tous les éléments de l'application.

Table 9.3 – Procédures stockées : Application

`add`	Ajoute une bijection entre les valeurs spécifiées. Si une telle bijection existe déjà, ou si le domaine ou le codomaine est déjà utilisé par une bijection existante, une erreur est retournée.
`setCodomain`	Remplace la valeur du codomaine correspondant à un domaine donné par celle spécifiée. Si le domaine n'existe pas, il est créé. Si la nouvelle valeur est utilisée par une autre bijection, une erreur est retournée.
`setDomain`	Remplace la valeur du domaine correspondant à un codomaine donné par celle spécifiée. Si le codomaine n'existe pas, il est créé. Si la nouvelle valeur est utilisée par une autre bijection, une erreur est retournée.
`remove`	Supprime la bijection existant entre les valeurs spécifiées. Si une telle bijection n'existe pas, la procédure termine silencieusement.
`clear`	Supprime tous les éléments de la bijection.

Table 9.4 – Procédures stockées : Bijection

`add`	Ajoute l'élément spécifié à la famille. Le même élément peut être ajouté plusieurs fois, le nombre d'occurrences est alors tenu à jour.
`addFirst`	Ajoute l'élément spécifié à la famille. Cette procédure n'est créé que lorsque la famille référence implicitement une autre collection (par exemple lorsqu'elle résulte de la transformation d'un **bag**(**set**(**string**))). Dans ce cas, un **surrogate** doit être généré en utilisant `addFirst`, puis être utilisé avec `add` lors des ajouts suivants.
`remove`	Supprime l'élément spécifié de la famille. Si l'élément est présent plusieurs fois, seule une des occurrences est supprimée. Si l'élément n'est pas présent, la procédure termine silencieusement.
`removeAllOccurrences`	Supprime toutes les occurrences de l'élément spécifié dans la famille. Si l'élément n'est pas présent, la procédure termine silencieusement.
`clear`	Supprime tous les éléments de la famille.

Table 9.5 – Procédures stockées : Famille

`add`	Ajoute l'élément spécifié à la fin de la liste.
`insert`	Insère l'élément spécifié à un emplacement donné. L'élément qui était déjà présent à cet index et tous les éléments suivants sont « déplacés » vers la droite : leur index est incrémenté de 1. Si l'index donné est au delà des limites actuelles de la liste, l'élément est ajouté en fin de liste.
`remove`	Supprime l'élément présent à l'index spécifié. L'élément qui était déjà présent à cet index et tous les éléments suivants sont « déplacés » vers la gauche : leur index est décrémenté de 1. Si l'index donné est au delà des limites actuelles de la liste, et donc si aucun élément n'est associé à cet index, une erreur est retournée.
`clear`	Supprime tous les éléments de la liste.
`shiftLeft` et `shiftRight`	Ces deux procédures sont « privées ». Elles sont utilisées par les autres pour effectuer leurs mises à jour. Elles ne doivent en aucun cas être appelées directement.

Table 9.6 – Procédures stockées : Liste

`addChild`	Ajoute le lien père-fils spécifié dans l'arbre. Le noeud père spécifié doit déjà faire partie de l'arbre mais pas son nouveau fils. Toute autre configuration casserait la structure de l'arbre et renvoie donc une erreur.
`setFather`	Remplace le père d'un élément par celui spécifié. Toute une branche de l'arbre peut ainsi être « déplacée ». Le nouveau père doit déjà appartenir à l'arbre et ne pas faire partie de la branche déplacée, à moins que le noeud dont on remplace le père ne soit la racine. Dans ce cas, le nouveau père devient la racine de l'arbre.
`removeChild`	Supprime le lien père-fils spécifié de l'arbre. Si le noeud fils a lui même des fils, ces derniers sont supprimés récursivement. Si le lien père-fils spécifié n'existe pas dans l'arbre, la procédure termine silencieusement.
`removeAllChildren`	Supprime récursivement tous les fils d'un élément donné. Si l'élément n'avait pas de fils ou s'il n'appartient pas à l'arbre, la procédure termine silencieusement.
`clear`	Supprime tous les éléments de l'arbre.

Table 9.7 – Procédures stockées : Arbre

`add`	Ajoute un arc dans le graphe. Dans le cas d'un graphe acyclique, vérifie la présence d'un cycle et renvoie une erreur le cas échéant.
`remove`	Supprime l'arc du graphe. Si l'arc spécifié n'existe pas dans l'arbre, la procédure termine silencieusement.
`removePredecessors`	Dans le cas d'un graphe orienté uniquement, supprime les arcs entre l'élément spécifié et ses prédécesseurs. Si de tels arcs n'existent pas, la procédure termine silencieusement.
`removeSuccessors`	Dans le cas d'un graphe orienté uniquement, supprime les arcs entre l'élément spécifié et ses successeurs. Si de tels arcs n'existent pas, la procédure termine silencieusement.
`transferAll`	Supprime tous les arcs impliquant l'élément spécifié et crée des arcs entre tous les éléments auparavant en relation avec cet élément (s'ils n'existaient pas déjà). Si le graphe est orienté, les arcs créés le sont entre ses prédécesseurs et ses successeurs. Si aucun arc n'implique l'élément spécifié, la procédure termine silencieusement.
`clear`	Supprime tous les éléments du graphe.

Table 9.8 – Procédures stockées : Graphes

de leur collection englobante. Prenons l'exemple d'une liste de tirages du loto définie par $tirages$: **list**($tirage$: **list**($numéro$: **natural**)). La suppression d'un tirage de la première liste doit entraîner la suppression des sept numéros correspondant à ce tirage.

Pour conserver et exploiter le lien unissant ces deux collections, nous nous appuyons sur le type **surrogate**. Les **surrogate**s vont en effet par paire et nous conservons toujours le lien unissant celui référençant la collection externe et celui la représentant.

Ce lien a deux impacts importants sur les procédures générées :

- Lorsqu'un attribut correspond à une collection imbriquée, il contient systématiquement un **surrogate** qui sera inclus dans la clef de la relation. L'opération `clear` de cette collection prendra alors ce **surrogate** en tant que paramètre : elle ne supprimera que les éléments d'une collection particulière (identifiée par le **surrogate**).
- Lorsqu'un attribut référence une collection de manière implicite *via* un **surrogate**, les opérations entraînant la suppression de cette collection (e.g., `remove`, `clear`, mais aussi `set`), s'appuient sur l'opération `clear` de la collection imbriquée en lui fournissant la valeur du **surrogate** en paramètre. Cet appel peut récursivement en appeler d'autres quand plusieurs collections sont imbriquées.

Dans la table représentant l'attribut englobant, le **surrogate** est matérialisé dans un champ pour lequel une valeur unique est générée automatiquement à chaque création de nouvel enregistrement[9]. C'est cette valeur qui participera ensuite à la clef de la table représentant la collection imbriquée. Comme indiqué en section 9.2.2, une collection est donc toujours créée vide. Une fois le **surrogate** généré, il faut appeler d'autres opérations pour la remplir.

Pour éviter au programmeur de faire un `SELECT` supplémentaire pour obtenir le **surrogate** qui lui permettra ensuite de remplir la collection imbriquée, ce dernier est systématiquement retourné par la

[9] Ceci peut être réalisé grâce à la propriété `IDENTITY` sous SQL Server ou une `SEQUENCE` sous ORACLE.

procédure qui provoque sa création. Dans le cas où un attribut contient plusieurs collections imbriquées, un **surrogate** est retourné pour chacune d'entre elles.

Outre les collections imbriquées, il existe un autre cas où nous utilisons des représentants implicites : lorsque la répétition des valeurs est trop consommatrice en espace. Ce cas peut arriver lorsqu'on utilise un type à base de graphe. Dans ce cas, on doit en effet stocker les liens unissant des sommets. Ces sommets pouvant intervenir dans plusieurs liens, ils doivent être stockés plusieurs fois dans la table. Ainsi, dès lors que la taille requise par un sommet dépasse un certain seuil, il est plus intéressant de les stocker dans une table séparée et d'utiliser un **surrogate** pour les représenter dans le graphe principal (voir par exemple les transformations (9.31) et (9.32) concernant le **dag**).

Contrairement à l'introduction de représentants pour les collections imbriquées, ce cas n'a cependant aucun impact sur la signature des procédures stockées générées, seulement sur leur fonctionnement interne. Pour créer un lien, le programmeur appellera la procédure avec la description complète des deux sommets. La procédure récupérera alors les représentants de ces sommets à partir de l'application dédiée, ou les créera s'ils n'existent pas encore, avant de les utiliser dans la table principale. Si la définition des sommets est trop complexe pour une utilisation simple, il revient au concepteur de les réifier dans un nouveau type de haut niveau.

Limitations sur la duplication des valeurs. Le processus de transformation se veut le plus général possible, mais au-delà des restrictions déjà mentionnées au niveau de la conception des types, il existe quelques limitations concernant les collections de collections acceptant des valeurs dupliquées.

Observons de nouveau l'exemple du loto : $tirages$: **list**($tirage$: **list**($numéro$: **natural**)). La traduction en relationnel va créer deux tables, $tirages$ et $tirages\_tirage$ ainsi que les opérations correspondant aux listes. Lorsque le programmeur appellera $tirages\_add$ afin de créer un nouveau tirage, un **surrogate** sera généré afin de pouvoir ensuite manipuler le nouveau tirage (initialement vide) et en particulier lui ajouter des valeurs. Cette création est systématique.

En conséquence, si le programmeur veut créer deux listes identiques, il appellera deux fois $tirages\_add$ et aura donc un **surrogate** différent pour chaque liste. Il est donc impossible d'avoir deux fois la même collection dans la liste, à moins de s'assurer manuellement de l'égalité des valeurs. Si cette limitation est bloquante, la seule solution est de réifier la collection imbriquée explicitement afin de pouvoir utiliser une **reference**.

Ce problème se pose à l'identique pour les familles (**bag**s). Cependant, le fait que le concepteur préfère utiliser une famille à un ensemble montre expressément son besoin d'éléments dupliqués[10]. Nous procédons donc différemment. Comme indiqué dans la table 9.5, nous créons alors une opération `addFirst` qui génére un **surrogate** alors que l'opération standard `add` requiert le **surrogate** en paramètre.

9.2.5.3 Exemple du dag

Afin d'illustrer la création des scripts SQL, observons l'exemple d'un type complexe : le graphe acyclique orienté (**dag**).

Comme nous l'avons vu dans la section 9.2.5.2, nous nous situons au niveau relationnel-proche pour commencer la génération des scripts SQL. Agissant après la transformation en relationnel, nous disposons des attributs relationnels qui ont été générés pour représenter le **dag** avec lequel on travaille. Ceux-ci peuvent être jusqu'au nombre de trois :

[10] Rappelons que l'autorisation ou non d'éléments dupliqués est la seule différence entre un ensemble et une famille.

- La relation principale, toujours présente ;
- Éventuellement une application contenant les sommets si ceux-ci prennent beaucoup d'espace ;
- Éventuellement la fermeture du graphe.

Nous allons commencer par nous intéresser au cas le plus courant, c'est-à-dire un **dag** qui n'a pas d'attribut dédié à ses sommets et qui s'appuie sur une fermeture pour détecter les cycles.

Création des tables. Les premiers scripts à écrire sont ceux des tables correspondant aux attributs relationnels. Partant au départ d'un attribut relationnel-proche $name$: **dag**$(a : T)$, nous allons obtenir deux relations qui suivent la même structure : un prédécesseur et un successeur, tous deux de type T.

Il existe un cas particulier que nous n'avons pas abordé dans la transformation en relationnel. Si le **dag** est imbriqué dans un autre attribut, nous obtenons au final, *via* notamment les transformations (9.4), (9.6) et (9.9), un attribut relationnel-proche de la forme :

$$\begin{array}{l} name : \textbf{dag}(\textbf{tuple}(id : \textbf{surrogate}, \\ \qquad\qquad\qquad a : T)) \end{array}$$

Si la transformation en relationnel de cet attribut est correcte, nous remarquons très vite qu'elle n'est pas optimale. En effet, le **surrogate** va être dédoublé dans les relations résultantes (il sera alors présent à la fois dans le prédécesseur et le successeur). Pour éviter cette situation, nous allons exploiter les propriétés du **surrogate** pour nous en servir uniquement comme identifiant de la collection et éviter qu'il ne prenne part à ses éléments. Nous obtenons ainsi :

$$\begin{array}{l} name : \textbf{dag}(\textbf{tuple}(id : \textbf{surrogate}, \\ \qquad\qquad\qquad a : T)) \end{array} \quad \Rightarrow \quad \begin{array}{l} name : \textbf{relation}(id : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad aPredecessor : T, \\ \qquad\qquad\qquad\qquad aSuccessor : T) \\ nameClosure : \textbf{relation}(id : \textbf{surrogate}, \\ \qquad\qquad\qquad\qquad\qquad aPredecessor : T, \\ \qquad\qquad\qquad\qquad\qquad aSuccessor : T) \end{array} \tag{9.37}$$

La même optimisation est possible pour tous les types à base de graphe, c'est-à-dire le **graph**, le **tree** et le **dag**.

Création des index. Comme nous allons le voir un peu plus loin, les opérations de mise à jour du **dag** vont exploiter les arcs dans les deux sens. Pour optimiser leurs performances, nous allons donc créer des index supplémentaires sur les tables.

En plus de leur clefs primaires, les relations vont recevoir chacune un index inverse permettant d'obtenir rapidement les prédécesseurs de n'importe quel sommet. Le cas échéant, ces index uniques incluront le **surrogate** identifiant la collection.

Détection des cycles. La disponibilité de la fermeture simplifie grandement cette tâche. Lors de l'ajout d'un nouvel arc dans le graphe, il nous suffit en effet de vérifier si le prédécesseur ne fait pas partie de la fermeture du successeur. Si tel était le cas, la création du nouvel arc provoquerait un cycle et serait donc refusée.

Mise à jour de la fermeture. Si la fermeture réduit grandement les temps d'insertion et de nombreuses requêtes, il est toutefois compliqué de la maintenir à jour. Sa création après chaque modification (ou lot de modifications) du graphe étant trop coûteuse, nous privilégions une approche incrémentale. Les procédures d'ajout et de suppression d'arcs dans le graphe mettent donc à jour à la fois la relation principale et la fermeture. Nous nous appuyons pour cela sur un algorithme dérivé de la proposition de Dong *et al.* [36].

Extraction des sommets dans un attribut. Lorsque le type des sommets du **dag** est trop volumineux, on extrait ces sommets de l'attribut principal pour les stocker dans une application dédiée. Ils seront représentés dans la relation principale par un identifiant de type **surrogate**.

Cette différence d'implémentation n'a cependant aucun effet sur la signature des opérations. Le programmeur manipule toujours le type d'origine (il doit néanmoins faire une jointure pour les sélections). Charge à la procédure stockée de chercher dans la table des sommets le **surrogate** correspondant à une valeur, et de le créer si la valeur n'existe pas encore dans le graphe. Pour permettre une obtention rapide des ces représentants, nous créons par défaut un index inverse sur l'application contenant les sommets.

Absence de fermeture. Si la disponibilité de la fermeture facilite grandement certaines requêtes, sa maintenance requiert potentiellement beaucoup d'espace. Aussi, si une utilisation particulière ne nécessite pas la présence d'une fermeture, le concepteur peut décider de s'en passer. Il nous faut alors nous appuyer sur un autre mécanisme pour détecter les cycles : le rang des sommets.

Le rang d'un sommet indique le nombre de sommets présents dans le chemin le plus long menant à lui. Lorsque l'on crée un nouvel arc $a \to b$ dans le **dag**, nous tentons de mettre à jour le rang de b et des sommets atteignables *via* b. Si cela nous amène à mettre à jour le rang de a, un cycle est détecté et nous annulons l'opération.

Plutôt que de calculer le rang des sommets à chaque requête, nous le stockons dans la base. Si les sommets sont stockés dans un attribut dédié, le rang peut tout simplement venir compléter la portée de l'application. Dans le cas contraire, une application spécifique doit être créée pour maintenir cette information.

9.2.6 Lien avec le modèle

La traduction des facettes se fait très simplement en utilisant une première transformation avant d'appliquer le processus normal :

$$name : \textbf{facet}([a :] T) \quad \Rightarrow \quad \begin{array}{c} name : \textbf{map}(id : Entity, \\ [a :] T) \end{array} \tag{9.38}$$

L'identifiant introduit comme domaine de l'application correspond à l'identifiant de l'entité supportant la facette. On remarquera que si le domaine de la facette est vide, l'application générée est équivalente à une relation $name$: **relation**$(id : Entity)$.

9.2.7 Le noyau

En plus du stockage des entités et des données qui les composent, le SGBD doit supporter les opérations de modèle (les principales étant l'attachement et le détachement de facettes). En outre, nous exploitons aussi la base de données pour héberger le méta-modèle. L'ensemble des éléments indispensables au fonctionnement du système, et à la réception de la traduction relationnelle du modèle, compose

ce que nous appelons le *noyau* du système. Il peut être vu comme l'amorce (ou *bootstrap*) nécessaire à la création d'une application et de son modèle.

Création du *bootstrap*. Grâce à l'algorithme introduit dans ce chapitre, nous sommes capables de traduire en relationnel tous les types du modèles DOAN. Nous pouvons aussi nous appuyer sur cet algorithme pour créer le méta-modèle. Le reste des opérations, en revanche, est créé de façon *ad hoc* et peut différer selon le SGBD utilisé. La gestion des erreurs, par exemple, peut s'appuyer sur la valeur de retour des procédures stockées dans le cas de SQL Server, alors qu'elle doit s'inscrire dans le mécanisme des exceptions sous ORACLE[11]. Le choix de la cible d'implémentation peut ainsi avoir un effet important sur les opérations du noyau.

À chaque SGBD supportant DOAN correspond donc une version des scripts du noyau. Ces scripts peuvent être écrits de façon *ad hoc* ou encore s'appuyer sur des patrons [46, 3], notamment pour les éléments du méta-modèle.

Paramétrage. Certaines options du système doivent être choisies dès l'installation, car elles influencent directement les scripts générés :

- `CascadingUnplug` : Si cette option est active, le détachement d'une facette provoque le détachement de toutes ses facettes impliquées (si elles ne sont pas impliquées par d'autres facettes supportées ou attachées explicitement).
- `LastMainFacetUnplug` : Cette option indique si le détachement de la dernière facette principale est autorisé. Si tel est le cas, l'entité est automatiquement détruite. Sinon, le détachement est interdit et l'entité doit être explicitement détruite.
- `Rejection` : Cette option indique si l'exclusion entre facettes est supportée ou non.
- `CoPlug` : Cette option indique si l'implication légère est supportée ou non.

À l'exception de `CoPlug`, toutes ces options sont actives par défaut (se référer à l'annexe A sur le cycle de vie pour voir un exemple de leur influence sur les scripts générés).

Opérations. Le *bootstrap* du système installe toutes les opérations du noyau et parmi elles les opérations fondamentales du cycle de vie :

- Attachement / Détachement d'une facette à une entité.
- Création / Condamnation / Restauration / Destruction d'une entité.

Ces opérations sont détaillées dans l'annexe A.

Adaptations particulières. Il peut être nécessaire d'adapter le noyau à des besoins particuliers. Un programmeur peut, par exemple, vouloir ajouter ou modifier les opérations du système pour y ajouter des fonctions de journalisation, tester des préconditions, ou encore déclencher des actions extérieures au noyau.

Pour permettre de telles adaptations, nous permettons l'insertion dans les opérations de morceaux de code appelés *code snippets*. Les actions représentées par ces *code snippets* sont exécutées lorsque des événements particuliers sont déclenchés, comme par exemple l'attachement d'une nouvelle facette à une entité. Ce mécanisme fait l'objet de l'annexe B. Il se rapproche de ce qui a été fait pour les frames avec les *dæmons* (cf. section 6.1.1) ou encore de la programmation par aspects [61, 72].

[11] ORACLE ne permet pas aux procédures stockées de retourner de valeurs autrement que par l'utilisation explicite d'un paramètre `OUTPUT`, mais fournit en contrepartie une gestion des exceptions plus riche.

Sécurité. Comme nous l'avons vu en section 2.4, la sécurité des données est un aspect important des applications de gestion des connaissances. Parmi les implémentations possibles, nous avons choisi de fournir par défaut une sécurité s'appuyant sur le concept de liste de contrôle d'accès (ACL, *Access Control Lists*) [85, 9]. L'annexe C détaille l'approche et décrit ses avantages et ses inconvénients.

9.3 Conclusion

Nous avons introduit dans ce chapitre une implémentation possible du modèle DOAN. Notre choix s'est porté sur le modèle relationnel qui offre de nombreux atouts. Parmi eux, nous bénéficions directement de ses capacités à supporter la charge, des bonnes performances des requêtes ensemblistes, et de sa part de marché dominante.

La principale contribution de notre implémentation est l'algorithme de traduction d'un système de types riche, vers des relations et des procédures stockées associées. Il permet au concepteur d'exploiter des données applicatives complexes de façon simple et précise tout en s'appuyant sur les SGBDR présents sur le marché, et ainsi profiter de leurs performances et possibilités de montée en charge.

Bien qu'il puisse fonctionner dans un mode complètement automatique, le processus de traduction peut être adapté pour répondre à des besoins particuliers. Les résultats restent consistants et reproductibles, et sont donc plus compréhensibles par les concepteurs et programmeurs que s'ils étaient créés de façon *ad hoc*. Des opérations contrôlent les mises à jour des données, alors que le programmeur peut s'appuyer directement sur SQL pour les interroger. Qui plus est, les résultats de la traduction ne sont pas liés à un SGBD particulier mais peuvent cependant profiter d'implémentations spécifiques.

Le processus de traduction a été créé dans le cadre du modèle décrit dans le Chapitre 8. Il est cependant suffisamment général pour être utilisé dans un autre contexte, en particulier pour traduire les types ODMG originaux en relations.

Si la démarche suivie est proche de l'approche MDA (*Model Driven Architecture*) [75], les transformations ont en revanche été programmées de façon *ad hoc* en utilisant le langage Java (voir la capture d'écran en figure 9.3). Elles pourront cependant, avec l'apparition d'outils de transformation de modèles, être traduites dans un environnement plus ouvert.

Nous n'avons présenté dans la section 9.2 qu'une partie des traductions possibles. À titre d'exemple, nous avons utilisé une liste d'adjacence pour transformer un arbre en relationnel. D'autres approches sont également possibles, en particulier les approches *Materialized Path* [96], *Nested Sets* [23] et leur généralisation *Nested Intervals* [97]. Elles sont plus performantes, mais forcent aussi le programmeur à écrire des requêtes plus complexes.

Notre but n'était pas d'être exhaustif, mais de proposer une solution « par défaut » à chaque situation. Sous un aspect pratique, nous envisageons de n'introduire de nouvelles traductions que si des besoins les nécessitant apparaissent. Le concepteur pourra alors utiliser des indications sur les types (*hints*) pour piloter l'algorithme et choisir ces nouvelles transformations.

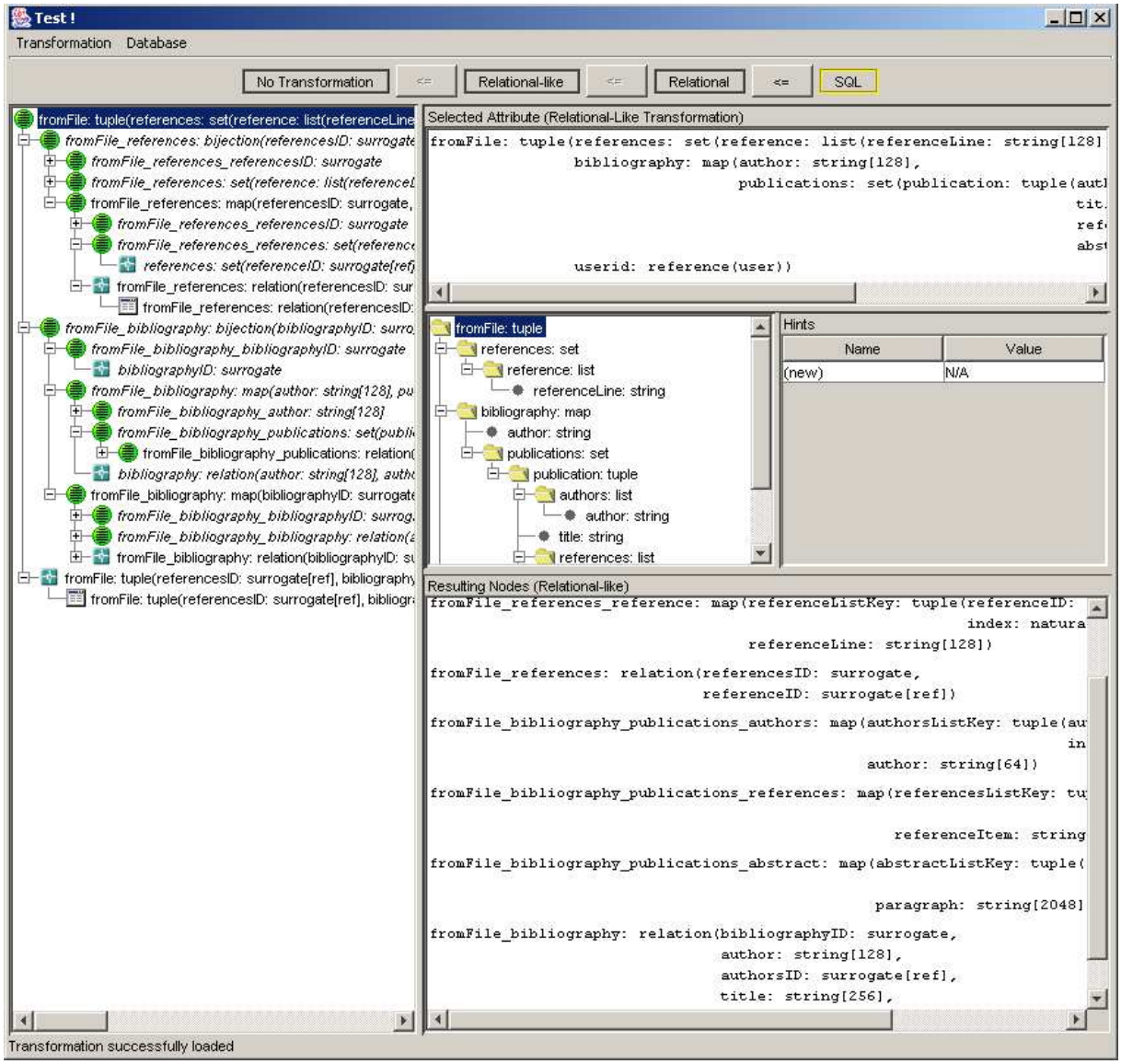

Figure 9.3 – Outil de traduction

CHAPITRE 10

Performances

Afin d'évaluer notre implémentation relationnelle, nous avons choisi de transposer une base de connaissances trouvée dans un autre modèle : l'exemple « newspaper » *de Protégé-2000 [77]. Protégé est un système à base de* frames *utilisé communément pour modéliser et exploiter de la connaissance, et l'exemple choisi correspond au domaine de l'édition de journaux.*

Notre but, dans ce chapitre, est de démontrer l'efficacité de notre implémentation relationnelle en réponse au besoin de montée en charge exprimé au chapitre 2, et non de comparer précisément chaque modèle (voir pour cela le chapitre 7 et la section 8.5 du chapitre 8).

10.1 Transposition du modèle de l'exemple *« newspaper »*

Notre objectif n'est pas de simuler un système à base de *frames*, mais d'examiner le comportement de notre modèle quand il doit faire face à d'importantes demandes de tenue en charge, un domaine dans lequel les modèles à base de *frames* sont pauvres [58]. Aussi, nous n'avons pas essayé de traduire tous les concepts de ces modèles. Nous nous sommes contentés de ceux disponibles dans Protégé, et plus précisément de ceux exploités par l'exemple choisi.

La plupart des traductions sont simples et directes. La transformation des classes est semblable à celle décrite pour les modèles à objets en section 8.5.2. Une classe et ses *slots* sont transformés en une facette dont le type est un n-uplet d'attributs. Les classes abstraites deviennent des facettes, alors que les classes normales deviennent des facettes principales.

Tous les types de Protégé sont aisément transposés dans notre système de types. Les seules particularités concernent les types multi-valués et les énumérations. Les premiers sont traduits par des ensembles, les secondes par des contraintes sur les valeurs.

D'autre part, comme pour la simulation du modèle entité-association (section 8.5.1), les associations sont réifiées dans des facettes. La hiérarchie de classes, quant à elle, est simplement simulée à l'aide de contraintes d'implication.

Seul un concept ne peut pas être transposé dans le modèle DOAN : les contraintes d'application. Nous avons décidé de les implémenter directement à l'aide de déclencheurs (*triggers*) sur les tables correspondantes.

Les bases de test. Pour parvenir à tester la tenue à la charge de notre implémentation, nous avons préparé cinq versions de la base de connaissance, l'originale, et quatre autres beaucoup plus volumineuses, contenant respectivement :

1. 3 auteurs, 6 journaux, et 9 articles;
2. base originale + 10 auteurs, 100 journaux, et 1,000 articles

3. base originale + 10 auteurs, 1,000 journaux, et 10,000 articles,
4. base originale + 100 auteurs, 1,000 journaux, et 100,000 articles,
5. base originale + 100 auteurs, 10,000 journaux, et 1,000,000 articles.

10.2 Résultats

Pour effectuer les tests, nous avons utilisé un *Bi-Pentium III* à *733 MHz* avec *1 Go* de mémoire vive, le système d'exploitation *Windows 2000 Server* opérant le système de gestion de base de données relationnelles (SGBDR) *SQL Server 2000* et Java, *via* le moteur *JRE 1.4.2*.

Nous avons utilisé deux versions de Protégé-2000 : une fonctionnant de façon standard sur le système de fichier et une autre fonctionnant avec une base de données *via* JDBC (*Java Database Connectivity*) [82]. La version JDBC de Protégé s'appuie sur une unique relation contenant toute la base de connaissance, modèle inclus. Elle s'appuie elle aussi sur le SGBDR *SQL Server 2000*.

Une première observation concerne la limitation importante de Protégé en terme de montée en charge. En effet, nous n'avons pas été en mesure de tester la cinquième version de la base de connaissance (plus d'un million d'instances). Lorsque Protégé s'appuie sur le système de fichier, il doit charger l'intégralité de la base en mémoire, dépassant ainsi les capacités de la machine. À partir de la quatrième version, qui consomme 730 Mo, nous pouvons estimer que l'empreinte mémoire de la cinquième dépasse 6 Go. Dans la version JDBC, les temps de réponse étaient simplement trop importants pour être mesurés.

Une autre limite de Protégé est l'importance du temps de chargement nécessaire avant de pouvoir effectuer une requête (au dessus de 25 minutes pour la quatrième version de la base de connaissances, aussi bien pour la version fichier que JDBC).

Nous avons exécuté nos requêtes de test à partir d'un programme Java, utilisant JDBC pour DOAN et l'API Java fournie pour Protégé. De plus, nous avons explicitement parcouru tous les résultats des requêtes de façon à avoir des temps d'exécution comparables. En effet, JDBC ne retourne les résultats qu'à la demande.

Première requête. Examinons tout d'abord une première requête très simple retournant la liste des employés, incluant les auteurs, qui sont bien payés. Comme nous pouvons le voir dans la figure 10.1, les temps de réponse sont corrects pour toutes les versions. Nous pouvons néanmoins déjà observer les faibles performances de la version JDBC de Protégé.

Deuxième requête. Notre seconde requête récupère les articles dont les auteurs sont bien payés, et renvoie un nombre très important de résultats pour les grosses bases de connaissances. Toutefois, il est très peu probable qu'une application nécessite de si gros ensembles résultants. Elle se contentera généralement des premiers résultats en fonction d'un tri prédéterminé. En conséquence, comme le montre la figure 10.2 nous avons aussi testé la même requête en nous contentant de parcourir uniquement les 100 premiers résultats.

Alors que cette requête montre clairement l'incapacité de Protégé à traiter d'importants volumes de données, nous pouvons observer que notre implémentation ne met que 655 millisecondes à répondre avec une base dépassant le million d'instances. Plus important encore, le temps de réponse reste constamment en dessous de la barrière des 10 millisecondes si nous nous contentons des 100 meilleurs résultats.

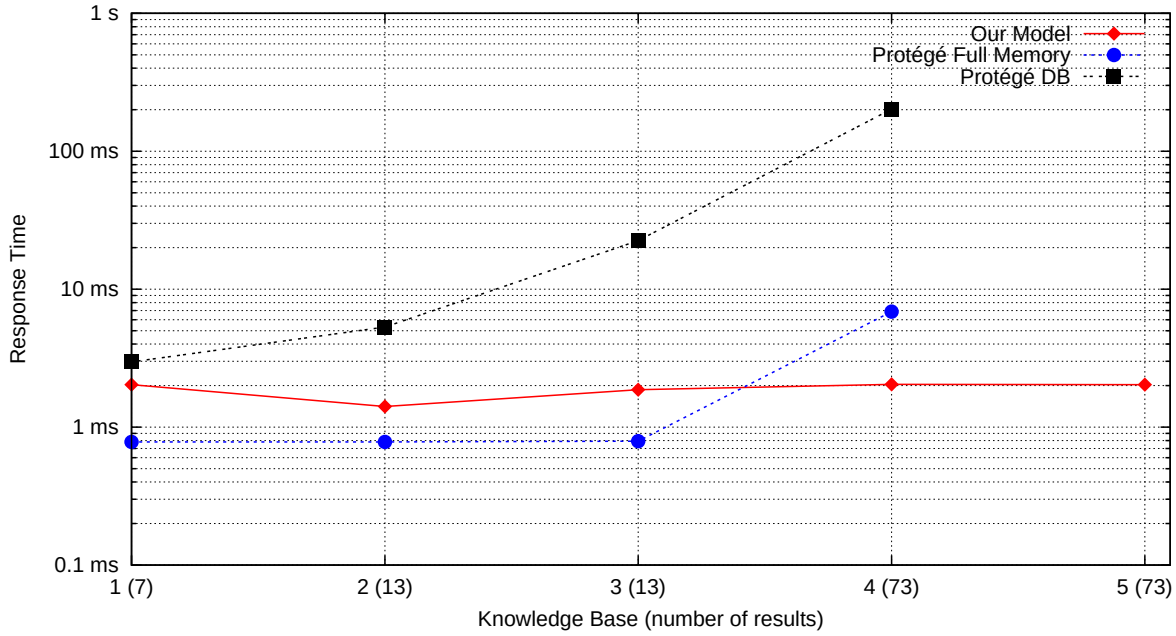

Figure 10.1 – Requête n°1

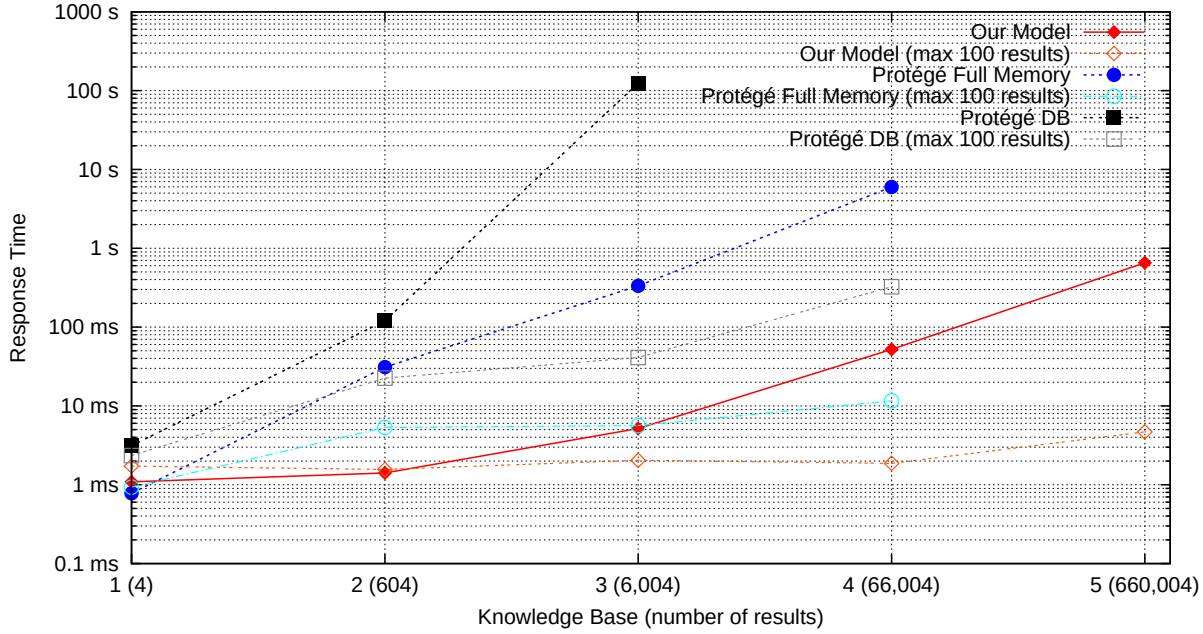

Figure 10.2 – Requête n°2

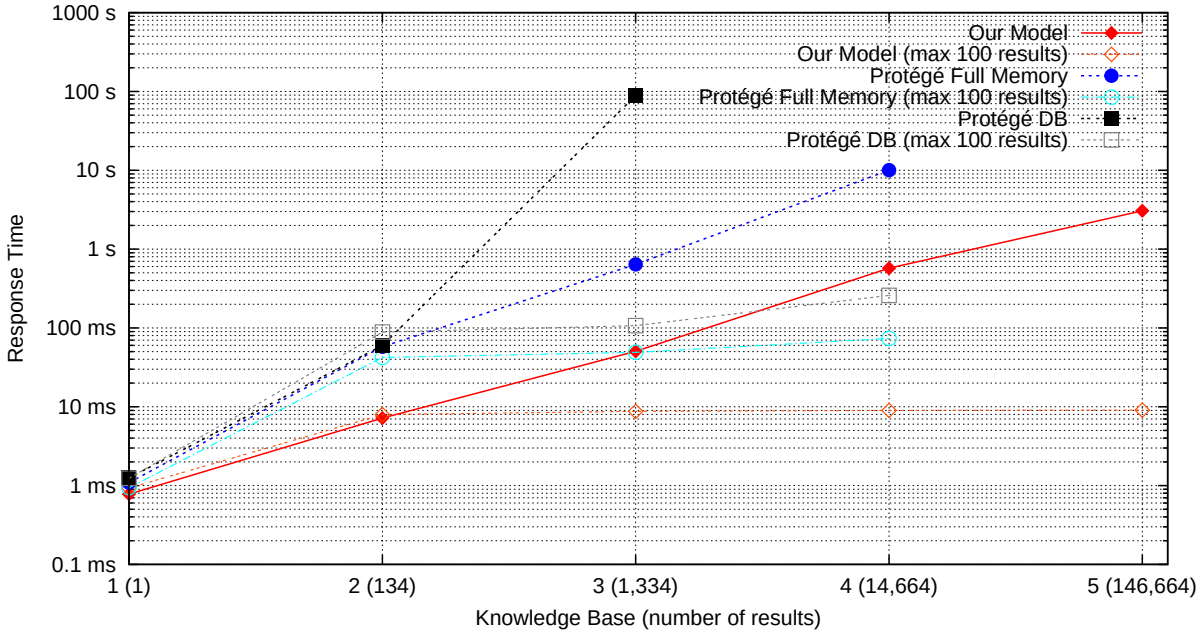

Figure 10.3 – Requête n°3

Troisième requête. Dans cette dernière requête, compliquons encore nos demandes en restreignant la précédente requête aux articles qui sont urgents et dont le nombre de pages est petit. Si l'ensemble résultant est plus faible, la complexité supplémentaire implique des temps de réponse plus lents, comme l'illustre la figure 10.3. Une seule exception à ce constat : la version JDBC de Protégé. En se limitant aux 100 meilleurs résultats, le temps de réponse est plus lent, comme attendu. Il est cependant plus rapide dès lors que tous les résultats sont renvoyés et donc que la restriction par rapport à la requête précédente agit sur leur nombre. Nous observons ici que la communication avec la base de données est le véritable goulot d'étranglement de Protégé.

10.3 Conclusion

Comme nous nous y attendions (cf. section 6.1.5), Protégé est adéquat pour la gestion de petites bases de connaissances, mais ne supporte pas bien le passage à l'échelle. La version s'appuyant sur un fichier exhibe des temps de réponse acceptables mais est rapidement limitée par les ressources mémoire disponibles. La version JDBC n'a pas une telle limitation, mais est, en contrepartie, beaucoup moins efficace.

L'approche choisie par Protégé de s'appuyer sur une unique relation dans le SGBD est simple et flexible. Toutefois, les résultats observés montrent que, bien qu'il rende l'algorithme de traduction en relationnel plus complexe, notre choix d'une structure de données proche du schéma représenté est beaucoup plus efficace.

Notre implémentation relationnelle de DOAN tire pleinement parti des avantages offerts par les SGBDR en terme de tenue en charge et de performances. Elle répond de manière satisfaisante aux besoins exprimés, ne montrant des signes de faiblesse que lorsque des ensembles résultants très volumineux ont besoin d'être parcourus. De plus, le concepteur d'application peut avantageusement tirer

parti des fonctionnalités des SGBDR pour optimiser le modèle en fonction de ses besoins essentiels (en particulier en créant des index adéquats).

PARTIE IV

Conclusion générale

CHAPITRE 11

Conclusion générale

Comme nous l'avons introduit au début de ce mémoire, cette thèse a été réalisée dans le but de répondre à un problème industriel concret.

Ce chapitre résume les principales contributions de notre proposition, mais expose aussi les limites de l'approche employée.

11.1 Contributions principales

Dans le cadre de la solution qu'elle édite, la société Arisem requiert un modèle ne trouvant pas de réponse adéquate dans la littérature. À partir de la problématique exprimée au chapitre 2, nous avons identifié trois « grands » besoins :

- Une importante expressivité afin de pouvoir représenter des structures complexes ;
- Une importante flexibilité afin de pouvoir gérer des données hétérogènes, les analyser de multiples façons, et supporter leurs évolutions ;
- Le support de fortes montées en charge afin de pouvoir traiter des volumes de documents très importants.

11.1.1 Modèle DOAN

Pour répondre à ces besoins, nous avons proposé le modèle DOAN (*DOcument ANnotation Model*). Ce modèle est construit autour du mécanisme fondamental d'agrégation de facettes.

Plutôt que de représenter les données d'une entité à l'aide d'attributs ou *via* l'instanciation de classes, nous les regroupons dans des « points de vues » que nous appelons facettes. Cette approche nous permet de bénéficier des facilités de définition, de regroupement et de manipulation d'objets que donnent les classes sans en supporter les contraintes. En particulier, le support d'une facette par une entité est dynamique : contrairement à l'instanciation de classes, une entité peut évoluer dans le temps en obtenant de nouvelles facettes et/ou en en perdant d'anciennes. Comme nous l'avons vu au chapitre 5, les modèles à base de rôles permettent de répondre en partie à ce besoin mais restent limités car ils conservent la notion de classe.

Afin de pouvoir représenter des domaines complexes, cette flexibilité est complétée par des contraintes, qui, bien que simples, permettent d'exprimer des besoins complexes. Elles sont plus souples que celles de nombreux modèles traditionnels sans pour autant pénaliser l'expressivité. Comme nous l'avons démontré en section 8.5.2, les contraintes d'exclusion et d'implication permettent notamment de simuler très simplement un arbre de spécialisation de classes.

Comme les modèles basés sur XML, et contrairement aux modèles à objets, DOAN permet le support transparent de points de vues complètement indépendants sur les entités. Notre problématique se concentre en effet sur la représentation des informations ; le code applicatif n'est pas à considérer. Dans

cette situation, il n'est pas nécessaire de fournir de mécanisme d'encapsulation ni d'assignation tardive (*late binding*). L'accès aux données d'une entité est toujours réalisé statiquement à l'aide de facettes.

Sur le même principe que les modèles à rôles, le modèle DOAN offre ainsi un accès contextuel aux entités. En évitant la dualité classe / rôle, les facettes sont cependant plus simples et flexibles que les rôles.

La notion de facette ne doit pas non plus être confondue avec celle de vue, qui est utilisée dans les systèmes de gestion de bases de données relationnels ou à objets. Contrairement aux vues, les facettes contiennent intrinsèquement leurs données.

Pour simplifier le travail du concepteur, le modèle propose également un riche système de types. Très proche de la proposition de l'ODMG, il s'en distingue par l'apport des types *relation* et *bijection*, ainsi qu'un ensemble de types à base de graphes (*arbre*, *graphe*, et *graphe acyclique orienté*), particulièrement utiles pour les applications en intelligence économique.

Grâce à l'utilisation des facettes, de leurs contraintes, et de son système de types, le modèle DOAN nous permet de répondre aux deux premiers besoins exprimés : l'expressivité et la flexibilité. Le troisième, les performances, est l'objet de son implémentation relationnelle.

11.1.2 Implémentation relationnelle

Nous avons choisi les Systèmes de Gestion de Bases de Données Relationnels (SGBDR) comme la cible principale de notre implémentation. En effet, en plus d'être matures et très répandus, ils présentent des performances et des possibilités de montée en charge en adéquation avec les besoins exprimés.

La principale difficulté de cette implémentation était la représentation des types complexes de DOAN. Nous avons pour cela proposé un algorithme de traduction de types proches de la proposition de l'ODMG dans le modèle relationnel. La traduction se déroule en trois phases rapprochant progressivement les types de leur implémentation finale : séparation des collections imbriquées en plusieurs collections, transformation en types purement relationnels, et finalement transformation dans une variante relationnelle dédiée au SGBDR choisi. Elle est entièrement automatique, mais peut être adaptée à des besoins ou à des choix d'implémentation particuliers. La reproductibilité de ses résultats aide les concepteurs dans leur maîtrise du processus.

Selon sa complexité, un type est ainsi transformé récursivement en une ou plusieurs relations. Des procédures stockées générées à leurs côtés contrôlent les mises à jour, alors que les accès en lecture sont effectués directement en SQL. Cette approche nous permet, d'une part, d'assurer la consistance des données, et, d'autre part, de rendre possible de nombreuses optimisations améliorant les performances d'accès. Le reste du modèle est implémenté soit à l'aide du même algorithme de traduction (ex. : méta-modèle), soit directement en relationnel (ex. : opérations du noyau).

Bien que créé pour DOAN, cet algorithme peut être utilisé avec succès pour tout type proche de la proposition de l'ODMG afin d'en obtenir une traduction relationnelle. Cela peut être particulièrement utile pour utiliser des types complexes, tels que les graphes acycliques orientés, dans un SGBDR.

Comme nous l'avons observé au chapitre 10, cette implémentation nous permet de répondre aux besoins de montée en charge des applications envisagées. D'autres cibles sont cependant possibles, comme par exemple les SGBD XML ou à objets. L'approche est alors la même : une traduction dans le modèle choisi, des accès en modification contrôlés par une API, et des accès en lecture directement dans le langage de requête (respectivement *X-Query* et OQL).

11.2 Discussion

11.2.1 Limites de l'approche

Bien qu'il réponde correctement aux besoins exprimés, le modèle DOAN souffre de certaines limitations. Certaines sont le résultat de choix délibérés. Ainsi, comme nous l'avons vu en section 8.3.2, les associations se limitent à l'utilisation de références unidirectionnelles. Cette restriction présente l'avantage de la simplicité pour les cas courants, sans pour autant empêcher la modélisation d'associations complexes grâce à la réification.

Pour les mêmes raisons de simplicité, nous avons choisi d'utiliser un méta-modèle simple et surtout séparé du modèle lui-même. En effet, l'expérience acquise avec le précédent modèle d'Arisem montre que les développeurs ne sont généralement pas à l'aise avec une intrication trop importante des notions méta avec celles du modèle lui-même. Ainsi, plutôt que de considérer, par exemple, les facettes comme de simples entités supportant la facette FACET, nous avons préféré les modéliser à part.

Une autre limitation concerne le support des facettes dans DOAN : il est impossible pour une entité de supporter deux fois la même facette. Si un tel besoin apparaît, il doit être modélisé à l'aide d'une collection de références. Une solution à ce problème serait de donner une identité propre aux facettes, sur le même principe que les identifiants de rôles (RID), introduits en section 5.2.2.

D'autres aspects manquent à DOAN et pourront faire, eux aussi, l'objet de développements futurs. C'est notamment le cas d'un mécanisme d'historique ou de *versioning* des entités. En effet, du fait de la simplicité donnée à l'application pour faire évoluer les entités, il est possible que celles-ci changent de façon importante au cours de leur existence. Si un concepteur veut conserver une trace des opérations d'attachement et de détachement réalisées sur les entités, il doit aujourd'hui créer un système *ad hoc* externe au modèle. L'intégration d'un mécanisme permettant de gérer versions et alternatives directement au modèle ou à son implémentation de référence fait donc sens.

Comme nous avons également pu l'observer, l'implémentation relationnelle du modèle répond correctement aux besoins exprimés de support des montées en charge. Ces performances sont cependant obtenues au prix d'une certaine complexité : le programmeur est directement confronté à la structure relationnelle. Nous avons cherché à limiter cette difficulté en fournissant des traductions reproductibles des types de haut niveau vers le modèle relationnel. La tâche du programmeur peut aussi être simplifiée à l'aide d'une bibliothèque de macros et de *best practices*.

Enfin, notons que le modèle a été conçu pour répondre à une problématique de modélisation de données. Cette restriction nous a permis de simplifier certains aspects (ex. : méthodes) au détriment de la généricité.

11.2.2 Réflexions complémentaires

Lors de la création de l'implémentation, les SGBD relationnels se sont imposés comme le choix logique. Ils sont en effet à la base de la solution actuelle éditée par Arisem. Un autre choix aurait compliqué la migration des données des clients tout en sortant du champ d'expertise des développeurs. Il aurait toutefois été intéressant de profiter de cette implémentation pour étudier des systèmes plus récents, comme par exemple les SGBD XML.

Aujourd'hui, de nouvelles traductions ne sont pas envisagées. En revanche, comme nous le notions en section 7.3, une importation / exportation au format OWL est une extension intéressante.

Notons aussi que tout en étant adaptée à la problématique métier étudiée, l'approche hybride que nous avons employée demande un investissement important. En plus d'une bonne maîtrise des modèles

relationnels, elle nécessite en effet l'apprentissage de nouveaux concepts (facettes et contraintes) par les concepteurs et programmeurs.

L'évolution de la demande client nous montre d'autre part que DOAN ne répond pas idéalement à tous les besoins du domaine de la gestion des connaissances. À titre d'exemple, Arisem est de plus en plus amenée à quitter le secteur de la veille et de l'intelligence économique pour se positionner sur celui des moteurs de recherche. La simple recherche de documents sur un Intranet, en particulier, bénéficie grandement de l'analyse sémantique. En effet, l'exploitation du contexte client à travers une base de connaissances permet, dans ce cas, d'obtenir une très bonne pertinence des résultats. Si le modèle DOAN peut être utilisé pour ce type d'applications, sa flexibilité n'est pas nécessaire. Un modèle plus simple, dédié purement à la recherche est alors plus approprié.

Bibliographie

[1] Serge ABITEBOUL et Anthony J. BONNER. Objects and views. In James CLIFFORD et Roger KING, réds., *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 238–247. ACM Press, 1991.

[2] Antonio ALBANO, Roberto BERGAMINI, Giorgio GHELLI et Renzo ORSINI. An object data model with roles. In Rakesh AGRAWAL, Seán BAKER et David A. BELL, réds., *19th International Conference on Very Large Data Bases*, pages 39–51, Dublin, Ireland, 1993. Morgan Kaufmann.

[3] APACHE JAKARTA. Velocity. http://jakarta.apache.org/velocity/, 2004.

[4] ARISEM. Arisem. http://www.arisem.com/.

[5] ARISEM. Présentation générale de OpenPortal4U. Whitepaper, Arisem, Paris, France, 2002.

[6] Malcolm ATKINSON, François BANCILHON, David DEWITT, Klaus DITTRICH, David MAIER et Stanley ZDONIK. The Object-Oriented Database System Manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223–240, Kyoto, Japan, 1989.

[7] Charles W. BACHMAN et Manilal DAYA. The role concept in data models. In *Proceedings of the Third International Conference on Very Large Data Bases*, pages 464–476, Tokyo, Japan, 1977. IEEE Computer Society.

[8] François BANCILHON. Object-oriented database systems. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 152–162. ACM, 1988.

[9] John BARKLEY. Comparing simple role based access control models and access control lists. In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 127–132, New York, NY, USA, 1997. ACM Press.

[10] Tim BERNERS-LEE, Roy FIELDING et Larry MASINTER. Uniform resource identifiers (URI): Generic syntax. http://www.ietf.org/rfc/rfc2396.txt, 1998.

[11] Scott BOAG, Don CHAMBERLIN, Mary F. FERNÁNDEZ, Daniela FLORESCU, Jonathan ROBIE et Jérôme SIMÉON. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, 2005.

[12] Grady BOOCH. *Object Oriented Design with Applications*. Benjamin-Cummings, 1991.

[13] Ronald BOURRET. XML and Databases. http://www.rpbourret.com/xml/XMLAndDatabases.htm, 2004.

[14] R.J. BRACHMAN et J.G. SCHMOLZE. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[15] Tim BRAY, Dave HOLLANDER et Andrew LAYMAN. Namespaces in xml. http://www.w3.org/TR/REC-xml-names/, 1999.

[16] Dan BRICKLEY et R. V. GUHA. RDF vocabulary description language 1.0: RDF schema. http://www.w3.org/TR/rdf-schema/, 2004.

[17] Jeen BROEKSTRA, Arjohn KAMPMAN et Frank van HARMELEN. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In I. HORROCKS et J. HENDLER, réds., *First International Semantic Web Conference (ISWC2002), number 2342 in Lecture Notes in Computer Science*, pages 54–68, Sardinia, Italy, 2002. Springer Verlag, Heidelberg Germany.

[18] Diego CALVANESE, Giuseppe De GIACOMO et Maurizio LENZERINI. Description logics: Foundations for class-based knowledge representation. In *Proc. of the 17th IEEE Sym. on Logic in Computer Science (LICS 2002)*, pages 359–370, 2002.

[19] Diego CALVANESE, Giuseppe De GIACOMO, Maurizio LENZERINI et Daniele NARDI. Reasoning in expressive description logics. In Alan ROBINSON et Andrei VORONKOV, réds., *Handbook of Automated Reasoning*, pages 1581–1634. Elsevier Science Publishers, 2001.

[20] Diego CALVANESE, Maurizio LENZERINI et Daniele NARDI. Description logics for conceptual data modeling. In Jan CHOMICKI et Gunter SAAKE, réds., *Logics for Databases and Information Systems*, pages 229–263. Kluwer, 1998.

[21] Bernard CARAYON. Intelligence économique, compétitivité et cohésion sociale / Rapport au Premier Ministre, juin 2003.

[22] R.G.G. CATTELL, Douglas K. BARRY et OTHERS, réds. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann Publishers, 2000.

[23] Joe CELKO. *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann, 2004.

[24] V. K. CHAUDHRI, A. FARQUHAR, R. FIKES, P. D. KARP et J. P. RICE. OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings of AAAI*, pages 600–607, Madison, Wisconsin, United States of America, 1998.

[25] P.P. CHEN. The entity-relationship model-towards a unified view of data. *CM Transactions on Database Systems*, 1(1):9–36, 1976.

[26] COMMISSARIAT GÉNÉRAL DU PLAN. *Intelligence économique et stratégique des entreprises / travaux du groupe présidé par Henri Martre*. La Documentation Française, 1994.

[27] Stéphane COULONDRE et Thérèse LIBOUREL. An integrated object-role oriented database model. *Data & Knowledge Engineering*, 42(1):113–141, 2002.

[28] Mohamed DAHCHOUR, Esteban ZIMÁNYI et Alain PIROTTE. Object-oriented modeling with roles. In Anne Banks PIDDUCK, John MYLOPOULOS, Carson C. WOO et M. Tamer ÖZSU, réds., *14th International Conference on Advanced Information Systems Engineering, CAiSE'02*, Toronto, Canada, 2002. Springer.

[29] C. DATE et H. DARWEN. *A Guide to the SQL Standard*. Addison-Wesley, 1997.

[30] N. DAY. MPEG-7 Daring to Describe Multimedia Content. *XML Journal*, 1(6), 2000.

[31] Bernard DEFLESSELLES et Jean MICHEL. Rapport d'information sur la participation de capitaux étrangers aux industries européennes d'armement, Enregistré à la Présidence de l'Assemblée nationale, No 222, mars 2005.

[32] Nicolas DESSAIGNE. K-Mining - Architecture technique. Rapport technique, Arisem, Paris, France, 2003.

[33] Nicolas DESSAIGNE. Les ontologies dans l'entreprise pour le non structuré : enjeux et cas concrets. In *AFIA2003, journée Web Sémantique*, Laval, France, 2003.

[34] Nicolas DESSAIGNE et Alain GARNIER. AB-STRAT - Rapport technique. Convention Ministère de la Recherche No 02M4689, Arisem, Paris, France, 2004.

[35] Nicolas DESSAIGNE et José MARTINEZ. A model for describing and annotating documents. In *EJC2005, 15th European-Japanese Conference on Information Modelling and Knowledge Bases*, Tallinn, Estonia, 2005.

[36] Guozhu DONG, Leonid LIBKIN, Jianwen SU et Limsoon WONG. Maintaining transitive closure of graphs in SQL. *International Journal of Information Technology*, 5:46–78, 1999.

[37] Andrew EISENBERG, Jim MELTON, Krishna KULKARNI, Jan-Eike MICHELS et Fred ZEMKE. SQL:2003 has been published. *SIGMOD Rec.*, 33(1):119–126, 2004.

[38] R. ELMASRI, J. WEELDREYER et A. HEVNER. The category concept: an extension to the entity relationship model. *Data & Knowledge Engineering*, 1(1):75–116, 1985.

[39] Ramez ELMASRI et Shamkant B. NAVATHE. *Fundamentals of Database Systems, 4/E*. Addison-Wesley, 2003.

[40] Thomas ERL. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, chapter 7: Integrating XML and Databases. Prentice Hall PTR, 2004.

[41] Eckhard D. FALKENBERG. Concepts for modelling information. In *Proceedings of IFIP Working Conference on Modelling in Data Base Management Systems*, pages 95–109, Freudenstadt, Germany, 1976. North-Holland.

[42] Dieter FENSEL, Ian HORROCKS, Frank van HARMELEN, Deborah L. MCGUINNESS et Peter F. PATEL-SCHNEIDER. OIL: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[43] Mark S. FOX, J. Mark WRIGHT et David ADAM. Experiences with SRL: an analysis of frame-based knowledge representations. In *Proceedings from the first international workshop on Expert database systems*, pages 161–172, Kiawah Island, South Carolina, United States of America, 1985. Benjamin-Cummings Publishing Co., Inc.

[44] Bertrand FRAYSSE. La vérité sur... l'offensive de la CIA en France. *Challenges*, 227, juin 2004.

[45] Alain GARNIER et Nicolas DESSAIGNE. Ontologies in production environment: Stakes and concrete examples. In *OntoWeb5 SIG4 : Industrial Applications*, Sundial Island, Florida, United States of America, 2003.

[46] Benjamin GEER, Mike BAYER et Jonathan REVUSKY. The FreeMarker Template Engine. http://freemarker.sourceforge.net/, 2005.

[47] Georg GOTTLOB, Michael SCHREFL et Brigitte ROCK. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.

[48] Tom R. GRUBER. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[49] Nicola GUARINO. Concepts, attributes and arbitrary relations: some linguistic and ontological criteria for structuring knowledge bases. *Data and Knowledge Engineering*, 8(3):249–261, 1992.

[50] P. HALL, J. OWLETT et S. TODD. Relations and entities. In G. M. NIJSSEN, réd., *Modelling in DataBase Management Systems*, pages 201–220. North Holland, 1976.

[51] Terry HALPIN. Object role modeling (ORM/NIAM). In Peter BERNUS, Kai MERTINS et Günter SCHMIDT, réds., *Handbook on Architectures of Information Systems*, pages 81–101. Springer, 1998.

[52] Terry HALPIN. *Information Modeling and Relational Databases - From Conceptual Analysis to Logical Design*. Morgan Kaufmann Publishers, 2001.

[53] Terry HALPIN. Uniqueness constraints on objectified associations. *Journal of Conceptual Modeling (issue 29)*, 2003.

[54] Terry HALPIN et Anthony BLOESCH. Data modeling in UML and ORM: a comparison. *Journal of Database Management*, 10(4), 1999.

[55] Richard HULL et Roger KING. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[56] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 9899:1990: Programming languages — C*. International Organization for Standardization, 1990.

[57] Ivar JACOBSON, Magnus CHRISTERSON, Patrik JONSSON et Gunnar ÖVERGAARD. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1992.

[58] P. KARP. The Design Space of Frame Knowledge Representation Systems. Rapport technique 520, SRI International AI Center, 1992.

[59] Setrag N. KHOSHAFIAN et George P. COPELAND. Object identity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'86*, pages 406–416, Portland, Oregon, United States of America, 1986. ACM Press.

[60] Gregor KICZALES, Jim des RIVIERES et Daniel G. BOBROW. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[61] Gregor KICZALES, John LAMPING, Anurag MENDHEKAR, Chris MAEDA, Cristina Videira LOPES, Jean-Marc LOINGTIER et John IRWIN. Aspect-Oriented Programming. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 1997. Springer.

[62] Graham KLYNE et Jeremy J. CARROLL. Resource description framework (RDF): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, 2004.

[63] KNOWINGS. Enquête 2003 « vision des dirigeants en matière de knowledge management ». http://www.knowings.com, 2003.

[64] Bent Bruun KRISTENSEN. Object-oriented modeling with roles. In John MURPHY et Brian STONE, réds., *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71. Springer-Verlag, 1996.

[65] Miltiadis D. LYTRAS et Tom R. GRUBER. Dr. Tom Gruber's interview: "Every ontology is a treaty". *Bulletin of AIS Special Interest Group on Semantic Web and Information Systems*, 1(3):4–8, 2004.

[66] Yogesh MALHOTRA. Deciphering the Knowledge Management Hype. *The Journal for Quality and Participation*, 21(4):58–60, 1998.

[67] Heikki MANNILA et Kari-Jouko RÄIHÄ. *The Design of Relational Databases*. Addison-Wesley, 1992.

[68] Deborah L. MCGUINNESS. Description logics emerge from ivory towers (position paper). In *International Semantic Web Working Symposium*, 2001.

[69] Deborah L. MCGUINNESS. Ontologies come of age. In Dieter FENSEL, James HENDLER, Henry LIEBERMAN et Wolfgang WAHLSTER, réds., *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, 2002.

[70] Deborah L. MCGUINNESS, Richard FIKES, James HENDLER et Lynn Andrea STEIN. DAML+OIL: An ontology language for the semantic web. *IEEE Intelligent Systems*, 17(5):72–80, 2002.

[71] MICROSOFT CORPORATION. *Microsoft C# Language Specifications*. Microsoft Press, 2001.

[72] Russell MILES. *AspectJ Cookbook*. O'Reilly, 2004.

[73] Marvin MINSKY. A framework for representing knowledge. In Patrick Henry WINSTON, réd., *The Psychology of Computer Vision*, pages 211–277. New York: McGraw-Hill, 1975.

[74] Tom M. MITCHELL, John ALLEN, Prasad CHALASANI, John CHENG, Oren ETZIONI, Marc RINGUETTE et Jeffrey C. SCHLIMMER. Theo: A framework for self-improving systems. In K. VANLEHN, réd., *Architectures for Intelligence*, pages 323–355. Erlbaum, 1991.

[75] Jishnu MUKERJI et Joaquin MILLER. MDA Guide Version 1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01, June 2003.

[76] Shir NIJSSEN et Terry HALPIN. *Conceptual Scheme and Relational Database Design*. Prentice-Hall, 1989.

[77] Natalya Fridman NOY, Ray W. FERGERSON et Mark A. MUSEN. The knowledge model of protégé-2000: Combining interoperability and flexibility. In *2th International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000)*, Juan-les-Pins, France, 2000.

[78] OMG. Object Management Group Adopts Unified Modeling Language and Meta Object Facility Specifications. http://www.omg.org/news/pr97/umlpr.html, 1997.

[79] OMG. Unified Modeling Language Specification, Version 1.5. Rapport technique, OMG, March 2003.

[80] S. PEPPER et G. MOORE. XML Topic Maps (XTM) 1.0. http://www.topicmaps.org/xtm/1.0/, 2001.

[81] Ruben PRIETO-DIAZ et Peter FREEMAN. Classifying software for reusability. *IEEE Software*, 4(1):6–16, 1987.

[82] George REESE. *Database Programming with JDBC and Java, Second Edition*. O'Reilly & Associates, 2000.

[83] James RUMBAUGH, Michael BLAHA, William PREMERLANI et Frederick EDDY. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[84] James RUMBAUGH, Ivar JACOBSON et Grady BOOCH. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[85] Ravi SANDHU et Pierangela SAMARATI. Access Control: Principles and Practice. *IEEE Communications*, 32(9):40–48, 1994.

[86] Michael K. SMITH, Chris WELTY et Deborah L. MCGUINNESS. OWL web ontology language guide. http://www.w3.org/TR/owl-guide/, 2004.

[87] Guy L. STEELE, JR. et Richard P. GABRIEL. The evolution of Lisp. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 231–270, Cambridge, Massachusetts, United States, 1993. ACM Press.

[88] Mark STEFIK. An examination of a frame structure representation system. In *Proceedings of the Sixth International Joint Conference on Artifical Intelligence*, pages 845–852, Tokyo, Japan, 1979.

[89] Friedrich STEIMANN. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000.

[90] Lynn Andrea STEIN, Dan CONNOLLY et Deborah L. MCGUINNESS. DAML-ONT initial release. http://www.daml.org/2000/10/daml-ont.html, 2000.

[91] Jianwen SU. Dynamic constraints and object migration. *Theoretical Computer Science*, 184(1-2):195–236, 1997.

[92] SUN MICROSYSTEMS. Java technology. http://java.sun.com/.

[93] Ralph R. SWICK et Henry S. THOMPSON. The cambridge communiqué. http://www.w3.org/TR/schema-arch, 1999.

[94] A. TAIVALSAARI. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, septembre 1996.

[95] Toby. J. TEOREY, Dongqing YANG et James P. FRY. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2):197–222, 1986.

[96] Vadim TROPASHKO. Trees in SQL: Nested Sets and Materialized Path. http://www.dbazine.com/oracle/or-articles/tropashko4, avril 2004.

[97] Vadim TROPASHKO. Nested Intervals Tree Encoding in SQL. *SIGMOD Record*, 34(2):47–52, 2005.

[98] W3C. World Wide Web Consortium Issues RDF and OWL Recommendations. http://www.w3.org/2004/01/sws-pressrelease.html.en, 2004.

[99] Gio WIEDERHOLD et Ramez ELMASRI. The structural model for database design. In Peter P. CHEN, réd., *Entity-Relationship Approach to Systems Analysis and Design. Proc. 1st International Conference on the Entity-Relationship Approach*, pages 237–258. North-Holland, 1979.

[100] Roel WIERINGA et Wiebren de JONGE. The identification of objects and roles – object identifiers revisited. Rapport technique, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1992.

[101] Roel WIERINGA et Wiebren de JONGE. Object identifiers, keys, and surrogates: object identifiers revisited. *Theory ans Practice of Object Systems*, 1(2):101–114, 1995.

[102] Raymond K. WONG, H. Lewis CHAU et Frederick H. LOCHOVSKY. A data model and semantics of objects with dynamic roles. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 402–411. IEEE Computer Society, 1997.

Liste des tableaux

Table des figures

Partie I — Introduction et analyse des besoins

Partie II — État de l'art

Partie III — Proposition

Partie IV — Conclusion générale

Annexes

Table des matières

Partie I — Introduction et analyse des besoins

Partie II — État de l'art

Partie III — Proposition

Partie IV — Conclusion générale

Annexes

Index

Annexes

ANNEXE A

Cycle de vie

Cette annexe présente les différents états qu'une entité traverse au cours de son cycle de vie, ainsi que les opérations liées (attachement, détachement, etc.).

A.1 États des entités

L'un des besoins du modèle est de permettre la restauration des entités détruites « par erreur ». Il est important de prendre en compte ce problème dès les premiers stades de la réflexion car il a d'importants impacts sur la gestion du cycle de vie des entités.

A.1.1 Restauration

Une première approche consiste en l'utilisation d'un *journal* pour pister les modifications faites durant la suppression, de façon à les restaurer si besoin. Toutefois, cette approche ne permet de restaurer que la dernière entité et correspond davantage à une option d'annulation.

Une seconde approche est de simplement marquer les objets comme « condamnés » en les plaçant dans une « poubelle » de façon à pouvoir les restaurer dans leur état complet à tout moment.

Nous avons privilégié la seconde approche, cette dernière étant plus simple et offrant plus de fonctionnalités. Toutefois, elle a des impacts importants, qui peuvent être vus autant comme des avantages que des inconvénients en fonction des besoins :

- Il est possible de condamner une entité même s'il existe des références dessus ; seule la destruction finale est alors bloquée. La présence d'une entité dans la poubelle n'est donc qu'indicative.
- Comme aucune référence n'est vérifiée, il est possible de condamner toute instance. Cette fonctionnalité est particulièrement utile quand quelqu'un veut détruire un ensemble d'entités contenant de nombreuses références entre elles (éventuellement des cycles). Avec une suppression directe, un tel ensemble serait indestructible sans une gestion spécifique, complexe et coûteuse. En utilisant une poubelle, cette tâche peut être efficacement déléguée à un processus asynchrone.
- Lorsqu'elle est dans la poubelle, une entité n'est pas accessible directement, tout du moins en utilisant les vues standards car comme expliqué ci-dessous, les vues d'administration permettent d'accéder aux instances condamnées. Une application peut cependant montrer cet état dans les entités la référençant avec, par exemple, une icône spécifique.

A.1.2 Deux ensembles d'entités

Durant sa durée de vie, une entité peut donc être dans deux états :

- Vivante
- Condamnée

Quel que soit son état, une entité est toujours cohérente et fonctionnelle, c'est-à-dire que même si elle est condamnée, ses références sortantes et entrantes ne sont pas modifiées. Les mises à jour ne sont appliquées que lors de la destruction effective. L'état « condamnée » n'est donc qu'indicatif.

En correspondance avec ces états, nous distinguons deux ensembles d'entités :

- $Entity$, l'ensemble de toutes les entités vivantes ;
- $EntityBin$, l'ensemble de toutes les entités condamnées.

Ces deux ensembles correspondent à des vues du noyau dans le SGBDR sous-jacent. Les deux incluent tous les champs additionnels normaux (ex. : date de création, ACL[1], etc.).

Nous définissons $EntityAll$ comme l'ensemble de tous les objets du système, tel que :

$$EntityAll = Entity \cup EntityBin$$

$$Entity \cap EntityBin = \emptyset$$

Cet ensemble est aussi disponible en tant que vue dans le SGBDR avec la même information que les autres, plus un champ additionnel « *state* » qui correspond à l'état courant de l'entité : vivante ou condamnée.

D'autre part, nous définissons $EntityDestructionCandidate$ comme un sous-ensemble de $EntityBin$ qui contient tous les objets de $EntityBin$ qui sont disponibles pour suppression. La destruction effective étant asynchrone, la durée pendant laquelle une entité peut rester dans cet état n'est pas connue.

$$EntityDestructionCandidate \subset EntityBin$$

Les règles indiquant si une instance présente dans la poubelle est aussi candidate à la destruction dépendent de l'application. La destruction peut entre autres être déclenchée par un utilisateur, dépendre d'un paramètre de « longévité » minimale, ou encore dépendre du nombre total d'entités dans la poubelle (analogue à la définition de la taille de la poubelle de Windows). $EntityDestructionCandidate$ étant une simple vue sur le SGBDR sous-jacent, il est aisé de l'adapter à tout besoin.

A.1.3 Interactions entre les états

La figure A.1 nous montre les deux états possibles d'une entité et leurs interactions.

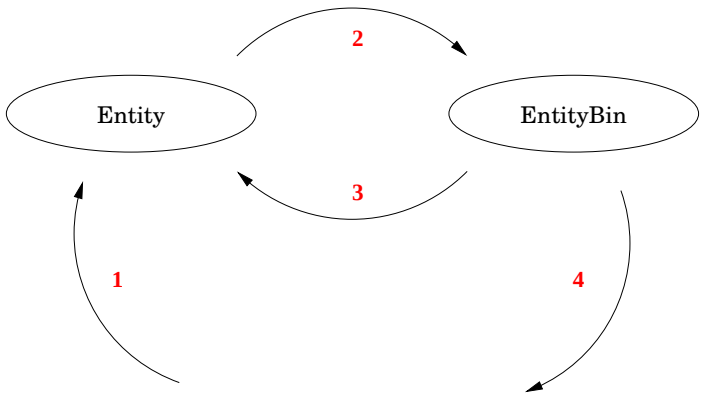

Figure A.1 – Intéractions entre les états d'une entité

Comme nous pouvons l'observer, il existe quatre opérations influant sur l'état d'une entité :

[1] Voir l'annexe C sur la sécurité.

1. `Entity_Create` : Création d'une entité. Elle est directement vivante et disponible dans l'ensemble $Entity$.
2. `Entity_Condemn` : Condamnation d'une entité. Elle n'est pas détruite mais simplement placée dans la poubelle. La présence de références ne bloque pas cette action.
3. `Entity_Restore` : Restauration d'une entité. Elle est de nouveau disponible dans un état cohérent.
4. `Entity_Destroy`: Destruction d'une entité. Cette opération est généralement appelée par un processus asynchrone et non par un utilisateur. Pour être détruite, une entité doit être candidate, c'est-à-dire appartenir au sous-ensemble $EntityDestructionCandidate$, et il ne doit pas y avoir de référence dessus. Sinon, elle reste dans $EntityBin$.

Quelques notes sur la destruction. Plusieurs cas peuvent empêcher la destruction d'une entité :

- L'entité est référencée par une autre instance qui n'est pas candidate à la destruction : elle est indestructible et le restera tant qu'elle est référencée de cette façon.
- L'entité est référencée par une instance elle-aussi candidate à la destruction : elle est indestructible pour l'instant, mais dès que cette autre entité sera détruite (c'est-à-dire que la référence sera supprimée), elle deviendra destructible. La destruction réelle aura donc lieu dans une future passe du processus.
- L'entité fait partie d'un ensemble d'objets ayant des références entre eux (cycle). Comme la gestion de ce cas particulier est potentiellement coûteuse, elle n'est déclenchée que lorsqu'un tel état a une forte probabilité d'exister (par exemple si le nombre d'entités indestructibles est stable entre deux passes du processus de destruction).

A.2 Opérations de base

Cette section décrit les quatre opérations introduites ci-dessus ainsi que les deux opérations les plus fondamentales dans la durée de vie d'une entité : l'attachement et le détachement de facettes.

A.2.1 Attachement et détachement de facettes

Ces deux opérations dirigent la vie des entités, et sont à la base des opérations de création et de destruction. Mais avant de les décrire, introduisons une option du noyau qui a une influence sur leur implémentation[2].

Option `CascadingUnplug`. Cette option est une option d'installation du noyau, désactivée par défaut. Si elle est active, le détachement d'une facette provoque le détachement de toutes ses facettes impliquées (si elles ne sont pas impliquées par d'autres facettes supportées ou attachées explicitement). Pour pouvoir agir ainsi, de nouvelles données sont ajoutées dans la relation contenant la liste des facettes supportées par chaque entité. Pour chaque facette, nous y indiquons le nombre de facettes l'impliquant et si elle est attachée explicitement ou non. La vue devient ainsi `(IdEntity, IdFacet, PluggedCount, Explicit)`, l'attribut `PluggedCount` contenant le nombre d'attachements par implication plus le possible attachement explicite.

[2] Les options `CoPlug` et `Rejection` (cf. section 9.2.7) étant très simples, leur influence n'est pas détaillée dans les opérations pour ne pas alourdir le pseudo-code.

L'activation de cette option met aussi à disposition deux événements du noyau supplémentaires qu'un développeur peut exploiter grâce à des *code snippets* : `OnPreGlobalUnplug` et `OnPostGlobalUnplug` (cf. pseudo code ci-dessous et annexe B).

Opération d'attachement (*plug*). Cette opération attache une facette et ses éventuelles facettes impliquées à une entité spécifiée. Si elle est réussie, toutes ses fonctionnalités sont immédiatement disponibles pour l'entité (les valeurs par défaut sont utilisées à la création).

```
Entity_Plug( IdEntity, IdFacet, IdActor ) [CascadingUnplug=Off]
  • If Facet is already plugged return
  • Begin transaction
    – Execute Kernel.OnPreGlobalPlug( IdEntity, IdFacet, IdActor ) code snippets
    – /* Beginning of the compilable fragment */
    – If the entity supports a facet in rejection with this facet or an implied one,
      return an error
    – Foreach ImpliedFacet (including IdFacet) ordered by PlugLevel desc
      * If ImpliedFacet is not already plugged
      * Execute Kernel.OnPrePlug( IdEntity, IdFacet, IdActor ) code snippets
      * Insert into tF<ImpliedFacet.Name>
      * Execute <ImpliedFacet>.OnPlug( IdEntity, IdActor ) code snippets
      * Insert ImpliedFacet in the Entity FacetList
      * Execute Kernel.OnPostPlug( IdEntity, IdFacet, IdActor ) code snippets
    – /* End of the compilable fragment */
  • End transaction
  • Execute Kernel.OnPostGlobalPlug( IdEntity, IdFacet, IdActor ) code snippets
```

Ce pseudo code s'appuie sur le fait qu'une facette appartient à la liste de ses facettes impliquées. Le *plug level* d'une facette correspond au nombre de facettes qui l'impliquent directement ou indirectement. Plus une facette est profonde dans le graphe, plus son *plug level* est haut.

Le cœur de cette fonction peut aisément être compilé pour chaque facette. En mode compilé, l'opération d'attachement doit simplement associer l'identifiant de la facette avec sa procédure et l'appeler.

L'option `CascadingUnplug` de l'installation du noyau implique un pseudo-code légèrement différent quand elle est activée :

```
Entity_Plug( IdEntity, IdFacet, IdActor ) [CascadingUnplug=On]
  • If Facet is already plugged explicitly return
  • Begin transaction
    – Execute Kernel.OnPreGlobalPlug( IdEntity, IdFacet, IdActor ) code snippets
    – /* Beginning of the compilable fragment */
    – If the entity supports a facet in rejection with this facet or an implied one,
      return an error
    – Foreach ImpliedFacet (including IdFacet) ordered by PlugLevel desc
      * If ImpliedFacet is already plugged
        · Increment ImpliedFacet PluggedCount
      * else
      * Execute Kernel.OnPrePlug( IdEntity, IdFacet, IdActor ) code snippets
      * Insert into tF<ImpliedFacet.Name>
      * Execute <ImpliedFacet>.OnPlug( IdEntity, IdActor ) code snippets
      * Insert ImpliedFacet in the Entity FacetList
      * Execute Kernel.OnPostPlug( IdEntity, IdFacet, IdActor ) code snippets
    – /* End of the compilable fragment */
    – Declare IdFacet as explicitly plugged
  • End transaction
  • Execute Kernel.OnPostGlobalPlug( IdEntity, IdFacet, IdActor ) code snippets
```

Opération de détachement (*unplug*). Cette opération détache une facette d'une entité spécifiée. Les éventuelles facettes impliquées ne sont détachées que si l'option `CascadingUnplug` est activée et si elle ne sont plus utiles à l'entité.

```
Entity_Unplug( IdEntity, IdFacet, IdActor ) [CascadingUnplug=Off]
  • If Facet is not plugged return
  • Begin transaction
    – Execute Kernel.OnPreUnplug( IdEntity, IdFacet, IdActor ) code snippets
    – If the entity supports a facet implying the specified one return an error
    – Execute <Facet>.OnUnplug( IdEntity, IdActor ) code snippets
    – Delete from tF<Facet.Name>
    – Delete from the Entity FacetList
  • End transaction
  • Execute Kernel.OnPostUnplug( IdEntity, IdFacet, IdActor ) code snippets
```

S'il existe une référence sur l'instance (*via* la facette détachée), un *rollback* de la transaction remettra le système dans son état préalable.

Le cas des facettes principales est particulier. En effet, si la facette détachée est la dernière facette principale, l'entité n'a plus de raison d'exister. Une option de l'installation du noyau, `LastMainFacetUnplug`, indique si le détachement de la dernière facette principale est autorisé (vrai par défaut). Si c'est le cas, l'entité est automatiquement détruite. Sinon, le détachement est interdit et une erreur est retournée. Le seul moyen de détruire une entité est alors d'appeler directement l'opération `Entity_Destroy`.

La gestion de ces cas est réalisée par les *code snippets* `PreUnplug` et `PostUnplug` (cf. annexe B) qui sont différents selon la configuration.

```
Entity_Unplug( IdEntity, IdFacet, IdActor ) [CascadingUnplug=On]
  • If Facet is not plugged return
  • Begin transaction
    – Execute Kernel.OnPreGlobalUnplug( IdEntity, IdFacet, IdActor ) code snippets
    – If IdFacet PluggedCount > Explicit (indicates that this facet is needed by another
      one) return an error
    – /* Beginning of the compilable fragment */
    – Foreach ImpliedFacet (including IdFacet) ordered by PlugLevel asc
      * Decrement ImpliedFacet PluggedCount
      * If ImpliedFacet PluggedCount = 0
      * Execute Kernel.OnPreUnplug( IdEntity, IdFacet, IdActor ) code snippets
      * If the entity supports a facet implying the specified one return an error
      * Execute <Facet>.OnUnplug( IdEntity, IdActor ) code snippets
      * Delete from tF<Facet.Name>
      * Delete from the Entity FacetList
      * Execute Kernel.OnPostUnplug( IdEntity, IdFacet, IdActor ) code snippets
    – /* End of the compilable fragment */
  • End transaction
  • Execute Kernel.OnPostGlobalUnplug( IdEntity, IdFacet, IdActor ) code snippets
```

Pour éviter toute confusion, une tentative de détachement d'une facette impliquée par une autre facette supportée retourne toujours une erreur. Prenons une facette attachée à la fois explicitement et en conséquence d'implications. Si le détachement explicite de cette facette ne faisait qu'enlever la partie « explicite » d'une facette, l'utilisateur pourrait avoir du mal à comprendre le fait qu'elle soit encore présente.

A.2.2 Création d'entités

Une entité est toujours créée avec une facette principale.

Entity_Create(IdMainFacet, IdActor, IdEntity output)
- Begin transaction
  - Execute Kernel.OnPreCreate(IdMainFacet, IdActor) code snippets
  - Create the entity and obtain the new IdEntity.
  - Execute Entity_Plug(IdEntity, IdMainFacet, IdActor)
  - Execute <MainFacet>.OnCreate(IdEntity, IdActor) code snippets
- End transaction
- Execute Kernel.OnPostCreate(IdMainFacet, IdEntity, IdActor) code snippets

A.2.3 Condamnation et restauration d'entités

Comme nous l'avons vu, il est possible de restaurer une entité condamnée. La condamnation d'un objet ne doit donc jamais déclencher d'action irréversible sur les données. Il est toutefois possible d'utiliser les *code snippets* des opérations de condamnation et de restauration en concordance pour mettre à jour des informations.

Condamnation. La fin du cycle de vie d'une entité commence avec un appel à `Entity_Condemn`. Une fois celle-ci condamnée, toutes ses données deviennent indisponibles puisque leur accès doit passer par la vue correspondant à l'ensemble $Entity$ présenté en section A.1.2. Rappelons que les objets condamnés ne sont pas dans $Entity$ mais dans $EntityBin$.

Dans la majorité des paradigmes à objets, les instances peuvent être subordonnées à d'autres quant à leur création et destruction. En effet, un objet appartenant à un espace de nommage hiérarchique aura souvent besoin de supprimer ses fils lorsqu'il sera détruit. Cependant, cette approche peut être très coûteuse. Cette implémentation n'est pas performante dès que les ensembles à supprimer sont importants, comme c'est souvent le cas dans les applications de gestion des connaissances.

Nous préférons ainsi sortir le problème de condamnation récursive en dehors du noyau, et laisser l'application décider de sa stratégie. Cette approche est courante. À titre d'illustration, la suppression d'un dossier non-vide est généralement impossible en utilisant l'API bas niveau d'un système de fichier. C'est un « *helper* », outil ou API de plus haut niveau, qui itère sur le contenu et demande la destruction fichier par ficher, puis détruit le dossier lui-même une fois qu'il est vide.

Entity_Condemn(IdEntity, IdActor)
- If entity is already condemned return
- Begin transaction
  - Execute Kernel.OnPreCondemn(IdEntity, IdActor) code snippets
  - Foreach plugged Facet ordered by PlugLevel asc
    - Execute <Facet>.OnCondemn(IdEntity, IdActor)
  - Internally condemn the entity (hide it from public views)
- End transaction
- Execute Kernel.OnPostCondemn(IdEntity, IdActor) code snippets

Si dans un *code snippet* `OnCondemn` donné, un programmeur veut condamner un autre objet, il peut appeler l'opération `Entity_Condemn` récursivement. Cependant, cette approche supporte mal les accroissements d'échelle et doit être évitée.

Restauration. L'opération de restauration permet l'annulation d'une condamnation. Si l'opération de condamnation n'a pas déclenché de modification irréversible dans la base de données, la restauration remet l'entité spécifiée dans son état précédent.

Entity_Restore(IdEntity, IdActor)
- If entity is not condemned return

- Begin transaction
  - Execute Kernel.OnPreRestore(IdEntity, IdActor) code snippets
  - Foreach plugged Facet ordered by PlugLevel desc
    - Execute <Facet>.OnRestore(IdEntity, IdActor)
  - Internally restore the entity
- End transaction
- Execute Kernel.OnPostRestore(IdEntity, IdActor) code snippets

Si des actions spécifiques étaient réalisées dans des *codes snippets* `OnCondemn`, elles doivent avoir leur actions opposées dans les *code snippets* `OnRestore`.

A.2.4 Destruction d'entités

Cette opération réalise la destruction effective des entités. C'est à cette étape que les détachements sont effectués. `Entity_Destroy` est généralement appelée par un processus asynchrone sur les entités de $EntityDestructionCandidate$, mais peut être déclenchée manuellement par un administrateur. Dans ce cas, l'entité n'a pas besoin d'avoir été préalablement condamnée.

`Entity_Destroy( IdEntity, IdActor )`

- Begin transaction
  - Execute Kernel.OnPreDestroy(IdEntity, IdActor) code snippets
  - Foreach plugged Facet ordered by PlugLevel asc
    - Execute Entity_Unplug(IdEntity, IdFacet, IdActor)
  - Internally destroy the entity
- End transaction
- Execute Kernel.OnPostDestroy(IdEntity, IdActor) code snippets

Si l'option d'installation du noyau `CascadingUnplug` est active, seules les facettes qui ont été attachées explicitement sont sélectionnées pour être détachées. En effet, les autres seront alors détachées récursivement.

ANNEXE B

Code snippets

Les opérations contrôlant le cycle de vie des entités (création, attachement / détachement de facettes, destruction) sont créées lors de l'installation du noyau de DOAN (cf. annexe A). Elles sont alors figées et il est difficile de les modifier ensuite. Pour permettre au concepteur de changer leur comportement selon la situation, nous permettons l'ajout de traitements spécifiques sous la forme de code snippets *(ex. : tests de préconditions / postconditions, journalisation, ou toute autre action).*

Les actions représentées par les code snippets *sont exécutées lorsque des événements particuliers sont déclenchés, comme par exemple l'attachement d'une nouvelle facette à une entité. Ce mécanisme se rapproche de ce qui a été fait pour les frames avec les* dæmons *(cf. section 6.1.1) ou encore de la programmation par aspects [61, 72].*

Dans cet annexe, nous présentons les événements pouvant être contrôlés par des code snippets*, et les moyens disponibles au programmeur pour ce faire.*

B.1 Événements du système

Comme le détaille l'annexe A, plusieurs opérations peuvent être effectuées sur les entités : création, condamnation, restauration, destruction, attachement et détachement de facettes. L'exécution de chacune de ces opérations déclenche plusieurs événements décrits ci-dessous.

B.1.1 Noyau

- **`OnPreCreate`** : Cet événement est déclenché avant la création d'une nouvelle entité.
- **`OnPostCreate`** : Cet événement est déclenché après la création d'une nouvelle entité. Attention, l'entité est déjà créée et la transaction est terminée. Il est alors impossible de s'appuyer sur cet événement pour exécuter un *rollback* de la transaction de création.
- **`OnPreCondemn`** : Cet événement est déclenché avant la condamnation d'une entité.
- **`OnPostCondemn`** : Cet événement est déclenché après la condamnation d'une entité. Attention, l'entité est déjà condamnée et la transaction est terminée. Il est alors impossible de s'appuyer sur cet événement pour exécuter un *rollback* de la transaction de condamnation.
- **`OnPreRestore`** : Cet événement est déclenché avant la restauration d'une entité.
- **`OnPostCondemn`** : Cet événement est déclenché après la restauration d'une entité. Attention, l'entité est déjà restaurée et la transaction est terminée. Il est alors impossible de s'appuyer sur cet événement pour exécuter un *rollback* de la transaction de restauration.
- **`OnPreDestroy`** : Cet événement est déclenché avant la destruction d'une entité.
- **`OnPostDestroy`** : Cet événement est déclenché après la destruction d'une entité. Attention, l'entité est déjà supprimée et la transaction est terminée. Il est alors impossible de s'appuyer sur cet événement pour exécuter un *rollback* de la transaction de destruction.

- **`OnPrePlug`** : Cet événement est déclenché avant l'attachement d'une facette à une entité.
- **`OnPostPlug`** : Cet événement est déclenché après l'attachement d'une facette à une entité. Contrairement aux autres événements « post opération », celui-ci intervient à l'intérieur de la transaction. En effet, plusieurs facettes peuvent être attachées récursivement si des contraintes d'implication sont présentes. L'événement `OnPostGlobalPlug` intervient, quant à lui, après tous les attachements et une fois la transaction terminée.
- **`OnPreGlobalPlug`** : Cet événement est déclenché avant l'attachement d'un ensemble de facettes à une entité. Cet ensemble peut être réduit à une seule facette si elle n'en implique pas d'autres. Alors que l'événement `OnPrePlug` est appelé pour chaque facette, celui-ci n'est appelé qu'une fois.
- **`OnPostGlobalPlug`** : Cet événement est déclenché après l'attachement d'un ensemble de facettes à une entité. Attention, contrairement à `OnPostPlug`, la transaction est déjà terminée lors de l'appel à cet événement.
- **`OnPreUnplug`** : Cet événement est déclenché avant le détachement d'une facette d'une entité.
- **`OnPostUnplug`** : Cet événement est déclenché après le détachement d'une facette d'une entité. Selon que l'option `CascadingUnplug` est été choisie ou non lors de l'installation du noyau, cet événement n'a pas le même comportement. Si le `CascadingUnplug` est désactivé, seule une facette à la fois peut être détachée et cet événement intervient alors après la transaction de détachement. Dans le cas contraire, il fonctionne comme `OnPostPlug`, c'est-à-dire qu'il est déclenché avant la fin de la transaction.
- **`OnPreGlobalUnplug`** : Cet événement n'est disponible que si l'option `CascadingUnplug` est activée. Il est déclenché avant le détachement d'un ensemble de facettes d'une entité.
- **`OnPostGlobalUnplug`** : Cet événement n'est disponible que si l'option `CascadingUnplug` est activée. Il est similaire à `OnPostGlobalPlug` mais est déclenché pour le détachement.

La lecture de l'annexe A permet de comprendre avec plus de précision quand ces événements sont déclenchés, et notamment l'impact de l'option `CascadingUnplug` de l'installation du noyau.

B.1.2 Facettes

Les événements suivant sont spécifiques à chaque facette. Ils ne sont déclenchés que pour la facette à laquelle ils correspondent.

- **`OnPlug`** : Cet événement est déclenché lorsque la facette est attachée à une entité.
- **`OnUnplug`** : Cet événement est déclenché lorsque la facette est détachée d'une entité.
- **`OnCondemn`** : Cet événement est déclenché lorsque qu'une entité supportant la facette est condamnée.
- **`OnRestore`** : Cet événement est déclenché lorsque qu'une entité supportant la facette est restaurée.

B.1.3 Facettes principales

Les facettes principales ont tous les événements des facettes plus un événement spécifique intervenant à la création d'une entité.

- **`OnCreate`** : Cet événement est déclenché lorsqu'une entité est créée en utilisant cette facette principale.

B.2 Code Snippets

Pour chacun des événements décrits dans la section précédente, le système exécute une procédure stockée dédiée. Ce sont ces procédures dont nous allons permettre la modification par le concepteur ou le programmeur.

Plusieurs besoins orthogonaux peuvent nécessiter la modification des procédures associées aux événements (ex. : journalisation et sécurité). Aussi, plutôt que de demander la modification explicite de ces procédures, nous nous appuyons sur un mécanisme de *code snippets* que nous concaténons pour les créer.

B.2.1 Définition

Un *code snippet* est un triplet $(source, fragment, content)$ où
- la $source$ identifie la « provenance » du morceau de code (ex. : le besoin ayant conduit à sa création, ou encore le nom du programmeur) ;
- le $fragment$ identifie le morceau de code pour la $source$ spécifiée (ex. : le numéro du *code snippet*) ;
- le $content$ correspond au code (SQL) devant être exécuté.

Dans le contexte d'un événement particulier, chaque paire $(source, fragment)$ doit être unique. Pour chaque événement, une liste de *code snippets* est maintenue à l'aide d'une API dédiée.

Si un *code snippet* détecte une erreur nécessitant l'annulation de l'opération en cours, il doit alors retourner une erreur à l'aide de l'opération système `sEntity_ErrorAdd` (cette fonction annule la transaction courante).

Chaque opération correspondant à un événement a une signature implicite donnant accès au contexte d'exécution. Ce contexte contient des variables ayant une sémantique précise, comme par exemple `IdActor` qui correspond à l'acteur, humain ou logiciel, exécutant le code. Le contenu du contexte, et donc la signature de l'opération, dépend de l'événement. `OnPostCreate`, par exemple, fournir l'identifiant de la nouvelle entité (`IdEntity`), celui de la facette principale ayant servi à la créer (`IdMainFacet`) et celui de l'acteur ayant provoqué l'opération (`IdActor`). Si des informations nécessaires ne sont pas disponibles dans le contexte d'exécution, elles doivent être obtenues à partir de ce dernier avec des instructions SQL.

Pour éviter l'exécution dynamique de tous les *code snippets* à chaque événement, il est possible de les compiler dans une procédure stockée. Dans une telle configuration, l'ajout d'un nouveau *code snippet* provoque la recompilation de tous les *code snippets* de l'événement.

B.2.2 Exemples

Choisissons de représenter un *code snippet* comme suit[1] :

$Source$	$Fragment$
`Content`	

Un *code snippet* peut être très simple comme par exemple l'enregistrement dans un journal d'une opération de création :

[1] Le code SQL des exemples de cette section correspond à la version SQL Server 2000 du noyau.

Logging	1

```
exec sLogging_Add 'info', Date(), 'actor '
      + @idActor + ' try to create a new entity with main facet ' + @idMainFacet
```

Il peut aussi être plus complexe et effectuer plusieurs opérations. L'attachement d'une facette peut, par exemple, être conditionné aux valeurs d'une facette impliquée :

PreConditions	*SalaryCheck*

```
declare @salary int
select @salary = Salary from tEmployee where IdEntity = @idEntity
if (@salary < 30000)
begin
      declare @idError int
      exec @idError = sEntity_ErrorAdd( 0, @idActor, @idEntity, @idFacet,
            "sManager_OnPlug", "An employee cannot become a manager if his/her income
            is less than 30,000 euros."
      return @idError
end
```

De manière générale, toute opération peut être exécutée dans un *code snippet*. Pour éviter tout état inconsistant, il est cependant fortement déconseillé de mettre à jour les tables du noyau et les tables générées automatiquement par l'algorithme de traduction des types. En cas de besoin, le concepteur peut créer des tables annexes et indépendantes du système.

B.3 API SQL

Cette API est générée automatiquement lors de l'installation du noyau du système. Les types utilisés pour *source*, *fragment* et *content* dépendent du SGBD sous-jacent. Dans le cas de SQL Server 2000, nous utilisons `nvarchar(64)` pour la source et le fragment, et `nvarchar(4000)` pour le *content*. Si le code d'un *snippet* dépasse 4000 caractères, il doit être divisé en plusieurs fragments qui seront concaténés lors de la compilation de l'opération.

Si `<Name>` désigne `Kernel`, `Facet` ou `MainFacet` selon l'événement concerné, pour chaque événement `<Event>` du noyau, d'une facette ou d'une facette principale, la vue suivante donne accès à ses *code snippets* :

Nom	Champs	
`<Name>_<Event>CS`	`Index`	Position du *code snippet* dans la liste
	`Source`	Source du *code snippet*
	`Fragment`	Fragment du *code snippet*
	`Content`	*Content* du *code snippet*

Les opérations suivantes permettent de manipuler et mettre à jour les *code snippets* :

Nom	Paramètres	
`s<Name>_<Event>CS_Add`	`Source`	Source du nouveau *code snippet*
	`Fragment`	Fragment du nouveau *code snippet*
	`Content`	*Content* du nouveau *code snippet*
	`IdActor`	L'acteur effectuant l'opération

Cette procédure ajoute le morceau de code spécifié par le triplet (`Source`, `Fragment`, `Content`) à la fin de la liste de *snippets* l'événement. Si un morceau de code existe déjà pour les `Source` et `Fragment` spécifiés, il est tout d'abord supprimé.

Nom	Paramètres	
`s<Name>_<Event>CS_Remove`	`Source`	Source du *code snippet* à supprimer
	`Fragment`	Fragment du *code snippet* à supprimer
	`IdActor`	L'acteur effectuant l'opération

Cette procédure supprime le morceau de code spécifié.

Nom	Paramètres	
`s<Name>_<Event>CS_AddBeforeSource`	`BSource`	La source devant laquelle ajouter le nouveau *code snippet*
	`Source`	Source du nouveau *code snippet*
	`Fragment`	Fragment du nouveau *code snippet*
	`Content`	*Content* du nouveau *code snippet*
	`IdActor`	L'acteur effectuant l'opération

Cette procédure insère le morceau de code spécifié par le triplet (`Source`, `Fragment`, `Content`) avant tout *snippet* de la source `BSource`. S'il n'y a aucun *code snippet* de source `BSource`, il est ajouté à la fin de la liste. Si un morceau de code existe déjà pour les `Source` et `Fragment` spécifiés, il est tout d'abord supprimé.

Nom	Paramètres	
`s<Name>_<Event>CS_AddBeforeFragment`	`BSource`	La source devant laquelle ajouter le nouveau *code snippet*
	`BFragment`	Le fragment devant lequel ajouter le nouveau *code snippet*
	`Source`	Source du nouveau *code snippet*
	`Fragment`	Fragment du nouveau *code snippet*
	`Content`	*Content* du nouveau *code snippet*
	`IdActor`	L'acteur effectuant l'opération

Cette procédure fonctionne comme la précédente, mais permet en plus de spécifier le fragment devant lequel est ajouté le morceau de code.

Nom	Paramètres	
`s<Name>_<Event>CS_RemoveSource`	`Source`	Source du *code snippet* à supprimer
	`IdActor`	L'acteur effectuant l'opération

Cette procédure supprime tous les *code snippets* de source `Source` de la liste.

Nom	Paramètres	
`s<Name>_<Event>CS_Clear`	`IdActor`	L'acteur effectuant l'opération

Cette procédure supprime tous les *code snippets* de la liste.

ANNEXE C

Sécurité

La sécurité des données est un critère indispensable à toute application critique. Si une sécurité forte n'est pas disponible pour protéger l'information, beaucoup d'acteurs dans l'entreprise pourraient refuser d'utiliser l'application, quels que soient ses apports. La sécurité des données est particulièrement importante quand l'application est déployée et utilisée par de nombreux utilisateurs ayant de nombreux rôles différents.

Pour répondre à ce besoin, nous avons décidé d'intégrer la sécurité directement dans le noyau du système, de façon à pouvoir associer des attributs de sécurité à toute instance à moindre coût.

Capitalisant sur un standard industriel largement utilisé, nous avons choisi d'implémenter le concept de liste de contrôle d'accès (ACL, Access Control Lists*) [85, 9]. Une ACL est simplement une liste d'entrées (ACE,* Access Control Entries*), chacune attribuant ou refusant une permission à un acteur. Ainsi, chaque entité du système a une ACL qui définit les acteurs pouvant y accéder et leurs droits.*

Cette annexe présente en premier lieu le modèle de sécurité que nous avons choisi avant de détailler son implémentation avec l'approche ACL. Nous terminerons en examinant comment une application doit interroger le système pour tenir compte de la sécurité.

C.1 Modèle de sécurité

C.1.1 Acteurs

La notion d'*acteur* correspond, dans le modèle de sécurité, à tout ce qui peut agir sur le système. L'exemple le plus typique d'acteur est un utilisateur de l'application. Mais plutôt que de se concentrer sur les utilisateurs, la notion plus ouverte d'acteur permet de contrôler l'accès d'autres entités, tels que les processus.

Dans notre modèle, les acteurs sont simplement des entités supportant la facette ACTOR, alors que les utilisateurs sont des entités supportant la facette USER. Puisqu'un utilisateur peut toujours être vu comme un acteur, nous avons l'implication suivante :

USER **implies** ACTOR

La facette USER contient des propriétés supplémentaires inutiles aux acteurs, comme un *login* par exemple.

Si le noyau du système se satisfait de la notion d'acteur, l'implémentation de DOAN propose aussi la notion de groupe aux applications. Elle simplifie grandement la gestion de la sécurité. Comme pour les deux concepts introduits précédemment, nous créons une facette ACTORGROUP. Les groupes d'acteurs sont organisés à l'aide d'un graphe acyclique orienté (DAG, *Directed Acyclic Graph*) permettant aux groupes d'être composés d'autres groupes, mais interdisant tout cycle. Ainsi, un groupe peut contenir aussi bien des sous-groupes que directement des acteurs.

Un acteur et un groupe ont un rôle particulier : l'acteur *anonymous* et le groupe *everyone*. *Anonymous* est l'acteur par défaut et est utilisé par tous les utilisateurs non authentifiés ; il a des droits minimaux sur le système. Il n'y aucun sens à lui attribuer des droits spécifiques. *Everyone*, comme son nom l'indique, contient tous les acteurs du système. Lui attribuer un droit revient à l'attribuer à tous les acteurs, y compris *anonymous*.

De plus, pour simplifier l'administration du système, nous introduisons le groupe *administrators*. Les acteurs de ce groupe ont un contrôle total sur toutes les instances du système. La plupart des opérations d'administration du système sont en outre restreintes à ce groupe. Sur le même principe, l'utilisateur *administrator*, membre du groupe *administrators*, est automatiquement créé lors l'installation du noyau. Cet utilisateur et ce groupe ne peuvent être détruits. Ceci nous assure qu'il existe au moins un utilisateur capable de faire toutes les opérations à tout moment.

Il existe, d'autre part, un acteur *system*, appartenant lui aussi au groupe *administrators*. Cet acteur n'est pas un utilisateur, il représente simplement le système lui-même. C'est le propriétaire de tous les objets du noyau.

Bien que cela puisse paraître plus simple, nous interdisons qu'un groupe soit considéré comme un acteur. Pour l'intérêt de la traçabilité, nous imposons que des acteurs individuels, et non des groupes, exécutent les actions du système. Quelques opérations peuvent cependant agir indifféremment sur des acteurs ou des groupes. Nous introduisons pour cela une facette dédiée, ACTORORACTORGROUP, et les implications correspondantes :

ACTOR **implies** ACTORORACTORGROUP

ACTORGROUP **implies** ACTORORACTORGROUP

C.1.2 Droits

Un *droit* est une permission nécessaire à un acteur pour effectuer une action sur un objet. Le droit `condemn` par exemple autorise l'acteur à condamner l'entité.

Certains droits peuvent en requérir d'autres. Par exemple, le droit `modify` requiert le droit `view` : pour pouvoir modifier une entité, un acteur doit d'abord pouvoir la voir. Pour répondre à ce besoin, nous organisons les droits sous la forme d'un DAG : attribuer le droit `modify` à un acteur lui attribue automatiquement le droit `view`. D'autres systèmes utilisent parfois des paquets de droits. Par exemple, un paquet `edit` peut inclure les droits `discover`, `view` et `modify`. Ce comportement est aisément simulé à l'aide d'implications dans le dag. Pour créer un paquet, il suffit de créer un nouveau droit impliquant les droits le composant.

La figure C.1 montre le dag par défaut des droits. Le droit `full-control` implique récursivement tous les autres droits du système. Comme son nom l'indique, il permet aux acteurs d'effectuer toutes les opérations possibles sur les entités. Si un nouveau droit est créé, une implication depuis `full-control` est automatiquement ajoutée.

Les droits `view rigths` et `grant rights` ont un comportement spécifique. Pour éviter tout risque, ils permettent respectivement de voir et d'attribuer uniquement les droits que l'acteur effectuant l'action a déjà. De plus, pour supprimer un droit à un acteur, le `full-control` de l'entité est nécessaire.

Le propriétaire d'une entité a automatiquement le droit `full-control` dessus. Ce droit est également nécessaire pour pouvoir changer le propriétaire d'une entité.

Pour plus de simplicité, nous avons décidé de ne gérer les droits que de manière additive. Il est ainsi impossible de refuser explicitement un droit à un acteur, c'est-à-dire qu'il est impossible d'accorder un

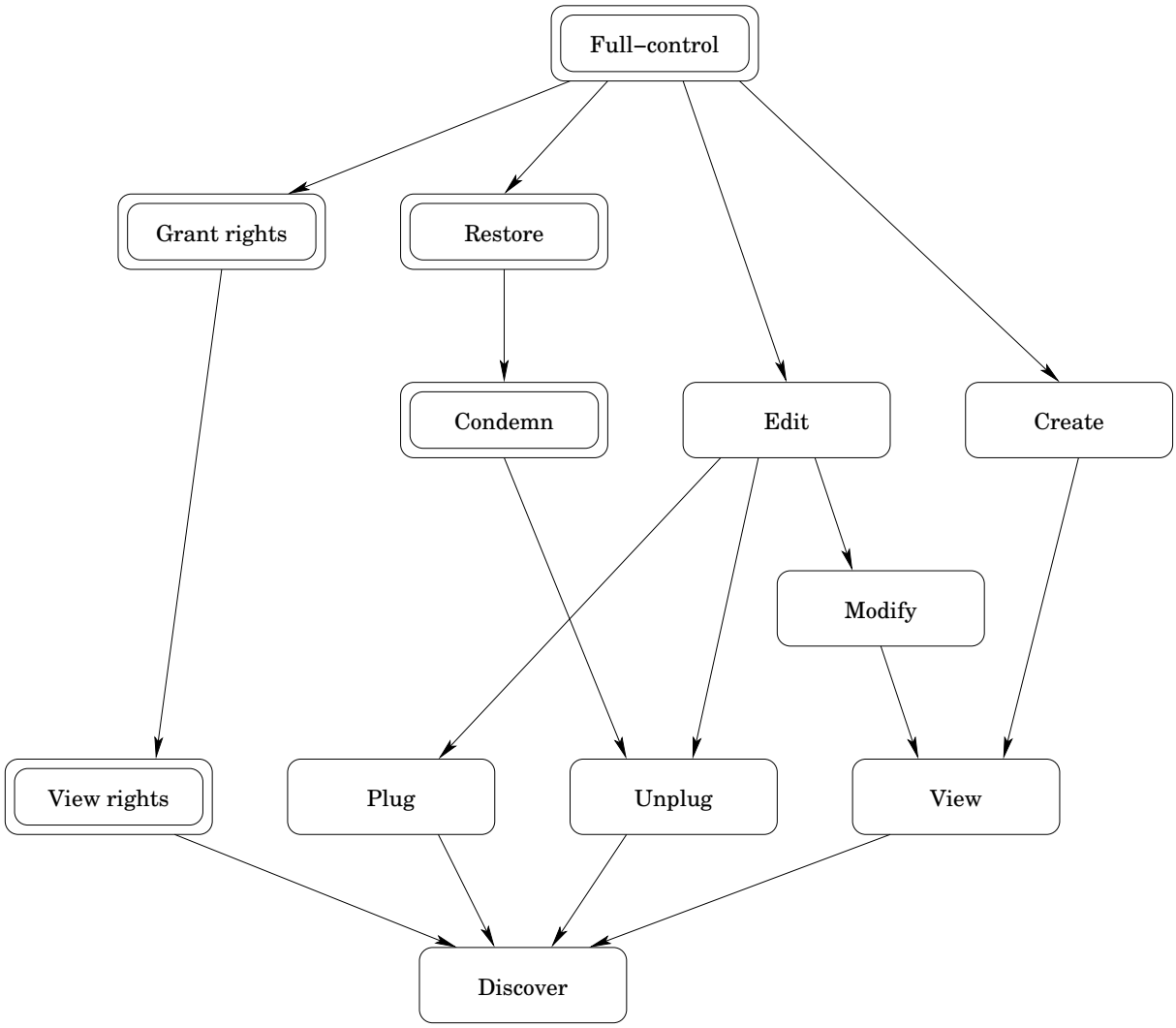

Figure C.1 – Graphe des droits du système

droit à un groupe d'acteurs à l'exception d'un sous-ensemble de celui-ci. Si un tel comportement est nécessaire, il faut créer un sous-groupe dédié et lui donner explicitement le droit voulu.

Spécialisation des droits. Comme nous venons de le voir, les droits sont utilisés pour contrôler ce que chaque acteur peut faire avec les entités. Cependant, ce modèle de sécurité n'est pas suffisamment précis : l'utilisateur peut avoir besoin de restreindre certains droits à des facettes spécifiques. Par exemple, il peut vouloir accorder à un acteur le droit `view` sur la facette AUTEUR d'une entité, mais pas sur sa facette EMPLOYÉ. S'il est possible de restreindre des droits à des facettes, ces derniers concernent cependant par défaut toutes les facettes supportées, simplifiant ainsi la gestion des droits dans une configuration « standard », tout en permettant un paramétrage précis de la sécurité en cas de besoin.

Pour plus de simplicité, si les droits peuvent être limités à des facettes spécifiques, ils ne peuvent pas leur être dédiés. Imaginons, par exemple, qu'un concepteur crée le droit `run`, qui contrôle quels acteurs peuvent exécuter un processus représenté dans le système par une entité supportant la facette PROCESSUS. Ce nouveau droit n'est pas réduit à cette facette et peut être appliqué à toutes les facettes du système.

De plus, certains droits ne peuvent pas être spécialisés à des facettes. C'est le cas de `full-control`, `condemn`, `restore`, `view rights` et `grant rights` (ils sont représentés dans la figure C.1 par une double ligne). `Full-control` implique, par définition, tous les droits sur une entité, ce qui suppose que toutes les facettes sont concernées. Par ailleurs, les droits `condemn` et `restore` concernent le cycle de vie de l'instance dans son entier. Pour condamner une entité, le droit de détacher toutes les facettes est nécessaire. Cela n'a donc pas de sens d'attribuer le droit `condemn` pour seulement une partie des facettes supportées. Qui plus est, cela traduirait un trou de sécurité : le refus de la condamnation par le système indiquerait que l'entité supporte des facettes non accessibles à l'acteur. Pour les autres droits, la simplicité guide nos choix : voir et accorder des droits (`view rights` et `grant rights`) pour une sous-partie des facettes serait très complexe.

Droits et cycle de vie des entités. Observons maintenant l'influence du modèle de sécurité sur la gestion du cycle de vie des entités. Nous avons introduit des droits `plug` et `unplug` dans le dag des droits. Ils sont simplement utilisés par les opérations d'attachement et de détachement. Ainsi, pour attacher une nouvelle facette à une entité par exemple, un acteur a besoin de posséder le droit `plug` pour cette facette sur l'entité. Implicitement, l'acteur doit aussi être capable de voir la facette et donc de posséder le droit `view` sur l'instance de la facette elle-même[1].

Les opérations de condamnation et de restauration fonctionnent de manière similaire. Cependant, comme indiqué plus haut, ces droits concernent forcément les entités dans leur entier.

Nous avons décrit jusqu'ici les droits permettant de contrôler les actions d'une entité existante. La création de nouvelles entités suit des règles différentes. En effet, n'existant pas encore, elles ne peuvent pas être contrôlées par des droits. La création d'une nouvelle entité nécessite une facette principale. En conséquence, nous allons la conditionner à la possession du droit `create` sur l'instance de la facette principale elle-même.

C.1.3 Opérations

Définissons tout d'abord les actions fondamentales nécessaires au modèle de sécurité.

[1] Le même modèle de sécurité peut être utilisé pour les instances du modèle et celles du méta-modèle.

Attribuer un droit. Un acteur peut attribuer un droit seulement s'il possède le droit `grant right` pour l'entité et si lui-même possède le droit qu'il veut accorder. Les fonctions suivantes décrivent cette action dans les cas où le droit est restreint à une facette et où toutes les facettes de l'entité sont concernées.

$$Grant : \begin{array}{rcl} Actor \times Actor \times Right \\ \times Entity \times Facet & \rightarrow & boolean \\ (a1, a2, r, e, f) & \mapsto & \end{array}$$

```
if (isGranted(a1, grant rights, e)
  and (isGranted(a1, r, e, f))
{
   create the grant entry (a2, r, f) for e
   return true
}
return false
```

$$Grant : \begin{array}{rcl} Actor \times Actor \times Right \\ \times Entity & \rightarrow & boolean \\ (a1, a2, r, e) & \mapsto & \end{array}$$

```
if (isGranted(a1, grant rights, e)
  and (isGranted(a1, r, e))
{
   create the grant entry (a2, r) for e
   return true
}
return false
```

Supprimer d'un droit. La suppression d'un droit fonctionne sur le même principe. En revanche le contrôle total de l'entité est alors requis.

$$Remove : \begin{array}{rcl} Actor \times Actor \times Right \\ \times Entity \times Facet & \rightarrow & boolean \\ (a1, a2, r, e, f) & \mapsto & \end{array}$$

```
if (isGranted(a1, full-control, e)
{
   remove the entry (a2, r, f) for e
   return true
}
return false
```

$$Remove : \begin{array}{rcl} Actor \times Actor \times Right \\ \times Entity & \rightarrow & boolean \\ (a1, a2, r, e) & \mapsto & \end{array}$$

```
if (isGranted(a1, full-control, e)
{
   remove the entry (a2, r) for e
   return true
}
return false
```

Fonctions utilitaires. Les fonctions suivantes permettent de déterminer si un acteur possède un droit particulier pour une entité. Elles sont entre autres utilisées par celles que nous venons d'introduire. Par ailleurs,

- La fonction `own` renvoie `true` si l'acteur spécifié est le propriétaire de l'objet.
- La fonction `administrator` renvoie `true` si l'acteur spécifié est membre du groupe *administrators*.

$$isGranted : \begin{array}{l} Actor \times Right \\ \quad \times Entity \times Facet \\ (a, r, e, f) \end{array} \begin{array}{c} \rightarrow \\ \\ \mapsto \end{array} \begin{array}{c} boolean \\ \\ \texttt{true if own(a, e)} \\ \texttt{true if administrator(a)} \\ \texttt{true if } \exists \texttt{ grant entry (a, r, f) for e} \\ \texttt{true if } \exists \texttt{ implied grant entry} \\ \quad \texttt{(a, r, f) for e} \\ \texttt{false otherwise (implicit deny)} \end{array}$$

$$isGranted : \begin{array}{l} Actor \times Right \times Entity \\ (a, r, e) \end{array} \begin{array}{c} \rightarrow \\ \\ \mapsto \end{array} \begin{array}{c} boolean \\ \\ \texttt{true if own(a, e)} \\ \texttt{true if administrator(a)} \\ \texttt{true if } \exists \texttt{ grant entry (a, r) for e} \\ \texttt{true if } \exists \texttt{ implied grant entry} \\ \quad \texttt{(a, r) for e} \\ \texttt{false otherwise (implicit deny)} \end{array}$$

C.2 Implémentation relationnelle

Observons maintenant comment nous avons traduit ces besoins dans le modèle relationnel.

C.2.1 ACL (*Access Control Lists*) et ACE (*Access Control Entries*)

Une implémentation basique du modèle de sécurité introduit ci-dessus résulterait en un système très peu réactif. Bien qu'elle serait suffisante pour un faible nombre d'instances, l'augmentation de ce dernier engorgerait très vite le système. Pour permettre une plus grande réactivité, nous faisons l'hypothèse que dans le cadre d'un usage normal, de nombreuses instances partagent le même comportement de sécurité. Typiquement, une application a besoin de seulement trois définitions de sécurité différentes pour toutes les entités d'un même utilisateur : *privée*, *publique*, et éventuellement *partagée* dans une communauté.

À partir de cette hypothèse, nous avons choisi d'adopter les notions couramment utilisées d'ACL et de d'ACE [85, 9] pour implémenter notre modèle de sécurité. Chaque entité est ainsi associée à une unique ACL, c'est-à-dire une simple liste d'ACE. Une ACE est un triplet

$$(\text{ACTORORACTORGROUP}, Right, Facet)$$

La facette est une valeur optionnelle : si elle est omise, l'ACE concerne toutes les facettes du système.

La bonne réactivité du système repose sur le fait que de nombreux objets peuvent partager la même ACL. En guise d'illustration, nous pouvons observer que les applications organisent généralement leurs entités dans un ou plusieurs espace(s) de noms. Prenons l'exemple d'un espace arborescent. Par analogie aux systèmes de fichiers, les fils d'un nœud « héritent » par défaut de la sécurité de leur père et partagent ainsi la même ACL.

La figure C.2 illustre ces notions. Un groupe d'entités, représenté à gauche, partage la même ACL. Celle-ci contient une liste d'ACE, qui attribuent à des acteurs ou des groupes d'acteurs des droits sur ces entités. La dernière ACE a la particularité de ne concerner qu'une seule facette.

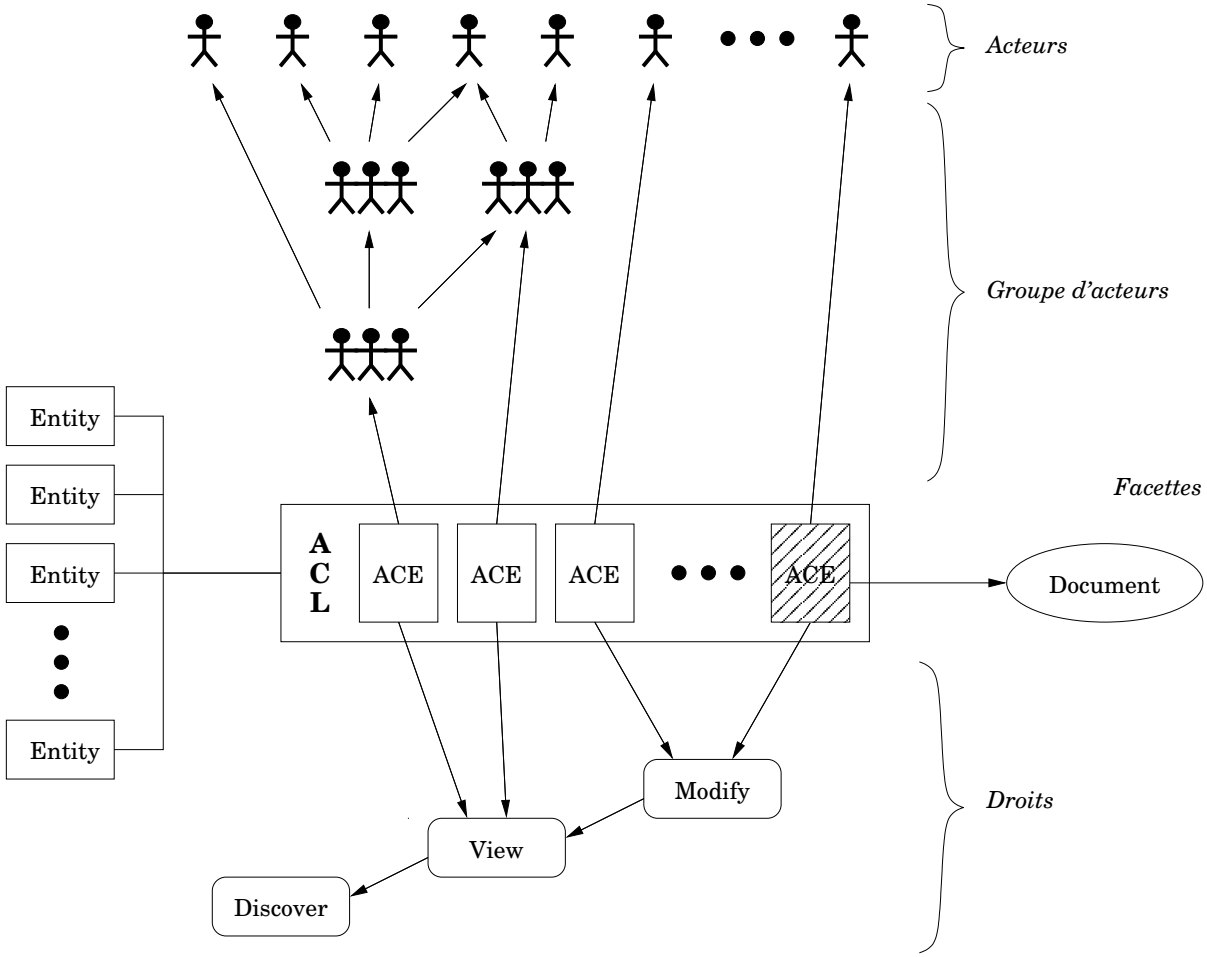

Figure C.2 – Exemple d'ACL et d'ACE

C.2.2 Traduction relationnelle

Comme nous l'avons décrit en section C.1, les acteurs et les groupes d'acteurs sont des entités qui supportent des facettes spécifiques. Sur le même principe, nous créons les droits comme de simples instances supportant la facette RIGHT. Le singleton système possède le dag de droits comme l'une de ses propriétés :

$$\begin{aligned} system : \textbf{tuple}(&..., \\ &rights : \textbf{dag}(right : \text{RIGHT}), \\ &...) \end{aligned}$$

En utilisant l'algorithme de traduction relationnelle présenté au chapitre 9 nous obtenons les scripts de création suivants en SQL :

```
-- Singleton system (SQLServer script):
---------------------------------------
if exists (select * from [dbo].[sysobjects]
    where id = object_id(N'[dbo].[tSystem]')
      and OBJECTPROPERTY(id, N'IsUserTable') = 1)
  drop table [dbo].[tSystem]
GO

CREATE TABLE [dbo].[tSystem] (
  [SingletonKey] [bit] DEFAULT 1 PRIMARY KEY CHECK ([SingletonKey] = 1),
  [...],
  [rightsID] [surrogate] IDENTITY(1,1) NOT NULL,
  [...]
)
GO

INSERT INTO [dbo].[tSystem] DEFAULT VALUES
GO

-- Relation system_rights (SQLServer script):
---------------------------------------------
if exists (select * from [dbo].[sysobjects]
    where id = object_id(N'[dbo].[tSystem_rights]')
      and OBJECTPROPERTY(id, N'IsUserTable') = 1)
  drop table [dbo].[tSystem_rights]
GO

CREATE TABLE [dbo].[tSystem_rights] (
  [rightsID] [surrogate] NOT NULL,
  [rightPredecessor] [RightRef] NOT NULL,
  [rightSuccessor] [RightRef] NOT NULL,
  CONSTRAINT [PK_tSystem_rights] PRIMARY KEY CLUSTERED (
    [rightsID],
    [rightPredecessor],
    [rightSuccessor]
  )
)
GO
```

```
-- Relation system_rightsClosure (SQLServer script):
-----------------------------------------------------
if exists (select * from [dbo].[sysobjects]
    where id = object_id(N'[dbo].[tSystem_rightsClosure]')
      and OBJECTPROPERTY(id, N'IsUserTable') = 1)
  drop table [dbo].[tSystem_rightsClosure]
GO


CREATE TABLE [dbo].[tSystem_rightsClosure] (
  [rightsID] [surrogate] NOT NULL,
  [rightPredecessor] [RightRef] NOT NULL,
  [rightSuccessor] [RightRef] NOT NULL
)
GO


-- Reverse index speeding up the lookup of predecessors
if exists (select name from [dbo].[sysindexes]
    where name = 'IX_tSystem_rights_reverse' )
  drop index [dbo].[tSystem_rights].[IX_tSystem_rights_reverse]
GO


CREATE NONCLUSTERED INDEX IX_tSystem_rights_reverse
    ON [dbo].[tSystem_rights]
  ( [rightsID], [rightSuccessor], [rightPredecessor] )
GO


-- Unique clustered index replacing the primary key.
-- Duplicate keys are ignored => insertion of values already existing
-- are ignored.
if exists (select name from [dbo].[sysindexes]
    where name = 'IX_tSystem_rightsClosure' )
  drop index [dbo].[tSystem_rightsClosure].[IX_tSystem_rightsClosure]
GO


CREATE UNIQUE CLUSTERED INDEX IX_tSystem_rightsClosure
    ON [dbo].[tSystem_rightsClosure]
  ( [rightsID], [rightPredecessor], [rightSuccessor] )
  WITH IGNORE_DUP_KEY
GO


-- Reverse index speeding up the lookup of predecessors
if exists (select name from [dbo].[sysindexes]
    where name = 'IX_tSystem_rightsClosure_reverse' )
  drop index [dbo].[tSystem_rightsClosure]
    .[IX_tSystem_rightsClosure_reverse]
GO
```

```
CREATE NONCLUSTERED INDEX IX_tSystem_rightsClosure_reverse
    ON [dbo].[tSystem_rightsClosure]
  ( [rightsID], [rightSuccessor], [rightPredecessor] )
GO


-- View on the dag closure inclusing identity edges
if exists (select * from [dbo].[sysobjects]
    where id = object_id(N'[dbo].[vSystem_rightsClosureExtended]')
    and OBJECTPROPERTY(id, N'IsView') = 1)
  drop view [dbo].[vSystem_rightsClosureExtended]
GO


CREATE VIEW [dbo].[vSystem_rightsClosureExtended]
AS
  select * from [dbo].[tSystem_rightsClosure]
  union select distinct [rightsID], [rightPredecessor], [rightPredecessor]
    from [dbo].[tSystem_rights]
  union select distinct [rightsID], [rightSuccessor], [rightSuccessor]
    from [dbo].[tSystem_rights]
GO
```

Comme indiqué en section 9.2.5.2, des procédures stockées sont également générées pour manipuler les données.

Généralement, les instances supportant RIGHT partagent la même ACL qui attribue le droit `view` à `everyone` et implicitement `full-control` aux administrateurs (qui ont ce droit pour toutes les entités du système).

Les ACL et les ACE ne doivent en revanche pas être créées « directement » par l'application. Il nous paraît donc inutile de les représenter comme des entités à part entière et nous préférons les garder comme de purs objets système. Ainsi, nous utilisons une relation dédiée :

ACL	
`idACL`	**integer**
`idACE`	**integer**
`actorOrGroup`	**ActorOrActorGroupRef**
`right`	**RightRef**
`facet`	**MetaFacet**
primary key (idACL, idACE)	

Pour indiquer qu'une ACE est définie pour toutes les facettes, nous utilisons un alias dédié `allFacets` dont la valeur est indisponible pour les facettes (par exemple `-1`). Nous pourrions utiliser la valeur `NULL`, mais les comparaisons ne seraient alors pas consistantes avec les autres facettes (nécessité d'utiliser `IS NULL` et `IS NOT NULL`).

C.2.3 Problématique de l'accès

Jusqu'ici, l'implémentation décrite répond à nos premiers besoins, c'est-à-dire une sécurité flexible appliquée au niveau des entités, acteurs, et facettes. Concentrons-nous maintenant sur les problèmes de performances.

Comme nous l'avons vu, les entités peuvent partager leurs ACL et en conséquence, le nombre d'ACL différents devrait rester raisonnable (nous l'estimons à quelques dizaines pour un usage normal[2]). Le nombre d'acteurs, en revanche, peut devenir très important (de lors de quelques milliers) quand l'application est déployée.

Partant de ces hypothèses, déterminer qu'un acteur a un droit particulier sur une entité est une tâche très coûteuse qui deviendrait vite prohibitive lors de la montée en charge du système. Une optimisation est donc nécessaire.

L'approche que nous avons choisie est de maintenir la liste des ACL auxquelles chaque acteur a accès, pour chaque droit et chaque facette : $(actor, right, facet, IdACL)$. Nous remarquerons qu'un index sur la relation contenant les ACL est insuffisant. En effet, cette relation référence aussi des groupes d'acteurs, alors que nous nous intéressons ici aux acteurs individuels.

Appelons cette nouvelle relation `AccessList`[3] :

AccessList	
`actor`	**ActorRef**
`right`	**RightRef**
`facet`	**MetaFacet**
`idACL`	**integer**
primary key (actor, right, facet)	

Comme dans la précédente relation, l'alias `allFacets` est utilisé pour représenter l'ensemble des facettes du système.

Le nombre total de droits et d'ACL étant supposé faible, l'obtention de la liste des ACL pour lesquelles un utilisateur a une permission donnée est rapide, quel que soit le nombre d'acteurs. Avec une telle liste à disposition, les opérations de contrôle d'accès décrites en section C.1.3 sont très simples : l'application doit simplement vérifier que l'ACL de l'entité à laquelle un acteur veut accéder est dans la liste de ses ACLs autorisées pour les droits et facettes appropriées à l'action.

Cependant, si cette approche nous permet de contrôler efficacement les accès aux objets, nous sommes maintenant confrontés au problème de maintenance de cette liste. Commençons par examiner deux approches antagonistes :

- Mettre à jour la liste à chaque fois qu'une modification de sécurité est effectuée ;
- Mettre à jour la liste à chaque fois qu'un acteur se connecte au système.

Par modification de sécurité, nous signifions toute modification des droits (voir les opérations décrites en section C.1.3), mais aussi de l'appartenance d'acteurs à des groupes. En effet, si un acteur sort d'un groupe par exemple, il ne doit plus avoir accès aux entités qu'il ne voyait que *via* le groupe.

La première approche a l'avantage de maintenir une liste constamment à jour avec les éléments de sécurité et ainsi d'éviter tout risque d'accès non autorisé. Cependant, de nombreuses mises à jour risquent d'être réalisées sans besoin, dès lors que les acteurs concernés ne sont pas connectés à ce moment là. De plus, si de nombreux acteurs sont concernés par une modification, le coût de celle-ci risque d'être très important.

La seconde approche présente l'avantage de ne maintenir la liste que pour les acteurs actifs. Cependant, durant tout le temps de connexion d'un acteur, les changements ne seront pas réfléchis sur ses droits.

[2] Le droit implicite `full-control` du propriétaire d'une entité n'a pas besoin d'être inclus dans son ACL (voir la fonction $isGranted$ en section C.1.3 et l'usage en section C.3.

[3] `AccessList` est en fait une vue, son implémentation est cachée.

De plus, alors que les modifications de sécurité peuvent être rares, le système doit vérifier, et mettre à jour si besoin, les droits des acteurs à chaque connexion.

De ce constat, nous choisissons d'adopter une approche intermédiaire :

- Chaque fois qu'une modification de la sécurité intervient, la relation `AccessList` est mise à jour pour tous les acteurs concernés qui sont connectés. Tous les autres acteurs concernés sont « marqués » pour une mise à jour future (en utilisant un simple booléen dans la facette ACTOR).
- À chaque connexion d'un acteur marqué, une mise à jour est effectuée.

C.2.4 Administrateurs

Comme nous l'avons défini en section C.1.1, les administrateurs possèdent le droit `full-control` sur toutes les instances du système. Ainsi, toutes les ACL contiennent l'ACE `(administrators, full-control, allFacets)`.

Plutôt que d'insérer cette ACE à chaque création d'ACL et ensuite maintenir la relation `AccessList`, nous préférons voir `AccessList` comme une vue et utiliser une union avec tous les acteurs du groupe `administrators` (la fermeture du dag nous permet de l'obtenir à moindre coût).

C.3 Usage

La sécurité doit être vérifiée dans deux cas :

- Quand une lecture de données est effectuée (instruction `SELECT`) ;
- Quand une opération est appelée (instruction `EXEC`).

C.3.1 Sélections

Le moyen le plus simple de gérer la sécurité est de cacher sa complexité dans une fonction dédiée :

$$isGranted : Actor \times Right \times Entity \times Facet \rightarrow boolean$$

Ainsi, une requête est alors composée d'une clause `WHERE` de la forme :

```
... WHERE isGranted( 34, 1, 189, 490 ) ...
```

Cette clause peut être clarifiée à l'aide de fonctions « *helper* » :

```
... WHERE isGranted(34, rightId('view'), 189, facetId('myFacet')) ...
```

La fonction `isGranted` vérifie que l'ACL correspondante à l'entité fait bien partie de la liste des ACL pour lesquelles l'acteur a la permission spécifiée (droit et facette). L'appeler avec la valeur `allFacets` (alias pour la valeur `-1`) pour la facette indique que nous voulons vérifier le droit pour toutes les facettes de l'entité.

Toutefois, si l'utilisation d'une fonction rend la sécurité très simple à utiliser, elle a un coût important comparée à une sélection directe. En effet, la fonction inclut elle-même une sélection qui est réalisée pour chaque enregistrement candidat et non une fois uniquement[4]. Elle ne supporte pas du tout une montée en charge. Ainsi, elle doit être évitée dans le cadre de sélections (elle peut toujours servir pour des tests unitaires).

[4] Observation réalisée avec le plan d'exécution par défaut de SQL Server 2000.

Pour résoudre ce problème tout en conservant la sélection aussi simple que possible, nous créons une nouvelle fonction qui, au lieu de gérer toute la sécurité, renvoie simplement la liste des ACL auxquelles un acteur à accès :

$$ACLsFor : Actor \times Right \times Facet \rightarrow 2^{ACL}$$

Ainsi, si nous voulons par exemple restreindre une recherche aux entités supportant MYFACET auxquelles l'acteur `34` a accès, nous utiliserons :

```
...
  INNER JOIN Entity ON myFacet.id = Entity.id
    AND ( Entity.owner = 34 OR Entity.idACL
    IN ( SELECT * FROM ACLsFor(34, rightId('View'), facetId('myFacet')) )
...
```

En utilisant des requêtes exprimées ainsi, la détermination de la liste d'ACL auxquelles un acteur a accès n'est effectuée qu'une fois, quel que soit le nombre d'enregistrements candidats. En contrepartie, la requête est plus complexe : une jointure sur la relation représentant les entités du système et une vérification du propriétaire de chaque entité sont nécessaires.

Les sélections mettant en jeu des entités référencées *via* leurs facettes ont besoin d'une attention spéciale. En effet, nous devons vérifier dans ce cas si l'acteur a accès à l'objet référencé. Observons un exemple :

```
SELECT Facet1.*, f2.info FROM Facet1
  INNER JOIN Entity On Entity.id = Facet1.id
    AND ( Entity.owner = 34 OR Entity.idACL
      IN ( SELECT * FROM
        ACLsFor(34, rightId('View'), facetId('Facet1')) ) )
  LEFT OUTER JOIN
    ( SELECT Facet2.id, Facet2.info FROM Facet2
      INNER JOIN Entity ON Facet2.id = Entity.id
      AND ( Entity.owner = 34 OR Entity.idACL
        IN ( SELECT * FROM
          ACLsFor(34, rightId('View'), facetId('Facet2')) )
    ) AS f2
    ON Facet1.facet2 = f2.id
```

La sélection de cet exemple n'est pas restrictive. Il est fréquent qu'une application ne montre qu'un sous-ensemble des résultats d'une requête (elle utilisera par exemple une pagination pour montrer tous les résultats). Dans ce cas, le `LEFT OUTER JOIN` n'influençant pas le nombre d'enregistrement retournés, la requête doit être faite en deux temps et n'effectuer la jointure externe qu'après la première restriction. Cette simple optimisation peut apporter d'importants gains de performances.

Comme l'illustre l'exemple précédent, les sélections peuvent devenir très complexes dès que la sécurité doit être vérifiée. Un moyen simple de simplifier ces sélections est l'utilisation de macros. Observons comment notre dernier exemple peut bénéficier d'une nouvelle macro `GRANTED` :

```
SELECT Facet1.*, f2.info FROM Facet1
  INNER JOIN GRANTED('Facet1', 34, rightId('View'))
  LEFT OUTER JOIN
    ( SELECT Facet2.id, Facet2.info FROM Facet2
      INNER JOIN GRANTED('Facet2', 34, rightId('View'))
    ) AS f2
    ON Facet1.facet2 = f2.id
```

En transformant légèrement notre macro, nous pouvons obtenir une requête encore plus compacte :

```
SELECT Facet1.*, f2.info
  FROM GRANTED('Facet1', 34, rightId('View'))
  LEFT OUTER JOIN
    ( SELECT Facet2.id, Facet2.info
      FROM GRANTED('Facet2', 34, rightId('View'))
    ) AS f2
    ON Facet1.facet2 = f2.id
```

Sans utiliser de macros, il serait aussi possible de simplifier notre requête en utilisant une fonction. Dans le dernier exemple, `GRANTED` pourrait ainsi être une fonction renvoyant l'ensemble des entités auxquels un acteur à accès pour la facette donnée. Cependant, l'ensemble des enregistrements correspondant devrait alors être créé entièrement avant de pouvoir tester les conditions, conduisant à de faibles performances.

C.3.2 Opérations

Toutes les opérations du noyau requièrent un identifiant d'acteur en paramètre de façon à sécuriser leurs appels. Une des premières actions de chaque opération est donc de vérifier l'accréditation de cet appelant.

Les actions travaillant avec seulement une entité à la fois peuvent tirer profit de la fonction `isGranted`, alors que celles travaillant sur des ensembles devront utiliser la méthodologie décrite dans la section précédente.

Reste la question de l'adaptation de la sécurité des opérations : comment changer son comportement ou même la désactiver ? La solution consiste à externaliser les vérifications de sécurité en dehors de l'action principale. Les *code snippets* (cf. annexe B) sont particulièrement adaptés à ce besoin. Leur fonctionnement nous permet même de cumuler plusieurs filtres de sécurité successifs s'il en est besoin.

Le modèle DOAN (DOcument ANnotation Model) Modélisation de l'information complexe appliquée à la plateforme Arisem Kaliwatch Server

Nicolas DESSAIGNE

Résumé

Nous présentons dans cette thèse le modèle DOAN (*DOcument ANnotation Model*), destiné à répondre aux besoins de modélisation de la société Arisem.

Arisem est éditeur de logiciels dans le domaine de la gestion des connaissances. La plateforme que l'entreprise propose s'inscrit dans le cycle collecte / analyse / diffusion de l'information. À partir de données de nature hétérogène et d'origines diverses (ex. : Internet, intranet, base de données), elle procède à différentes analyses (ex. : classement automatique, extraction de concepts émergents), afin de fournir des informations synthétiques et utiles à l'utilisateur. Partant de cette problématique, nous avons identifié trois besoins principaux pour le modèle : *expressivité*, *flexibilité* et *performances*.

Dans le cadre de cette thèse, nous avons développé un modèle basé sur le paradigme d'agrégation de facettes, qui permet aux concepteurs de décrire des données complexes, hétérogènes et évolutives. Au-delà de la simple notion de document, il rend possible la représentation d'objets métiers, comme par exemple des annotations ou des arbres de catégorisation. Complété par un système de types riches et par la capacité d'exprimer des contraintes entre facettes, ce modèle nous permet de répondre aux besoins d'expressivité et de flexibilité.

Nous proposons d'autre part un algorithme permettant de traduire les éléments du modèle DOAN en une implémentation relationnelle. Une fois le modèle instancié, les accès en modification sont contrôlés à l'aide de procédures stockées afin de garantir la consistance des données. Les accès en consultations sont en revanche effectués directement à l'aide de requêtes SQL. Les concepteurs peuvent ainsi faire des requêtes à la fois complexes et performantes, tirant parti au maximum des possibilités du système de gestion de bases de données. Cette approche permet une montée en charge importante et répond aux besoins de performances.

Mots-clés : Modélisation d'information, traduction relationnelle, gestion des connaissances.

Abstract

This thesis introduces the DOAN model (*DOcument ANnotation Model*), which aims at answering the modelling needs of the company Arisem.

Arisem is a software vendor acting on the knowledge management market. It offers a platform that processes data during the collection, analysis and dissemination steps of the information flow. As it works on heterogeneous data coming from various sources (e.g., internet, intranet, databases), it performs several analyses (e.g., automatic classification, extraction of emergent concepts), in order to provide synthetic information to the user. From this problematic, we have identified three main needs for the model: *expressivity*, *flexibility* and *performance*.

Within the context of this thesis, we have developed a model based on the facet aggregation paradigm. It enables designers to describe complex, heterogeneous and evolving data. Beyond the simple notion of document, it permits the representation of business objects, such as annotations or classification trees. Supplemented by a rich type system and the ability to express constraints between facets, this model enables us to answer the needs of expressiveness and flexibility.

On the other hand, we propose an algorithm able to translate DOAN elements into a relational implementation. Once the model is instantiated, update accesses are controlled by using stored procedures in order to ensure data consistency. Read accesses are in return performed directly by using SQL queries. Designers are thus able to use complex queries without sacrificing performance, exploiting database management systems functionalities. This approach scales well and answers the needs of performance.

Keywords: Information modelling, relational translation, knowledge management.