La correspondance d'objets dans la recherche d'informations : une expérimentation sur un SGBD OO

Philippe Mulhem

THESE

présentée par

Philippe MULHEM

pour obtenir le titre de DOCTEUR
de l'université Joseph Fourier - Grenoble 1
(arrêté ministériel du 5 juillet 1984 et du 30 mars 1992)
spécialité : INFORMATIQUE

.....

La Correspondance d'Objets dans la Recherche d'Informations : Une Expérimentation sur un SGBD OO

.....

Date de soutenance : 16 septembre 1993

Composition du jury :

Président :	**M. Pierre-Claude Scholl**
Rapporteurs :	**M. Patrick Bosc**
	M. Jacques Le Maitre
Examinateurs :	**Mme. Marie-Françoise Bruandet**
	M. Yves Chiaramella

THÈSE PRÉPARÉE DANS LE CADRE DU PROJET ARISTOTE
AU SEIN DU LABORATOIRE DE GENIE INFORMATIQUE - IMAG
A L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

Je tiens à remercier :

Mr Pierre-Claude Scholl, Professeur à l'Université Joseph Fourier de Grenoble, pour avoir accepté de présider ce jury de thèse, et pour l'intérêt qu'il a porté au travail présenté ici.

Mr Patrick Bosc, Professeur à l'Ecole Nationale Supérieure de Sciences Appliquées et de Technologie de Lannion, d'avoir bien voulu rapporter ce travail (je sais enfin écrire c'est-à-dire!).

Mr Jacques Le Maitre, Chargé de Recherche au Groupe Représentation et Traitement des Connaissances du CNRS à Marseille, pour la tâche de rapporteur qu'il a réalisée pour cette thèse.

Mme Marie-Françoise Bruandet, Professeur à l'Université Joseph Fourier, pour les matins et les après-midis longs, difficiles, mais fructueux, durant lesquels la fumée de mes idées s'est mélangée à celle de ses cigarettes pour donner ce résultat. "ET conceptuellement?", me diras-tu. "Ah, et bien ... euh .... oui chef!", répondrais-je.

Mr Yves Chiaramella, Professeur à l'Université Joseph Fourier et Directeur du Laboratoire de Génie Informatique, pour l'intérêt qu'il a porté à ce travail, et pour la machine à café de compétition de la Kfet.

Mr Michel Adiba, pour m'avoir accueilli au sein du groupe Aristote.

Les membres actifs du groupe Aristote, et en particulier Jean-Pierre (EPS man), Catherine (Villarsss-lesss-Dombesss-woman), Hervé (the Martin's Touch), les super Christines (R5-

sans-démarreur et on-me-demande-tellement-de-choses), Florence (ELEN-2, le retour), Bruno (Defoudess), Mourad ("Bien sûr!"), Marie-Christine (VTT-squash-ski-surf-canyoning woman), Christian (Christiang), ainsi que tous les autres!

Les joyeux drilles que j'ai eu le plaisir de côtoyer : Gilles (bien singulier), Sandrine (le Jas de Cucuron), Francis (impedance man), Laurence (CiMYOU-Wrappette), Walcelio (le petchi), les IHM-KIDS, the Adèle Team, Valérie, Liliane, ...

Joe et Pat

Table des Matières Globale

CHAPITRE I

Introduction

CHAPITRE I

Introduction

Le point de départ de notre travail a été de doter un système de gestion de bases de données d'une composante recherche d'informations. Notre objectif est donc de permettre l'expression de requêtes spécifiques à la recherche d'informations en comparant une requête aux documents de la base. Un élément qui permet de traiter une part de ce problème est de proposer des facilités pour décrire les correspondances entre les structures représentant la sémantique des documents et de la requête.

On remarque qu'historiquement, la recherche dans le domaine des bases de données a volontairement été axée sur des problèmes tels que : les optimisations de requêtes, les transactions, ... Cependant, si les langages de requêtes de ces systèmes ont un grande puissance d'expression, ils ne permettent que des comparaisons prédéfinies entre des éléments de la base et des données de la requête. Or, nous considérons que le fait de définir des prédicats et de les utiliser dans le but de comparer des données est nécessaire dans certains types d'applications. Dans notre travail, nous entendons par "Correspondance" de deux données l'utilisation de prédicats non-prédéfinis par le système sur ces données. Ceci veut dire que pour effectuer des correspondances, il faut pouvoir exprimer les nouveaux prédicats créés.

Nous avons dirigé notre travail sur la réalisation de prédicats dans le cadre d'applications ayant des fonctionnalités de Recherche d'Informations. Les Systèmes de Recherche d'Informations (SRI) ont pour but de permettre l'accès à des données, actuellement principalement textuelles, en utilisant le contenu sémantique de ces données. La représentation de ces données, aussi bien que la

représentation de leur sémantique, peut être structurée. En fait, la stratégie suivie par un Système de Recherche d'Informations lors du traitement d'une requête est le suivant :

i) la requête est traduite en une représentation exprimant sa sémantique,

ii) cette représentation est comparée à la représentation sémantique des éléments d'une base de document, appelé corpus.

Habituellement, les SRI gèrent leurs données de façon ad-hoc, c'est à dire sans faire appel à des systèmes de Gestion de Bases de Données (SGBD). Or, l'utilisation de SGBD pour gérer les documents et les représentations de la sémantique de ces documents permettrait de faciliter la réalisation de SRI, ou de composantes d'applications ayant des besoins en Recherche d'Informations. Ces facilités permettrons de décrire comment réaliser des prédicats qui comparent les descriptions sémantiques d'éléments de la base de données.

Aujourd'hui, de nouveaux types de SGBD sont développés dans les laboratoires de recherche, et voient le jour comme produits industriels.

Citons les SGBD déductifs, basés sur des langages comme LDL [Tur86] ou Datalog [Gar90]. Ces SGBD déductifs gèrent des bases explicites (comme une base de données "habituelle"), mais aussi infèrent des éléments de la base par l'utilisation de déductions. De tels systèmes peuvent retrouver tous les ancêtres d'une personne, alors que seules les liaisons enfant -> parents sont stockées explicitement.

D'autres SGBD, appelés actifs [Loh91, Ros89, Med92], permettent d'exprimer des règles du type Evénement-Condition-Action [Jar92], les événements pouvant aussi bien être internes à la base (par exemple "répercussion" sur des mises à jour) ou bien externes au système (changement de date …).

Enfin, le dernier type de SGBD dont nous parlons ici sont les SGBD à objets (aussi appelés SGBD Orientés Objets, ou SGBDOO en abrégé). Ces systèmes intègrent les éléments des langages Orientés Objets [Bai87] tels que les notions de classes, d'héritage, d'encapsulation, de méthodes, mais de plus ils gèrent les notions typiquement "bases de données" telles que la persistance des objets et les aspect transactionnels, et proposent de plus des langages de requêtes pour interroger les bases stockées.

Les SGBD à objets semblent les systèmes les plus aptes à gérer les applications auxquelles nous nous intéressons. Ceci vient en particulier de la notion d'objet, qui permet de représenter simplement des données ayant des structures

complexes.
Les raisons supplémentaires qui nous ont fait choisir ces systèmes sont les suivantes:
• Des SGBD Orientés Objets, qui ne sont plus des prototypes de recherche, sont disponibles actuellement. Ceci veut dire que, même si tous ces SGBD n'ont pas exactement la même approche, les problèmes de l'orientation objet sont pour une part dominés (par exemple, des solutions pour l'optimisation des requêtes ont été implantées). De plus, les concepts de l'orienté objet sont relativement standardisés "de fait". En nous basant sur une telle approche, nous espérons donc, tout en restant général, pouvoir appliquer nos résultats sur différents systèmes.
• Notre travail se situe dans le cadre du projet ARISTOTE [Adi92], qui vise à développer un environnement de développement d'applications orientées objet.

Dans de tels SGBD, les représentations des éléments comparés sont des objets, et c'est pourquoi nous nous sommes intéressés à la comparaison d'objets complexes. Le fait de permettre l'utilisation de prédicats non-prédéfinis dédiés à des comparaisons d'objets est un élément de réponse intéressant à mettre en œuvre.
Nous nous limitons, au cours de ce travail, à des prédicats que nous qualifions de stricts, c'est à dire des prédicats qui sont capables de déterminer si deux objets sont "proches" ou non (mais incapables de trouver dans quelle mesure des objets correspondent). Si l'utilisation de tels prédicats est comparable à celle des prédicats prédéfinis, leur définition est plus difficile à appréhender, et c'est sur ce point particulier que nous nous focalisons.
Pour aider un programmeur au cours de telles définitions, nous avons utilisé des travaux réalisés pour les modèles de Recherche d'Informations. Nous proposons une méthodologie pour l'élaboration par un programmeur du processus de correspondance d'une requête et d'un document. Notre approche permet de décomposer en deux parties la tâche de conception, l'une théorique qui permet d'exprimer une sémantique pour le processus de comparaison, et l'autre opérationnelle pour formuler ces correspondances en utilisant un langage proche de ceux de requêtes des SGBDOO.

Organisation de la thèse

Dans un premier temps nous allons expliquer certains points qui nous

intéressent pour la suite sur les SGBD Orientés Objet, puis l'existant dans le domaine des bases de données sur les requêtes imprécises. Ensuite, nous expliquerons l'approche choisie basée pour la partie théorique sur la logique modale reliée au modèle logique de Recherche d'Informations, dans le but de comparer des objets complexes par un outil dont les différentes phases reposent sur l'utilisation de la logique modale.
Le plan que nous allons suivre est le suivant:

Chapitre II

• Dans cette partie, nous définissons beaucoup plus en détail que dans cette introduction certains des avantages de l'orientation objet, en particulier face aux systèmes de bases de données relationnels. Nous allons nous intéresser plus spécifiquement dans cette analyse à trois sujets qui sont les identificateurs d'objets, les liens inter-objets et les langages de requêtes de ces systèmes.

Chapitre III

• Nous étudions des SGBD qui ont intégré des notions de "correspondances floues", c'est à dire des notions qui utilisent plus de “sémantique” pour comparer les éléments d'une base de données que de simples correspondances habituellement rencontrées. Ces systèmes ont la particularité d'avoir une approche similaire à celle des systèmes de recherche d'informations, dans la mesure où l'on essaie de trouver des éléments de la base qui sont **proches sémantiquement de la requête, ou d'éléments de référence**. Cette étude vérifie que les propositions au niveau des SGBD utilisent une expression sémantique assez pauvre des éléments de la base, et que cette sémantique n'est plus suffisante lors de son utilisation dans un cadre à objets complexes.

Chapitre IV

• La quatrième partie de ce document est consacrée à la correspondance d'objets complexes. Nous décrivons les éléments des systèmes de recherche d'informations, et nous expliquons en détail l'approche de C.J. Van Rijsbergen, qui décrit le *modèle logique de recherche d'informations*. Nous proposons un modèle théorique basé sur la logique modale. L'idée est la suivante : le document constitue un monde initial, si la requête n'y est pas évaluable, ce monde est étendu à l'aide de connaissances supplémentaires et le processus est réitéré jusqu'à l'obtention d'un monde impliquant le document. L'intérêt de présenter un modèle logique est de donner un moyen d'expression indépendant de

l'implantation et de guider le conception. Nous donnons deux exemples d'utilisation de nos travaux. La conception des prédicats est décomposée en deux phases, l'une est la phase théorique, et la seconde consiste en la déclaration des différentes parties du traitement de ces prédicats en utilisant un langage algébrique qui est une extension de celui proposé par Zdonik [Zdo89a]. Cette dernière partie utilisera intensivement la notion de liens entre objets. Dans cette partie, nous nous limitons à l'une des fonctionnalités fondamentales d'un SRI qui est la **fonction de correspondance**.

Chapitre V

• Ce chapitre permet de replacer notre travail par rapport au développement d'une application de bases de données, et aussi de situer clairement quels sont les extensions qui vont devoir être effectuées pour proposer une approche réellement utilisable par un programmeur. Ces extensions se situent principalement au niveau des valuations de correspondances.

Chapitre VI

• Nous passons ensuite à la réalisation effectuée sur le système O_2. O_2 est un système représentatif des SGBDOO actuels aussi bien au niveau des concepts que de son modèle de données. Nous abordons donc ici les choix que nous avons faits pour implanter notre outil. Comme nous l'avons dit plus haut, on se situe au niveau d'une application de ce système, en créant des méthodes et des classes.

Chapitre VII

• La conclusion nous permet de situer nos travaux par rapport à l'existant, en fixant les apports que nous faisons . Nous allons de plus exprimer les suites à plus long terme, comparé au chapitre VI, à donner à ce travail, en particulier à un niveau "reconstruction du résultat d'une requête".

CHAPITRE II

L'approche Orienté Objet, pourquoi?

CHAPITRE II

L'approche Orienté Objet, pourquoi?

I. Introduction

Les systèmes de gestion de bases de données relationnelles apparus dans les années 70 ont permis de s'affranchir de la structure physique des bases de données. Ces systèmes ont donc de ce fait apporté un plus non négligeable par rapport aux systèmes réseaux ou hiérarchiques. Cependant, si les systèmes relationnels sont tout à fait adaptés à des données structurellement simples, ils atteignent leurs limites, par exemple dans des cas de gestion de documents complexes. En fait, la simplicité du modèle est à la fois un avantage dans les cas simples et un inconvénient dans les cas "complexes". Or, ces cas complexes, comme la CAO, c'est-à-dire des cas où l'on veut intégrer des liaisons entre éléments de la base, demandent de plus en plus les capacités de stockage et de traitement des bases de données.

Dans cette partie, nous ne cherchons pas à examiner en détail les caractéristiques des SGBD Orientés Objet, comme ce qui a été fait par F. Bancilhon dans [Ban88], ou dans le manifeste de l'orienté objet [Atk89]. En effet, nous ne nous intéressons pas ici à des notions courantes dans les SGDB OO, comme l'héritage, le polymorphisme, les méthodes, la persistance, …, car nous considérons ces éléments comme faisant partie à l'heure actuelle des connaissances générales dans le domaine des bases de données. Par contre, nous traitons des éléments qui vont nous servir par la suite. Notre but est de montrer que les SGBD OO permettent des représentations de données suffisamment riches pour intégrer, par exemple, la notion de documents complexes et de termes d'indexations complexes des systèmes de recherche d'informations.

Nous nous intéressons aussi aux langages de requêtes de ces SGBD orientés objet, pour bien situer l'avantage acquis par cette approche SGBD.

Nous allons nous concentrer sur les avancées des SGBD orientés objets suivant les trois axes suivants :

- les identificateurs d'objets, qui sont la base même de la notion d'objets complexes,
- les liens entres objets, qui permettent de créer des objets complexes,
- les langages de requêtes, qui ont permis de réduire le dysfonctionnement rencontré entre un langage de programmation et un langage de requête.

Nous avons choisi ces trois directions, car la démarche que nous allons utiliser lors de la correspondance d'objets utilise de façon intensive la notion de lien entre objets, et que cette notion est basée sur les identificateurs d 'objets. Le dernier point est consacré aux langages de requêtes de ces systèmes, car nous allons nous baser dans la suite sur des travaux effectués sur ces langages.

Au cours de ce chapitre, nous citons différents systèmes, comme O_2 [Deu91], ORION [Bane87], ONTOS [And88], ENCORE [Zdo86], OODAPLEX [Day89] (l'extension Objet de DAPLEX [Cha83]), EXODUS [Car87], ObjectStore [Lam91], POSTGRES [Sto86] (extension Objet de INGRES [Sto76]) et O-RAID [Bha92, Sri92]. Nous indiquons les propositions de ces systèmes suivants les axes choisis.

II. Certains “plus” de l'Orienté Objet

II.1. les identificateurs d'objets

Pour bien comprendre les avantages de cette notion d'identificateur d'objet, aussi appelé *oid* (pour ***o***bject ***id***entifier), nous nous ramenons au relationnel. Nous examinons un exemple où la notion d'identificateur d'objet est réellement une avancée pour faciliter la tâche d'une personne utilisant un SGBD.

Considérons une relation *Personne* ayant les attributs suivants:
- *id*: numéro d'identificateur (entier),

- *nom*: le nom d'une personne (chaîne de caractères),
- *prénom*: le premier prénom d'une personne (chaîne de caractères),
- *âge*: âge de la personne (entier),
- *enfant*: numéro d'identificateur d'un enfant de la personne.

La déclaration de cette relation serait donc celle de la figure II1, en utilisant la syntaxe de INGRES [ING91].

```
create table Personnes
  (
  id          integer,
  nom  varchar(30),
  prénom      varchar(20),
  âge  integer,
  enfant      integer
  )
```

Figure II1: Déclaration de la relation Personnes.

Nous ne nous préoccupons pas ici de normaliser (cf. [Elm89]) cette relation, ce qui n'apporterait rien à notre explication. Nous savons juste que chaque attribut de cette relation est atomique, c'est à dire que la valeur que peut prendre cet attribut est un type de base entier, réel, chaîne de caractères, …

Considérons une personne, "Jean Térieur", qui a deux enfants prénommés respectivement "Alain" et "Alex". Une partie de la relation *Personne* est décrite dans le tableau de la figure II2.

Personnes

id	*nom*	*prénom*	*âge*	*enfant*
555	Térieur	Jean	25	721
555	Térieur	Jean	25	890
721	Térieur	Alain	2	
890	Térieur	Alex	1	

Figure II2: Une partie de la relation Personnes.

La pauvreté structurelle du modèle relationnel, qui rappelons-le ne permet de n'avoir que des types de bases pour les attributs d'une relation, ne permet pas d'exprimer explicitement que les enfants de "Jean Térieur" ont une quelconque liaison avec leur père. A l'intérieur de la relation *Personnes*, on sait juste que l'entier *721* est la valeur de l'attribut *enfant* de "Jean Térieur" et aussi la valeur

du champ *id* du n-uplet qui correspond à "Alain Térieur".
En fait, si cette sémantique de référence n'est pas proposée par le modèle de données, c'est par l'expression d'une requête que l'on exprime ces liens. Pour retrouver dans la relation *Personnes* "les noms et prénoms des enfants de Jean Térieur" (requête **R1**), on peut poser la requête en SQL [Pas88] dans la figure II3. C'est par l'intermédiaire d'un produit que cette liaison père<->enfants est réalisée. Nous allons revenir dans la partie suivante sur les liens entre objets.

```
Select p2
from Personnes p1, Personnes p2
where p1.id = 555
  and p2.id = p1.enfant
```

Figure II3: Requête SQL pour les enfants de Jean Térieur.

Des systèmes NF^2 (pour *N*on *F*irst *N*ormal *F*orm) ont proposé des extensions en permettant des attributs multivalués et des attributs "sous-relations" [Ari83] , [Pis86]. Dans ce cas, ces systèmes permettent d'écrire que "Jean Térieur" a plusieurs enfants, en utilisant une relation *Personnes_v2*, identique à *Personnes*, mais qui possède un attribut *enfants* qui contient les identificateurs des enfants d'une personne à la place de l'attribut *enfant* de la relation *Personnes*. Cette relation est décrite dans la figure II4.

Personnes_v2

id	*nom*	*prenom*	*âge*	*enfants*
555	Térieur	Jean	25	{721, 890}
721	Térieur	Alain	2	
890	Térieur	Alex	1	

Figure II4: Une partie de la relation Personnes_v2.

Cette méthode permet d'exprimer explicitement le fait que les entiers 721 et 890 correspondent à "Jean Térieur", mais on ne sait toujours pas explicitement que ces entiers correspondent aux enfants de cette personne.

Ces modèles NF^2 se sont donc intéressés aux constructeurs applicables, mais n'ont pas remis en cause les domaines de bases qui existent dans le relationnel en première forme normale. Ces extensions du modèle de données ne permettent

toujours pas de représenter par un *nom* un regroupement de données. Un nom est en fait, dans le langage plus informatique, ce qu'on appelle un *identificateur*. Comme le modèle relationnel n'était pas suffisamment puissant, les modèles de données orientés objets ont tenu compte de ce point et ont permis la manipulation de telles références, appelées identificateurs d'objets.

Dans tous les systèmes orientés objet que nous avons étudiés, chaque objet peut être vu comme un couple (identificateur, valeur), où l'*identificateur* est un "pointeur" géré par le système et auquel l'utilisateur n'a pas accès, et la *valeur* représente les données qui sont stockées dans cet objet.
Il convient ici de parler du concept de valeur :
les valeurs représentent des propriétés attachées aux objets, elles ne correspondent pas à des entités ayant une existence propre dans le monde réel [Jar90]. Une valeur n'existe que par rapport à l'objet qui la contient, et ne peut être partagée, on revient donc à l'idée de *valeur* du relationnel. Dans ORION et ENCORE, seules des valeurs de bases sont possibles (entier, réel, …), alors que le système O_2 permet de manipuler des valeurs construites.

Reprenons notre exemple de personnes. En utilisant la syntaxe du système O_2, la déclaration du type qui correspond à une personne pourrait être celle de la figure suivante :

```
add class Personnes
public type tuple (   id : integer,
                      nom : string,
                      prénom : string,
                      âge : integer,
                      enfants : set(Personnes))
end;
```

Figure II5: Déclaration de la classe Personne.

Suivant cette déclaration, nous retrouvons les mêmes éléments que dans la relation NF^2.
Cependant, et c'est là qu'un pas est franchi, les enfants d'une personnes sont déclarées comme étant eux-mêmes des personnes.
Nous représentons les objets qui correspondent aux n-uplets de la relation Personnes_v2 dans la figure II6. Dans cette figure, nous reprenons les notations de [Zdo89a], c'est à dire qu'un objet n-uplet est représenté par "<>", un ensemble par "*" et un élément de base (valeurs entières, réelles, …) par un point "•". A

droite de ces représentations figures l'identificateur de l'objet correspondant. Dans cette figure, l'objet ayant l'identificateur *o1* correspond à "Jean Térieur", l'objet *o2* à "Alain" et l'objet *o3* à "Alex".

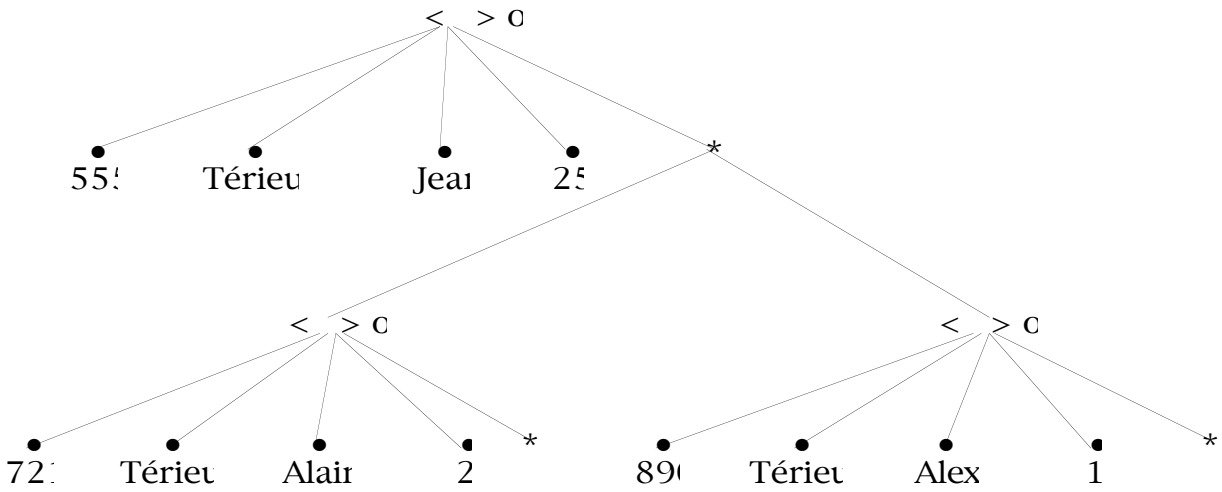


Figure II6: Les objets Personnes.

Ceci veut dire que pour un utilisateur qui veut poser la requête **R1**, il lui suffira de trouver la personne qui correspond à "Jean Térieur", puis d'accéder *directement* à ses enfants.
Si l'on considère que toutes les personnes de la base sont dans l'ensemble nommé *Les_Personnes*, en adoptant l'approche de O_2 et celle du modèle de données EXTRA pour EXODUS, la requête est décrite dans la figure II7 (exprimée dans le langage de requête de O_2, appelé O_2SQL [Ban89]).

```
select tuple(enfants : p->enfants)
from p in Les_Personnes
where p->id = 555
```

Figure II7: Requête sur des objets.

On se rend donc bien compte que la structuration des objets, grâce aux références inter-objets, est un acquis très intéressant en ce qui concerne la facilité d'écriture de requêtes.

II.2. Les liens inter-objets

Nous avons, dans la partie précédente, exprimé au travers de "références simples" les liaisons parents<->enfants. En fait, il existe plusieurs types de liens possèdant chacun certaines caractéristiques. Nous décrivons ici trois types de liens rencontrés dans les systèmes existants. En fait, quand on parle de liens inter-objets, il faut sous-entendre entre objets et/ou valeurs, comme nous allons le voir. Dans cette partie, nous allons reprendre la terminologie de [Jar90], dans laquelle les types de liens définis sont: les liens de Propriété, les liens de Composition, et les liens de Référence.

De plus, nous allons regarder aussi les liens que nous nommons "bidirectionnels", c'est à dire des liens gérés par le système utilisables dans les deux directions départ<->cible. Ces liens facilitent encore le travail d'un utilisateur, car ce dernier n'a plus à gérer explicitement les inverses des liens, en particulier tous les problèmes qui adviennent lors de la destruction d'objets.

II.2.1. Les Liens de Propriété

Ces liens se situent entre un objet et une valeur et non pas entre deux objets. On les retrouve dans O_2, ENCORE, ORION, dans le SGBD extensible EXODUS, dans l'extension objet de DAPLEX appelée OODAPLEX, ainsi que dans des systèmes relationnels étendus tels que O-RAID et POSTGRES.

Revenons sur l'exemple de la classe *Personnes* (figure II5). Nous utilisons ce type de lien pour exprimer que les identificateurs, noms, prénoms et âges des personnes sont des valeurs et non pas des objets. En effet, il n'est pas utile de créer des classes qui correspondent à des chaînes de caractères ou à des entiers, nous voulons juste utiliser ces valeurs de base sans les partager *explicitement* entre plusieurs objets. De plus, si on veut enlever de la base l'objet correspondant à "Jean Térieur", c'est à dire détruire cet objet, les valeurs contenues dans cet objet n'ont plus de raison d'exister.
C'est ce type de lien qui a été utilisé lors de la déclaration de la classe Personne, pour représenter que l'attribut *enfants* est une valeur ensemble qui va *référencer* (cf. partie II.2.3.) des objets *Personnes*.

Cette notion de lien de propriété se rapproche des attributs relationnels. En effet,

un n-uplet d'une relation ne contient que des valeurs qui lui sont propres et qui, en cas de destruction du n-uplet, sont elles-mêmes détruites.

II.2.2. Les liens de Composition

Ces liens ont pour cible non plus des valeurs mais des objets. Suivant le type décrit par S. Jarwah, un objet cible O_c d'un lien de Composition est partageable, mais possède une contrainte existentielle avec les points de départs (objets ou valeurs) de ces liens : si un point de départ *d* est détruit, alors O_c le sera aussi, *sauf* si O_c est la cible d'autres liens de Composition.
Reprenons l'exemple de la classe Personnes, mais en considérant que les *enfants* d'un objet Personne sont les cibles de liens de Composition. Cette classe nommée *Personnes_v2* pourrait être déclarée de la manière suivante (suivant une notation proche de celle de O_2, et en utilisant le mot clé Comp pour indiquer les liens de composition) dans la figure suivante :

```
add class Personnes_v2
public type tuple (id : integer,
            nom : string,
            prénom : string,
            âge : integer,
            enfants : set(Comp Personnes_v2))
end;
```

Figure II8: Déclaration de la classe Personne_v2.

Dans ce cas, nous représentons dans la figure II9 les objets qui correspondent à "Jean Térieur", ainsi que ses enfants, en rajoutant un objet correspondant à "Marie Térieur", 26 ans, la mère de "Alain" et "Alex". Dans cette figure, nous avons numéroté les liens de Composition parents<->enfants de 1 à 4.

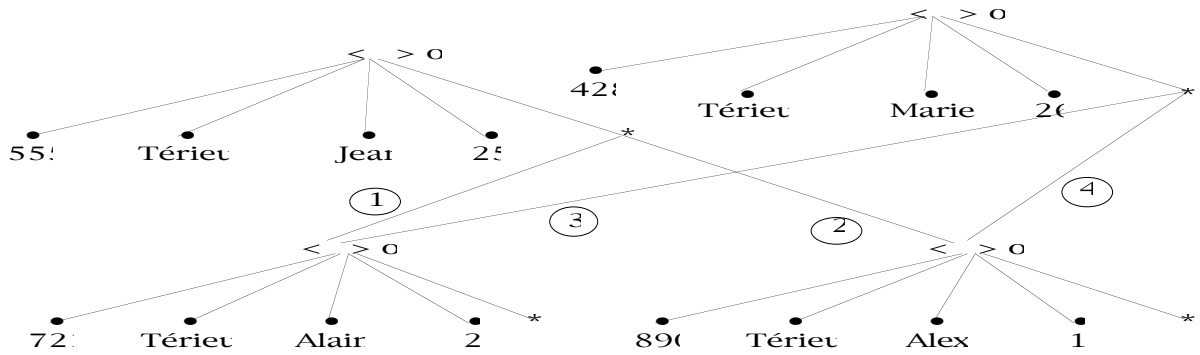


Figure II9: La "famille" Térieur.

Suivant la configuration de la figure II9, si l'objet *o1*, qui correspond à "Jean Térieur", est ôté de la base de données, les liens numérotés 1 et 2 sont donc détruits (car la valeur de l'attribut *enfants* de l'objet *o1* est elle aussi détruite, cf. partie II.2.1.). Les objets *o2* et *o3* ne sont pas détruits, car ils sont la cible des liens de Composition numérotés 3 et 4 qui partent de l'objet *o4*.
Par contre, si par la suite l'objet *o4* est détruit à son tour, les liens numérotés 2 et 3 vont être éliminés. Les objets *o2* et *o3*, qui ne sont plus référencés par aucun lien de Composition, vont donc être détruits à leur tour, automatiquement.

Ce type de lien se rencontre dans un système comme ORION, et dans ce système ces liens sont appelés "partagés dépendant du père" (*shared dependant links*). De plus, ORION propose des liens de composition *non-partagés* (liens "non-partagés dépendants du père"), qui interdisent à un objet d'entrer dans la composition d'autres objets. Une approche similaire est réalisée dans le langage de programmation pour bases de données PEPLOM [Dec93], avec les liens nommés `own`.
Dans le modèle de données EXTRA pour le système EXODUS, les liens `own ref` permettent de déclarer des liens de Composition partagés "originaux", en effet un objet ne peut être la cible que d'un seul lien de composition mais de plusieurs liens de référence (cf. partie suivante), la contrainte existentielle se faisant par rapport au lien de composition. Cette dernière approche peut certainement poser des problèmes, car un objet peut être détruit sans que l'un de ses pères soit au courant.

De tels liens sont utiles pour exprimer explicitement les dépendances fortes entre objets. Par exemple, dans le cas où un document textuel (un livre) est stocké dans une base de données, et que l'objet Livre est composé d'objets Chapitres (qui peuvent être dans plusieurs livres), si on veut qu'automatiquement les chapitres soient ôtés de la base, il suffit de définir des liens de Composition entre les livres et les chapitres. Si un objet-livre L est composé des objets-chapitres C1, C2 et C3 et que ces chapitres n'appartiennent pas à d'autres livres; si L est détruit dans la base, alors les chapitres de L le seront aussi.

II.2.3. Les liens de Référence

Les liens de référence relient deux objets , commeles liens de composition. La différence est que dans ce cas, il n'existe aucune contrainte existentielle entre les départs et les cibles de liens. Si un objet de départ O_d est détruit, alors l'objet cible O_c continuera d'exister sans aucune sorte de contrainte.

Comme O_2 ne permet que ce type de lien entre objets ou entre valeur et objet, si nous reprenons la déclaration de la classe *Personnes* de la figure II5 et le schéma II6 de description des objets *o1*, *o2* et *o3*, de la partie II.1, le fait de détruire l'objet *o1* laissera dans la base les objets *o2* et *o3*.

Ce type de lien existe dans ORION sous le nom de "lien de référence non partagé indépendant du père". PEPLOM, EXTRA, ENCORE proposent également de tels liens de référence.
Ces liens sont très utiles dans une applications de CAO par exemple, où une pièce d'un moteur existe indépendamment du fait qu'elle soit ou non montée dans un moteur.

II.2.4. Les liens bidirectionnels

Les liens bidirectionnels sont en fait des liens qui permettent, si par exemple un objet A référence un objet B, de gérer automatiqument un lien inverse de B vers A. Ces types de liens sont rencontrés dans le système ORION et dans le langage PEPLOM. Cependant, l'approche sous-jacente à ces deux systèmes est différente.

Dans ORION, on exprime au niveau de la structure d'une classe le type de lien entre éléments de la base, et le lien inverse est géré par l'utilisation d'une méthode prédéfinie, `parents_of`, décrite ci-dessous :

```
parents_of ( <liste de classes> <type du lien> )
```

Dans ce cas, pour revenir à notre exemple de la classe *Personnes*, nous atteindrions les parents de l'objet modélisant "Alex Térieur", en considérant que le lien enfants->parent est un lien de référence et en appliquant à cet objet la méthode :

```
parents_of( (Personnes) Shared)
```

Dans PEPLOM, ce lien inverse est à déclarer au niveau de la structure des objets, au moment de la déclaration du type *Personnes*, de la manière décrite dans la figure suivante :

```
typedef abstract Personnes
{
  struct {
       string(30)   nom;
       …

       Personnes enfants{} inv value.parents;
       Personnes parents{} inv value.enfants;
       …
            } value;
  …
};
```

Figure II10: déclaration de liens inverses dans PEPLOM.

L'approche de PEPLOM permet de ne garder que les liens inverses de certains liens, alors que dans ORION on obtiendra tous les liens inverses d'un certain type, sans indication explicite du lien dont on obtient l'inverse.
Dans [Lem92], il existe une fonction appelée ***invby*** permettant d'obtenir les liens inverses d'une fonction donnée.

II.3. Les langages de requêtes

Nous décrivons les éléments généraux que l'on rencontre habituellement dans les langages de requêtes des SGBD orientés objets que nous avons étudiés.
L'avantage habituel que l'on cite en matière de langage de requête d'un SGBD orienté objet est le fait que le dysfonctionnement (*impedance mismatch*) est réduit par rapport à un SGBD relationnel couplé à un langage de programmation.

Prenons l'exemple du SGBD relationnel ORACLE. Supposons que l'on utilise notre relation en première forme normale *Personnes* (figure II5 partie II.1) à l'intérieur d'un programme C.
Si nous voulons trouver toutes les personnes qui ont 26 ans, la démarche à suivre est la suivante [ORA88] :

- associer un curseur à la requête SQL:

```
EXEC DECLARE Curs_26 CURSOR FOR
     select nom, prenom
     from Personnes
     where âge = 26;
```

- ouvrir le curseur: `EXEC SQL OPEN Curs_26`
- accéder aux n-uplets les uns après les autres, par l'instruction:

```
EXEC SQL FETCH Curs_26 INTO :nom, :salaire
```

On remarque que le résultat multivalué d'une requête ne peut pas être employé comme une variable du langage C, ce qui a nécessité l'introduction de curseurs, sortes de *protocoles* de passage de données entre le langage de programmation et le SGBD. C'est ce problème qui est appelé *dysfonctionnement* entre le langage de programmation et le SGBD.

Dans les systèmes orientés objet, un effort a porté sur la réduction de ce dysfonctionnement. Ces systèmes fournissent en effet des gestionnaires de types qui gèrent la structure complexe des objets et des valeurs. Les résultats de requêtes sont des structures pouvant être des valeurs ou des objets, suivant les cas, mais ont surtout une structure que ce gestionnaire peut gérer.
Les langages de requêtes des SGBD orientés objet sont tous basés sur des extensions de SQL, à de très rares exceptions près comme OQL [Ala89]. Ces langages sont donc à base de blocs "`select … from … where …`", comme nous l'avons vu dans la partie II.1 lors d'une requête O_2 (figure II3). Certains les utilisent plutôt comme des expressions, comme PEPLOM et ObjectStore, d'autres comme des instructions dans EXTRA et O_2. Si une requête en O_2 vise à trouver les noms et les prénoms des personnes de l'ensemble *les_Personnes* qui ont 26 ans et que cette requête est exprimée à l'intérieur du corps d'une méthode, nous écrivons l'instruction de la figure suivante :

```
o2query(res,"select tuple(nom : i->nom, prenom : i->prenom) \
            from i in Les_Personnes \
            where i->age = 26");
```

Figure II11: Une requête O_2 non interactive.

Dans cette figure II11, le résultat de cette requête est un ensemble, et cette valeur est traitée comme les autres valeurs par le langage de programmation.

L'emploi de requêtes à l'intérieur d'un langage de programmation est donc réduit à des appels de fonctions ou à l'évaluation d'expressions, ce qui est beaucoup plus simple que dans un système relationnel, et présente un avantage indéniable.

III Conclusion

Nous avons dans cette partie exprimé brièvement les avantages de l'approche orienté objet par rapport aux SGBD relationnels au niveau du modèle de données. Les axes de description ont été les identificateurs d'objets, les liens entre objets et le langage de requêtes.
Nous n'avons pas abordé d'autres notions importantes des SGBD orientés objet, car notre but n'était pas de les décrire complètement, mais de citer leurs avantages suivants les axes qui nous intéressent. Cependant, nous citons ici des références intéressantes sur la gestion des objets dans EXODUS [Car86a], dans O_2 [Vel89] et dans ORION [Kim88], sur les versions d'objets dans ORION [Kim89a], sur des optimisations de requêtes dans EXODUS [Car86b] à base de règles ou des optimisations de requêtes algébriques [Zdo89b].

Nous avons montré que le but des fonctionnalités étudiées est de simplifier l'utilisation du système de gestion de bases de données.
En effet, l'introduction des identificateurs permet de simplifier, en l'enrichissant, l'expression des données de manière à refléter la réalité de façon plus "naturelle" que dans un SGBD relationnel par exemple, ainsi que de faciliter la manipulation de données "complexes".
Dans la partie consacrée aux liens entre objets, nous avons vu les contraintes existentielles introduites entre les départs et les cibles de ces liens, dont le but est de gérer des relations de types prédédinis entre objets. Ceci permet de ne pas laisser à un utilisateur ce travail de gestion de contraintes. Cette gestion est un surplus de travail pour un concepteur, et est sujette à des erreurs de programmation. Les avantages sont les mêmes pour les liens inverses entre objets.
Pour les langages de requête, on a cherché avec les SGBD orientés objet à diminuer le dysfonctionnement entre ces langages et les langages de programmation dédiés à l'écriture du corps des méthodes. Ces recherches visaient à simplifier l'écriture de requêtes dans le corps de programmes, pour que le programmeur n'ait à se concentrer que sur son application, et pas sur des problèmes liés aux limitations des systèmes utilisés. Les exemples que nous

avons pris dans cette partie sont bien entendu limités; nous n'avons pas parlé entre autres des opérations sur les ensembles imbriqués comme `flatten`, ou des opérateurs `nest` et `unnest` [Zdo89a, Dec93], mais notre but n'était pas de faire une présentation exhaustive des langages de requêtes, mais d'indiquer quels sont les avantages de ces langages dans les SGBD orientés objet.
Nous allons revenir dans le chapitre suivant sur les caractéristiques de certains de ces langages.

Notre démarche se situe dans le cadre de simplification de "la vie" de l'utilisateur qui écrit une application. Nous voulons lui permettre d'écrire facilement des processus de comparaisons d'objets complexes. Comme nous allons le voir dans la partie suivante, les outils existants sur ce sujet sont assez limités.

CHAPITRE III

Requêtes Evoluées dans les SGBD

CHAPITRE III

Requêtes Evoluées dans les SGBD

I. Introduction

Nous nous intéressons dans ce chapitre aux propositions faites par les systèmes existants en ce qui concerne les expressions et le traitement de requêtes non classiques dans des SGBD, que nous appelons *requêtes évoluées* par la suite. Ce chapitre se base en grande partie sur [Mul91]. Par requêtes évoluées, nous entendons des requêtes intégrant des éléments qui mettent en jeu plus de sémantique que de simples requêtes du type SQL [Pas88] pour les SGBD relationnels, ou de ses extensions objets comme le langage de requête d'O_2 [Deu91].
Classiquement, le traitement d'une requête par un SGBD correspond à ce que l'on peut qualifier une correspondance stricte. Ceci veut dire que le système détermine les éléments de la base de données qui vérifient exactement la requête. Ce processus peut être représenté graphiquement par la figure III1.

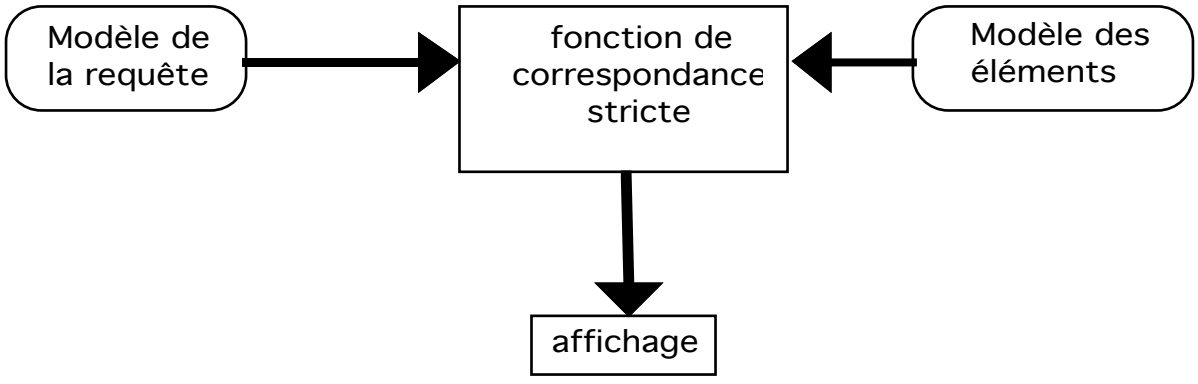


Figure III1: Traitement de requête classique par les SGBD.

Pour intégrer plus de souplesse dans ce processus, plusieurs solutions ont été

envisagées. En effet, les cibles du "flou" (i.e. de correspondances non strictes) sont différentes selon les systèmes.
Ce chapitre va s'articuler de la façon suivante. Tout d'abord, nous faisons une description générale des fonctionnalités supplémentaires nécessaires au traitement de ces requêtes évoluées. Ensuite, nous examinons suivant trois axes, que nous appelons "services", les requêtes évoluées des systèmes qui ont été étudiés:

- l'axe "emploi du schéma de la base de données",
- l'axe "facilité d'utilisation des structures des éléments de la base de données",
- l'axe "facilité d'expression de l'ordonnancement des réponses à une requête".

Ces trois points sont intéressants pour permettre l'expression de requêtes évoluées, même si dans ce travail nous nous sommes en grande partie limités au second axe cité ci-dessus.
Durant ces projections sur les axes ci-dessus, nous décrivons des propositions qui se situent dans un cadre qui se rapproche de nos travaux, tels que MULTOS [Tha90], VAGUE [Mot88] et les travaux de Pivert et Bosc [Bos91] basés sur une extension floue du modèle relationnel inspirée de la théorie des ensembles flous. Ces propositions ne se situent pas dans le contexte orienté objet, mais sont cependant intéressantes à examiner.
Dans le monde "objets", nous connaissons peu de propositions. Cependant, la notion d'i-égalité de Shaw et Zdonick pour le système ENCORE [Zdo89a] est un premier pas dans des correspondances plus souples d'objets. De plus, les propositions d'ORION qui visent à retrouver des objets reliés à un objet de départ suivant certaines contraintes sont intéressantes.
D'autres propositions, telles que celle basée sur l'emploi de regroupements flous [Kam90], et celle d'un langage flou basé sur un SGBD relationnel [Won90] vont également être étudiées.

II. Le traitement de requêtes évoluées

Le but du traitement de requêtes évoluées est de permettre à un système de faciliter le travail d'une personne qui pose une requête à un SGBD.

Une première approche est de faciliter l'écriture de la requête, puis *simplement* de l'étendre, c'est à dire de la traduire, pour qu'elle soit traitée comme une

requête non évoluée. Dans ce cas, c'est une phase de prétraitement qui effectue cette extension. Ce prétraitement, comme nous allons le voir par la suite, est utilisé en grande partie dans le but de simplifier l'utilisation du schéma d'une base de données.

Une autre approche consiste à faciliter l'expression des différentes correspondances réalisées durant le traitement de la requête évoluée (ces correspondances pouvant être strictes c'est-à-dire conclure "ça correspond" ou bien "ça ne correspond pas", ou bien non-strictes, c'est à dire donner une valeur de correspondance valuée située dans un intervalle prédéfini, qui est habituellement [0, 1]).
Dans le cas d'une requête non-stricte, c'est-à-dire d'une requête qui permet le calcul de la pertinence des éléments de la base qui correspondent plus ou moins à la requête, le souci de faciliter la tâche de l'utilisateur est toujours présent. En effet, l'intérêt de tels processus est de pouvoir exprimer en une seule requête ce qui, sans requête évoluée, devrait être décomposé en plusieurs.
Dans ce cas, les éléments du résultat sont ordonnés suivant leur valeur de pertinence estimée par le système (cf. partie V de ce chapitre). Dans cette optique, le traitement d'une requête évoluée trouve les éléments "idéaux" qui correspondent à la requête, ainsi que les "bons" éléments qui sont proches de l'idéal de la requête. Toute la difficulté est bien entendu de définir les termes "idéal" et "bon" pour les éléments de la base que l'on recherche.

Une dernière façon de faciliter l'expression d'une requête est, pour un système, d'évaluer si la réponse globale est jugée *bonne*, et de générer automatiquement une reformulation de la requête dans le cas contraire. Cette évaluation peut se baser sur plusieurs points, comme par exemple : le nombre de réponse, la pertinence moyenne de ces réponses, mais aussi sur une typologie de l'utilisateur. Le problème est alors de reformuler en essayant d'améliorer le résultat, tout en restant le plus proche possible de la requête initiale. Nous n'avons pas considéré ce point comme discriminant, au vu du peu de propositions faites à ce sujet.

D'après ce que nous venons de dire, le schéma du traitement de requêtes évoluées est donné dans la figure III2.

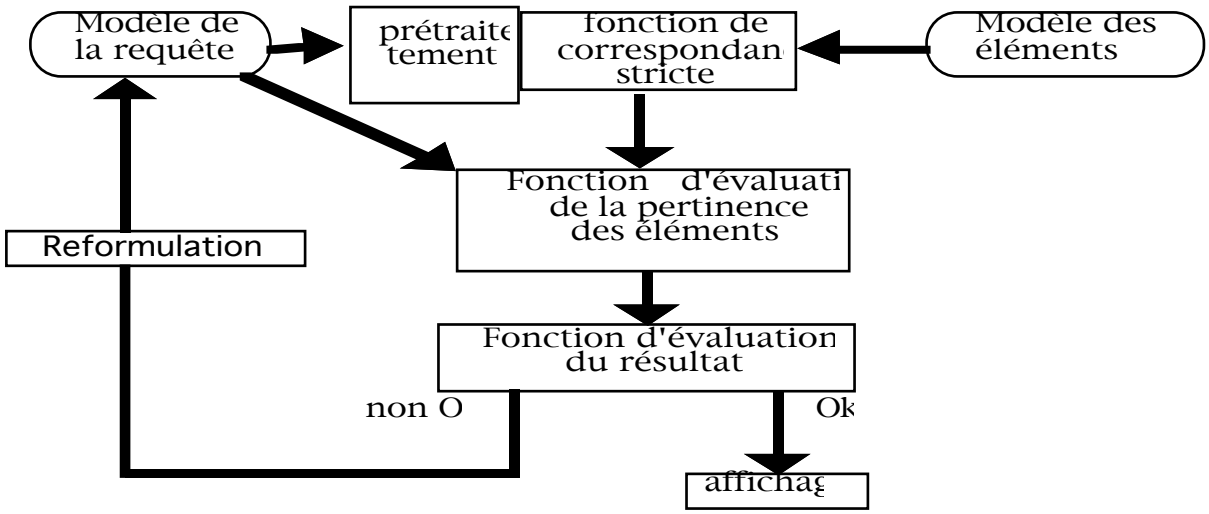


Figure III2: Traitement de requête évoluée.

Nous résumons ici les différentes parties du traitement de requêtes évoluées:

• Le prétraitement vise, pour un utilisateur, à utiliser des raccourcis d'écriture qui permettent une simplification de la requête. C'est au système d'utiliser la connaissance du schéma de la base pour "résoudre" les simplifications. De telles facilités sont permises avec des systèmes comme O^2FDL et OQL.
• La fonction de correspondance stricte est en fait un processus qui est similaire à celui des bases de données traditionnelles, qui intègre des fonctionnalités supplémentaires, comme dans ORION, ENCORE et le système basé sur des regroupements flous de [Kam90].
• La fonction d'évaluation de la pertinence des éléments permet de donner la valeur de similarité entre les éléments sélectionnés par la fonction de correspondance stricte. On rencontre dans MULTOS et le langage de requête flou pour bases de données [Won90] de telles fonctions.
• La fonction d'évaluation du résultat a pour but de déterminer dans quelle mesure le résultat est satisfaisant. VAGUE et MULTOS proposent des éléments simples pour réaliser cette fonction.
• La fonction de reformulation est liée à la fonction d'évaluation du résultat, et réalise une transformation de la requête initiale, en incluant les raisons pour lesquelles l'évaluation du résultat est jugée mauvaise.
• La fonction d'affichage, donne les résultats qui correspondent à une requête. Ce résultat est le but ultime d'une requête, et c'est à sa vue que l'on estime la qualité du traitement de requêtes floues.

Les systèmes que nous allons décrire dans la suite de ce chapitre n'incluent pas toutes les fonctionnalités ci-dessus, mais sont déjà un premier pas dans la réalisation de traitements les prenant en compte.

III. Les services "faciliter l'emploi du schéma d'une BD"

Dans cette approche, on ne remet pas en cause le processus habituel de traitement de bases de données, on rajoute seulement des éléments qui permettent de faciliter la formulation de requêtes en utilisant des connaissances sur le schéma de la base. Le traitement de ces simplifications est utilisé durant la phase de prétraitement (cf. figure III2).
Deux systèmes proposent ce genre de services, O^{2}FDL et OQL, sans se préoccuper de la notion d'ordonnancement.

III.1. O^{2}FDL

O^{2}FDL (Object Oriented Functional Data Language) est un langage fonctionnel de manipulation de données développé par M. Mannino, I. Choi et D. Batory à Austin [Man90]. Nous ne nous intéresserons qu'à son langage de requête.
Les constructeurs que l'on rencontre habituellement dans les bases de données orientées objet se retrouvent dans O^{2}FDL, nous ne nous étendrons pas sur ce sujet ici.
Considérons deux classes, les professeurs et les d'étudiants. Nous considérons que les classes Etudiant et Professeur sont décrites par un *nom* qui est une chaîne de caractères. De plus, il existe une fonction sur les étudiants qui permet de trouver les enseignants d'un étudiant donné, fonction multivaluée. Cette structure est décrite par le schéma de la figure III3, les flèches doubles indiquent une fonction multivaluée, et les flèches simples des fonctions monovaluées.

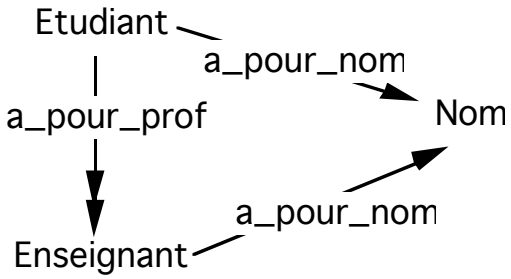


Figure III3: Un exemple de schéma O^{2}FDL

Examinons une requête qui exprime le fait que l'on cherche les noms des professeurs d'un étudiant. La requête exprimée en O^2FDL sans simplification serait:

```
Etudiant!.a_pour_prof.LR{nom_prof := a_pour_nom}        (1)
```

Cette requête est décomposée de la manière suivante:

• parmi chaque objet persistant de la classe Etudiant, noté “Etudiant!”,
• on veut les professeurs de cet étudiant, noté “a_pour_prof”,
• et le résultat de cette requête est un ensemble de n-uplets possédant un attribut nommé *nom_prof*, attribut dont la valeur est donnée par l'application de la fonction “a_pour_nom” à chaque objet professeur trouvé.
• la composition des ces différentes fonctions est exprimée par la notation pointée “.”.

Dans le cadre du prétraitement d'une requête, il est possible de "simplifier" la requête (1) en tenant compte des points suivants:

• dans le cas où il n'existe pas d'ambiguïté, on peut remplacer le nom d'une fonction de liaison d'une classe à une autre par le nom de la classe d'arrivée de la fonction,
• on peut omettre l'expression de la fonction “!” lors de l'expression d'une requête, celle-ci sera par défaut appelée sur le nom de classe en première position dans la requête,
• il est possible de ne pas utiliser le nom de fonction “LR”, et de se limiter à l'emploi des parenthèses ouvrantes et fermantes pour déclarer l'utilisation de cette fonction.

Sur l'exemple de la fonction (1), on peut employer la première simplification pour le passage de la classe Etudiant à la classe Enseignant, de même que pour le passage de la classe Enseignant vers la classe Nom. L'expression de la formule (1) devient donc la suivante, en se servant de toutes les simplifications à notre disposition:

```
Etudiant.Enseignant.{nom_prof := Nom}     (2)
```

III.2. OQL

Le langage de requête pour manipuler des bases de données orientées objet OQL (object Query Language) a été défini à l'université de Floride [Ala89].
Par rapport au cas précédent, un pas est franchi dans la mesure où cette idée d'utiliser les noms de classes plutôt que les noms des liens est étendue à des liaisons indirectes entre classes.
OQL opère sur le modèle sémantique de données OSAM [Su88], qui permet de définir des classes d'objets associées par des relations d'héritages, ainsi que par des relations de composition ou bien d'interactions contraintes.
Une requête en OQL est considérée comme une fonction qui renvoie une sous-base de la base de départ. On peut donc rapprocher un résultat de requête d'une vue de la base qui est comme un masque.
Le modèle de données a les caractéristiques que l'on retrouve habituellement dans les modèles de données orientés-objets, c'est à dire qu'il possède une hiérarchie d'agrégation, une hiérarchie de généralisation, chaque objet a un identificateur unique.
Prenons l'exemple d'une base de données Université décrite par le schéma de la figure III4, dans lequel:
• un trait non-barré indique un lien d'agrégation (qui est un lien de référence) de cardinalité non-nulle, et un trait barré exprime le fait que le lien d'agrégation peut avoir une cardinalité nulle,
• un “A” signifie un lien d'agrégation, et un “G” un lien de généricité.

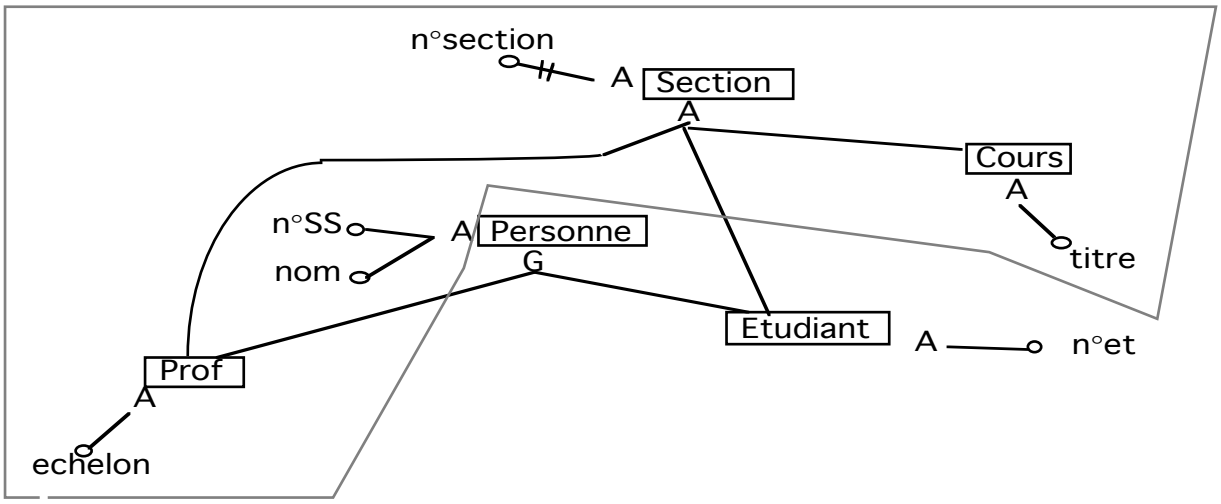


Figure III4: Le schéma de la base de donnée Université.

Comme nous l'avons dit plus haut le résultat d'une requête en OQL est une sous-base qui se rapproche d'une vue.

Posons la requête **(R)** définie comme suit, dont le résultat a pour schéma la partie de la base de données Université *entourée de pointillés* :

```
context Prof * Section * Cours
  select  n°ss, nom, echelon, n°section, titre
display
```

Cette requête utilise les classes Prof, Section et Cours pour déterminer les cours des professeurs d'après les sections où ils enseignent. Si le diagramme d'extension normalisé[0] initial est représenté figure III5, le fait d'utiliser "Prof * Section * Cours" dans la partie *context* de la requête va indiquer que le résultat ne contiendra que les objets des trois classes qui sont liés trois par trois, ce qui nous donne le diagramme extensionnel du résultat figure III6.

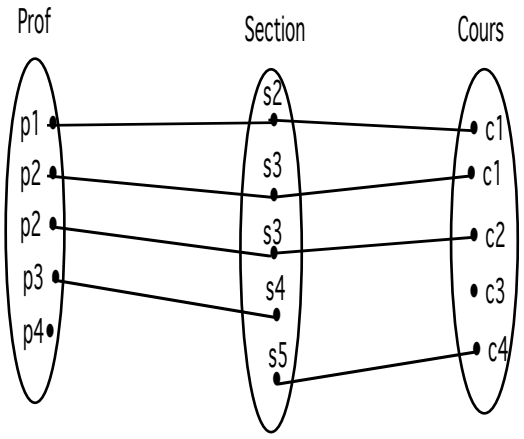


Figure III5: Diagramme extensionnel normalisé d'une sous-base.

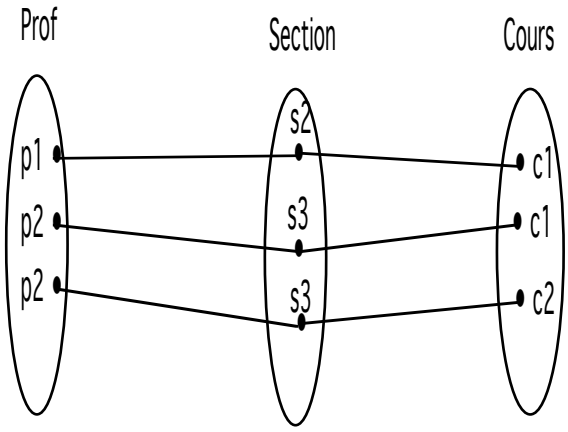


Figure III6: Diagramme extensionnel normalisé du résultat de la requête (R).

Ce résultat (i.e. schéma et diagramme extensionnel) constitue donc bien une vue de la base de départ.

La marche à suivre pour poser une requête en OQL est la suivante : d'abord

[0] La normalisation consiste à n'avoir que des liens 1:1 (en créant des doubles dans les "ensembles" d'objets).

spécifier la sous-base désirée, en décrivant son schéma et l'ensemble des types de structures extensionnelles qui nous intéressent, et ensuite les opérations appliquées sur les classes de cette sous-base. Ce langage de requête diffère des langages de requêtes habituels en orienté objet, du fait que l'on ne privilégie pas une classe (ou un type) qui est le point de départ de la requête, mais que toutes les classes de la sous-base de départ ont même importance.
On remarque deux sortes de prétraitement quand on exprime une requête en OQL.

• Le premier se rapproche de ce qui est proposé dans O^{2}FDL. En effet, on n'indique pas le nom de la liaison entre deux classes, mais le nom des classes (quand il n'y a pas ambiguïté), par exemple "Prof*Section*Cours" (l'opérateur "*" va permettre de ne garder que les triplets d'objets regroupant un objet de chaque classe tels qu'un objet Prof est relié à un objet Section, lui même en relation avec un objet Cours). Quand des classes sont reliées par plusieurs liens, il convient d'indiquer le nom du lien utilisé dans la requête.

• Le second se trouve quand on emploie des liaisons via des classes intermédiaires. Par exemple, au lieu d'écrire "Prof*Section*Cours", s'il n'y a pas d'ambiguïté dans le chemin on peut exprimer "Prof*Cours". Le système se chargeant de trouver les objets professeurs et cours liés via les sections. Ce dernier point est une généralisation de ce que propose O^{2}FDL, où on ne peut exprimer que des liaisons entre deux classes directement en relation. Dans la requête **(R)**, on aurait donc pu se limiter à indiquer “Prof*Cours” dans la partie *context*.

III.3. Discussion

Nous venons d'expliquer ce que proposent certains systèmes pour faciliter l'expression de requêtes, en **permettant de ne pas utiliser le schéma d'une base de données tel qu'il a été défini**.
On remarque que ces prétraitements uniquement basés sur le schéma de la base ne sont proposés que sur des systèmes orientés objet. En effet, à cause de la structure plate des n-uplets de bases de données relationnelles, ces dernières ne nécessitent pas ce genre de facilité. Ces facilités se rapprochent de l'utilisation de requêtes graphiques [Ando92], dans la mesure où l'on utilise une notion de contiguïté géographique, simple comme dans O^{2}FDL ou bien transitive comme dans OQL, pour faciliter l'expression de requêtes.

La proposition d'O^{2}FDL possède l'avantage de ne pas avoir à se préoccuper obligatoirement du nom des fonctions qui permettent le lien entre les classes. Le traitement de ces simplifications n'est réellement qu'une interprétation de la requête initiale de manière à fournir à la fonction de correspondance la requête "complète" (i.e. sans simplification) qui correspond à la requête initiale.

OQL va plus loin, dans la mesure où, il permet d'"inférer" les relations transitives entre classes, mais utilise ces relations au niveau de l'interprétation de la requête ainsi qu'au niveau extensionnel de la base. L'opérateur d'association "*", par exemple, filtre les objets suivant des règles précises que nous avons vues plus haut. Cet opérateur est donc aussi une facilité offerte pour l'expression d'une requête. Par rapport à SQL, où à ses dérivés pour les objets, OQL réalise des opérations qui permettent de factoriser l'expression de requêtes.

Si nous restons au niveau de l'expression de liens entre classes, ces propositions ne sont utilisables que dans le cas où il n'existe pas dans le schéma d'une base plusieurs chemins qui vont d'une classe à une autre. Dans ce cas, qui peut être qualifié d'ambiguïté de chemin entre classes, on en revient aux techniques habituelles où le nom explicite du lien (fonction pour O^{2}FDL ou lien d'agrégation pour OQL) doit être utilisé pour ôter cette ambiguïté.
De plus, il semble que ces facilités sont utilisables seulement dans le cas d'une bonne connaissance du schéma de la base. En effet, comment savoir que le lien entre les classes Prof et Section dans l'exemple d'OQL n'indique que les sections dans lesquelles enseigne un professeur **cette année**. Si cette liaison n'a pas la même sémantique pour une personne qui pose une requête, les résultats risquent d'être folkloriques!
Ces facilités ne sont donc utilisables que lorsque l'on connaît très précisément le schéma de la base, mais ont cependant le gros avantage de n'avoir à se souvenir, dans la plupart des cas, que d'un ensemble restreint de noms. De plus, une modification du schéma n'entraîne pas obligatoirement la modification des requêtes, ce qui est intéressant. Ce dernier point n'est pas valable dans tous les cas, car le système peut alors se trouver face à des ambiguïtés qui amèneront à une nouvelle expression de requête.

Ces approches constituent une première phase qui indique que maintenant, on commence à se préoccuper de faciliter l'expression des requêtes. Notre travail a également un souci de faciliter l'expression de requêtes, même si nous nous

situons au niveau des objets et non pas du schéma de la base.

IV. Les services "Emploi de structure"

Nous nous intéressons ici aux propositions qui sont faites sur l'utilisation de la structure des éléments de la base lors du traitement des requêtes.
Par rapport au cas précédent, nous ne nous intéressons plus à la définition syntaxique de la classe des objets, mais à des traitements qui diffèrent suivant chaque objet.
Le traitement de ces propositions ne se situe donc plus au niveau du prétraitement des requêtes, mais au niveau de la fonction de correspondance et/ou au niveau de l'évaluation de la pertinence des éléments sélectionnés. Ces approches permettent de simplifier l'expression d'une requête qui prend en compte la structure des éléments de la base, sans avoir à écrire explicitement, en utilisant un langage de programmation, ce genre de traitement.

Nous différencions ici les traitements explicites, c'est à dire exprimés dans la requête, des traitements implicites qui sont réalisés automatiquement.

Des propositions n'existent que dans le cadre d'éléments complexes, que ce soient des objets ou non.
- Au niveau des traitements explicites, nous allons nous intéresser à trois propositions:

• ORION [Kim89a] propose une méthode qui permet de se déplacer dans la structure des objets
• l'algèbre de requêtes développée par Shaw et Zdonick pour le système ENCORE [Zdo86], qui inclut la définition de l'égalité d'objets complexes basée sur la structure de ces objets. Des versions simplifiées de cette égalité existent dans le système O_2 , ORION [Kim89b], OQL et LIFOO [Lem89].
- Pour les traitements implicites, MULTOS [Rab90] réalise des reformulations sur les attributs textuels de manière automatique pour "élargir" le champ de recherche dans le cas d'un premier échec, et le traitement de regroupements (*clusters*) flous [Kam90] permet l'utilisation de concepts plutôt que d'attributs des données de la base.

IV.1. Les traitements de structures explicites

Ces traitements sont des nouvelles opérations sur les éléments de la base.

IV.1.1. ORION

Nous allons commencer par décrire l'opérateur "recurse", avant de traiter les méthodes "pères", "ancêtres" et "descendants".

Dans ORION [Kim89c], l'opérateur *recurse* est utilisé sur des objets dont la représentation du type de la classe par le schéma de hiérarchie-composition (SHC) forme un cycle. Nous n'allons pas ici entrer en détail dans la description d'un schéma de hiérarchie-composition d'ORION. Un tel schéma ressemble à ce qui à été décrit pour OQL, c'est à dire que le SHC représente à la fois les liens hiérarchiques entre les classes (“G” dans OQL) et les liens de composition d'attributs (“A” dans OQL).

Nous allons décrire sur un exemple l'utilisation de l'opérateur *recurse*. Considérons une classe Employé qui possède les attributs suivants:

- un nom qui est une chaîne de caractères,
- une référence vers un autre employé qui est le directeur direct de cette personne.

Le SHC de cette classe est donné dans la figure III7.

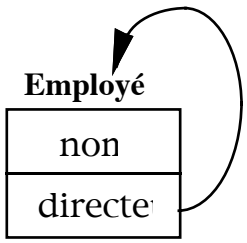


Figure III7: Description de la classe Employé.

Si nous voulons obtenir tous les directeurs d'un employé de nom "DURANT", c'est-à-dire effectuer une requête récursive, il va suffire d'écrire la requête suivante:

```
select :E (recurse directeur)
from Employés :E
where :E nom = "DURANT
```

Le traitement appliqué à cette requête est le suivant:
L'expression ":E (recurse directeur)" donne un ensemble d'objets de la classe Employés qui contient les objets résultats de :

:E directeur
:E directeur directeur
…

Le résultat de la requête est un ensemble d'ensembles d'objets de la classe Employés, c'est à dire un ensemble d'Employés (les directeurs) par Employé.

Le second type de simplification d'ORION n'est pas limité aux simples traitements de requêtes, mais dénote un souci de reformulation de la structure d'un objet de manière à faciliter l'expression des traitements sur ces derniers.
Nous ne décrivons ici que la méthode *descendants* qui nous a paru la plus intéressante dans le cadre de notre étude, bien que les méthodes *pères* et *ancêtres* soient également proposées.

Ces méthodes tiennent compte des types de liens entre les objets. Dans ORION, ces liens sont au nombre de trois (cf. Chapitre II partie II.2) :

- **les liens de référence dépendants du père**,
- **les liens de référence indépendants du père,**
- **les liens de propriété**.

La méthode *descendants* possède les caractéristiques suivantes :

• descendants *niveau types_des_descendants type_des_liens* : cette méthode donne l'ensemble des descendants de niveau *niveau* (0 pour tous les descendants, 1 pour les fils, 2 pour les fils et les petits-fils, …) de type *types_des_descendants* via le *type_des_liens*. Si par exemple on représente un moteur (du type Moteur) sous la forme d'un objet complexe, composé de sous-parties, etc, on peut retrouver toutes les courroies (de type Courroie) qui composent ce moteur par cette méthode.

IV.1.2. ENCORE

Dans l'algèbre de requêtes décrite par Zdonik et Shaw, il est possible de savoir à quelle profondeur de structure deux objets sont égaux récursivement.

Les prédicats proposés permettent de conclure sur une "identité" de structure entre deux objets. [Zdo89a], [Zdo89b] et [Zdo90] donnent la définition de l'i-égalité. Deux objets sont 0-égaux s'ils ont même identificateur d'objet, de plus :

• deux "objets ensembles" sont i-égaux s'ils ont même cardinalité et s'il y a une correspondance entre les éléments de ces ensembles, et que ces objets sont i-1-égaux,

• deux objets ("non-ensembles") sont i-égaux s'ils ont même type, et que leurs attributs correspondants ont des valeurs i-égales.

Dans [Zdo89b], des exemples d'objets i-égaux sont donnés :
Prenons trois objets de la base S1, S2 et S3 qui ont la structure décrite dans la figure suivante :

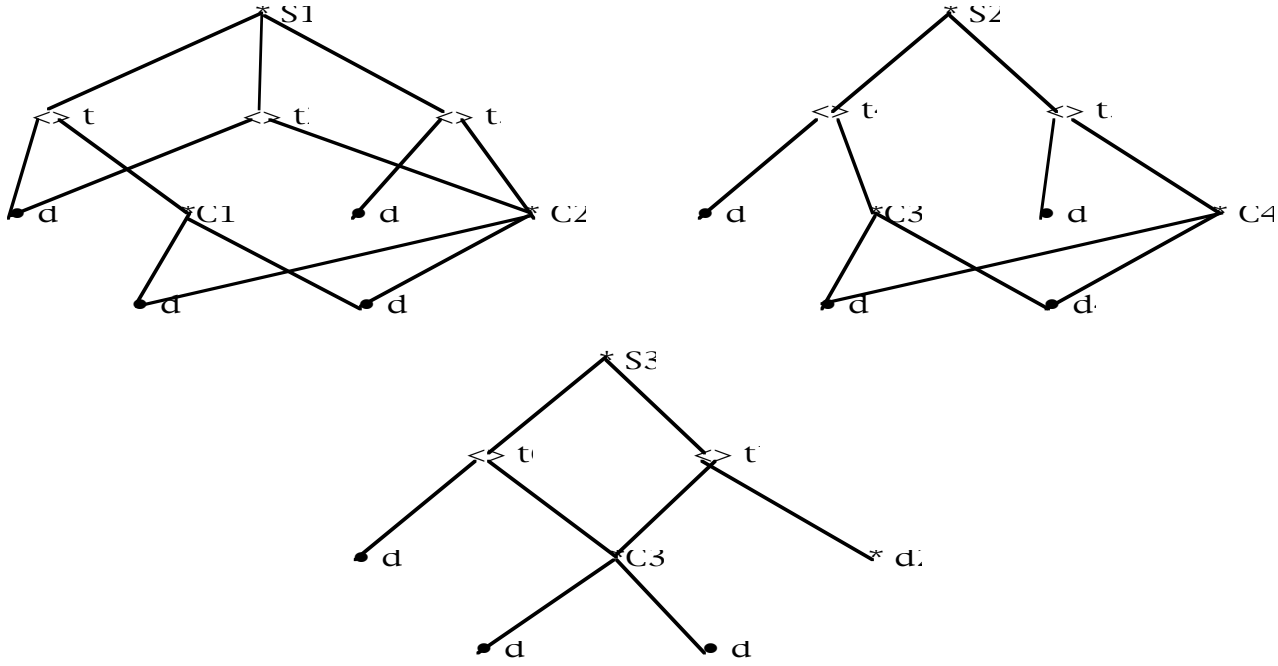


Figure III8: Exemple d'objets pour l'i-égalité.

Pour vérifier si ces objets sont i-égaux, il convient de partir des objets les plus profonds dans la représentation structurelle des objets. Les "objets ensembles" C3 et C4 sont 1-égaux, il en découle que les objets n-uplets t4 et t6 et les objets t5 et t7 sont 2-égaux. Les "objets ensembles" S2 et S3 sont 3-égaux, alors que S1 n'est lié par aucune relation d'égalité à S2 ou S3 (car S1 et S2 n'ont pas le même

nombre d'éléments).

IV.1.3. MULTOS

MULTOS est un un système de gestion de documents bureautique incluant des fonctionnalités de recherche d'informations. Pour le traitement de structures explicites, le système MULTOS propose une approche un peu plus "intelligente" que ce que nous venons de voir. En effet, il permet de tenir compte des types (textuels) des attributs des documents gérés.

Dans MULTOS, dont le but est de gérer les documents bureautiques, les documents possèdent une structure conceptuelle sur laquelle portent les requêtes des utilisateurs. Par exemple, pour des lettres d'offre, la structure conceptuelle peut être donnée dans la figure III9.

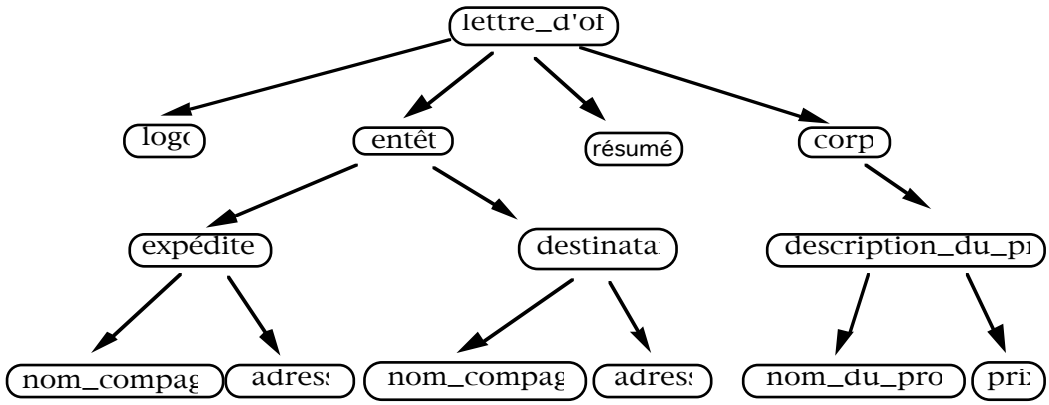


Figure III9: La structure conceptuelle d'une lettre d'offre.

Le système valorise, en analysant le document, chacun des attributs terminaux (nom_compagnie, …) de l'arbre qui représente la structure conceptuelle d'une lettre d'offre.
Si on veut chercher si un document parle de "base de données", il suffit de poser la requête suivante, en employant le mot-clé TEXT :

```
FIND DOCUMENTS Article
(TEXT CONTAINS "base de données") HIGH
```

Nous reviendrons par la suite sur le mot HIGH, dans la partie V de ce chapitre consacrée aux possibilités d'ordonnancement. La requête ci-dessus ne fait pas

référence explicitement à la structure des textes. L'utilisateur, lorsqu'il pose sa requête, peut ignorer la représentation de ces documents. Comme le système a la connaissance complète du type, de la structure des lettres d'offres et de la manière de traiter une requête sur ces documents, il n'est pas nécessaire que l'utilisateur connaisse la structure logique des documents. Cette facilité donnée à l'utilisateur est possible car les types de documents et les stratégies de recherche sont figés.

IV.1.3. Discussion

L'intérêt de méthodes prédéfinies dans ORION, est de permettre à l'utilisateur d'utiliser des méthodes prédéfinies suffisamment générales, sans qu'il connaisse la structure encapsulée des objets. Si de telles méthodes ne sont pas proposées et que le programmeur a besoin de tels outils, il est donc obligé de les écrire. L'écriture de ces méthodes prend du temps et est sujette à des erreurs de programmation, ce qu'évitent des méthodes prédéfinies qui réalisent ces tâches. L'intérêt de ces méthodes est d'utiliser explicitement les déplacements dans des objets de manière déclarative.
D'une certaine manière, on peut considérer que ces méthodes renvoient des "vues" de l'objet receveur, ces vues permettant de comparer des objets qui peuvent avoir des structures différentes.
Cette approche n'est pas sans rapport avec la nôtre, mais il faut cependant indiquer que ces vues n'utilisent que la structure des objets et qu'il en découle une pauvreté sémantique de ces méthodes. Pour notre travail, l'ajout de sémantique supplémentaire sur les objets étend ce que les méthodes prédéfinies d'ORION permettent à un niveau uniquement structurel.

Comme nous l'avons dit précédemment, les méthodes prédéfinies, par opposition au *recurse*, peuvent être utilisées en dehors d'une requête. Avec l'opérateur de requête *recurse*, on indique le nom de l'attribut que l'on veut suivre récursivement. Cet opérateur n'a pas été exprimé sous forme de méthode, car il n'est pas possible d'indiquer le nom d'un attribut comme paramètre effectif d'une méthode dans ORION. Donc, par opposition aux méthodes prédéfinies dans lesquelles tous les chemins sont explorés, avec *recurse* on n'en utilise qu'un seul. Cet opérateur est donc plus "fin" que les méthodes prédéfinies au niveau des déplacements dans les structures d'objets.
On peut en fait différencier ces deux propositions en qualifiant les méthodes de

recherche en *largeur* dans la structure des objets, et l'opérateur de recherche en *profondeur*. Ces deux possibilités permettent donc de résoudre des problèmes sur les objets qui entrent dans ces deux catégories.

MULTOS étend la notion introduite par les méthodes d'ORION. En effet, la connaissance utilisée par le système n'est plus seulement structurelle, mais également sémantique, dans la mesure où le système sait que lorsqu'une requête utilise le mot *TEXT*, il doit rechercher les termes de la requête dans toutes les composantes textuelles des document considérés. Il faut remarquer que MULTOS a pour but la gestion de documents bureautiques, ce qui est une application bien spécifique. Un système de gestion de bases de données se devant de pouvoir répondre à *tous* les cas possibles, il est nécessairement plus général, et c'est pourquoi ORION ne propose pas ce genre de facilité.

L'approche d'ENCORE est radicalement différente de ce qui est fait dans ORION ou dans MULTOS, car on ne se base plus sur un seul objet, mais sur deux objets que l'on compare suivant certaines caractéristiques. Cette approche peut être vue comme cumulant ce que fait ORION avec ses méthodes prédéfinies, en parallèle sur les deux objets comparés, et un processus de comparaison sur les résultats de ces déplacements.
Donc, en plus d'un processus de simplification d'écriture de la requête, l'i-égalité exécute un processus de comparaison sur les objets. Comme dans ORION, ce processus de comparaison est très limité, dans la mesure où seule la structure des objets est utilisée comme source de connaissance.
Une comparaison plus fine d'objets nécessite de prendre en compte d'autre(s) connaissance(s), ce que nous faisons dans notre travail.

IV.2. Les traitements de structures implicites.

Nous nous intéressons maintenant à des traitements permettant à un utilisateur de poser une requête sans se préoccuper de la structure des données recherchées. Ces traitements sont implicites, dans la mesure où la personne qui pose la requête ne se pose pas de question sur la structure des objets. Ce dernier élément est différent des traitements de requêtes explicites, dans lesquels la structure devait être à l'esprit lors de la formulation d'une requête.

IV.2.1. Utilisation de regroupements flous

Ici, l'utilisateur peut poser des requêtes **sans une connaissance précise de sa structure**. Dans cette partie, nous décrivons comment dans [Kam90], les auteurs utilisent une technique de clusters flous pour permettre ce type de requêtes. Dans l'article, les auteurs ne se basent que sur des "descripteurs qualitatifs numériques", c'est à dire sur des approximations des attributs numériques.

Pour permettre des requêtes évoluées, le concepteur de la base utilise trois concepts qui sont:

- les termes flous : par exemple : *âgé, jeune, très âgé.*
- les concepts flous : par exemple : *âge d'employé, âge de machine.*
- les types de concepts flous : par exemple : *âge.*

Un type de concept flou est une abstraction d'un ensemble de concepts flous. Un type de concept flou décrit les termes flous pour tous les concepts flous de ce type. Un concept flou est une fonction d'attributs des éléments de la base. Prenons par exemple le concept flou *âge d'employé* de type *âge*. Ce concept flou sera par exemple "Date_courante - Date_naissance" pour un employé quelconque (l'opérateur "-" étant la soustraction sur des dates). Pour un autre concept comme *âge de machine* de type *âge*, la fonction de calcul peut être la suivante : "Date_courante - Date_de_mise_en_service". Un terme flou est la caractéristique d'un concept flou que l'on peut exprimer dans une requête. Par exemple "âge d'employé jeune" indique que l'on cherche, pour le concept flou âge d'employé, la caractéristique "jeune" décrite par ce terme flou.

Nous allons dans l'exemple qui suit utiliser une base de données sur des cigarettes. Nous utilisons un seul type de concept flou *longueur*, qui correspond à l'ensemble de termes flous suivant : {très court, court, moyen, long, très long}. Nous avons deux concepts flous qui sont *longueur_du_filtre* et *longueur_de_tabac*, tous deux de type *longueur*. L'expression de la formule de ces deux concepts n'est pas utile ici.
Une requête de l'utilisateur, posée sous forme de langue naturelle, peut être la suivante :
"Je veux les cigarettes qui ont une longueur de filtre courte et une longueur de

tabac longue".

Quand l'utilisateur pose une requête, son traitement est divisé en trois parties pour chaque requête élémentaire, une requête élémentaire étant définie comme une partie de la requête qui correspond à un concept flou caractérisé par un terme flou :

1. trouver si le terme flou existe. Dans l'exemple donné, pour la première requête élémentaire ce terme est "court", pour la seconde "long".
2. trouver si le concept flou existe. Dans notre exemple le premier concept flou est "longueur_de_filtre" et le second "longueur_de_tabac".
3. renvoyer les éléments des regroupements qui correspondent à cette requête élémentaire.

Suivant les opérateurs qui relient les requêtes élémentaires, on fait des unions ou des intersections de clusters pour donner le résultat d'une requête complète. Dans notre exemple, on fait une intersection des regroupements d'éléments de la base qui correspondent à chaque requête élémentaire, car les deux parties de requêtes sont liées par une conjonction.

IV.2.2. MULTOS

Nous avons déjà parlé du système MULTOS dans la partie sur les traitements de structures explicites. Examinons un exemple de requête sur un attribut de la structure conceptuelle d'un document. Nous considérons le type de document *Lettre_d'offre* (cf. partie précédente IV.1.3). La requête suivante :

```
FIND DOCUMENTS Lettre_d'offre
WHERE (.résumé CONTAINS " base de données") HIGH
```

signifie que l'on cherche des lettres d'offres qui ont dans leur résumé les termes "base de données".
La démarche suivie lors du traitement de cette requête est la suivante:
• la recherche des termes "base de données" va être effectuée tout d'abord dans la partie de texte correspondant au résumé du document,
• si les termes recherchés ne sont pas dans le résumé du document, alors une recherche est effectuée sur *tous* les attributs textuels du document.
Le traitement implicite qui est automatiquement réalisé est la *reformulation* de

la requête pour rechercher les termes de la requête sur l'ensemble des éléments textuels de ce document.

IV.2.3. Discussion

Nous venons de décrire deux solutions pour traiter les structures des éléments d'une base de données de manière implicite.
Une limite du système se basant sur des clusters flous, est le fait de ne pas proposer des prédicats binaires sur les concepts flous, par exemple "longueur de cigarette environ ≤ 30 cm" en particulier à cause du fait que les bornes des valeurs des attributs ne sont pas indiquées comme caractéristiques des clusters. Si ces bornes existaient, par exemple deux clusters de bornes respectives [0,25] et [25,50] pour le concept "longueur" de cigarette, le calcul serait fait de la manière suivante : i) on cherche à quel(s) cluster(s) appartient la valeur "30" pour la longueur de cigarette, ii) on cherche, dans ce(s) cluster(s) tous les éléments qui vérifient cette requête élémentaire, iii) on fait l'union de l'ensemble de ces éléments avec les clusters dont la borne supérieure est plus petite que "30". Comme ces bornes n'existent pas, il est nécessaire de passer en revue une grande partie des éléments de la base avant d'être certain d'obtenir tous les éléments qui correspondent à la requête.
Cependant, cette approche possède l'intérêt de permettre des requêtes sans tenir compte à un seul moment de la structure des éléments de la requête. Il faut signaler que ces propositions sont limitées à des attributs numériques, mais qu'a priori elles sont applicables à d'autres types de données possédant des relations d'ordre total.

MULTOS utilise une stratégie qui semble éloignée de ce qui est décrit ci-dessus avec les clusters flous. Il utilise la structure des documents pour élargir le champ de recherche initial lorsque celui-ci n'est pas suffisant. Cependant, on peut comparer ces deux approches en considérant le nom de l'attribut textuel sur lequel porte la requête comme un concept flou, qui sous-entend une recherche dans tous les attributs textuels du document si l'attribut du document ne correspond pas à la requête. Il est clair que ceci n'est qu'un *rapprochement*, car la notion de condition n'apparaît pas dans le traitement des clusters flous.
On peut réaliser un parallèle entre notre travail et ce que propose MULTOS. En effet, nous tentons d'étendre notre recherche, lors de la comparaison de deux éléments de la base de données, cependant nous utilisons des données

supplémentaires à ces éléments.

V. Les services "Ordonnancement"

V.1. Introduction

Dans ce que nous avons vu jusqu'à présent, nous ne nous sommes pas préoccupés d'un point important, si l'on se focalise sur les requêtes "évoluées", qui est l'ordonnancement des réponses à une requête suivant leur pertinence pour cette requête. En effet, si on a la possibilité d'exprimer finement que certains éléments de la base correspondent mieux que d'autres à une requête, il semble naturel de fournir comme résultat les éléments qui correspondent très bien, puis ceux qui correspondent un peu moins bien, etc… Cet ordonnancement se situe dans la "fonction d'évaluation de la pertinence des éléments" (cf. figure III2).
Basés sur cette analyse, différents systèmes se sont intéressés à cet ordonnancement en prenant des voies différentes. Le langage de requête flou pour bases de données [Won90] réalise un traitement qui ordonne les réponses à une requête suivant leurs attributs numériques. MULTOS propose une approche relativement similaire, mais sans se limiter à des attributs numériques. Avec le système VAGUE [Mot88], on arrive à des systèmes souples qui permettent de définir à un utilisateur ses propres notions de similarités entre éléments de la base. Finalement, nous allons nous intéresser à l'approche de Pivert et Bosc [Bos91], qui exprime comment modéliser, grâce à la théorie des ensembles flous, les notions de similarités entre éléments d'une base de données.

V.2. Un langage de requête flou pour bases de données.

Dans le langage de requête flou décrit par Wong et Leung dans ([Won90]), une requête est composée d'une partie stricte et d'une partie floue. La partie stricte est ce que l'on trouve habituellement dans un langage de requête d'un SGBD.
La partie floue permet de donner des valeurs de préférence pour certaines valeurs d'attributs, ces attributs ayant uniquement des domaines numériques.
Les auteurs se basent sur la théorie des ensembles flous citée dans [Raj88]. Dans cette partie, l'évaluation de la requête ne se limite pas uniquement à additionner les valeurs trouvées pour chaque partie de la requête. On utilise des opérateurs flous appliqués sur les résultats des parties élémentaires de la

requête. Les parties élémentaires de la requête sont appelées *tables de distribution*. Elles consistent à associer à chaque valeur d'un attribut une mesure située dans l'intervalle [0,1]. Cette mesure permet à l'utilisateur d'indiquer de façon assez fine les valeurs qui l'intéressent le plus. La partie floue d'une requête est exprimée de la manière suivante :

Req_floue ::= (*op_flou* exp_floue [exp_floue]*)
exp_floue ::= (*nom_attribut table_de_distribution poids*) | Req_floue

La table de distribution est donc composée de deux listes :

1. une *liste de valeurs croissantes de l'attribut* ,
2. une *liste des valeurs de préférences correspondantes* à chacune des valeurs de la liste précédente.

La première partie est une liste de valeurs croissantes de l'attribut considéré, cet attribut doit donc être d'un domaine qui possède une relation d'ordre comme les entiers ou les réels, ou sur lequel une telle relation peut être définie. La seconde liste, qui exprime les valeurs de préférences de chacune des valeurs de l'attribut, est composée de réels compris entre 0.0 et 1.0. Plus la valeur de préférence est grande, plus la valeur correspondante de l'attribut est préférée.

L'analyse de cette table, pour une valeur d'attribut donnée, va **calculer**, à partir de la position de la valeur de l'attribut dans la liste des valeurs ordonnées de façon croissante, la valeur de l'élément courant pour cette requête élémentaire. Ce calcul réalise la fonction d'appartenance de l'élément pour cette partie de requête. Pour notre exemple sur les personnes d'une entreprise, la table de distribution d'appartenance pourra être pour l'attribut *âge* :

((18 19 20 21 25 29 30 31 32)
(0.0 0.5 1 0.5 0.2 0.5 1 0.5 0.0))

Si, pour une personne p, la valeur de l'attribut âge est 23 pour un des éléments de l'ensemble auquel on applique la requête, la valeur de la correspondance avec la requête sera de (0.5+0.2)/2 ce qui donne 0.35 (plus la valeur de correspondance est grande, plus l'élément correspond à la requête).
Le critère d'ordonnancement des réponses est lié aux préférences données par l'utilisateur.

Au niveau du langage de requêtes, un certain nombre d'opérateurs sont autorisés : *and*, *or* et un opérateur spécifique *poll*. Les opérateurs *and* et *or* sont ceux de la théorie des sous-ensembles flous et ont la même sémantique. L'opérateur *poll* va servir dans le cas où chaque critère compte de manière identique, par exemple les personnes en bonne forme qui ont la taille la plus grande. En effet, le résultat fourni par cet opérateur est la moyenne des résultats des tables de distribution données pour les opérandes de cet opérateur. On peut également signaler qu'il est possible d'exprimer le fait qu'une table de distribution a plus d'importance qu'une autre; ceci permet d'exprimer des préférences, non plus sur les valeurs des attributs des éléments de la base, mais sur des parties de la requête. Le *and* et le *or* ne sont pas pondérés.

V.3. MULTOS et OFFICER

Nous avons déjà cité le système MULTOS ([Ber88], [Rab90]) précédemment. OFFICER ([Cro90]) est un système de recherche d'informations. Nous avons regroupé ces deux systèmes en raison de l'identité de propositions au niveau de l'ordonnancement des réponses de leur langage de requêtes. Ces deux systèmes permettent d'indiquer une valeur de préférence pour les valeurs des attributs, ainsi que des valeurs d'importance des parties de la requête.
Dans OFFICER, si on s'intéresse, par exemple, à l'attribut *date* (dont le domaine de valeur est {Janvier, …, Décembre}), et que l'on cherche des documents parus "aux alentours" du mois de septembre, on l'exprimera dans la requête de la façon suivante :

date = août ACCEPTABLE
date = septembre PREFERRED
date = octobre ACCEPTABLE

Si l'utilisateur caractérise l'égalité "date = août" par le terme ACCEPTABLE, cela signifie que la valeur "août" pour l'attribut "date" est pour lui moyennement intéressante; par contre l'égalité "date = septembre" avec la caractéristique PREFERRED indique que c'est cette valeur d'attribut qui nous intéresse le plus; les documents qui vérifient cette égalité seront fournis en tête dans le résultat. Le processus sous-jacent qui traite la requête associe un poids plus important au document, si une condition vérifiée est caractérisée par PREFERRED et plus faible pour la caractéristique ACCEPTABLE. Le poids associé au terme

PREFERRED est de 1, et le poids associé à ACCEPTABLE est 0,3. Ces valeurs sont prédéfinies et constantes. Pour toutes les valeurs de l'attribut qui ne sont pas indiquées dans la requête (dans l'exemple précédent les mois de janvier,..., juillet, novembre, décembre), le traitement revient à considérer qu'elles sont indiquées dans la requête avec la caractéristique NOT ACCEPTABLE, ce qui donne le poids de 0 à cette partie de requête. La somme des poids de chaque partie de la requête est effectuée pour chaque document : si une requête comporte deux parties et que les poids, pour un document donné, sont de 0,3 pour la première et de 1 pour la seconde, alors la valeur pour le document considéré sera de 0,65. Le résultat obtenu pour chaque document permet l'ordonnancement de la réponse suivant les préférences. En fait, cette pondération n'est pas la seule dans OFFICER. En effet, une pondération supplémentaire est utilisée pour valoriser des parties de requêtes, et indiquer quelles parties sont plus importantes que d'autres.
On peut bien entendu appliquer la même technique sur des attributs numériques ou bien sur des chaînes de caractères.

Dans MULTOS, un effort supplémentaire a été fait : on peut indiquer des intervalles avec des préférences sur les attributs numériques.
Supposons qu'il existe un attribut *prix* de domaine réel. Si on cherche les documents pour lesquels l'attribut prix est environ 5000 (Fr), le langage de requête permet de préciser la notion d'"environ" de la façon suivante :

```
( .prix BETWEEN(4000, 4500) ACCEPTABLE,
        BETWEEN(4501, 5500) PREFERRED,
        BETWEEN(5501, 6000) ACCEPTABLE )
```

Du point de vue mesure, on attribue une préférence moyenne si l'attribut *prix* à une valeur comprise entre 4000 et 4500 de même qu'entre 5501 et 6000; par contre on privilégie la préférence de la valeur de l'attribut entre 4501 et 5500. Pour une valeur de l'attribut en dehors de ces intervalles la préférence est nulle. Si on n'a que cette condition dans la requête, les éléments qui ont la valeur de leur attribut comprise entre 4501 et 5500 seront donnés dans le résultat avant ceux dont cette valeur est dans l'un des deux autres intervalles.

Ce traitement revient à permettre d'utiliser de nouvelles relations d'ordre pour des attributs bases de données. Ces attributs ne sont pas liés aux concepts

véhiculés par le texte.

Revenons sur les valeurs d'importance au niveau des requêtes.
Dans ces deux systèmes, la valeur d'importance de chaque partie de la requête peut prendre trois valeurs qui sont : LOW, MEDIUM ou HIGH. Chacune de ces valeurs dénotant un ordre d'importance croissant dans la requête. Quand le système traite une requête, il va assigner à chaque partie de cette dernière une valeur correspondante à l'importance indiquée. Plus un élément vérifiera des parties importantes de la requête, plus il sera considéré comme répondant correctement à celle-ci. Les éléments qui vérifieront le mieux une telle requête seront les premiers fournis dans la réponse.
Nous voulons les lettres d'offres (cf partie IV.3.1) satisfaisant le critère suivant :

• *les lettres envoyées par la société "DURANT & Fils" qui parlent de compatibles PC.*

ou bien

• *les lettres reçues par la même société "DURANT & Fils" qui parlent de compatibles PC.*

On n'avait pas jusqu'à présent le moyen d'indiquer que l'on voulait d'abord les lettres reçues, puis celles envoyées, et il fallait poser deux requêtes. Le fait que l'on accorde plus d'importance à la partie qui traite des lettres reçues qu'à celle qui s'intéresse aux lettres envoyées, ainsi que le fait que ces lettres parlent de "compatibles PC" est important, peut s'exprimer dans MULTOS de la façon suivante :

```
FIND DOCUMENTS TYPE = Lettre_d'Offre
WHERE ( .entête.destinataire.nom_compagnie = "DURANT & Fils" ) HIGH
      OR ( .entête.expéditeur.nom_compagnie = "DURANT & Fils" ) MEDIUM
    ( TEXT CONTAINS "compatible PC") HIGH
```

V.4. VAGUE

Ce système est basé sur le SGBD relationnel INGRES ([Sto76]).
Commençons par regarder quelles sont les caractéristiques des métriques utilisées pour comparer des valeurs d'attributs ou des n-uplets.
Une métrique est définie sur un "domaine" qui est le nom de l'attribut sur lequel est appliquée la métrique. Dans le cas de comparaisons de n-uplets, la métrique

est basée sur l'attribut-clé de la relation. Elle est caractérisée par un type que nous allons décrire par la suite. Une métrique a un nom qui est un nom de métrique prédéfinie, ou bien un nom de relation. Elle est décrite de plus par un diamètre, un rayon et une sémantique. Le diamètre est la borne supérieure de toutes les distances entre les valeurs de l'attribut. Si on utilise une métrique qui est basée sur un attribut entier et qui calcule la valeur absolue de la différence entre deux valeurs de cet attribut, et si les entiers de cet attribut sont compris entre *a* et *b*, le diamètre de cette métrique sera |a - b|. Le rayon décrit la notion de voisinage standard. Si on a la distance entre deux valeurs (ou entre deux n-uplets), le système va déterminer si ces valeurs sont proches ou non, suivant que cette distance est inférieure ou non au rayon de la métrique associée.
Les types de métriques utilisables sur des valeurs d'attributs sont de deux sortes :

• les *métriques calculées* : la distance entre deux valeurs est uniquement calculée. Sur un attribut comme le *salaire* d'un employé, la distance peut être la valeur absolue de la différence entre deux nombres (ceci est un nom de métrique prédéfinie : ABS). Ces métriques peuvent porter sur des valeurs non numériques, par exemple la comparaison de deux chaînes de caractères dont le nom est STRING (nom de métrique prédéfinie).

• les *métriques tabulées* : une distance entre deux valeurs est tabulée si elle est trouvée uniquement à partir d'une recherche dans une relation créée spécialement à cet effet. Trouver la distance entre deux valeurs d'attributs revient à chercher dans cette table. Par exemple si on veut avoir la distance entre deux villes d'après leur nom, on réalise une table comme celle donnée ci-dessous :

VOISINAGE

Valeur	Valeur2	Distance
"Grenoble"	"St Martin d'Hères"	5
"Grenoble"	"Pont de Claix"	5
"Grenoble"	"Lyon"	80
...	...	...

On pourra retrouver les villes qui sont à une distance inférieure à un

certain seuil de Grenoble en passant en revue cette relation.

Décrivons maintenant les métriques qui permettent de calculer des distances entre des n-uplets. Il n'y a qu'un seul type de métriques qui réalise ces comparaisons. Ce sont les *métriques référentielles*, qui sont basées sur un domaine qui est la clé d'une relation, et les calculs sont vus comme étant ceux de la distance entre les clés de n-uplets. Ce type de métrique utilise d'autres métriques.
Nous prenons par exemple une relation "FILM(titre, metteur_en_scène, catégorie, qualité)". Le titre d'un film et le metteur en scène sont des chaînes de caractères. La catégorie d'un film est une chaîne de caractères, par exemple "Drame", "Science-Fiction",… La qualité d'un film est un réel entre 0 et 4 et correspond à une note (subjective) attribuée à ce film.

Définissons les métriques sur cette relation dans le tableau suivant :

Domaine	Métrique	Type	Diamètre	Rayon	Sémantique
titre	STRING	calcul	1	0, 2	Films avec titre identique
	FILM	référence	10	2	Films identiques
metteur_en_scène	STRING	calcul	1	0, 2	metteurs en scène de même nom
catégorie	CATEGORIE	table	3	1	catégories similaires
qualité	ABS	calcul	4	0, 5	films de même qualité

Comment calcule-t-on la distance entre deux films ?
Premièrement, les descriptions des deux films sont retrouvées d'après leur clé. Les distances entre les valeurs notées d_1 pour l'attribut metteur_en_scène, d_2 pour catégorie et d_3 pour qualité sont obtenues d'après les métriques définies dans le tableau ci-dessus. Ces distances individuelles sont pondérées grâce à des valeurs calculées.
Dans le cas présent, ces valeurs sont respectivement 2.83, 1.47 et 2.20. La distance entre les deux films sera donnée par la formule suivante :

$$\sqrt{(1.83*d_1)^2+(1.47*d_2)^2+(2.20*d_3)^2}$$

Revenons sur les pondérations associées aux distances. Leur but est triple : i) tenir compte du fait que différentes métriques sont utilisées, ii) refléter l'importance relative des attributs des n-uplets comparés, iii) garantir que le résultat est dans une fourchette prédéfinie pour rester dans le diamètre indiqué pour cette métrique.

Ce système propose de plus d'ordonner les résultats d'une requête suivant les valeurs de distances croissantes avec la requête, c'est-à-dire suivant des ressemblances de plus en plus faibles.

Quand l'utilisateur pose une requête floue grâce à ce système, il n'utilise qu'un seul prédicat supplémentaire à ceux habituellement rencontrés dans des requêtes SQL qui est *similar_to*. Dans l'implantation décrite, cet opérateur est exprimé par "?=".
On a vu qu'on pouvait utiliser plusieurs métriques sur un seul attribut (cf. dans le tableau des métriques le domaine *titre*). Au cours de l'évaluation de la requête, le système demande de choisir la métrique voulue après avoir donné les sémantiques de chacune des métriques possibles sous forme d'un menu. C'est alors que servent les sémantiques associées aux métriques stockées lors de leur définition (cf. tableau de définition des métriques).

V.5. L'approche par les ensembles flous.

Les travaux de P. Bosc et O. Pivert [Bos91b] ont visé à étendre la notion de relation de manière à réaliser des relations floues inspirées des ensembles flous, et à étendre le langage de requête SQL pour gérer ces relations.
De plus, ils ont montré que les processus des traitements de systèmes existants, tels que VAGUE ou MULTOS, peuvent être formalisés sous forme d'opérateurs sur des ensembles flous. Le but de ces travaux est de prouver que les ensembles flous permettent ces expressions, et peuvent même "améliorer" les processus en les resituant dans un cadre plus formel. Dans ce cas, les fonctions de correspondance et d'évaluation de la pertinence des éléments de la figure III2 (cf. partie II de ce chapitre) sont regroupées dans une même phase.

Dans un ensemble flou A, chaque élément *x* d'un référenciel R possède un degré d'appartenance, noté $\mu_A(x)$, compris dans l'intervalle [0, 1]. L'ensemble A est composé de couples $<x/\mu_A(x)>$, sans faire apparaître les couples de degré d'appartenance nul. Si on cherche un ensemble flou qui décrit les personnes qui ont environ 30 ans, il pourra être exprimé sous la forme:

{<26/0,2>,<27/0,4>,<28/0,6>,<29/0,8>,<30/1>,<31/0,8>,<32/0,6>,<33/0,4>,
<34/0,2>}

Sur de tels ensembles, des opérateurs sont applicables, tels que l'union et l'intersection. Par exemple, l'intersection et l'union de deux ensembles flous A et B (de référenciel commun R) peuvent être exprimés de la manière suivante:

- $\forall x \in R, \mu_{A \cup B}(x) = \min(\mu_A(x), \mu_B(x))$
- $\forall x \in R, \mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$

En fait les fonctions min/max ne sont pas les seules possibles. Dans ces travaux, la difficulté consiste à trouver les opérations sur les ensembles flous qui permettent un ordre sensé des éléments de la réponse à une requête. En fait, le rapport avec les ensembles flous est le suivant : on pose que plus un élément de la base correspond à la requête, plus il possède un degré d'appartenance à l'ensemble flou de la réponse qui est grand (c'est-à-dire proche de la valeur 1). En fait, le processus qui a été suivi est le même pour tous les systèmes étudiés:

• à chaque opérateur de requête, on recherche l'opération qui est effectuée durant le traitement d'une requête

• on associe ensuite à cette opération une fonction sur les ensembles flous, qui opère sur le ou les ensembles flous qui représentent chaque prédicat de la requête.

V.6. Discussion

La première approche étudiée en ce qui concerne le traitement de l'ordonnancement des réponses à une requête est le langage de requête flou pour bases de données. Ce langage fournit une approche relativement simple de traitement qui peut être fait pour permettre un ordre, mais il est limité à des attributs numériques, ce qui ne représente pas la totalité de ce que peut gérer un SGBD. L'avantage de cette approche est de ne pas avoir à écrire un grand nombre de requêtes pour obtenir ce que l'on cherche, car une seule suffit à fournir un ordre

utilisable. Si les valeurs numériques ne sont pas suffisantes, ou si on ne peut pas se rapporter à des valeurs numériques, alors cette approche atteint ses limites.

Avec MULTOS et OFFICER, on a la possibilité de ne pas se baser uniquement sur des valeurs numériques. On remarque que l'ordonnancement proposé est assez grossier (dans le cas des intervalles, toutes les valeurs qui sont dans le même intervalle ont même préférence), mais il faut aussi tenir compte du fait que, si on permet un grand raffinement dans l'expression de la requête, l'utilisateur ne saura plus différencier une valeur de préférence d'une autre, ce qui peut être préjudiciable. Cependant, étant donné que les intervalles fournis par l'utilisateur sont "fermés", c'est à dire que les bornes sont considérées comme incluses dans l'intervalle, il peut en découler des problèmes pour l'ordonnancement du résultat. Dans l'exemple cité durant l'étude de ces systèmes (partie V.3), on ne sait pas ce qui se passe si un document a un attribut *prix* qui possède la valeur 5550,5.

Dans les systèmes dont nous venons de parler, on ne propose pas de comparer des éléments complets de la base. Ceci veut dire que l'on ne peut pas exprimer le fait que, si on a un élément qui est intéressant, on recherche des éléments "proches" de cette référence. Cette approche nécessite une plus grande connaissance pour le système, car nous passons à un niveau où on essaie de rapprocher *sémantiquement* des éléments de la base, et que l'on ne donne pas, lors de l'expression de la requête, toute la connaissance nécessaire. Le système VAGUE permet l'expression de telles connaissances. Ces connaissances préexistantes à la requête sont exprimées par le concepteur de la base.

On peut rapprocher ce qui est fait dans VAGUE du modèle vectoriel ([Sal71]) en recherche d'informations. En effet, dans le modèle vectoriel, le contenu sémantique d'un document i est représenté par un vecteur de n coordonnées, noté D_i, chaque coordonnée indiquant l'importance du $n^{ième}$ terme dans le document. Une requête Q est également modélisée par un vecteur à n dimensions. La fonction de correspondance revient à évaluer la distance entre ces deux vecteurs.
Dans le cas de VAGUE, les deux vecteurs comparés sont des vecteurs qui correspondent à deux n-uplets, chaque vecteur ayant autant de coordonnées que le n-uplet auquel il correspond. Mais les valeurs des attributs n'ont aucune valeur d'indication d'importance. Dans ce cas, c'est uniquement par rapport à la différence entre ces deux vecteurs que l'on peut baser l'évaluation de la fonction

de correspondance.
Le problème de VAGUE vient toujours du problème du seuil d'une métrique, en-dessous duquel deux éléments sont "proches", et au-dessus duquel ils sont "éloignés". Ce seuil étant prédéfini, il se peut qu'il ne corresponde pas à ce que recherche l'utilisateur. Dans ce cas, VAGUE peut "jouer" sur ces seuils pour reformuler une requête.

Notre approche est assez similaire à celle de VAGUE, mais dans le contexte orienté objet, la structure des éléments de la base entre en jeu, et nous tentons d'utiliser la théorie de la logique modale pour aider à l'élaboration de telles correspondances. En cela, notre démarche se rapproche également du travail sur les ensembles flous de Pivert et Bosc.
Si cette utilisation des ensembles flous a un intérêt pour "unifier" l'approche "ordonnancement" des systèmes existants, la notion de structure des éléments de la base n'a pas été abordée, les travaux se situant dans un contexte relationnel.

VI Conclusion

Nous avons dans ce chapitre étudié les différentes approches de requêtes évoluées que nous connaissons dans les systèmes de gestion de bases de données. Les différents axes que nous avons mis en avant sont les facilités au niveau du schéma de la base de données, des éléments qui utilisent la structure des éléments de la base et les éléments qui permettent un ordonnancement du résultat d'une requête. Les systèmes qui possèdent au moins l'une de ces caractéristiques sont regroupés dans le tableau récapitulatif qui suit.

Ce tableau permet de se rendre compte que peu de systèmes ont englobé l'ensemble des facilités que nous avons déclarées. MULTOS est en fait le seul de ces systèmes à proposer des solutions dans les trois directions, cependant, de par son utilisation dédiée à la bureautique, il n'est pas suffisamment souple pour répondre à des besoins qui ne sont pas connus a priori.

SYSTEME	FACILITES SUR LE SCHEMA	FACILITES SUR LES STRUCTURES	ORDONNANCEMENT
O^2FDL	√		
OQL	√		
MULTOS	√	√	√
ORION		√	
O_2		√	
VAGUE		√	√
OFFICER			√
Regroupements flous	√		
Langage flou			√

On remarque que les SGBDOO proposent tous un tant soit peu de manipulation explicite des structures d'objets (ORION, O_2, O^2FDL, LIFOO, ...), mais sans dépasser un emploi strict de ces structures. C'est pourquoi, dans cette thèse, nous nous sommes tout d'abord préoccupés de comparaisons d'objets qui intègrent plus de sémantique que leurs simples structures. En fait, nous voulons suivre une démarche que l'on peut rapprocher de celle suivie pour le système VAGUE, en nous situant dans un cadre orienté objet.

Faciliter l'utilisation du schéma d'une base de données ne permet pas directement de comparer des objets, même si de telles fonctionnalités sont intéressantes. L'introduction de valuations dans les comparaisons est très utilisée dans le cas de systèmes de recherche d'informations, mais l'approche que nous avons suivie doit être définie clairement dans le cas strict, nous allons donc nous intéresser en grande partie à ce dernier cas dans la suite de ce travail.

CHAPITRE IV

La Correspondance d'Objets

CHAPITRE IV

La Correspondance d'Objets

I. Introduction

Nous avons vu dans les chapitres précédents que, bien que les SGBD Orientés Objet permettent une représentation plus "naturelle" d'éléments complexes que les Systèmes de Gestion de Bases de Données Relationnels, ils ne proposent pas de fonctionnalité réellement puissante pour comparer des objets complexes. Notre démarche va donc consister en l'élaboration d'un processus de conception de comparaison d'objets pour un programmeur d'application. Nous voulons simplifier son travail, en le guidant dans l'expression de ce processus.

Pour comparer des objets qui ont une certaine sémantique, nous avons choisi une approche Recherche d'Informations, car ce domaine s'est depuis plus de 20 ans intéressé à ce genre de problème de comparaison d'éléments (requêtes et documents) d'après leur sémantique respective. Nous nous basons sur une approche, proposée par C.J. Van Rijsbergen ([Rij90]), de modélisation de la recherche d'informations : *le modèle logique de recherche d'informations* . Ce modèle unificateur permet d'exprimer les modèles de Recherche d'Informations existants. Ce modèle, théorique, doit être instancié pour être utilisé de manière opérationnelle. Une manière d'instancier ce modèle logique est d'utiliser la logique modale. En effet, des travaux ([Nie90a,Nie90b], [Che92]) ont prouvé que la logique modale était capable de représenter *l'implication* entre une requête et un document au niveau modèle logique. Tout l'intérêt de l'utilisations de la logique modale vient du fait que nous allons pouvoir décrire pas à pas la démarche à suivre pour comparer les objets.

A partir de ce postulat (i.e. l'utilisation de la logique modale instanciant le modèle logique de RI), nous avons élaboré une démarche qui nous permet d'exprimer comment, et quand, deux objets correspondent. Pour cela, nous

décrivons un processus de conception qui va être composé de deux grandes parties, l'une au niveau logique, et l'autre au niveau des objets de la base. Ceci veut dire que le programmeur d'application va devoir exprimer sous forme logique la comparaison des éléments de la base, avant de déclarer chacune des phases du processus opérationnel qui correspond à une phase au niveau logique. Un point d'intérêt important pour nos travaux est aussi de déterminer les outils et/ou opérateurs nécessaires au niveau opérationnel pour faciliter la déclaration de processus de correspondances. Les avantages que nous attendons de cette approche sont les suivants :

- avantage méthodologique par l'emploi d'une architecture prédéfinie pour l'élaboration de comparaisons,
- avantage de portabilité des concepts par l'utilisation d'une partie logique d'expression de comparaison,
- avantage suivant une optique de validation de la correspondance, dans la mesure où l'on se base sur une formulation logique des comparaisons.

Dans ce chapitre, nous allons commencer par décrire les fonctionnalités d'un système de recherche d'informations, avant de regarder en détail le modèle logique de Recherche d'Informations proposé par C.J. Van Rijsbergen, puis la logique modale. Dans la partie suivante, nous verrons comment appliquer ces éléments à la correspondance d'objets complexes, en décrivant le processus de manière générale sur un exemple simple, puis en nous intéressant plus en détail à un exemple tiré d'un système de recherche d'informations médicales, RIME [Chi87, Berr88], développé au Laboratoire de Génie Informatique de Grenoble. La partie VI nous permettra de regarder d'un oeil critique l'une des parties du processus que nous décrivons. Nous conclurons par une comparaison entre notre approche et celle du système VAGUE proposé par A. Motro.

II Les Systèmes de Recherche d'Informations

Dans la partie précédente, nous avons commencé par annoncer que nous voulons rapprocher les bases de données de la recherche d'informations. En fait, ce choix a été fait principalement à cause de la connexion entre le domaine de la Recherche d'Informations et le but que nous nous sommes fixé :

- nous voulons intégrer plus de sémantique dans les comparaisons de données d'une base, de manière à fournir une réponse qui tient compte d'une “mesure”

indiquant la pertinence d'un document pour une requête,
• la Recherche d'Informations s'intéresse depuis longtemps (par exemple Salton en 1971 [Sal71]) à l'utilisation de connaissances plus ou moins évoluées dans le but de rechercher des documents qui répondent à une requête, en utilisant des représentations sémantiques de ces deux éléments comparés.

Nous allons ici entrer un peu plus en détail dans la description des systèmes de Recherche d'Informations. Nous allons tout d'abord décrire succinctement les différentes fonctionnalités d'un SRI, avant de faire un bref survol de différents modèles de Recherche d'Informations.

Les fonctionnalités d'un SRI

Dans les Systèmes de Recherche d'Informations, on peut décomposer le traitement de la requête selon le schéma de la figure IV1.

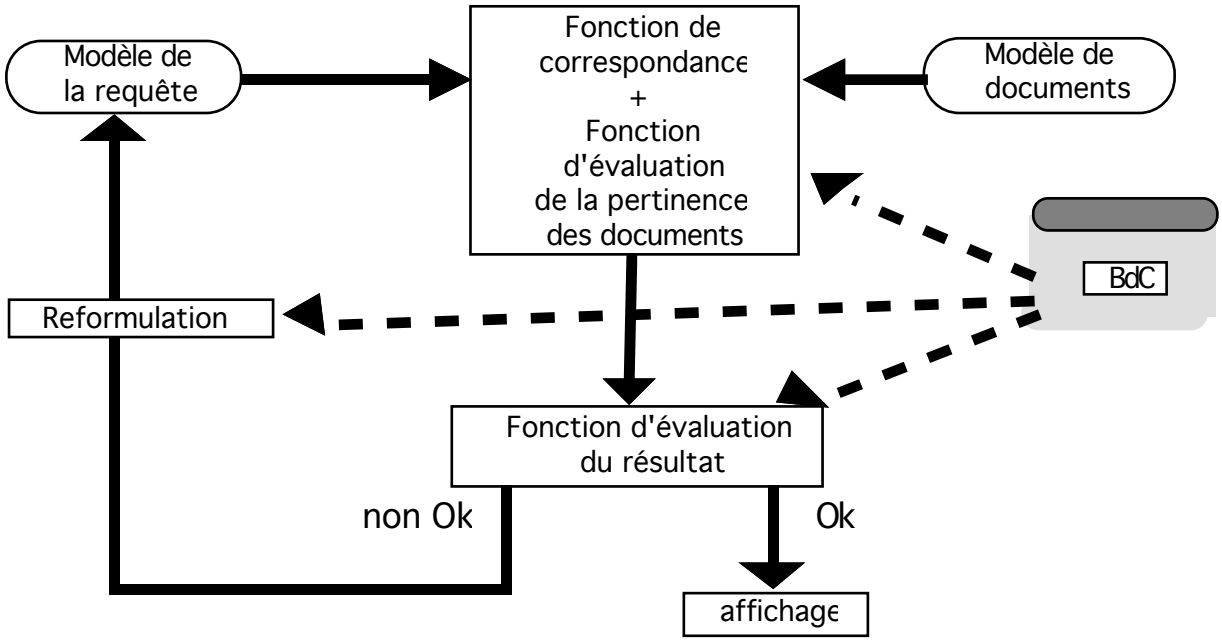


Figure IV1: Les fonctionnalités d'un SRI.

Le traitement d'une requête dans un SRI comprend les trois parties suivantes:

- **la réalisation d'une fonction de correspondance** entre le modèle des documents et le modèle de la requête ("pattern matching"). Cette fonction a pour

objectif de retrouver les documents dont le modèle correspond le mieux au modèle de la requête. Par modèle de documents, on entend la représentation de leur contenu sémantique et pour cela on utilise la connaissance des concepts qu'ils véhiculent. Le résultat de cette fonction est un ensemble de documents correspondants au moins à une partie de la requête ainsi qu'une évaluation de la correspondance de ces documents avec la requête. De ces valeurs de correspondance, on déduit **la pertinence des documents** retenus. Les documents sont alors **ordonnés** selon leur valeur de pertinence.

- **l'évaluation du résultat global** (ensemble de documents retrouvés) de la requête. Cette évaluation utilise d'autres éléments, tels que le nombre de documents, la typologie de l'utilisateur, la qualité des documents retrouvés, etc, et permet d'évaluer la qualité du résultat.

- **le processus de reformulation des requêtes.** Si le résultat d'une requête est jugé non satisfaisant, ce processus utilise une connaissance qui lui permet, étant donnée la requête initiale, de la modifier afin d'améliorer la qualité des réponses données à l'utilisateur.

Des modèles de Recherche d'Informations

Nous ne voulons pas ici analyser en détails les différents modèles de Recherche d'Informations qui existent, mais donner un aperçu de ces modèles. On pourra trouver plus de renseignements dans [Rij79] et dans [Sal85].

Le modèle vectoriel

Dans les systèmes de recherche d'informations vectoriels, un document est représenté par un vecteur à t dimensions, où t est le nombre de termes d'indexation du système. Le SRI SMART dans les années 70 était basé sur le modèle vectoriel [Sal71].
A chaque document i de la base, on associe un vecteur Di avec :

$$Di = (ai1, \ldots , ait)$$

où aij ($1 \leq j \leq t$) représente l'importance du terme j dans le document i.
Les aij sont compris entre deux valeurs a et b (déterminées à la conception du système), si le terme j n'apparaît pas dans le texte i alors aij est égal à a, par contre si le concept est très important dans le texte i alors aij est voisin de b.

Une requête est également formulée sous la forme d'un vecteur, et la pertinence d'un document pour une requête peut être calculée par un cosinus des deux vecteurs.

Le modèle probabiliste

Dans ce modèle, la tâche consiste à retrouver les documents dont la probabilité d'être pertinent est grande et la probabilité d'être non-pertinent est faible [Tur90] pour une requête.

On fait appel à deux probabilités pour un document D :

P(per | D) : probabilité de pertinence du document,
P(nonper | D) : probabilité de non-pertinence du document.

Grâce à ces deux probabilités, les documents peuvent être rangés dans un ordre décroissant en utilisant la fonction de recherche g(D) pour chaque texte :

$$g(D) = \frac{P(per \mid D)}{P(nonper \mid D)} = \frac{P(D \mid per) \cdot P(per)}{P(D \mid nonper) \cdot P(no}$$

avec :

- P(D | per) : probabilité pour le document D d'être dans l'ensemble des documents pertinents .
- P(D | nonper) : probabilité pour le document D d'être dans l'ensemble des documents non-pertinents.
- P(per) et P(nonper) des probabilités pré-évaluées de pertinence et de non-pertinence d'un document quelconque.

Si les probabilités P(per) et P(nonper) peuvent être calculées (sur un ensemble de documents et de requêtes donnés), en revanche on ne peut pas estimer les deux autres sans information sur les termes qui caractérisent chaque document. Une manière d'obtenir ces probabilités est de regarder les occurrences d'apparition des termes dans le texte du document.
Ce modèle permet l'utilisation de termes d'indexation non sémantiquement indépendants, mais dans ce cas les calculs de pertinence deviennent très

complexes. La relation entre les termes d'indexation et les documents peut être binaire ou non (c'est à dire que la relation entre un terme et un texte peut être plus fine que la notion d'appartenance ou de non-appartenance, la relation peut être valuée).

Le modèle booléen

On associe ici à chaque texte un ensemble de mots clés, les requêtes étant formées par des conjonctions et des disjonctions de termes d'indexation, par exemple : base ET donnée. On effectue une requête en associant au ET l'intersection des deux ensembles correspondants aux textes liés aux mots-clés, et au OU l'union de deux ensembles.
La requête "base ET donnée" donne :

$$E(\text{base}) \cap E(\text{donnée})$$

où E est une fonction qui renvoie l'ensemble des textes qui sont liés à un mot clé donné. Pour réaliser la fonction E on utilise la fonction F(D,q) de réprésentativité d'un terme q pour un document D. Cette fonction F peut avoir des valeurs binaires (avec habituellement un ensemble d'arrivée égal à {0,1}) ou non binaires (valeurs comprises le plus souvent dans l'intervalle [0,1]). Si on fait l'hypothèse qu'un terme q n'apparaît pas comme terme d'indexation dans le document D si F(D,q) est égal à 0 on a :

$$E(q) = \{ D \mid \lambda D \in \text{Base} \wedge F(D,q) \neq 0 \}$$

où Base est l'ensemble des textes de la base et 'a ∈ E' un prédicat VRAI si a est un élément de l'ensemble E, et FAUX sinon. La notation λD est la λ-notation qui indique que D est le paramètre de l'expression booléenne.

La relation F, si elle n'est pas binaire, mesure l'importance d'un terme dans un document. On peut ordonner la réponse à une requête en renvoyant tout d'abord ceux qui correspondent le mieux à la question posée.
Une limitation provient du fait que l'on ne tient pas compte de relations possibles entre termes. Cette faiblesse est levée par le modèle linguistique.

Le modèle linguistique

Les concepts qui servent de termes d'indexations ne sont plus seulement des mots, mais des groupes de mots. Dans un système comme IOTA [Def86] et RIME [Nie90a, Nie90b], ce sont les groupes nominaux qui sont utilisés.
On a donc la nécessité, au niveau de l'indexation des textes, d'analyser plus finement que dans les modèles précédents, pour extraire les concepts représentatifs, mais on ne désire pas parvenir à une compréhension détaillée du document. On se limite donc à une analyse de la syntaxe des textes, en ne résolvant pas obligatoirement les ambiguïtés syntaxiques (ceci étant possible grâce à la sémantique).
Si la requête est exprimée en langue naturelle, on peut l'analyser finement pour résoudre les ambiguïtés syntaxiques (en considérant qu'il est important de résoudre de telles ambiguïtés au niveau de la requête) comme le remarque A. Smeaton dans [Sme88]. En effet, si on laisse des ambiguïtés au niveau de l'interprétation d'une requête, il se peut que l'on fournisse des réponses complètement en dehors de ce qu'attend ce dernier.

Le modèle logique

Les différents modèles de Recherche d'Informations que nous venons de décrire ont tous des caractéristiques différentes. Cependant, C.J. Van Rijsbergen a proposé dans [Rij90] un "méta-modèle" , appelé le *modèle logique*, qui peut être instancié de manière à pouvoir exprimer les autres modèles de Recherche d'Informations. Il a été prouvé, dans [Nie90a], que ce modèle logique instancié par la logique modale floue permet de réaliser un SRI. L'avantage de ce modèle est de ne pas se focaliser sur un modèle de Recherche d'Informations donné, ce qui dans notre cas répond à notre souci de généralité. Voilà pourquoi nous avons choisi de nous baser sur ce modèle pour rapprocher les Systèmes de Gestion de Bases de Données Orientés Objet et les Systèmes de Recherche d'Informations.

III. Le Modèle Logique de Recherche d'Informations

Nous venons de voir ci-dessus différents modèles de Recherche d'Informations. Suivant la terminologie utilisée par J-P. Chevallet dans [Che92], nous pouvons caractériser les modèles booléens, vectoriels et probabilistes de modèles

opérationnels. Cela signifie que ces modèles sont utilisables tels quels pour réaliser un Système de Recherche d'Informations, car ils décrivent explicitement la sémantique des données comparées, et fournissent des opérateurs dédiés aux comparaisons entre les documents de la base et les requêtes. D'un autre coté, le modèle logique proposé par C.J. Van Rijsbergen peut être qualifié de *modèle théorique*. En effet, il expose *seulement* une analyse possible de la sémantique de la phrase suivante: "Un document D correspond à une requête Q".

III.1. L'idée de base du modèle logique

Le souci de C.J. Van Rijsbergen dans [Ris90a] a été, comme nous l'avons dit plus haut, de tenter d'unifier les modèles opérationnels de Recherche d'Informations existants.
Pour être général, ce modèle ne devait pas se limiter à une représentation de la sémantique des documents et des requêtes, et la formulation de Van Rijsbergen est la suivante:

"Etant données deux phrases x et y; une mesure de l'imprécision de x→y relativement à un ensemble d'informations donné est déterminé par l'extension minimale que l'on doit ajouter à l'ensemble d'informations, pour établir la vérité de x→y".

Dans le cadre de cette définition, le contenu sémantique du document est *x* et le contenu sémantique de la requête est représenté par *y*, et "x→y" représente en fait "le document x répond à la requête y".
Comme le note J-P. Chevallet, dans le cadre de la Recherche d'Informations, les "phrases" x et y sont des "ensembles d'informations", et "l'ensemble d'informations" est un "ensemble de connaissances".

J. Nie a dérivé de ce principe ce qu'il a appelé *le second principe d'incertitude*:

"Etant donnés deux ensembles d'informations x et y, la mesure d'incertitude de x→y relative à un certain ensemble K de connaissances donné est déterminé par l'extension nécessaire de x en x' et la certitude de la vérification de x'→y".

C'est ce second principe que nous allons suivre pour déterminer si un objet O_1 (correspondant à y) est capable de répondre à une requête si l'on cherche l'objet O_2 (correspondant à x).

Nous allons maintenant montrer sur un exemple très simple l'utilisation de ce modèle logique de Recherche d'Informations.

III.2. Exemple d'utilisation du modèle logique

Nous allons ici nous baser sur des exemples "géographiques" sur la Communauté Economique Européenne, pour situer sur un cas concret l'utilisation de ce modèle logique, et en particulier le second principe d'incertitude.

Considérons, dans une bibliothèque, un livre qui traite de “la Bretagne”, et une personne qui recherche des ouvrages sur "l'Europe".
Suivant le modèle logique de recherche d'informations, nous saurons si le document correspond à ce que recherche la personne si:

$$\text{la\_Bretagne} \vdash \text{l'Europe} \qquad (I_1)$$

Sans connaissance supplémentaire, cette implication n'est pas vérifiée. Par contre si l'on considère l'existence d'une connaissance **extérieure** sur une relation "de généralisation géographique" qui stipule que la Bretagne est une partie de l'Europe, nous arrivons, en *étendant la sémantique du livre*, à l'implication suivante:

$$\text{la\_Bretagne } \textit{et l'Europe} \vdash \text{l'Europe} \qquad (I_2)$$

Dans ce cas, l'implication est vraie, ce qui nous permet de conclure que le document répond, au moins partiellement, à ce que cherche l'utilisateur.
La valeur de cette implication peut être soit *stricte* (c'est dire que l'implication peut être Vraie ou Fausse), soit *non-stricte* (l'implication peut par exemple être pondérée par une valeur entre 0 et 1). Pour notre exemple, une valuation non-stricte de l'implication serait par exemple de 0,9 si l'on considère que l'Europe est un concept qui généralise très bien le terme Bretagne.

Dans le cadre de ce chapitre, nous allons nous intéresser à des calculs d'implications strictes. Ceci veut dire que nous sommes seulement capables d'indiquer qu'un objet correspond à un autre ou non, mais nous ne saurons pas dans quelle mesure.

IV. La logique modale et le modèle logique

Nous allons commencer dans cette partie par décrire succinctement la logique modale des propositions, en nous intéressant également plus particulièrement à la logique modale floue. Nous montrerons ensuite son application au modèle logique de Recherche d'Informations.
Notre but n'est pas ici de rester uniquement théorique, mais de déterminer comment le support théorique que constitue la logique modale peut aider à préciser le processus de correspondance d'objets.

IV.1. La logique modale des propositions

Dans ce paragraphe, nous nous intéressons à la logique modale des propositions de manière à situer les concepts importants de la "modalité". La logique modale des prédicats est très proche, et nous choisissons donc de nous limiter ici aux propositions.

Dans la logique *non-modale* des propositions, une formule F bien formée est décrite en BNF par:
F ::= F ∧ F | ¬F | P où P est une proposition atomique.
De plus, une valeur de vérité est associée à chaque formule F, grâce à une interprétation I, fonction qui donne, pour une proposition atomique de l'ensemble des propositions atomiques P, sa valeur de vérité Vrai ou Faux. Cette interprétation peut également être exprimée au travers de la méta-notation[1] "|=" rencontrée dans [Kle71], en utilisant une formule F_I qui exprime positivement les propositions de P qui sont Vraies, et négativement les propositions de P qui sont Fausses, de la manière suivante: "F_I |= F".

L'apport de la logique modale est que l'on peut faire **varier** la valeur de vérité associée à une proposition atomique, et donc la valeur de vérité d'une formule. Pour refléter ces variations, le concept de *monde* a été introduit.

Pour rester en contact avec la réalité, on va examiner cette notion sur un exemple.

[1] "F|=G" exprime le fait que la formule G est Vraie pour toutes les interprétations qui rendent F Vraie.

Sur terre, on peut parfaitement dire "le soleil est jaune". Cependant, cette phrase peut très bien se révéler Fausse sur une planète d'un autre système solaire où le soleil est vert (pourquoi pas?), c'est-à-dire dans *un autre monde*.
Dans ce cas, un certain nombre d'éléments supplémentaires sont utilisés pour décrire un tel système:

• L'évaluation d'une formule se réalise par rapport à un monde *m* qui correspond à une interprétation donnée des propositions. L'ensemble des mondes est appelé W. La fonction d'interprétation modale V renvoie l'ensemble des mondes dans lesquels les propositions de P sont Vraies. Elle est étendue à toute formule bien formée F. On notera par la suite “m |= F” l'interprétation d'une formule F dans le monde *m*.
• Les mondes peuvent être vus comme les noeuds d'un graphe, et les arcs dirigés de ce graphe sont représentés par la fonction d: W×W - {0,1}, fonction qui vaut 1 si les deux mondes sont reliés, et 0 sinon. On dira qu'un monde *m'* est un monde possible de *m*, si et seulement s'il existe un chemin dans le graphe des mondes qui mène de *m* à *m'*.
• La définition d'une formule F est modifiée, de manière à tenir compte de cette notion de monde :
F ::= F ∧ F | ¬F | ◊F | P où P est une proposition atomique.
L'opérateur de possibilité ◊ est intimement lié aux mondes dans lesquels on évalue une formule. En effet, ◊F est Vraie dans le monde *m* si la formule F est associée à Vraie, ou si F est associée à Vraie dans l'un des mondes possibles à partir de *m*.

La logique modale des propositions est finalement très stricte, au sens où aucune évaluation autre que Vrai ou Faux ne peut être faite. Or, comme nous avons toujours à l'esprit que nous voulons appliquer ce modèle à un cadre Recherche d'Informations, ce fait d'être strict est une limitation difficilement supportable. Une solution pour pallier ce genre d'inconvénient est d'introduire des éléments "flous" (i.e. non stricts) dans la logique modale, de manière à utiliser de la *logique modale floue*. Nous allons seulement indiquer les éléments supplémentaires ou ceux qui diffèrent de la logique modale *stricte*:

• La fonction δ n'a plus son domaine d'arrivée dans l'ensemble {0,1}, mais dans l'intervalle [0,1]. Cette fonction va donc définir dans quelle mesure il existe un passage d'un monde *m* à un monde *m'*. Ceci revient en fait à valuer les arcs du

graphe des mondes.
• La fonction V est modifiée par rapport à la logique modale stricte, dans la mesure où elle indique, pour une formule et un monde donné, la valeur "de vérité" de cette formule par un réel compris dans l'intervalle [0,1].
• Une fonction C, qui n'existait pas dans la logique modale stricte, indique, pour un monde donné et pour une proposition donnée, la valeur "de vérité" de cette proposition comprise dans l'intervalle [0,1].
• Une fonction $\Delta_{\Box}$, de [0,1]×[0,1] - [0,1], qui permet de combiner les valeurs de certitude de passage d'un monde à l'autre avec les valeurs de certitude des propositions et des formules.

Nous n'en dirons pas plus sur la logique modale floue des propositions, car dans ce chapitre nous allons nous focaliser sur l'emploi de la logique modale stricte, mais nous y reviendrons dans la chapitre suivant, pour situer comment des valuations peuvent être intégrées.

IV.2. Application au modèle logique de Recherche d'Informations

Les explications qui ont été données dans la partie précédente vont tout d'abord permettre le lien avec le modèle logique de Recherche d'Informations. Ensuite, nous nous basons sur l'exemple de la partie III.2 pour situer l'utilisation de la logique modale.

IV.2.1. L'idée générale

Rappelons que nous nous situons dans le cadre du second principe d'incertitude (partie III.1). L'idée choisie pour représenter en logique modale le modèle logique est la suivante (cf. [Che92]):
• le contenu sémantique du document et celui de la requête que nous comparons sont des formules logiques que nous appelerons respectivement F_D et F_Q. Remarquons ici que nous ne choisissons pas entre la logique des propositions ou celle des prédicats, car nous verrons par la suite que ces deux logiques peuvent être utilisées (cf. partie V.2), suivant la sémantique des objets que nous cherchons à comparer.
• le document sera considéré comme le *monde* initial dans lequel va être évaluée la requête. Ceci veut dire que, **"idéalement"**, nous cherchons à vérifier si

"$F_D |= F_Q$" (par la suite notée $\mathbf{F_{imp}}$) est valide ou non. Dans le cas de la logique modale, cette validité ne dépendra que du fait que F_Q est Vraie ou Fausse. Dans le cas de la logique modale floue, une solution simple peut être de trouver si $V(F_Q)$ est différente de 0. En fait, nous devons garder à l'esprit que les correspondances que nous cherchons à évaluer sont toujours basées sur une connaissance K_i (une connaissance K_i par monde *mi*). Ceci veut dire qu'en fait, nous allons plutôt chercher à évaluer "$F_D \wedge F_{ext}^i |= F_Q$" que F_{imp}. Les extensions de connaissance destinées à tenter de rendre valide l'implication du document sur la requête sont notées F_{ext}^i. L'esprit de l'élaboration de ces F_{ext}^i est le suivant:

-> la connaissance K_i permet d'obtenir les formules suivantes:

$$F_D =>_{Ki} E_1$$
$$\ldots$$
$$F_D =>_{Ki} E_n$$

la notation "$F_D =>_{Ki} E_1$" exprime en fait que, par l'utilisation de la connaissance K_i, la formule F_D implique la formule E_1. Nous appelerons par la suite les E_j, avec $1 \leq j \leq n$, des *extensions élémentaires* de F_D. La connaissance K_i peut utiliser F_D et F_Q, comme nous le verrons par la suite. Les formules E_j peuvent être des **extensions positives** (on ajoute des éléments non précédés d'une négation à la formule F_D), ou bien des **extensions négatives** (on ajoute des éléments précédés d'une négation à la formule F_D).

-> à partir de ces implications élémentaires, on obtient le résultat suivant:

$$F_D =>_{Ki} E_1 \wedge \ldots \wedge E_n$$

-> comme notre but est d'obtenir un modèle dans lequel on va évaluer F_Q, on a choisi de regrouper F_D et ses extensions dans une seule formule, grâce aux résultats suivants :

$$F_D, F_D =>_{Ki} E_1 \wedge \ldots \wedge E_n |= E_1 \wedge \ldots \wedge E_n$$

et $$F_D, F_D =>_{Ki} E_1 \wedge \ldots \wedge E_n |= F_D$$

ce qui nous permet de trouver :

$$F_D, F_D =>_{Ki} E_1 \wedge \ldots \wedge E_n |= F_D \wedge E_1 \wedge \ldots \wedge E_n$$

Cette formule nous permet de regrouper F_D et ses extensions, et nous allons nous servir du modèle de la formule "$F_D \wedge E_1 \wedge \ldots \wedge E_n$" pour évaluer F_Q, ce qui nous donne :

$$F_D \wedge E_1 \wedge \ldots \wedge E_n |= F_Q \qquad (Fo1)$$

Par rapport à ce que nous avons écrit plus haut, la formule F_{ext}^i est en fait "$E_1 \wedge \ldots \wedge E_n$".

La formule *Fo1* indique que l'on considère F_Q dans un monde où F_D et son extension sont considérées comme Vraies, ce qui est finalement assez naturel, si l'on estime que la base de données possède des informations valides.
Chacune de ces *i* extensions va être utilisée de la manière suivante:
- on va tenter de vérifier "$F_D \wedge F_{ext}^i \models F_Q$", pour un i donné, ceci veut dire que l'on est dans le monde m_i,
- si cette formule est valide, alors le document répond à la requête
- si cette formule n'est pas valide, on génère une nouvelle formule F_{ext}^{i+1}, qui étend encore le document, et on évalue de nouveau l'implication, …

Dans la suite de ce document, nous utiliserons indifféremment l'expression "d'implication de $F_D \wedge F_{ext}^i$ vers F_Q" pour la formule Fo1.
Revenons sur le terme "idéalement", en situant ce que peuvent être les *extensions négatives* de F_D. En fait, il se peut, et c'est même fort probable, que l'on ne puisse pas conclure sur la validité de la formule F_{imp}. En effet, certains éléments de F_Q (peu importe le fait que l'on soit en logique des propositions ou des prédicats) qui n'apparaissent pas dans F_D n'ont aucune valeur de vérité si l'on s'en tient strictement à la formule F_{imp}. Nous devons donc, au niveau logique, *indiquer explicitement* une valeur de vérité de ces éléments, au travers d'une extension au document, appelée F_{ext}^0, qui permet de statuer sur les valeurs de vérité de ces éléments. Donc, la formule initiale F_{imp} devient en fait F_{imp+}, qui correspond à "$F_D \wedge F_{ext}^0 \models F_Q$". Nous allons revenir plus en détail par la suite sur cette connaissance initiale que nous utilisons. Il est à noter que l'ajout d'une telle formule se rapproche de ce qui est fait implicitement dans les SGBD déductifs par l'hypothèse du monde fermé. Nous allons voir par la suite pourquoi nous devons considérer explicitement cette formule supplémentaire.
En fait, comme nous le verrons par la suite, une telle phase d'indication explicite des éléments de la requête qui *ne correspondent* à aucun élément du document existera pour chacune des formules F_{ext}^i.

IV.2.2. Exemple

Nous reprenons ici l'exemple que nous avons défini dans la partie II.2.

Dans ce premier cas, on cherche à savoir si un livre qui parle de "la Bretagne" répond à une requête d'un utilisateur qui cherche un livre sur "l'Europe". Comme

nous nous situons maintenant au niveau logique, nous allons exprimer ces phrases grâce à la logique des propositions:
• le document va avoir son contenu sémantique représenté par la formule F_D qui est une simple proposition: “Bretagne”,
• la requête, suivant une approche analogue, est représentée par la proposition “Europe”.

Suivant cet exemple, si nous voulons obtenir un résultat valué, nous devons utiliser la logique modale floue des propositions. Nous allons dans notre exemple nous limiter à utiliser la fonction δ floue, c'est-à-dire que nous allons pondérer les arcs qui relient les mondes, mais les interprétations des propositions seront strictes (i.e. la fonction C donne 1 ou 0).
D'après la partie III.2.1, la formule F_{imp} serait donc la suivante:

Bretagne |= Europe

Comme cela ne nous permet pas de conclure, nous allons considérer que toutes les propositions de la requête qui n'apparaissent pas dans le document sont Fausses. Cette considération est en fait le résultat de l'utilisation d'une connaissance qui est extérieure à l'expression même de F_{imp}.
Ceci nous donne une formule F_{ext}^0 qui est “¬Europe” (dans ce cas F_{ext}^0 n'est composée que d'une proposition qui est précédée d'une négation). Comme nous l'avons dit plus haut, nous choisissons délibérément d'indiquer cette connaissance "fermée" qui indique que tout ce qui n'est pas représenté explicitement dans la formule du document n'est pas dans le document, et doit donc être considéré comme faux pour la requête. En cela, nous suivons le même esprit que dans un Système de Recherche d'Informations.

Le monde initial *m0* correspond donc à l'interprétation rendant Vraie la formule “Bretagne ∧ ¬Europe”. Dans ce monde, la proposition “Europe” est Fausse. Ceci veut dire que, sans connaissance supplémentaire, le document ne répond pas à la requête.

Si nous considérons que nous pouvons étendre la formule F_D avec la proposition “Europe” pour générer le monde *m1*, et ceci en indiquant qu'il existe un arc entre *m0* et *m1* valué à 0, 9, le monde *m1* correspond à l'interprétation qui rend Vraie la formule “Bretagne Ê Europe”. Dans ce monde *m1*, la formule F_Q est Vraie. La

valeur 0, 9 est en fait la même que celle que nous avons définie plus haut, et qui pondère en fait l'extension de la requête.

Une fois que nous avons trouvé un monde possible à partir de *m0* qui rend F_Q Vraie, alors le document répond à la requête. Dans notre cas, le document répond à la requête avec une *pertinence* de 0,9 (plus la valeur est proche de 0, moins le document répond bien à la requête), qui est V(m0,Europe). Cette valeur (le maximum entre la valeur de vérité d'une formule dans le monde courant et de la valeur de possibilité de cette formule) est en fait due à l'expression de la fonction δ (valuation du passage d'un monde à un autre) et la fonction Δ (multiplication sur les réels):

$$\begin{aligned} MAX(C(m0, Europe), V(m0, \Diamond Europe)) &= MAX(0, \Delta(\delta(m0, m1), C(m1, Europe))) \\ &= MAX(0, \Delta(0{,}9\ ,1)) \\ &= MAX(0, 0{,}9) = 0{,}9 \end{aligned}$$

Donc, en utilisant la logique modale floue, on réussit à retrouver le résultat "expérimental" de la partie III.2, en suivant le schéma IV2.

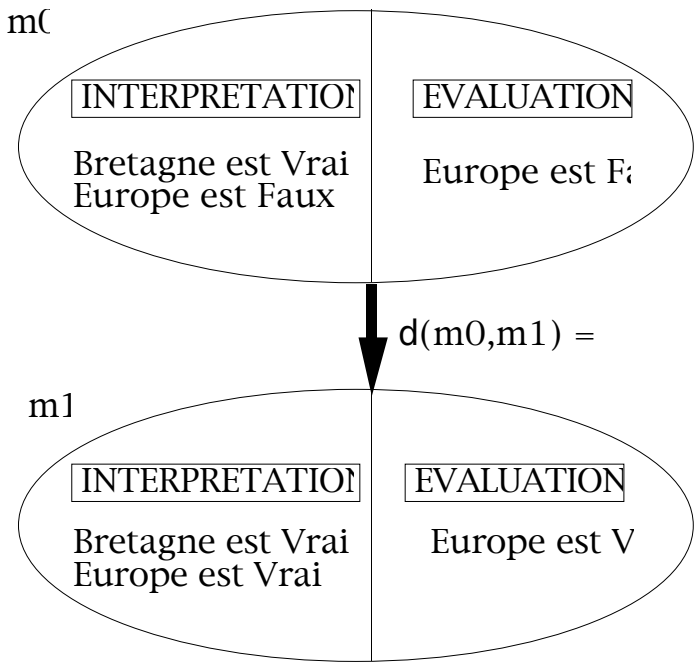


Figure IV2: Le traitement logique du premier exemple.

Dans le cas où on se limite à une correspondance stricte, il nous suffit de vérifier si la formule F_Q est possiblement Vraie dans le monde *m0*. Nous reprenons donc

la figure précédente, sans associer de pondération inter-monde on utilise donc la logique modale non floue. On constate que la formule F_Q est Vraie dans le monde *m1*, et ce monde est accessible à partir de *m0*, donc la formule F_Q est "possiblement Vraie" à dans le monde *m0*, ainsi le livre correspond à ce que recherche la personne. Il est certain que l'emploi de correspondances non-pondérées est très grossière, mais nous devons en passer par là si nous voulons étendre ces correspondances pour intégrer des valeurs moins strictes.

V. Application aux objets complexes

Nous entrons ici dans la partie du travail que nous avons réalisé. Nous voulons permettre à un concepteur d'exprimer de quelle manière un objet *B* peut être considéré comme une bonne réponse si l'on recherche l'objet *A*. Pour cela, nous avons choisi de décomposer cette expression suivant deux axes:

- nous voulons tout d'abord permettre d'exprimer de manière théorique, par l'utilisation de la logique modale appliquée au modèle logique de recherche d'informations, la comparaison des formules logiques représentant la sémantique des deux objets comparés,
- nous permettons à un concepteur d'exprimer, au travers d'un outil, les différentes parties de la comparaison logique en vue du traitement opérationnel qui manipulera les objets. Cet outil doit bien sûr proposer les éléments nécessaires pour réaliser cette déclaration.

L'intérêt de passer par une formulation théorique de la correspondance des objets permet de garder une certaine distance face au niveau opérationnel. De plus, nous allons montrer que le passage par la logique modale nous permet de définir différentes *phases* nécessaires à la comparaison d'objets via leur sémantique.

Cette partie va être décomposée en trois points. Dans le premier, nous allons examiner la façon de définir la comparaison de deux objets de manière théorique. Le second va traiter des caractéristiques que propose l'outil de comparaison d'objets, en situant ces caractéristiques du point de vue logique.
Ces deux premiers points vont être supportés par l'exemple très simple que nous avons introduit précédemment.
Cependant, pour prouver que notre approche ne se limite pas à des “cas d'école”, nous l'expliquerons dans la troisième partie sur un exemple plus "complexe",

basé sur les concepts du prototype de Recherche d'Informations RIME [Chi87,Berr88].

V.1. La correspondance théorique de deux objets

Comme nous l'avons vu dans les paragraphes précédents, nous allons décomposer l'élaboration de la correspondance entre deux objets.

Suivant cette description, *l'objet O_Q correspondant au contenu sémantique de la requête, ainsi que l'objet O_D correspondant au contenu sémantique d'un élément de la base doivent être représentés par des formules logiques.*
Dans la suite, nous utiliserons respectivement les termes *document* et *requête* pour *contenu sémantique du document* et *contenu sémantique de la requête.*

V.1.1. Description de l'exemple simple

Dans cet exemple, le document est représenté par une proposition, et la requête également.

Récapitulons cet exemple. La représentation F_D du contenu sémantique du document est

" Bretagne ".

La requête, quant à elle, va être représentée sous forme logique par F_Q:

" Europe ".

Nous voulons donc comparer deux formules logiques, chacune d'elles étant initialement composée d'une proposition.

V.1.2. Définition du premier monde

Tel que nous l'avons décrit initialement, nous devons commencer par décrire explicitement les valeurs de vérité des propositions de F_Q qui n'ont pas de valeur de vérité en se basant seulement sur la formule F_D.

On veut donc comparer deux termes que nous appellerons pr_d et pr_q, car la

formule F_D ne contient que pr_d, et la formule F_Q que la proposition pr_q.

Nous voulons générer F_{ext}^0 qui étend la formule du document, en indiquant explicitement qu'"Europe" est une proposition Fausse.

Le but de cette indication est double:

• on veut exprimer les différences "élémentaires" entre les deux formules, pour pouvoir situer avec facilité les points sur lesquels se focaliser pour "rapprocher" ces formules. Cette stratégie se rapproche de la technique de décomposition d'un problème. En effet, on tente tout d'abord de décomposer la correspondance "globale" de deux objets en correspondances élémentaires, qui vont servir à réaliser cette correspondance globale.
• au niveau opérationnel, nous n'avons pas de mécanisme de déduction dans un SGBD Orienté Objet, il faut donc remplacer cette déduction en explicitant les valeurs de vérité de tous les éléments de la requête.

Les différences élémentaires correspondent en fait à des implications élémentaires entre les parties de la requête, implication au sens *implication logique* suivant une certaine connaissance, comme nous l'avons expliqué précédemment.

Dans notre cas, on peut se trouver dans deux situations:

• soit pr_d et pr_q sont la même proposition pr, et dans ce cas il y a une correspondance immédiate entre F_Q et F_D, car de manière évidente le modèle qui rend Vrai pr_d permet d'interpréter pr_q comme Vrai.
• soit les deux propositions ne sont pas les mêmes, et nous devons donner explicitement à pr_q la valeur de vérité Faux. Ceci découle du fait que nous considérons que deux propositions différentes n'ont pas la même sémantique, et que ce postulat, sans connaissance supplémentaire sur des liaisons entre concepts, nous indique que la proposition de la requête est Fausse.

Ce que nous voulons ici, c'est exprimer comment nous pouvons obtenir le résultat ci-dessus. Nous gardons à l'esprit que le but final de la formulation logique est de permettre une expression opérationnelle de correspondances d'objets, c'est pourquoi nous allons exprimer grâce à l'utilisation de fonctions sur des

propositions l'élaboration de l'extension initiale du document. Au niveau opérationnel, les traductions de ces fonctions constitueront le point le plus important à réaliser.

Dans l'exemple qui nous intéresse, nous constatons que l'extension initiale du document est uniquement "négative", c'est-à-dire précédée d'une négation car on indique ce qui n'est pas dans le document. Ceci veut dire que la formule F_{ext}^0 qui étend F_D ne contiendra "au plus" qu'une proposition "négative". La démarche que nous suivons est formée de deux niveaux qui sont :
1. une fonction appelée $G_{K:0}$ pour déterminer si la proposition de la requête est la même que celle du document,
2. une expression qui permet de formuler, à partir du résultat de $G_{K:0}$, comment déduire la valeur de vérité de la proposition de la requête.

Nous commençons par le point 2), en considérant que la fonction $G_{K:0}$ renvoie la proposition pr_q, si celle-ci ne correspond pas à la proposition pr_d du document:

$$F_{ext}^0 = \neg(G_{K:0}(pr_q, pr_d))$$

La notation “F = G” sur des formules logiques indique seulement que la formule G *a pour nom* F, ce qui veut dire que dans les formules utilisant F, il faut remplacer F par son expression qui est G.

La fonction $G_{K:0}$ est la suivante:

$G_{K:0}$: proposition × proposition - proposition
$G_{K:0}(pr_q, pr_d)$ = si ¬(pr_q est la même proposition que pr_d)
alors pr_q
sinon *rien*

Dans l'expression de la fonction $G_{K:0}$:
• K:0 représente la connaissance initiale.
• le point important réside dans l'emploi du prédicat **est la même proposition que**, qui opère entre deux propositions.

Pour se rapprocher de ce que nous avons décrit dans la partie précédente, la

connaissance initiale permet seulement dans notre cas simple d'indiquer si la proposition de la requête n'est pas impliquée par celle qui décrit le document.

Revenons à notre exemple. La formule logique F_D du document est “Bretagne” et F_Q est “Europe”. Il s'en suit que pr_q est *Europe* et que pr_d est *Bretagne* .

En nous basant sur ce que nous avons écrit plus haut, la fonction $G_{K:0}$(Europe, Bretagne) renvoie le résultat Europe. La formule F_{ext}^0 est donc la suivante : ¬(Europe) .
En se basant sur le modèle de la formule “Bretagne ∧ ¬(Europe)”, la formule F_Q n'est pas valide, ce qui montre qu'il n'y a pas de correspondance "immédiate" entre la requête et le document.

V.1.3. Définition du second monde

Dans notre démarche, on va considérer que nous aurons une génération *linéaire* de mondes. Ceci veut dire que chaque monde est la cible et le point de départ d'un et d'un seul arc inter-monde (sauf le monde initial qui n'est la cible d'aucun lien, et le monde final qui n'est le point de départ d'aucun lien inter-monde). Cette démarche semble logique dans notre cas, car la connaissance que nous allons utiliser est "unidirectionnelle", et se base sur des relations de généricité du genre de celles utilisées dans les thésaurus.

Le traitement va donc toujours se baser sur le monde *mi* pour générer le monde *mi+1*, mais aussi sur la formule F_Q et *sur de la connaissance supplémentaire*.
Le travail doit donc essentiellement répondre à la question : “comment fait-on pour générer un nouveau monde?”.

Nous devons ici différencier ce que nous appelons la connaissance "externe" de la connaissance "interne". La connaissance interne est celle qui nous permet de définir les fonctions et expressions qui permettent de réaliser les formules F_{ext}^i. La connaissance externe est une connaissance que nous supposons exister indépendamment de la génération des formules F_{ext}^i. De plus, la connaissance interne doit être en mesure d'utiliser la connaissance externe.

Dans la génération du monde initial de notre exemple, nous n'avons utilisé que de la connaissance interne, en supposant en particulier que les prédicats de la

fonction $G_{K:0}$ ne nécessitaient pas d'autre connaissance.

Si la formule F_Q était fausse dans le monde *m0*, l'extension de F_D pour le monde *m1 comprend* deux choses:
-> l'extension positive de F_D pour les génériques de pr_d,
-> l'extension négative de F_D qui suit le même esprit que dans le monde *m0*, mais qui n'est plus basée sur pr_d et pr_q, mais sur les génériques de pr_d et sur pr_q.

Comme ces extensions sont toutes impliquées, via une certaine connaissance de F_D, elles seront toutes connectées par une conjonction à F_D pour créer le monde m1.
Nous avons choisi de regrouper ces deux formules sous le nom générique de $F_{ext\_sup}{}^1$, car ces extensions sont des éléments supplémentaires de F_D.

La formule $F_{ext\_sup}{}^1$ qui reflète ces deux extensions est la suivante:

$$F^1_{ext\_sup} = \bigwedge_{p \in F_{K:1}(pr_d)} p \quad \bigwedge_{p \in G_{K:1}(pr_d, pr^n_{ext0})} \neg\, p$$

Dans l'expression de cette formule, pr^n_{ext0} représente la proposition de $F_{ext}{}^0$ qui est précédée d'une négation (c'est en fait pr_q si le document ne correspond pas à la requête).
La fonction $F_{K:1}$ a pour but de renvoyer les propositions qui correspondent aux termes génériques de pr_d, et la fonction $G_{K:1}$ permet de dire si la proposition pr^n_{ext0} est la même proposition que l'un des génériques de pr_d. Ces fonctions sont décrites comme suit :

$F_{K:1}$: proposition $\rightarrow$ { proposition }
$F_{K:1}(pr_d)$ = génériques(pr_d)

$G_{K:1}$: proposition $\times$ proposition $\rightarrow$ proposition
$G_{K:1}(pr_d, pr^n_{ext0})$ = si $\neg$(pr^n_{ext0} est la même proposition que q
et q $\in F_{K:1}(pr_d)$)
alors pr^n_{ext0}
sinon *rien*

Dans l'expression de ces fonctions, "génériques(pr_d)" est un appel à la connaissance extérieure qui permet de donner les propositions qui correspondent aux génériques du terme qui se rapporte à la proposition pr_d. Nous constatons donc que cet appel réalise ce qui était représenté par une *implication* pour l'extension du document, ce qui nous permet de l'utiliser pour étendre F_D.

Nous revenons maintenant sur notre exemple. Exprimons au travers de l'arbre de la figure IV3 les relations de généricité que nous allons utiliser. Dans cet arbre, les liens sont dirigés du terme spécifique vers le terme générique (il est certain que la structure d'un thésaurus est plutôt un graphe, mais nous nous limitons ici à représenter la partie utilisée par la connaissance dont nous avons besoin).

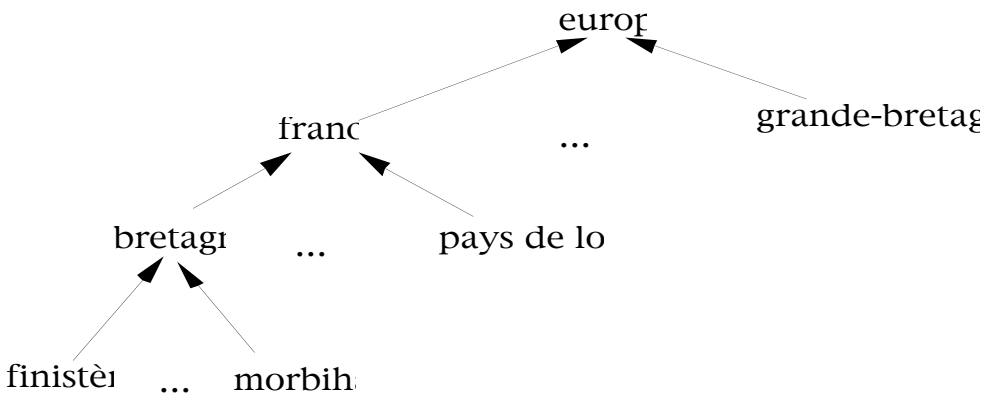


Figure IV3: Les liaisons sémantiques utilisées par l'exemple.

Suivant ce schéma, la fonction $F_{K:1}$(Bretagne) a pour résultat la proposition “France”. La fonction $G_{K:1}$(Bretagne, Europe) quant à elle fournit le résultat “Europe”, car la proposition “Europe” n'est pas la même que la proposition “France”.

Le modèle dans lequel est évalué F_Q est donc celui de la formule “Bretagne ∧ France ∧¬Europe”, et cette interprétation rend Faux. Le document ne correspond donc toujours pas à la requête.

V.1.4. Définition des autres mondes

Si nous voulons étendre encore la formule F_D pour tenter de trouver que F_Q est Vraie, nous avons plusieurs solutions possibles:

1. soit augmenter incrémentalement l'ensemble des génériques du document,
2. soit se contenter de remplacer les génériques g_i du monde précédent par les génériques de ces g_i.

La seconde solution semble a priori plus intéressante, car on ne testera pas *n* fois des propositions qui ne correspondent pas dans les mondes précédents. Nous choisissons donc de suivre cette seconde solution. On remarque donc qu'ici, on est guidé par un souci "opérationnel" pour simplifier les choses quand cela s'avère possible. Ce choix nous permet de dire que nous n'allons pas garder explicitement les éléments de la formule $F_{ext}{}^1$ du monde précédent, car on sait qu'aucun de ces éléments ne permet de trouver que le document correspond à la requête, dans une nouvelle extension $F_{ext}{}^2$ de F_D dans le monde *m2*. Cette incrémentation est toujours composée d'une partie *positive*, pour étendre F_D, et d'une partie *négative*, pour fournir une valeur de vérité à pr_q si elle est non-incluse dans l'extension positive.

La formule $F_{ext}{}^2$, qui correspond à $F_{ext\_sup}{}^2$, est donc générée grâce à de la connaissance externe sur les génériques des termes, et sur la connaissance interne du processus qui exprime que l'on ne garde pas les extensions positives du monde précédent.
Le fait d'utiliser cette "transitivité" ne nous fait pas sortir du cadre logique défini plus haut, nous filtrons seulement les éléments qui ne servent pas.
La formule $F_{ext\_sup}{}^2$ est la suivante :

$$F^2_{ext\_sup} = \bigwedge_{p \in F_{K:2}(E^p_{ext1})} p \quad \bigwedge_{p \in G_{K:2}(E^p_{ext1}, pr^n_{ext1})} \neg p$$

Dans l'expression ci-dessus, nous utilisons une notation $E^p{}_{ext1}$. Elle référence seulement l'ensemble des propositions de $F_{ext}{}^1$ qui ne sont pas précédées d'une négation. Ceci représente en fait des génériques de pr_d.
Les fonctions F et G sont telles que:

$$F_{K:2} : \{ \text{proposition} \} \rightarrow \{ \text{proposition} \}$$
$$F_{K:2}(E^p{}_{ext1}) = \{ q \mid q \in \text{génériques}(r) \wedge r \in E^p{}_{ext1} \}$$

$G_{K:2}$: { proposition } × proposition → proposition
$G_{K:2}(E^p_{ext1}, pr^n_{ext1})$ = si ¬(pr^n_{ext1} est la même proposition que q
et q ∈ $F_{K:2}(E^p_{ext1})$)
alors pr^n_{ext1}
sinon *rien*

Ces fonctions sont en fait très proches de ce qui est fait pour le monde *m1*, avec la différence que l'on emploie des ensembles de propositions au lieu d'une seule proposition (qui est pr_d).

Si nous revenons sur notre exemple, nous obtenons les éléments suivants:
• l'ensemble $E_{ext1}{}^p$ est { France },
• $F_{K:2}$({France}) renvoie le singleton { Europe },
• $G_{K:2}$({France},Europe) ne renvoie rien,
• le modèle du monde *m2* est "Bretagne ∧ Europe", et dans ce monde la formule F_Q (c'est à dire "Europe") est Vraie. Le document D correspond donc à la requête Q.

Le passage d'un monde *mi* à *mi+1*, pour i>2, sera le même que celui de *m1* à *m2*, car on remarque que le processus utilisé entre *m1* et *m2* est réutilisable si l'on se base sur les mêmes connaissances. Donc le passage d'un monde *mi* à un monde *mi+1*, avec i≥1, suit le schéma suivant, par l'utilisation de l'extension $F^{i+1}_{ext\_sup}$:

$$F^{i+1}_{ext\_sup} = \bigwedge_{p \in F_{K:i+1}(E^p_{exti})} p \quad \bigwedge_{p \in G_{K:i+1}(E^p_{exti}, pr^n_{exti})} \neg p$$

avec :

$F_{K:i+1}$: { proposition } → { proposition }
$F_{K:i+1}(E^p_{exti})$ = { q | q ∈ génériques(r) ∧ r ∈ E^p_{exti}}

$G_{K:i+1}$: { proposition } × proposition → proposition
$G_{K:i+1}(E^p_{exti}, pr^n_{exti})$ = si ¬(pr^n_{exti} est la même proposition que q
∧ q ∈ $F_{K:i+1}(E^p_{exti})$)
alors pr^n_{exti}
sinon *rien*

V.1.5. Récapitulatif

Nous avons exprimé ici les éléments à mettre en œuvre pour réaliser une correspondance simple d'éléments qui sont exprimables en utilisant la logique propositionnelle.

Pour exprimer les formules qui étendent la formule du document, en vue de définir un monde d'interprétation de la formule de la requête, nous nous basons sur des ensembles de propositions décrites à partir de ces formules. Ces ensembles seront bien sûr à établir dans le contexte opérationnel de la correspondance. De plus, ces formules se basent également sur des fonctions qui seront à définir au niveau opérationnel. Nous avons différencié la "connaissance interne" de la "connaissance externe" dans l'élaboration des formules d'extensions. La connaissance externe que nous utilisons pour réaliser les implications élémentaires doit se baser implicitement ou explicitement sur la notion d'implication pour que le processus soit valable.
Nous remarquons que l'élaboration des mondes *m0* et *m1* est différente de celle des autres mondes, et cette différence va être reflétée au niveau opérationnel.

V.2. La correspondance opérationnelle de deux objets

Dans cette partie, nous allons montrer comment faire le lien entre le processus opérationnel qui va réellement comparer les objets et la partie logique que nous venons de décrire. Nous allons donc exprimer quelles sont les structures des objets que l'on compare.
Tout d'abord, nous allons définir les différentes phases de ce processus, en nous basant sur l'architecture générale du développement de la comparaison. Nous allons donc définir un découpage fonctionnel du processus, en tenant compte d'éléments liés au "monde réel" dans lequel va s'effectuer cette correspondance opérationnelle.
Ensuite, nous allons décrire les différents éléments de chacune de ces phases en utilisant l'exemple sur lequel nous nous sommes basés dans la partie précédente. Nous exprimerons donc la comparaison opérationnelle des objets documents et des objets requêtes.

V.2.1. Les propriétés du SGBD sous-jacent

Bien que la partie théorique que nous avons définie soit indépendante d'un système quelconque, la partie opérationnelle de nos travaux doit tenir compte du système sur lequel elle opère.
Nous sommes dans un cadre où nous allons comparer des objets d'une base de données gérée par un SGBD orienté objet. Nous devons donc, avant d'aller plus loin, fixer les caractéristiques d'un SGBD Orienté Objet que nous estimons nécessaires.
Nous ne décrivons ici que les parties qui sont directement liées à notre travail, en nous basant sur [Atk89] :

• La notion d'identificateur d'objet est supportée par le système.
• Les liens inter-objets de référence simples existent.
• Le système supporte également la notion de type.
• Le système propose les constructeurs *ensemble* et *n-uplet*.
• On atteint la valeur d'un objet ensemble quelconque par l'appel de la méthode `contenu` appliquée à cet objet.
• On peut atteindre tous les attributs et méthodes des objets (tout est public) durant le processus de correspondance.

Ces éléments ne comprennent pas pour l'instant le langage de requête du SGBD. En effet, nous allons par la suite définir en détail les éléments du langage de requête que nous utiliserons.

V.2.2. Architecture fonctionnelle de l'outil de comparaison proposé

Nous redéfinissons tout d'abord le cadre des comparaisons que nous proposons. Le processus de comparaison doit être appelé, comme une fonction, à l'intérieur du corps d'une méthode, ou bien comme un prédicat dans l'expression d'une requête du SGBD utilisé. Ceci implique que ce processus doit recevoir les objets qu'il compare, et qu'il renvoie un résultat qui indique si un objet O_D correspond à un objet O_Q.

Nous voulons essayer de déterminer de façon claire, grâce aux différentes phases

de ce processus, comment situer chacun des éléments de "la partie logique". Ces phases sont une aide à la conception pour le programmeur.

La figure IV4 montre les différentes partie d'un tel processus.

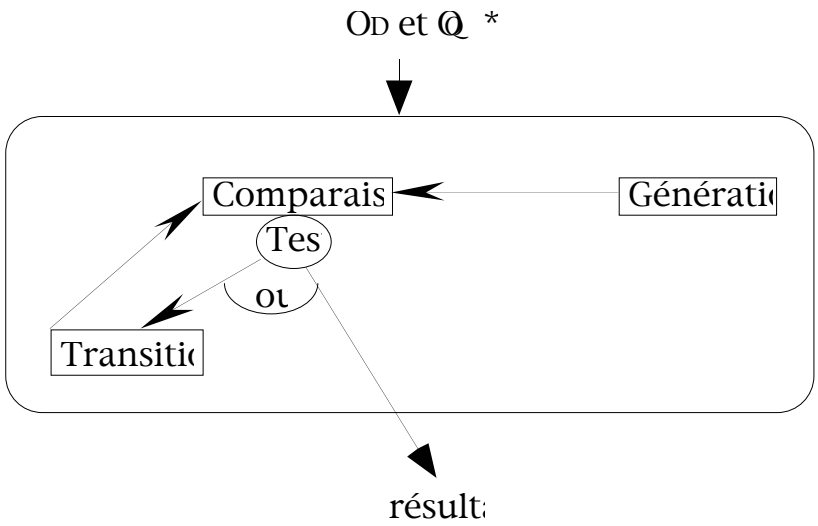


Figure IV4: Le processus de correspondance.

Passons un par un les éléments de la figure IV4:

• Les objets O_D et O_Q sont les entrées du processus de correspondance. Signalons qu'il n'est pas aberrant de penser que l'on peut fournir au processus des éléments qui vont aider ce processus (c'est ce que représente le caractère "*"), mais ce point n'est pas développé ici.
• La phase de **Génération** correspond à la génération du premier monde de la correspondance. Cette phase va permettre en outre d'initialiser les éléments dont on va se servir durant le processus.
• La phase de **Comparaison** permet l'évaluation de l'objet O_Q dans le monde qui aura été créé. En effet, comme nous l'avons dit précédemment, nous nous limitons dans un premier temps à regarder les implications élémentaires du document, mais nous devons trouver la valeur de la formule logique totale qui correspond à la requête.
• La phase de **Transition** permet de trouver un nouveau monde si l'objet O_Q ne correspond pas encore à O_D. Cette phase permet de retrouver de nouvelles implications élémentaires du document, et de réévaluer les éléments de la requête.
• Dans cette description on remarque une *étape*, **Test**, que nous n'appelons pas

intentionnellement phase. Cette étape ne correspond pas à un élément du processus logique défini précédemment. En effet, comme nous nous situons dans un cadre opérationnel, nous ne pouvons pas nous permettre (du point de vue *temps de réponse*) de rechercher indéfiniment si des objets correspondent ou non. D'après notre démarche, on ne trouve que les correspondances qui réussissent, et on ne trouve jamais que deux objets ne correspondent pas. Cette étape de Test est donc destinée à exprimer quand *couper* le processus, c'est à dire quand stopper la boucle Transition↔Comparaison.

• Finalement, le processus de correspondance fournit un résultat, qui n'est pas valué numériquement quand on utilise la logique modale simple comme nous le faisons. Le résultat est donc Vrai ou Faux.

Nous reprenons maintenant en détail chacune de ces parties dans le cadre de notre exemple simple.

V.2.3. La structure des objets de l'exemple simple

La classe `TI` décrit en fait des termes d'indexations qui sont simplement des chaînes de caractères. La définition de cette classe est donnée dans la figure IV5. Par exemple, un TI qui correspond au terme *Bretagne* sera un objet qui possède un attribut chaîne de caractères qui est "bretagne". Cette classe possède une méthode, `génériques`, qui sera le point d'appel de la connaissance extérieure à notre processus. Nous reviendrons sur ce dernier point par la suite.

```
Class TI
       type tuple(terme : string),
       method génériques: set(TI)
end;
```

Figure IV5: Définition de la classe TI.

Donc, si nous reprenons les notations utilisées dans la partie IV.1.2 du chapitre III, nous pouvons représenter l'objet qui correspond à la formule du document de l'exemple dans la figure IV6. Cet objet est un n-uplet (un n-uplet est représenté par "<>"), référençant une chaîne de caractère (représentée par "•") qui est le terme d'indexation.

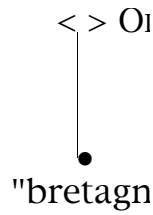


Figure IV6: L'objet document de l'exemple.

Un objet requête est lui aussi un objet de la classe TI décrite ci-dessus. Si nous reprenons l'exemple de la requête dont la formule est “Europe”, l'objet qui lui correspond sera celui de la figure IV7.

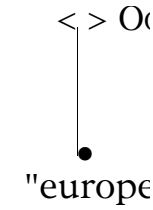


Figure IV7: L'objet requête de l'exemple.

D'après ce qui est fait dans les SGBD orientés objet (cf. les 2 chapitres précédents), **la comparaison directe de tels objets serait toujours fausse**.

V.2.4. La phase de Génération

Cette phase correspond à la génération du premier monde *m0* de la partie logique.
Examinons tout d'abord l'élément que l'on attend de l'exécution d'une telle phase:
• Nous voulons générer le monde initial, c'est-à-dire donner une valeur de vérité à chacun des termes d'indexation de la requête.

Etant donné que nous ne sommes plus dans le monde logique, mais dans le monde objet, et que ces valeurs de vérité ne font pas partie initialement des objets, il va falloir trouver "une solution" pour exprimer ces valeurs. La solution qui nous semble la plus "naturelle" est de générer un objet, appelé O_{ref}, qui va servir de liaison entre O_Q et sa valeur de vérité par l'intermédiaire de liens de référence. Nous choisissons, dans notre processus, de stocker dans cet objet la valeur de vérité "élémentaire" d'O_Q.
De plus, cette phase va nous permettre d'initialiser les éléments propres au

processus de Correspondance que nous décrivons. Si nous gardons à l'esprit que le processus va nécessiter la gestion des génériques de l'objet de la requête, nous allons choisir de les stocker dans un ensemble nommé E_D, un ensemble de TI, qui sera initialisé à vide durant la phase de Génération.

Nous avons donc choisi de représenter la déclaration d'une phase de génération de la façon décrite figure IV8.

```
Génération
Nom : <identificateur>
Pour OD : <instruction>
Pour OQ : <instruction>
```

Figure IV8: La déclaration d'une phase de Génération.

La partie *Nom* est facile à comprendre, par contre nous allons nous focaliser plus sur les deux dernières parties d'une telle déclaration, "*Pour* O_D" et "*Pour* O_Q".
Ces deux parties vont servir à initialiser des valeurs associées aux composants d'O_D et O_Q.
L'initialisation de l'ensemble des génériques, qui sera utilisée plus tard pour le document, sera faite dans la partie *Pour* O_D, car cet ensemble va servir à étendre le document.
L'initialisation de la valeur de vérité de l'unique élément d'O_Q au moyen d'O_{ref} sera faite dans la partie *Pour* O_Q.

Avant de montrer la définition de cette phase sur notre exemple, nous décrivons les éléments que nous devons proposer, et nous fixons nos notations, dans le but de savoir à quoi ressemble une *instruction* de la figure IV8. En fait, ce que nous avons appelé *instruction* sera en fait une affectation, et nous définissons ici les éléments de la partie droite de telles affectations.

Définitions des "opérateurs" utilisés

Sur les définitions suivantes, A et B sont des ensembles d'objets de type T (A et B sont donc de type *ensemble(T)*), et *o* est un objet de type *T*:
-> Les opérateurs basés sur des ensembles. Ces opérateurs ont pour but de retrouver et d'opérer sur les objets qui sont le document et la requête.

- L'*union(T)* donne comme résultat un nouvel ensemble d'objets qui

contient les objets de l'ensemble A et ceux de l'ensemble B. Ce résultat est du type *ensemble(T)*. De même, les opérateurs comme l'*Intersection* ou la *Différence* sur les ensembles existent.

• L'opérateur de requête *Image(A, f)*, inspiré de [Zdo89a]. Cet opérateur renvoie un ensemble de type *ensemble(Tf)* si le résultat de f est du type Tf. La fonction f est appliquée à chaque objet de A, et son résultat est inséré dans celui de l'opérateur :

$$\text{Image}(A, f) = \{ f(x) \mid x \textit{ in } A \}$$

• L'opérateur de requête *Project(A, <(A1,f1), …, (An, fn)>)*, inspiré de l'opérateur du même nom de [Zdo89a], donne comme résultat un ensemble d'objets n-uplets de type <A1:T1, …An:Tn>, où *Ti* est le type du résultat de la fonction *fi*. Dans le cas où une des fonctions renvoie un objet, l'attribut Ai contiendra une référence simple vers cet objet. Durant l'exécution de cet opérateur, la valeur d'un attribut Ai est le résultat de la fonction fi. On utilisera la l-notation lors de l'expression des fonctions. Cet opérateur est défini comme suit :

$$\text{Project}(A, <(A1,f1), \ldots, (An, fn)>) = \{ <A1:f1(x)) \ldots, An: fn(x)> \mid x \textit{ in } A \}$$

• L'opérateur de requête *Select(A, p)*, donne comme résultat un ensemble de type *ensemble(T)*, sous-ensemble de A. Le résultat contient des références simples sur les objets de A qui vérifient le prédicat *p* :

$$\text{Select}(A, p) = \{ x \mid (x \textit{ in } A) \wedge p(x) \}$$

• L'opérateur *Flatten(C)*, avec C un ensemble du type *ensemble(ensemble(T))*, va renvoyer un ensemble résultat de l'union de tous les ensembles éléments de C. On ôte donc un niveau d'imbrication à C en enlevant les répétitions (c'est-à-dire les objets en double) :

$$\text{Flatten}(C) = \{ r \mid \exists t (t \textit{ in } C) \wedge (r \textit{ in } t) \}$$

-> Les constructeurs d'ensembles ou de n-uplets. Ces éléments servent entre autres lors de l'initialisation des éléments propres au processus de correspondance.

• La fonction *set(o)* renvoie comme résultat un ensemble (singleton) de type *ensemble(T)* qui contient *o*.

• La fonction *<(A1:f1), …, (An:fn)>* renvoie comme résultat un n-uplet de type <A1:T1, … An : Tn>, avec les fonctions *fi* qui renvoient des résultats de type T_i.

• La fonction *élément(A)*, valide uniquement si A est un singleton, renvoie comme résultat l'élément contenu dans A.

-> une fonction basée sur la structure des objets. Cette fonction va permettre d'exprimer de façon plus déclarative la manière d'obtenir les parties liées à la sémantique des objets comparés, quand on sait quel chemin suivre pour les obtenir.

• une fonction *recurse(o, chemin)*, qui permet de trouver tous les objets reliés à *o* suivant le chemin *chemin* et ceci récursivement. Le résultat de cette fonction est l'ensemble des éléments du type de l'attribut *chemin* de l'objet *o*.

-> Un prédicat opérant sur les ensembles d'objets. Ce prédicat permet de refléter au niveau opérationnel le quantificateur existentiel.

• le prédicat de requête *existe(A, λx p)* est Vrai quand le prédicat *p* est vérifié par au moins l'un des éléments de l'ensemble A.

-> Des prédicats qui opèrent sur tout type d'objet.

• les prédicats d'i-égalité (cf chapitre III), "$=_i$" (0≤i) et d'i-inégalité "$<>_i$". Ici, c'est la notion d'i-égalité de [Zdo89a] qui est traitée. Pour nous, ces prédicats vont opérer aussi bien sur des valeurs que sur des objets. Deux objets sont 0-égaux s'ils ont même identificateur, et deux valeurs sont 0-égales si elles correspondent à la même valeur (on peut associer ce dernier point à une 1-égalité sur les valeurs). Le prédicat $<>_i$ correspond à la négation du prédicat $=_i$.

-> Une méthode prédéfinie qui traite la structure des objets. Cette méthode permet de ne pas se préoccuper du lien à suivre, lorsque l'on veut trouver les objets d'un certain type reliés à un objet de départ.

• Nous avons besoin d'une méthode prédéfinie, *desc(profondeur, type d'objet, type du lien)*. Cette méthode donne les descendants à une certaine *profondeur* (la valeur 0 indiquant tous les objets), qui sont du type *type d'objet*, et reliés au receveur de la méthode par un certain *type de lien*. Comme nous l'avons déjà signalé précédemment, différents types de lien existent. Cependant, nous avons décidé dans la partie V.2.1 de ce chapitre sur la description du SGBD sous-jacent que nous nous limitions au lien de référence simple. Pour l'instant, nous n'utiliserons que le lien de référence appelé `Ref`, mais nous gardons à l'esprit ces autres liens.

Notations utilisées

-> Nous définissons ici les notations qui vont nous être utiles par la suite:

- Nous choisissons la notation fléchée "->" pour l'appel de méthodes sur des objets, aussi bien que pour l'accès à un attribut public d'un objet, ou même pour l'accès à un attribut d'une valeur.
- L'objet *nil* correspond à l'objet vide.
- Les commentaires sont déclarés entre un "//" et la fin de la ligne courante.
- On utilise la notation ‡<expression booléenne 1>‡‡<expression booléenne 2>, qui est inspirée de la notation fonctionnelle. Une telle expression aura la sémantique suivante: la seconde expression booléenne sera évaluée si et seulement la première est Vraie, dans ce cas le résultat de cette expression sera celui de l'évaluation de la seconde expression booléenne; dans le cas contraire, cette expression est fausse. Ce type d'expression va permettre d'exprimer, à l'intérieur d'expressions, que des conditions ne doivent être évaluées que si des conditions strictement structurelles sont vérifiées sur les objets ou les valeurs.
- Nous permettons aussi l'emploi d'une notation `Reverse(A,A1,o)` qui est un raccourci d'écriture pour `Select(A, λx x->A1 =0 o)`. Cette notation, si elle n'ajoute rien à la puissance des outils que nous proposons, permet de faciliter un traitement qui va intervenir souvent pour trouver la valeur de vérité associée à des objets.
- Nous noterons l'instruction vide par `rien`.

Expression de la phase de Génération

En se basant sur les éléments ci-dessus, nous allons trouver la valeur de vérité de l'objet O_Q (en fait c'est l'extension négative du document), et préparer la structure de données qui stockera les extensions positives du document.

La partie déclaration de la valeur de vérité de l'objet O_Q qui correspond à la proposition pr_q sera déclarée dans la partie *Pour O_Q*. Pour donner les valeurs de vérité de chaque terme élémentaire de la requête (ici, il n'y a qu'une seule partie élémentaire), nous avons besoin des éléments suivants:

- on doit générer un objet qui va permettre de relier O_Q à sa valeur de vérité, en utilisant le constructeur *tuple*. L'objet généré est O_{ref}, et cet objet possède un seul

attribut, `vérité`, qui est un booléen.
• on doit ensuite affecter la valeur de vérité d'O_Q. Pour cela, on se base à la fois sur l'expression de $G_{K:0}$ et sur la formule $F_{ext}{}^0$:
-> si l'objet O_Q est 1-égal à l'objet O_D, alors la valeur de vérité associée à la proposition pr_q correspondant à O_Q est Vraie. C'est le prédicat d'1-égalité qui correspond au prédicat *est la même proposition que* de la fonction $G_{K:0}$, et le fait d'assigner la valeur de vérité *Vrai* ou *Faux* dépend de $F_{ext}{}^0$.
-> sinon, la valeur de vérité qui est associée à O_Q, via l'objet O_{ref} est *Faux*.

Pour initialiser l'ensemble des génériques d'O_D, appelé E_D, nous utiliserons le constructeur *set()*, dans la partie *Pour O_D*.
La déclaration de la phase de Génération, appelée `Gen_exemple`, est décrite dans la figure IV9.

```
Génération
Nom : Gen_exemple
Pour OD : ED = set();
Pour OQ : Oref = <(vérité, OD =1 OQ) >;
```

Figure IV9: Déclaration de la Génération de l'exemple simple.

La déclaration de la phase de Génération a donc permis de déterminer la valeur de vérité de la seule partie élémentaire de la requête. Comme nous avons affaire à un cas très simple, la phase de Comparaison qui évalue la requête globale va être ici très simple à exprimer.

En nous basant sur les objets de notre exemple, l'objet O_{ref} référence la valeur `Fausse`.

V.2.5. La phase de Comparaison

Cette phase a pour but de vérifier l'implication du document étendu vers la requête. Elle conclut donc sur la correspondance entre deux objets par Vrai ou Faux, qui indique si "$F_D \wedge F_{ext} \models F_Q$" est valide ou non.

Dans notre cas d'utilisation de la logique des propositions, il nous suffit de connaître l'ensemble des valeurs de vérité des propositions de la requête pour conclure sur l'implication (nous verrons dans le cas de RIME que la validité de

l'implication dépend aussi du document).

Dans le cas où le résultat de cette phase est Vrai, le processus de correspondance s'arrête et le résultat Vrai est fourni à l'application appelante. Sinon, une phase de Transition est exécutée. Nous reviendrons dans la suite sur le Test.

Une déclaration de phase de Comparaison va contenir:

• le nom de la phase,
• l'expression qui correspond à l'évaluation de la comparaison. C'est une expression booléenne.

La déclaration d'une phase de Comparaison est donc décrite par la figure IV10.

```
Comparaison
Nom : <identificateur>
Expression : <expression booléenne>
```

Figure IV10 : La déclaration d'une phase de Comparaison

Dans notre cas, pour l'exemple simple, l'analyse de la valeur de l'attribut `vérité` de l'objet O_{ref} suffit à déterminer si la requête est impliquée par le document, et la phase de Correspondance appelée `Cor_exemple` est décrite figure IV11.

```
Comparaison
Nom : Com_exemple
expression : Oref->vérité =0 Vrai
```

Figure IV11: Déclaration de la Comparaison pour l'exemple simple.

Nous n'avons pas utilisé d'éléments supplémentaires par rapport à ce qui a été posé dans la partie Génération.

Nous constatons donc que, comme l'objet O_{ref} contient la valeur `Faux`, la comparaison renvoie un résultat `Faux`, et que donc nous devons exécuter une phase de transition qui correspondra ici au passage du monde *m0* au monde *m1*.

V.2.6. La phase de Transition

On a vu dans la partie logique, que l'esprit de la génération du monde initial est le même que celui de la création d'un nouveau monde non-initial.
Au niveau opérationnel, une grosse différence entre la Génération et une Transition va venir du fait que l'on ne cherche plus à générer de nouveaux objets qui vont indiquer la valeur de vérité des parties élémentaires de la requête et les extensions du document, mais on va chercher à *modifier* ces objets.
Pour cela, nous pouvons utiliser une procédure prédéfinie pour la correspondance d'objets qui est

• `Apply(A, <(Ai,fi), …, (Aj,fj)>)`. Cette notation permet de donner aux attributs Ax ($1 \leq x \leq n$) des objets de l'ensemble A le résultat de l'exécution de la fonction fx. On ne modifie que les attributs exprimés dans cette procédure, les autres étant par défaut inchangés.
D'autres éléments supplémentaires peuvent être utilisés:

• On peut utiliser explicitement le numéro du monde sur lesquels on se base, grâce à `Nb_transition`, qui compte le nombre de transitions déjà effectuées.
En fait, à la vue de l'exemple simple donné, ces éléments ne seront pas utilisés.

Le modèle de la définition d'une phase de Transition est représenté dans la figure IV12. De même que pour la phase de Génération, on différencie les modifications apportées aux objets qui sont liés à O_Q de celles qui portent sur les objets liés à O_D.

```
Transition
Nom : <identificateur>
Pour OD : <instruction>
Pour OQ : <instruction>
```

Figure IV12: Modèle de la déclaration d'une phase de Transition.

Pour notre exemple, nous devons déclarer deux phases de Transition, car dans la partie logique, le passage du monde *m0* au monde *m1* est différent du passage d'un monde *mi* à un monde *mi+1* avec $i \geq 1$.

Regardons tout d'abord le passage du monde *m0* à *m1*. Nous allons réaliser

l'extension du document en nous basant sur la fonction $F_{K:1}$. En fait, nous ne stockons pas les valeurs de vérité des objets qui correspondent à cette extension, car nous considérons que ces extensions, comme les éléments de la requête, sont par définition Vrais. Si nous revenons à la définition de la fonction $F_{K:1}$, nous remarquons qu'elle fait appel à de la connaissance extérieure qui renvoie les propositions génériques d'une proposition. On savait donc qu'il existait une "base de connaissance" que nous pouvions utiliser, et que cette connaissance était accessible. Nous savons de plus que cette connaissance exprime d'une manière ou d'une autre des implications entre les termes, en fournissant les génériques d'un terme. Nous devons donc, au niveau opérationnel, permettre l'emploi de cette connaissance. Pour rester cohérent avec les définitions données précédemment, nous posons que l'appel à cette connaissance sera effectué via un appel de méthode sur les objets sur lesquels nous utilisons cette connaissance. Dans notre cas, nous utilisons de la connaissance sur les termes d'indexation, de la classe TI, et nous utiliserons la connaissance au moyen de la méthode `génériques`, qui renvoie les objets de la classe TI qui sont des génériques du receveur. Comme $F_{K:1}$ étend le document, le résultat de l'extension de O_D sera stocké dans E_D.

Au niveau de l'extension négative du document, qui se répercute au niveau de la valeur de vérité de l'unique composant d'O_Q, il va suffire de vérifier, comme nous l'avons exprimé dans la fonction $G_{K:1}$ que l'objet O_Q (qui correspond à pr^n_{ext0}) est 1-égal à l'un des objets d'E_D (que l'on vient de calculer ci-dessus). Si c'est le cas, alors la valeur de vérité que l'on va associer à O_Q est Vraie, et Fausse sinon. On suppose donc que la partie extension positive de la requête est faite antérieurement à la partie valorisation des parties de la requête, ce qui nous permet de ne pas écrire deux fois la même chose.

La définition de la phase de Transition pour réaliser l'équivalent du passage du monde *m0* au monde *m1* est donc celle donnée dans la figure IV13.

```
Transition
Nom : Tra_exemple_1
Pour OD : ED = OD->génériques
Pour OQ : Oref->vérité = existe(ED, λo o =1 OQ)
```

Figure IV13: Déclaration de la première Transition sur l'exemple simple.

Le résultat de cette Transition nous donne tout d'abord pour E_D un singleton qui contient un TI qui référence la chaîne de caractères "france" (en nous basant sur

la connaissance de la figure IV3). Dans ce cas, il n'y a aucun objet d'E_D qui est 1-égal à O_Q, et donc le document ne correspond toujours pas à la requête.

Il s'en suit l'exécution d'une nouvelle phase de Transition qui correspond à celle du passage du monde *m1* à *m2*.
Comme cette phase est très proche de celle que nous venons de définir, nous nous contentons de donner les différences avec celle-ci. La différence se situe au niveau de la partie extension positive du document (c'est-à-dire la partie *Pour O_D*). Au lieu de prendre les génériques de O_D, on va prendre les génériques des objets de l'ensemble E_D du monde précédent, et mettre ce résultat dans E_D. Pour cela, nous utiliserons l'opérateur Image pour obtenir les génériques des objets d'E_D, puis l'opérateur Flatten, pour obtenir un ensemble de TI.
La partie *pour O_Q* quant à elle ne change pas.
La phase de Transition, appelée Tra_exemple_2, qui correspond à ce passage est donc décrit dans la figure IV14.

```
Transition
Nom : Tra_exemple_2
Pour OD : ED = Flatten( Image( ED, λi i->génériques))
Pour OQ : Oref->vérité = existe(ED, λo o =1 OQ)
```

Figure IV14: Déclaration de la seconde Transition sur l'exemple simple.

A la fin de cette phase de Transition, l'ensemble E_D est un singleton dont l'élément contient la valeur "europe". Cet élément est 1-égal à O_Q, ce qui permet de donner à l'attribut `valeur` de l'objet O_{ref} la valeur `Vrai`.

La phase de Comparaison qui est exécutée par la suite trouve qu'O_Q correspond à O_D. Dans ce cas, le processus de correspondance renvoie à l'application appelante Vrai, pour indiquer qu'O_D répond à O_Q.

V.2.7. Déclaration du processus de Correspondance

Nous exprimons ici comment regrouper les différentes composantes du processus global de Correspondance.

Revenons sur l'étape de Test. Cette étape va être une expression booléenne qui peut utiliser la variable prédéfinie `Nb_transition` (qui comptabilise le nombre

de Transitions effectuées).
En fait, le processus d'utilisation de cette étape est le suivant:
• après la phase de Transition, si le résultat de la phase de Correspondance est Vrai (i.e. les deux objets correspondent), alors le résultat Vrai est fourni à l'application
• sinon, si la condition du Test est Vraie, une phase de Transition est exécutée, et si la condition du Test est Fausse, le résultat Faux est renvoyé à l'application.

L'étape de Test est un contrôle du processus de correspondance.

Le modèle de la déclaration d'un processus de correspondance (cf. figure IV15) va comprendre le nom de ce processus, le nom de ces différentes parties, le type des objets comparés et la déclaration du Test.

Comme nous l'avons constaté dans notre exemple simple, plusieurs phases de transition peuvent être employées suivant le nombre de transitions effectuées, ceci se reflète de la façon suivante: on décrit *quand* le processus doit effectuer une transition plutôt qu'une autre par une condition sur le nombre de transitions.
Nous avons également choisi de déclarer ici les variables locales au processus (comme l'ensemble E_Q de l'exemple simple) dans la partie `Local`.

```
Correspondance
Nom : <identificateur>
OD : < nom du type d'OD>
OQ : < nom du type d'OQ>
Local : <liste de déclaration de variables>
Génération : <identificateur>
Comparaison : [ <condition sur Nb_transition>
                | <identificateur> ]
Transition : <identificateur>
Test : <expression booléenne>
```

Figure IV15: Modèle de déclaration d'un processus de Correspondance.

Dans notre exemple, nous choisissons de dire que deux objets ne correspondent pas si on effectue plus de 4 transitions. La déclaration du processus de correspondance, appelée `Cor_exemple`, est donnée dans la figure IV16.

```
Correspondance
Nom : Cor_exemple
OD : TI
OQ : TI
Local :       ED : set(TI)
              Oref : type tuple(vérité : boolean)
Génération : Gen_exemple
Comparaison : Com_exemple
Transition :  [ Nb_transition = 0 | Tra_exemple_1]
                    [ Nb_transition > 0 | Tra_exemple_2]
Test : Nb_transition < 4
```

Figure IV16: Déclaration du processus de Correspondance de l'exemple simple.

V.2.8. Récapitulatif

Nous venons de faire, dans cette partie IV.2, la liaison entre le processus logique qui se base sur le modèle logique de correspondance instancié par la logique modale et un outil qui permet de traduire au niveau opérationnel cette correspondance.

Nous avons découpé le processus de correspondance en 3 phases principales qui représentent l'un des éléments du traitement que nous avons décrits par la logique modale.
La première phase, la `Génération`, correspond à une première interprétation pour les éléments de la requête pris indépendamment les uns des autres.
La `Comparaison` permet d'exprimer la manière d'évaluer l'objet requête en se basant sur les résultats de la phase précédente, qui peut être aussi bien une Génération qu'une phase de Transition.
La `Transition` permet quant à elle de passer d'un monde à autre, c'est-à-dire que l'on tente de garder les éléments du monde précédent qui sont intéressants, et on essaie de modifier les interprétations des éléments qui posent des problèmes, en utilisant de la connaissance extérieure.

L'exemple que nous avons utilisé dans cette partie est simple, c'est pourquoi nous voulons maintenant montrer dans la partie suivante que notre démarche fonctionne sur des cas plus complexes.

V.3. L'exemple de RIME

Nous décrivons ici sur un exemple complexe notre démarche.

V.3.1. Description de RIME

RIME (Recherche d'Informations MEdicales) [Chi87,Berr88] est un prototype de recherche d'informations développé par le Laboratoire de Génie Informatique de Grenoble. Dans ce système, l'accès à des images médicales (images radios et scanners) est réalisé par l'intermédiaire d'informations textuelles (les dossiers médicaux sont associés aux images dans la base de données). RIME est basée sur un modèle sémantique utilisant des représentations arborescentes dans lesquelles les nœuds feuilles sont des concepts du domaine, et les nœuds non-terminaux sont des opérateurs sémantiques qui dénotent des relations sémantiques spécifiques entre concepts et permettant la construction de concepts complexes.

On peut remarquer qu'en général si on représente de l'information sous forme structurée, il y a une sémantique associée à chaque élément de la structure. L'exemple de RIME entre donc bien dans ce cadre, et nous allons nous en servir comme base pour une explication détaillée de notre processus. Nous verrons que, si on est capable d'expliciter la sémantique et la connaissance sous-jacente à ces structures, on peut alors définir une méthode et les opérateurs nécessaires à la recherche de tels objets.

Dans le cas de RIME, le vocabulaire est dit "de spécialité", c'est-à-dire qu'il porte sur un domaine médical restreint. Dans ce domaine, les connaissances sémantiques sont très bien déterminées et, du fait du peu d'ambiguïté du langage médical, on peut exprimer cette sémantique par l'intermédiaire d'une grammaire. Une représentation possible des éléments décrits par cette grammaire est une mise en forme arborescente.

Tous les arbres manipulés sont le résultat d'une phase de compréhension, traduisant la langue naturelle française en structures sémantiques. Dans ce système, chaque concept, élémentaire ou non, possède un certain type, comme LESION, ORGANE, etc. De plus, les types (appelés classes sémantiques) sont organisés dans une hiérarchie. Examinons comment le type des concepts permet

de générer les arbres sémantiques. Par exemple, prenons un dossier D parlant d'"une opacité sur la face antérieure de l'humérus due à une tumeur". Le processus d'analyse commence par traiter la partie "face de l'humérus". Pour cela, il utilise les types des concepts de base "face" et "humérus", qui sont respectivement DETAIL et ORG (pour organe). A partir de ces types, et en utilisant la grammaire, le système détermine que le seul moyen de relier des instances de tels types est une relation a_pour_valeur_locative, le type de l'arborescence ainsi créée étant ORG. Ensuite, on relie la position "antérieur", de type POS, à cet arbre par une relation sémantique a_pour_valeur_locative, qui est la seule relation possible entre un organe et une position. L'analyse se poursuit de façon à créer l'arbre sémantique de la figure IV17, dans lequel les types de concepts sont entre crochets.

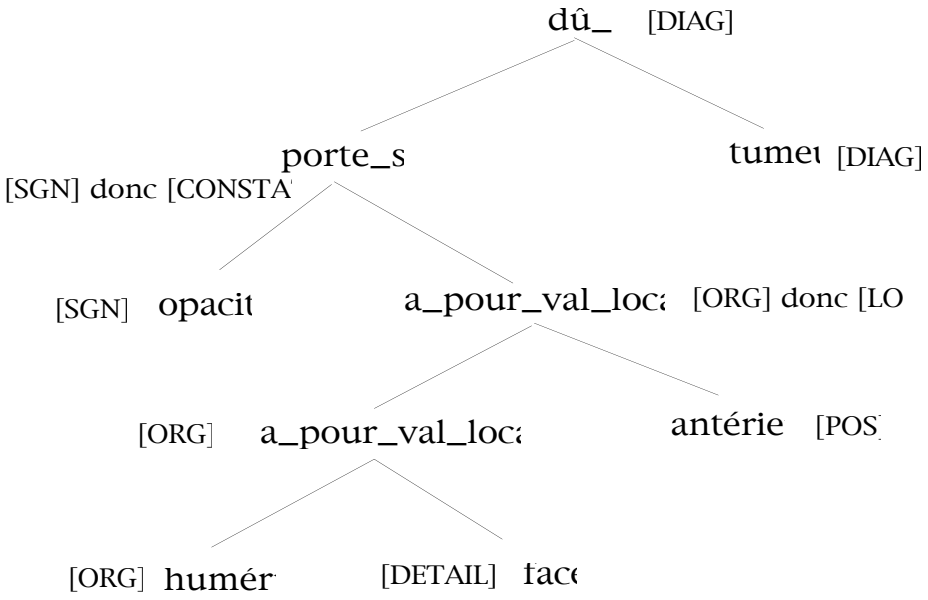


Figure IV17: Un exemple d'arbre sémantique

Nous nous limitons ici à des arborescences de termes complexes qui ne mettent pas en jeu d'opérateurs logiques tels que le ET ou le OU, ce que permet la modélisation des termes dans RIME. Par la suite, nous n'indiquerons plus les types des différents composants des arborescences.

Pour étendre les documents représentés par de telles arborescences, nous nous basons sur certaines des caractéristiques de ces arbres sémantiques décrits par J. Nie dans [Nie90a].
Dans RIME, la comparaison de concepts complexes (i.e. représentés par des

arbres non-réduits à un nœud), se base entre autres sur la substitution d'arborescences, et nous choisissons de tester notre démarche en utilisant des substitutions.

Prenons par exemple l'arbre *i'* de la figure IV18a. Cet arbre exprime le fait qu'il existe une relation porte_sur entre le concept élémentaire *rongement* et le concept élémentaire *vertèbre*. Nous exprimerons cet arbre par "porte_sur[rongement,vertèbre]". On remarque que, dans l'arbre *i'*, le nœud *porte_sur* de l'arbre est du même type que le concept *rongement*, ce que J. Nie exprime en disant que cette arborescence possède un concept gouverneur qui est *rongement*. En fait, J. Nie déduit de cette caractéristique le fait que le fils droit de la racine de *i'* est une *spécification* de son fils gauche, et que donc le fils gauche de la racine de *i'* est un générique de l'arborescence *i'*.
Dans notre cas, ces éléments sont du type SIGNE (noté SGN). De fait, il est clair qu'"un rongement qui porte sur une vertèbre" est un spécifique (au sens thésaurus) d'"un rongement".

La figure IV18a décrit que l'arbre *i*, qui était composé entre autre de *i'*, peut donc être "étendu", en disant que l'arbre *T* est aussi un descripteur valide du document. Le terme *étendu* est utilisé à dessein, car dans notre contexte, on va plutôt étendre l'arbre *i* de la figure IV18a, pour le transformer en graphe qui va contenir un lien entre le nœud *a_pour_valeur* et le nœud *rongement*, comme nous l'indiquons dans la figure 18b.

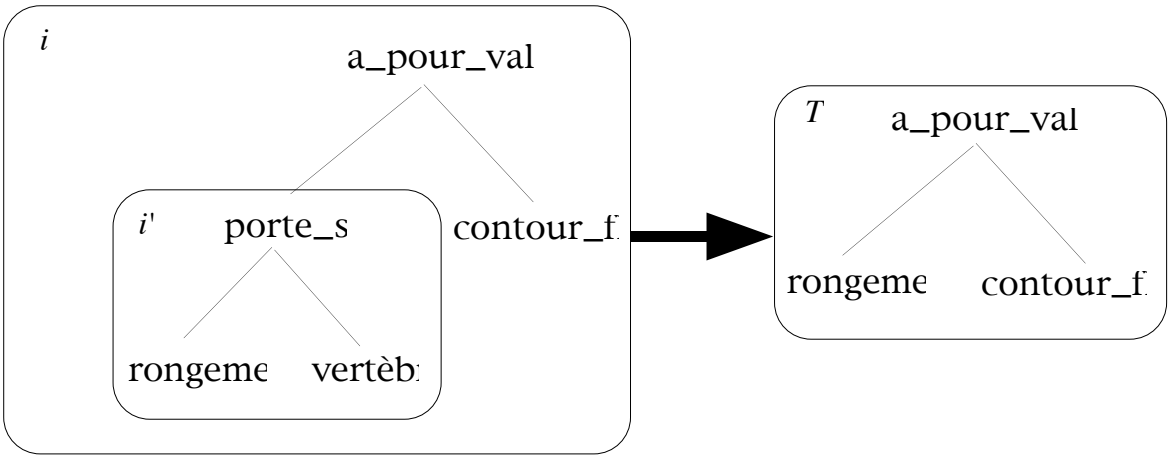


Figure IV18a: Exemple de substitution d'arbre.

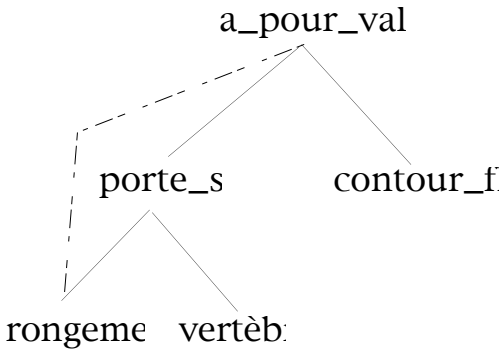


Figure IV18b : Visualisation dans un graphe du lien de substitution.

Comme nous le verrons par la suite, on différenciera, pour chaque nœud, le cas où le nouveau lien créé est une extension due à une substitution d'un fils gauche ou à celle d'un fils droit. Dans l'exemple de la figure IV18b, on crée un nouveau lien qui substitue le nœud `porte_sur`, et ce lien est donc une substitution "gauche" pour le nœud `a_pour_valeur`.

Nous ne nous intéressons dans la suite qu'aux substitutions d'arborescences sur des arbres possédant un *concept gouverneur*, car ce point nous permet de bien décrire notre démarche.
On peut résumer la démarche en disant que l'on a à notre disposition une série de règles de substitution. Ces règles proviennent de l'interprétation des arborescences et/ou des éléments de ces arborescences. Ces règles ont pour but de spécifier dans quelles mesures des inclusions d'arborescences sont "cachées" dans des arbres sémantiques.

V.3.2. L'approche logique de RIME

Si nous revenons au modèle que nous avons défini dans le début de ce chapitre, nous devons exprimer sous forme logique les "documents" et "requêtes" que nous voulons comparer. Nous allons dans cette partie nous intéresser à la formulation en logique modale du premier ordre des arborescences sémantiques, ainsi qu'aux répercutions au niveau logique de cette notion de substitutions d'arborescences.
Nous allons dans cette partie nous baser sur un exemple très simple qui représente la phrase : "opacité sur un poumon", dont la représentation par un arbre est donnée dans la figure suivante :

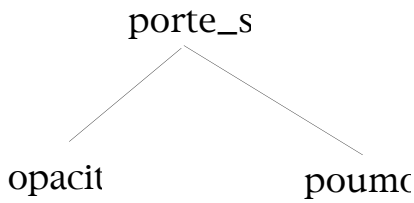


Figure IV19 : Un arbre sémantique simple.

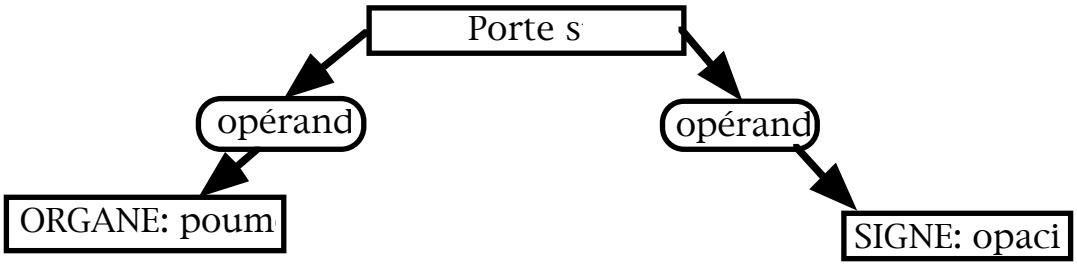
Choix de la représentation logique

Voyons la représentation logique choisie, ainsi que la raison de cette représentation. Pour nous aider à définir cette représentation, nous nous sommes fortement inspirés des travaux de SOWA sur les graphes conceptuels dans [Sow84]. Cependant, nous nous intéressons ici uniquement aux aspects structurels des Graphes Conceptuels, sans nous préoccuper de la partie des travaux de Sowa sur les opérateurs applicables à ces graphes.
Un graphe conceptuel est formé de concepts (notés ▭) et de relations (notées ⬭) entre ces concepts. De plus, les concepts sont typés, ceci veut dire que tout concept est un couple (type:référent), où le *type* est en fait un label, et le *référent* représente une instance de ce type. Nous allons passer en revue différentes manières d'exprimer les arborescences sémantiques, en indiquant les critères qui nous ont amenés à choisir une solution plutôt qu'une autre.
Par exemple, une phrase comme "il y a une opacité sur le poumon droit" pourrait être représentée par le graphe suivant appelé G :

D'après SOWA, la formule logique qui correspond à un graphe est obtenue par l'opérateur Φ, et donne dans le cas du graphe G la suivante:
"`ORGANE(poumon) ∧ porte_sur(opacité,poumon) ∧ SIGNE(opacité)`".
Cette représentation suppose qu'une relation sémantique de RIME est représentée par une relation dans les graphes. Or, cette représentation ne permet pas de différencier deux nœuds d'un arbre qui correspondent à des relations sémantiques de même nom, comme par exemple dans la figure IV17. La distinction entre des nœuds différents est cependant nécessaire, car la substitution d'arborescence dépend des instances des relations sémantiques (par

exemple on représente dans la schéma IV17 les deux relations *a_pour_val_locative* par des nœuds différents).
Cette solution ne semble donc pas correspondre à ce que nous voulons exprimer, nous en étudions donc une seconde.

Nous allons donc différencier ces instances de relations sémantiques par la solution suivante : chaque relation sémantique sera en fait représentée par un *type* de concept, le nœud d'une relation sera une instance du type correspondant. Pour notre exemple, on utilise donc le type *Porte_sur*, et on obtient le graphe suivant:



Nous avons choisi de ne pas instancier explicitement une instance de la relation *Porte_sur*, car il nous suffit juste de savoir qu'un tel nœud existe. En fait, ce choix vient simplement du fait que nous allons comparer des formules qui n'ont pas les mêmes *instances* de relations, mais nous nous baserons sur des relations *identiques* (i.e. des relations qui sont du même type, et qui ont leurs fils qui *correspondent*). La représentation logique du graphe ci-dessus est donc la suivante :
"`∃x ORGANE(poumon) ∧ Porte_sur(x) ∧ opérande1(x, poumon) ∧ opérande2(x, opacité) ∧ SIGNE(opacité)`".

La représentation ci-dessus utilise les types des concepts élémentaires. Or, comme nous l'avons dit précédemment, nous ne nous préoccupons pas de ces types, ils n'apparaîtront donc pas dans notre formulation logique finale. Rappelons que la connaissance sur les types est considérée comme gérée par la connaissance extérieure.
De plus, tous les raisonnements vont jouer sur les nœuds non-feuilles **et** leurs fils, nous choisissons donc de ne pas représenter la relation sous la forme de trois prédicats. Les relations sémantiques sont donc sous la forme d'un seul prédicat,

ce qui donne la formule "∃x Porte_sur(x, poumon,opacité)" au lieu de la formule "∃x Porte_sur(x) ∧ opérande1(x,poumon) ∧ opérande2(x,opacité)" pour représenter "il y a une opacité sur un poumon".

Donc, nous avons choisi d'exprimer un terme d'indexation complexe d'un dossier médical en utilisant les éléments suivants :
• les constantes représentent chacun des concepts élémentaires et chacune des relations sémantiques de RIME, par exemple *humérus* pour "humérus", *porte_sur* pour "porte_sur", …
• *rel*: un prédicat à trois places. rel(a,b,c) est Vrai *si et seulement si* la relation *a* (instance de la relation sémantique *rel*) existe entre les concepts qui correspondent à *b* et *c*. Un tel prédicat représente un nœud non-feuille d'un arbre sémantique.

Selon cette approche, le premier paramètre d'un prédicat *rel* sera toujours une variable.

Nous choisissons par la suite de ne pas représenter explicitement les quantificateurs existentiels, ce qui va permettre de simplifier les expressions des formules logiques.

Expression des substitutions d'arborescences

Nous représentons ici sous forme logique comment exprimer le fait qu'une substitution existe sur une partie de l'arborescence.
Nous ne nous intéressons ici qu'au résultat d'une telle substitution (nous verrons dans la partie suivante comment exprimer effectivement cette substitution).

En reprenant l'exemple de la figure IV18a, la formule logique de l'arbre sémantique *i* est "a_pour_valeur(x,y,contour_flou) ∧ porte_sur(y,rongement,vertèbre)".
Comme nous l'avons considéré dans cette figure, la relation y de type *porte_sur* est *substituable* (c'est-à-dire peut être substituée) par le concept *rongement*.
Nous étendons la formule logique qui correspond à *i*. Pour cette extension, nous cherchons tous les endroits où *y* apparaît, pour créer de nouvelles relations en substituant *y* par *rongement*.
Nous étendons donc la formule initiale en considérant que l'instance *x* de la

relation sémantique *a_pour_valeur* entre rongement et contour_flou existe. La formule entière du terme d'indexation correspondant au document indexé par *i* devient, après l'extension (où l'extension est notée en gras) :

"a_pour_valeur(x,y,contour_flou)
∧ porte_sur(y,rongement,vertèbre)
∧ **a_pour_valeur(x,rongement,contour_flou)**".

V.3.3. La correspondance logique

Nous regardons ici comment réaliser la correspondance entre les contenus sémantiques de deux arbres. Le premier correspond au contenu sémantique du document D de la figure IV17, et sa formule est :

"∃x,y,z,t dû_à(x,y,tumeur) ∧ porte_sur(y,opacité,z)
∧ a_pour_val_locative(z,t,antérieur)
∧ a_pour_val_locative(t,humérus,face)" **(F_D)**.

Le second représente la requête Q (figure IV20) exprimant que l'on recherche les documents traitant "d'opacité sur l'humérus due à une tumeur". La formule logique qui lui correspond est, en posant que le nom des variables[2] de F_Q sont tous différents des noms des variables de F_D.:

"dû_à(a,b,tumeur) ∧ porte_sur(b,opacité, humérus)" **(F_Q)**.

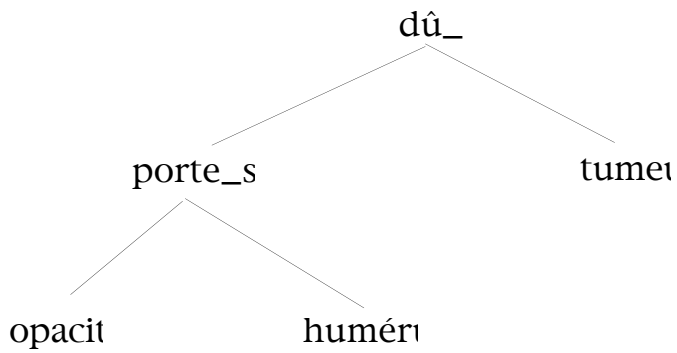


Figure IV20 : L'arbre de requête.

[2] Nous rappelons que, comme dans [Kle71], les minuscules *a, b, …, z* représentent des variables, et les majuscules comme *A, B, …, Z* des constantes dans les formules logiques que nous utilisons.

V.3.3.1. Définition du premier monde

La démarche que nous voulons suivre est la même que celle de la partie V.1.2, à la différence qu'il faut tenir compte du fait qu'il s'agit maintenant des prédicats et non plus des propositions.
Une différence primordiale, par rapport au cas de la partie V.1, est qu'ici le document **et la requête** sont des formules composées de plusieurs parties élémentaires.
L'élément principal de l'élaboration du monde initial est de donner une valeur de vérité à *tous* les éléments de la requête, pris un à un, par l'intermédiaire de la formule F_{ext}^0. En particulier, on va déterminer les éléments de la requête qui ne correspondent à aucun élément du document. Cette approche "prédicat par prédicat" permet de déterminer une condition nécessaire pour que la formule F_Q soit vraie. En effet, si l'un des prédicats de la requête pris à part n'est impliqué par aucun prédicat du document, alors le document ne répond pas à la requête. On pourra donc se baser sur ce point lors de le l'évaluation de l'implication. Par contre, si tous les prédicats de la requête, pris à part, sont impliqués par l'un des prédicats du document, alors on pourra vérifier si l'implication de tout le document sur la formule totale de la requête est vraie.

Pour exprimer la formule d'extension initiale F_{ext}^0, nous introduisons les notations suivantes:
- E_D : l'ensemble des prédicats[3] de F_D,
- E_Q : l'ensemble des prédicats de F_Q.

A partir de ces éléments, nous exprimons F_{ext}^0 telle que:

$$F_{ext}^0 = \bigwedge_{p \in G_{K:0}(E_Q, E_D)} \neg p$$

La fonction $G_{K:0}$ va seulement renvoyer les prédicats de F_Q qui ne sont impliqués logiquement par aucun prédicat de F_D. L'expression de F_{ext}^0 revient en fait à exprimer que le document implique la négation du prédicat en question, en utilisant une connaissance de la requête qui est l'ensemble des prédicats de cette requête.

[3] Quand nous écrivons *prédicat*, nous rappelons que les quantificateurs liés à des variables sont sous-entendus.

La fonction $G_{K:0}$ est décrite de la façon suivante:

$$G_{K:0} : \{ \text{prédicat} \} \times \{ \text{prédicat} \} \rightarrow \{ \text{prédicat} \}$$

$$G_{K:0}(E_Q, E_D) = \{ p \mid p \in E_Q \wedge \neg(q \textbf{ implique } p \wedge q \in E_D) \}$$

Si la formulation de F_{ext}^0 ne pose pas de problème, il nous faut cependant définir le prédicat **implique**, comme nous l'avons fait dans le cas simple pour le prédicat **est la même proposition**. Pour ce test, nous devons faire beaucoup plus attention qu'avec les propositions. En effet, les paramètres des prédicats peuvent être aussi bien des constantes que des variables.

Ce que nous avons "simplement" à faire, c'est de réaliser ce qui correspond à une implication logique entre deux prédicats *p1* et *p2*. Nous rappelons que les quantificateurs existentiels sont sous-entendus.
En fait, *p2* a une valeur de vérité qui peut être *déduite* de *p1*, et comme nous ne supposons pas de processus déductif au niveau opérationnel, nous devons effectuer explicitement ce travail. Pour nous, en posant que a,b,c,x,y et z sont des variables et A, B et C des constantes, “p1 **implique** p2” sera Vrai :

• si *p2* est de la forme `rel(a,b,c)`, et si *p1* est un prédicat ayant l'une des formes suivantes : `rel(x,A,B)` ou `rel(x,y,B)` ou `rel(x,A,y)` ou `rel(x,y,z)`,

• si *p2* est de la forme `rel(a,B, c)` de F_Q, et si *p1* est un prédicat ayant l'une des formes suivantes : `rel(x,B,y)` ou `rel(x,B,C)`,

• si *p2* est de la forme `rel(a, b, C)`, et si *p1* est un prédicat ayant l'une des formes suivantes : `rel(x,y,C)` ou `rel(x,B,C)`,

• si *p2* est de la forme `rel(a,B,C)`, et si *p1* est un prédicat ayant la forme suivante: `rel(x,B,C)`.

Cette description est une connaissance interne, car nous verrons qu'on peut l'exprimer sans avoir recours à une connaissance extérieure.

Génération du monde *m0*

Dans notre exemple, les ensembles E_D et E_Q sont:

• E_D = { dû_à(x,y,tumeur), porte_sur(y,opacité,z),
a_pour_val_locative(z,t,antérieur),
a_pour_val_locative(t,humérus,face) }
• E_Q = { dû_à(a,b,tumeur), porte_sur(b,opacité, humérus) }

D'après ce que nous venons de dire, nous obtenons les résultats suivants:
• "dû_à(x,y,tumeur) **implique** dû_à(a,b,tumeur)", donc on ne va pas étendre négativement le document avec la négation de "dû_à(a,b,tumeur)".
• aucun prédicat de E_D n'implique le prédicat "porte_sur(b,opacité, humérus)", la fonction $G_{K:0}(E_Q,E_D)$ renvoie le singleton { porte_sur(b,opacité, humérus) }, ce qui permet d'étendre la formule F_D par la formule F_{ext}^0 qui est la suivante :

"¬(porte_sur(b,opacité, humérus))".

Dans ce monde, la formule F_Q est fausse, car l'un de ses prédicats est Faux, nous devons donc générer de nouveaux mondes susceptibles de rendre cette formule Vraie.
On remarque la chose suivante, comme la formule F_Q est une conjonction de termes, une condition suffisante pour que F_Q soit Fausse est que l'un de ses prédicats le soit, nous y reviendrons dans la partie opérationnelle.

Nous venons donc de définir comment réaliser la formule d'extension initiale du document. Dans le cas où le document ne correspond pas à la requête après cette extension initiale, il faut tenir compte des substitutions d'arborescences.
Au niveau logique, la question sur la manière de vérifier si F_Q est valide ou non dans le monde défini par $F_D \wedge F_{ext}^0$ peut se baser sur la notion d'implication logique, et au cours du traitement opérationnel on devra exprimer explicitement la manière de réaliser les traitements liés à cette implication logique.

V.3.3.2. Définition du second monde

Nous allons générer plusieurs mondes, en tenant compte au fur et à mesure des éléments transitifs du type *x est substituable par y et y est substituable par z*, donc *x est substituable par z*.
En effet, si nous revenons à une vue "graphique" du processus de substitution, on crée de nouvelles relations limitées à **un** niveau de profondeur. Or, il est probable

que les substitutions jouent sur plusieurs niveaux. Dans ce cas, on augmentera, de monde en monde, la profondeur des liens de substitution. Ces nouvelles relations nécessiteront l'emploi du résultat des phases antérieures, c'est-à-dire des mondes précédents.

En fait, la difficulté va résider dans l'utilisation de la connaissance sur les substitutions pour créer réellement tous les nouveaux liens qui sont possibles, par une *sorte de fermeture transitive* (cf [Swa81]) sur les nouveaux liens de substitution obtenus au fur et à mesure de la génération *linéaire* de nouveaux mondes. En fait, l'arbre qui représente le document est un graphe particulier, qui possède la caractéristique d'avoir des liens de différents types, qui sont :

-> les liens fils-droits initiaux,
-> les liens fils-gauches initiaux,
-> les liens fils-droits supplémentaires, et
-> les liens fils-gauches supplémentaires.

La fermeture transitive se basera sur la connaissance des substitutions, et sur des règles sur les liaisons entre les types de liens qui sont, par exemple :

* la fermeture d'un lien droit-init(n1,n2) et d'un lien droit-init(n2,n3), n'est pas possible,
* la fermeture d'un lien droit-init(n1,n2) et d'un lien gauche-init(n2,n3) donne un lien droit-sup(n1,n3) si et seulement si le nœud n2 est substituable par le nœud n3,
…

En suivant la même démarche que dans le cas simple, nous commençons par décrire les notations suivantes:
• E^N_{ext0} : l'ensemble des prédicats de F_{ext}^0 qui apparaissent négativement (cet ensemble est en fait E_{ext0}, car toutes les extensions du premier monde sont négatives).

La formule F_{ext}^1 ne va être composée que de la formule $F_{ext\_sup}^1$ (cf. partie sur le cas simple en V.1.3). Comme dans le cas simple, nous avons ici une extension positive (c'est-à-dire les nouveaux prédicats créés) et une extension négative (i.e. les éléments de F_Q qui n'ont toujours pas de valeur de vérité sans cette

extension).

On n'a donc qu'à définir la formule $F_{ext\_sup}^{1}$. C'est à l'intérieur de cette formule que l'on va tenir compte du fait que des nœuds d'un arbre sémantique sont "substituables" ou non. La connaissance extérieure que nous utilisons se contente de nous dire si un nœud est substituable ou non, et c'est la connaissance interne qui crée le nouveau prédicat qui découle de cette substitution.

Examinons le processus que nous utiliserons :
si la relation représentée par rel(α,β,γ) est substituable par son fils gauche (β et γ étant des constantes correspondant à des concepts, ou des variables correspondant à des relations sémantiques), il faut trouver le prédicat *p* qui, dans F_D, utilise α en seconde ou troisième place (ceci revient à trouver le nœud de l'arbre qui est le "père" du nœud substituable). Ensuite, il reste à générer pour F_{ext}^{i+1} un nouveau prédicat en remplaçant α par γ dans une copie de *p*. Pour effectuer cette tâche, nous devons utiliser des fonctions qui manipulent des prédicats rel(α,β,γ) et qui sont les suivantes :

- des sélecteurs
  - `nom(`rel(α,β,γ)), qui renvoie le nom du prédicat, c'est-à-dire `rel`,
  - `place1(`rel(α,β,γ)), qui renvoie l'élément qui est en première place dans le prédicat, c'est-à-dire α, qui est une instance de la relation `rel`,
  - `place2(`rel(α,β,γ)), qui renvoie l'élément qui est en seconde place dans le prédicat, c'est-à-dire β, qui représente le fils gauche de la relation,
  - `place3(`rel(α,β,γ)), qui renvoie l'élément qui est en troisième place dans le prédicat, c'est-à-dire γ, qui est le fils droit de la relation,

- un constructeur
  - `fait_prédicat(`rel',α',β',γ'), une fonction de création d'un nouveau prédicat rel'(α',β',γ'), connaissant ses différents éléments rel', α' ,β' et γ'.

L'extension négative pour le document suit la même démarche que celle du monde initial.

D'après ce que nous venons de décrire, la formule $F_{ext\_sup}^{1}$ est la suivante:

$$F^{1}_{ext\_sup} = \bigwedge_{p \in F_{K:1}(E_D)} p \quad \bigwedge_{p \in G_{K:1}(E^{N}_{ext1}, E_D \cup F_{K:1}(E_D))} \neg\, p$$

Avec les fonctions définies comme suit:

$F_{K:1}$: {prédicat} → {prédicat}

$F_{K:1}(E_D)$ = { fait_prédicat(nom(r),place1(r),place2(p), place3(r)) |
$p \in E_D \wedge$ substituable(p)
$\wedge\ r \in E_D \wedge$ place2(r) = place1(p) }
$\cup$
{ fait_prédicat(nom(r),place1(r),place2(r), place2(p)) |
$p \in E_D \wedge$ substituable(p)
$\wedge\ r \in E_D \wedge$ place3(r) = place1(p) }

$G_{K:1}$: {prédicat} × {prédicat } → {prédicat}

$G_{K:1}(E^{N}_{ext1}, S2)$ = { p | $p \in E^{N}_{ext1} \wedge \neg$(q ∈ S2 ∧ q **implique** p) }

Si nous revenons à notre exemple, on a :

- E_D = { dû_à(x,y,tumeur), porte_sur(y,opacité,z),
  a_pour_val_locative(z,t,antérieur),
  a_pour_val_locative(t,humérus,face) },
- E_Q = { dû_à(a,b,tumeur), porte_sur(b,opacité, humérus) },
- E^{N}_{ext0} = { porte_sur(b,opacité, humérus) }.

A partir de ces ensembles, et en supposant que tous les nœuds du document sont substituables, on obtient comme résultat de $F_{K:1}(E_D)$ l'ensemble :
{ dû_à(x, opacité, tumeur), porte_sur (y, opacité, t),
a_pour_val_locative(z,humérus,antérieur) }.

Le résultat de $G_{K:1}(E_Q, E_D \cup F_{K:1}(E_D))$ est {porte_sur(b,opacité, humérus)}, ce qui veut dire que cette première opération n'a pas permis de trouver que le document correspond à la requête. En effet, ce résultat veut dire que, dans le document, on n'a pas trouvé qu'il existait une relation `porte_sur` entre les concepts élémentaires `humérus` et `antérieur`.
On doit donc générer le monde *m2*.

La figure IV21 représente en pointillés sur l'arborescence du document les *liens* créés.

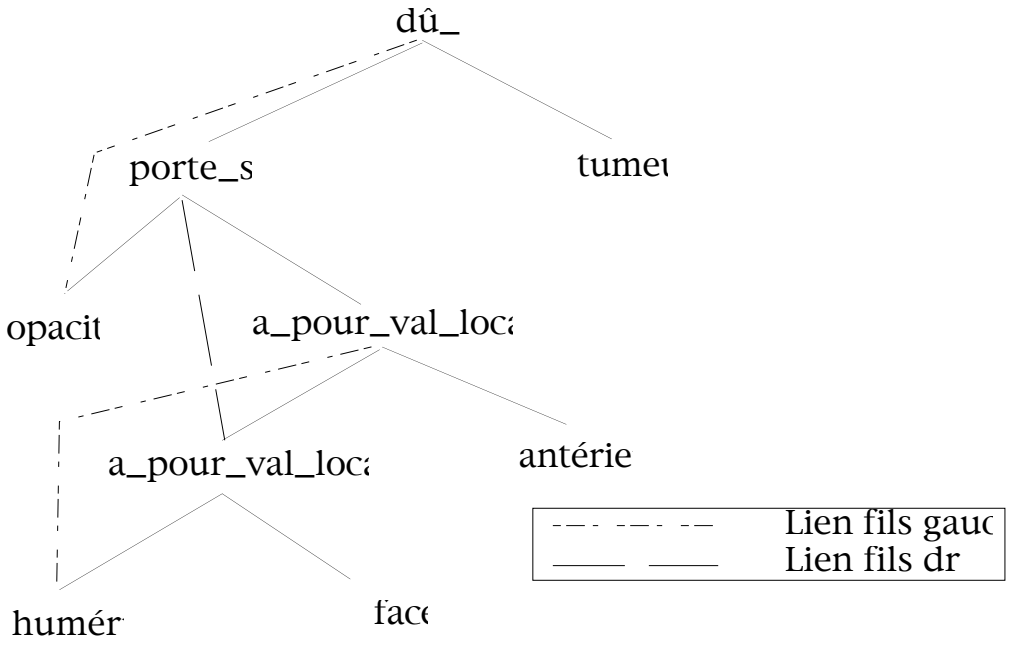


Figure IV21: Les liens créés par le premier monde.

V.3.3.3. Définition des autres mondes

Deux éléments majeurs différencient la création des mondes suivants de celle du second monde . Tout d'abord, on va utiliser les extensions positives de la formule du monde précédent *mi* pour créer la formule d'*extension incrémentale* du nouveau monde *mi+1*. Nous devons garder d'un monde à l'autre les éléments intéressants déjà obtenus. Une autre différence vient du fait que la fonction $F_{K:i+1}$ ne se basera pas uniquement sur E_D, mais aussi sur les résultats intéressants du monde précédent regroupés dans un ensemble noté E^P_{exti}, en réalisant la *fermeture transitive* au fur et à mesure de la création des mondes.

Les ensembles relatifs au monde *mi+1* obtenus à partir du monde précédent sont les suivants :

- E^P_{exti} : l'ensemble des extensions positives du monde mi,
- E^N_{exti} : l'ensemble des extensions négatives de mi (i.e. les prédicats précédés d'une négation).

L'extension incrémentale

C'est la première fois que nous rencontrons une extension incrémentale de F_D au cours de l'élaboration d'un monde. En effet, dans les cas précédents, on ne gardait pas les éléments des mondes qui étaient jugés intéressants, on se contentait à chaque fois de réinitialiser cette extension.
Ici, la démarche est différente, car nous devons garder au fur et à mesure les extensions positives de *tous* les mondes précédents. En effet, cette connaissance interne va nous servir à obtenir de nouvelles extensions.
D'une certaine manière, on peut donc dire que nous *incrémentons* les extensions de monde en monde, c'est pourquoi la formule qui répercute ces éléments *intéressants* est nommée $F_{ext\_inc}^{i}$, pour *formule d'extension incrémentale* du monde *mi*. La formule est la suivante :

$$F_{ext\_inc}^{i+1} \quad \bigwedge_{p \in E^{P}_{exti}} p$$

Cette extension du document est uniquement basée sur une connaissance de la sémantique que l'on veut accorder aux relations *inter-mondes*.

Dans l'exemple, les ensembles utilisés sont :
• E^{P}_{ext1} = { dû_à(x, opacité, tumeur), porte_sur (y, opacité, t), a_pour_val_locative(z,humérus,antérieur) }
• E^{N}_{ext1} = { porte_sur(b,opacité, humérus) }.

La formule, $F_{ext\_inc}^{2}$ est la suivante:
“dû_à(x, opacité, tumeur) ∧ porte_sur (y, opacité, t)
∧ a_pour_val_locative(z,humérus,antérieur)”.

L'extension supplémentaire

Comme nous l'avons indiqué précédemment, l'extension supplémentaire du monde *mi+1* va en fait peu différer de celle du monde *m1*. La formule $F_{ext\_sup}^{i+1}$ est la suivante :

$$F^{i+1}_{ext\_sup} = \bigwedge_{p \in F_{K:i+1}(E^P_{exti}, E_D)} p \quad \bigwedge_{p \in G_{K:i+1}(E^N_{ext1}, E_D \cup F_{K:i+1}(E^P_{exti}))} \neg p$$

On remarque deux éléments:

• les extensions négatives de ces deux formules ($F_{ext\_sup}^{i+1}$ et $F_{ext\_sup}^{1}$) sont les mêmes :

$G_{K:i+1}$: {prédicat} × {prédicat } × {prédicat } → {prédicat}

$G_{K:i+1}(E^N_{exti}, S2) = \{ p \mid p \in E^N_{exti} \wedge \neg(q \in S2 \wedge q$ **implique** $p) \}$

• l'extension positive ne se base plus uniquement sur E_D, mais aussi sur E^P_{exti}. Le problème est de trouver si les prédicats positifs du monde précédent sont substituables, et si cela est le cas il faut répercuter ce fait aux prédicats de E_D ainsi qu'à ceux de E^P_{exti}.

La fonction $F_{Ksup:i+1}$ est donc la suivante :

$F_{K:i+1}$: {prédicat} × {prédicat} → {prédicat}

$F_{K:i+1}(E^P_{exti}, E_D) = \{$fait_prédicat(nom(r),place1(r),place2(p), place3(r)) |
$p \in E^P_{exti} \wedge$ substituable(p)
$\wedge\ r \in (E_D \cup E^P_{exti}) \wedge$ place2(r) = place1(p) $\}$
$\cup$
{ fait_prédicat(nom(r),place1(r),place2(r), place2(p)) |
$p \in E^P_{exti} \wedge$ substituable(p)
$\wedge\ r \in (E_D \cup E^P_{exti}) \wedge$ place3(r) = place1(p) $\}$

Dans notre exemple, le résultat de la fonction $F_{K:2}$ est :
{ porte_sur(y,opacité,humérus)}.
Ceci vient du fait que nous obtenons un nouveau prédicat qui exprime la transitivité de "*z* est substituable par *t* et que *t* est substituable par *humérus*".
De plus, on constate que $G_{K:2}$ renvoie un résultat vide.

La formule complète $F_{ext\_sup}^{2}$ s'exprime par :
“`porte_sur(y,opacité,humérus)`”.

La formule correspondant au monde *m2* est :

“ dû_à(x,y,tumeur) ∧ porte_sur(y,opacité,z)
∧ a_pour_val_locative(z,t,antérieur)
∧ a_pour_val_locative(t,humérus,face)
∧ dû_à(x,opacité,tumeur) ∧ porte_sur(y,opacité,t)
∧ a_pour_val_locative(z,humérus,antérieur)
∧ porte_sur(y,opacité,humérus)”.

Dans l'interprétation qui rend vraie la formule du document étendue, la formule F_Q est Vraie, nous en concluons donc que les deux formules correspondent.

Le processus logique que nous venons de décrire permet donc de faire correspondre des formules logiques en mettant en jeu de la connaissance extérieure sur les substitutions d'arborescences. L'introduction de sémantique dans cette correspondance a permis de trouver que finalement elles étaient "proches".

Si l'on revient à la représentation sous forme d'arbre du document, on peut visualiser ces différents résultats. Le schéma de la figure IV22 montre en pointillé les liens représentants les extensions du document dans ce monde.
Si nous comparons l'arbre représentant le document avec celui de la requête, nous voyons que l'arbre de requête est *inclus* dans ce graphe.

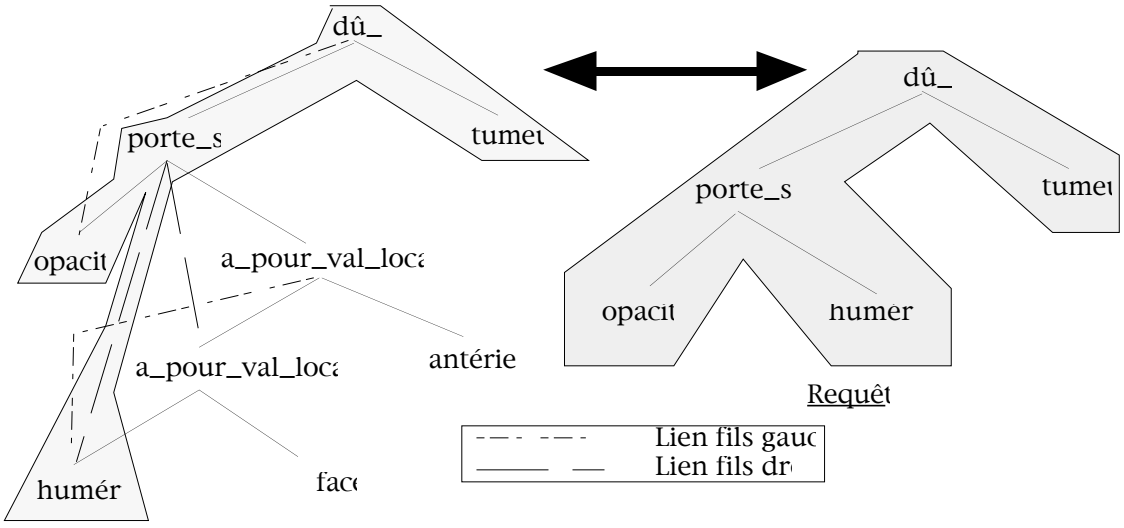


Figure IV22: Les liens créés dans le second monde, et l'inclusion de la requête.

V.3.4. La correspondance opérationnelle de deux arbres

Dans le paragraphe précédent nous avons expliqué comment on pouvait exprimer qu'un document donné correspond à une requête du point de vue logique. Notre objectif étant d'appliquer cette méthode dans une approche à objets, nous allons définir quels sont les différents éléments dont nous avons besoin, pour effectivement réaliser cette correspondance.

Nous insistons sur le fait que si nous utilisons ce qui est actuellement proposé par les SGBD Orientés Objet, ces deux objets ne sont pas comparables directement. En effet, ils n'ont ni la même structure ni le même contenu, mais nous cherchons à réaliser une sorte de test d'inclusion "intelligente" d'un objet dans un autre, via l'utilisation de connaissance non habituellement prise en compte dans les correspondances des SGBDOO.

V.3.4.1. La structure des objets "arbres sémantiques"

Pour cela, nous avons choisi d'exprimer les arbres en suivant une syntaxe proche de celle du système O_2. Le type d'un arbre est décrit dans la figure suivante.

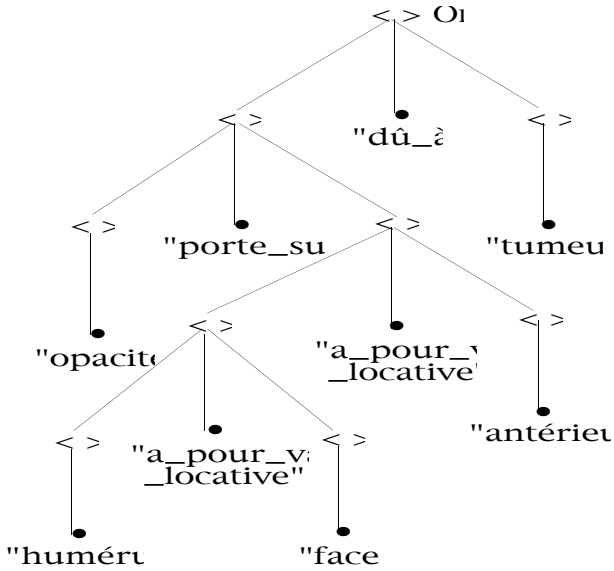
```
Class Nœud_sem
type tuple(    nom : string,
                      fils_g : Nœud_sem,
                      fils_d : Nœud_sem ),
method substituable: boolean
end;
```

Figure IV23: Définition du type des objets nœuds d'un arbre sémantique.

Les nœuds non-feuilles n-uplets correspondent aux prédicats *rel*, et les nœuds feuilles (c'est-à-dire ceux donc les attributs fils sont nil) aux constantes du domaine. L'attribut *nom* d'un nœud sémantique non-feuille contiendra le nom de la relation sémantique. Ce même attribut d'un nœud feuille contiendra le nom du concept. La méthode `substituable` est le point d'appel de la connaissance externe. Cette méthode renvoie Vrai si le receveur est substituable, et Faux sinon.

L'arbre sémantique de la figure IV17 sera représenté par l'objet O_D de la figure IV24a, et l'arbre de requête de la figure IV19 correspondra à l'objet O_Q de la

figure IV24b.



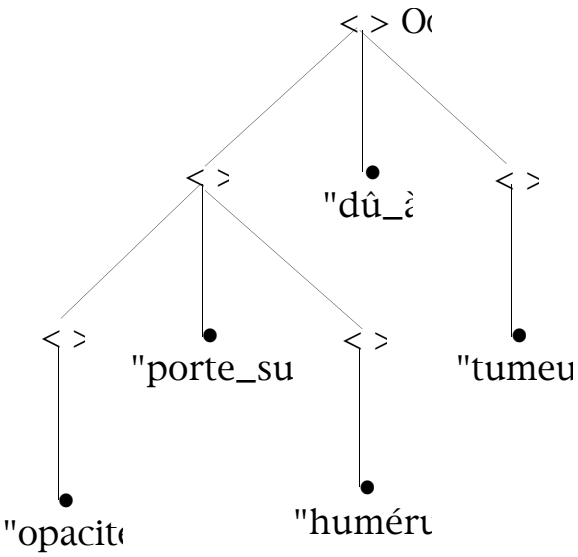


Figure IV24a: Un objet Document.

Figure IV24b: Un objet Requête

V.3.4.2. La phase de génération

Cette phase initialise les valeurs des éléments d'O_Q qui correspondent à un prédicat de F_D et les éléments qui vont stocker les extensions du document.

Structure des éléments créés par le processus

-> Pour les nœuds d'O_Q, les nouveaux objets générés ne contiendront que deux attributs (comme dans le cas simple), qui sont une référence vers l'objet d'O_Q appelée `term`, et une référence vers la valeur de vérité de cet objet appelée `vérité`. Le calcul de cette valeur de vérité va correspondre à la partie génération de la formule F_{ext}^0. Ces objets seront regroupés dans un ensemble appelé E_Q qui a donc la structure suivante :

```
EQ : type set(tuple(term : Nœud_sem, vérité : booleen ))
```

-> Pour l'extension d'O_D, nous avons choisi de construire des objets possédant trois attributs. Le premier, `term`, est une référence vers les nœuds non-feuilles d'O_D. Le second, `sup_g`, sera un ensemble de références vers les nœuds d'O_D qui représentent les nouveaux liens *gauches* qui ont été obtenus lors de l'extension du document. Le troisième attribut, `sup_d`, a le même rôle que `sup_g` mais pour les

liens droits. Ces objets seront regroupés dans l'ensemble appelé E_D qui a la structure suivante :

```
E_D : type set(  tuple(     term : Nœud_sem,
                           sup_g : set(Nœud_sem),
                           sup_d : set(Nœud_sem)));
```

L'initialisation de ces éléments

-> Pour la requête, cette génération utilise l'**implication** dont nous avons parlé précédemment. Pour exprimer cette condition au niveau opérationnel, nous allons tester les cas suivants, avec n_Q le nœud non-feuille courant de O_Q:

• si ce nœud possède deux fils non-feuilles, i.e. il correspond à un prédicat *rel* dont les secondes et troisièmes places sont des variables, et s'il existe un nœud n_D de O_D qui a le même nom de relation sémantique, alors n_Q correspond à un prédicat qui a une valeur de vérité. Ceci veut dire que n_Q a une valeur de vérité Vraie quand il est considéré seul, i.e. sans les autres nœuds de O_Q.

• si ce nœud correspond à un prédicat *rel(x,A,y)*, avec A une constante et y une variable, c'est-à-dire que son fils gauche est une feuille et que son fils droit n'est pas un feuille, et s'il existe un nœud n_D de O_D qui correspond à un prédicat *rel(b,A,c) ou rel(b,A,D)* avec c une variable et D une constante, alors n_D sera associé à la valeur Vraie.

• même démarche si n_Q correspond à *rel(x,y,A)*,

• si n_Q correspond à un prédicat avec A et B constantes, c'est à dire que n_Q possède deux fils feuilles, ce nœud ne peut être impliqué que par un nœud n_D correspondant à *rel(c,A,B)*.

-> Tout objet d'O_D possède initialement un ensemble vide pour les attributs de liens supplémentaires, car le premier monde ne fait pas appel à la connaissance sur les substitutions.

La figure IV25 décrit la déclaration de cette Génération.

Nous nous limitons ici à la génération des valeurs de vérités des objets de O_Q qui correspondent à un prédicat *rel(x,A,B)* .

La description complète de la partie se rapportant à O_Q est donnée en annexe 1.

```
Génération
Nom : Gen_complex
Pour OD : ED =
  Project( Select(
                    Union(OD -> desc(0,Nœud_sem, Ref), set(OD)),
              λi i->fils_g <>0 nil ) ,
     λd <term:d, sup_g:set(), sup_d: set()>)
   // on crée les nouveaux objets référençants ces nœuds non-feuilles de OD,
   // et on initialise les attributs de liens gauches et droits supplémentaires à l'ensemble vide
Pour OQ : EQ =
  Project(
      Select( Union(OQ -> desc(0,Nœud_sem, Ref), set(OQ)),
              λo o->fils_g <>0 nil),    // même chose pour OQ
      λd <( term,d),
           ( vérité,
            ( d->fils_g->fils_g =0 nil
              and d->fils_d->fils_g =0 nil
               and existe (
                     Select(Project(ED,λo  o->term),// les objets de OD
                    λe (    d->nom =0 e->nom
                        and e->fils_g->fils_g =0 nil
                        and e->fils_g->nom =0 d->fils_g->nom
                        and e->fils_d->fils_g =0 nil
                        and e->fils_d->nom =0 d->fils_d->nom)
// on crée ici les valeurs de validité pour les noeuds de OD qui ont deux fils-feuilles
...
```

Figure IV25: Déclaration de la Génération de l'exemple complexe.

Les résultats de cette phase sur les objets O_D et O_Q que nous avons décrits dans les figures IV24a et IV24b sont donnés dans les figures IV26a et IV26b.

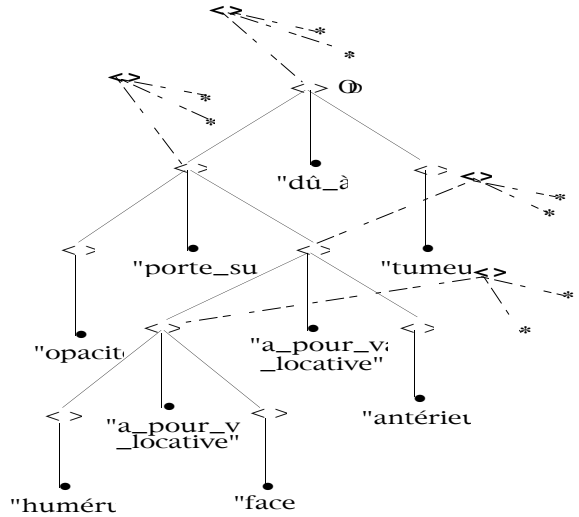


Figure IV26a: Les éléments générés pour O_D.

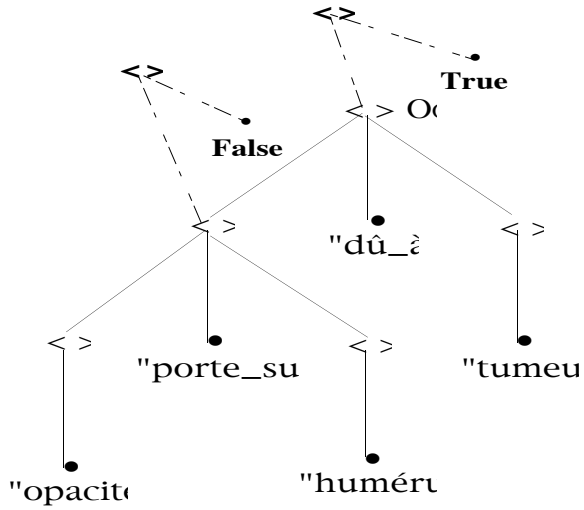


Figure IV26b: Les éléments générés pour O_Q.

V.3.4.3. La phase de Comparaison

Le processus d'évaluation de la requête dans le monde du document étendu est en fait un élément qui n'apparaît pas au niveau logique, mais nous devons explicitement réaliser l'implication globale des deux formules. Cette phase est décrite dans la figure IV27.
Comme nous l'avons remarqué dans la partie logique, une condition nécessaire pour que le document corresponde à la requête est que chaque élément de la requête ait une évaluation Vraie. Cette condition, bien que nécessaire, n'est pas suffisante pour que "$F_D \wedge F_{ext}^i \models F_Q$" soit valide. Il faut un processus supplémentaire pour vérifier l'inclusion de l'arbre de requête dans le document étendu. Nous rappelons ici deux notations :
• "‡ <cond1> ‡‡ <cond2>", qui correspond à une notation fonctionnelle qui renvoie le résultat de l'évaluation de la condition booléenne <cond2> si la condition <cond1> est Vraie, et qui renvoie Faux sinon,
• "reverse", qui permet de retrouver un élément d'un ensemble en se basant sur une 0-égalité de l'un de ses attributs.

```
Comparaison
Nom : Comp_complex
Expression :
   ‡ not existe(EQ, λe e->vérité =0 False)
   ‡‡ ( OQ->nom =0 OD->nom
       and ( Comp_complex(OQ->fils_g, OD->fils_g)
                  or existe(
                    reverse(ED, OD, term),
                          λx existe(
                            x->sup_g,
                                     λy Comp_complex(OQ->fils_g,y)))
                        )
       and ( Comp_complex(OQ->fils_d, OD->fils_d)
                 or existe(
                   reverse(ED, OD, term),
                         λx existe(
                           x->sup_d,
                                     λy Comp_complex(OQ->fils_d,y))
                             )
           )
        )
    or Comp_complex(OQ, OD->fils_g)
    or Comp_complex(OQ, OD->fils_d)
```

FigureIV27: Déclaration de la Comparaison dans le cas complexe.

Le résultat de cette phase de comparaison appliquée aux objets des figures IV25a-25b donne Faux. Il faut donc "effectuer un changement de monde", ce qui correspond à la phase de Transition suivante.

V.3.4.4. Les phases de Transition

Cette phase va donc s'attaquer à la fois à étendre O_D et à retrouver les valeurs de vérités élémentaires de chaque nœud de O_Q.
Il s'agit, grâce à ces phases, de construire les extensions du document (F_{ext}), constituées des deux extensions : positives (liées dans le processus à O_D), et négatives (liées dans notre processus à O_Q).

La première Transition

Cette première Transition correspond au premier changement de monde à partir du monde initial.

-> l'extension positive du document :
Nous suivons ce que nous avons décrit dans la partie V.3.3.2, en particulier l'expression de la fonction $F_{K:1}$, pour trouver les nœuds de O_D qui ont un fils substituable par les liens générés. La fonction *fait_prédicat* sera réalisée simplement en créant un nouveau lien dans l'ensemble de liens supplémentaires adéquat de l'objet qui stocke les liens supplémentaires d'un nœud.

-> l'extension négative, quant à elle, a la même expression que dans la phase de génération, à la différence qu'elle effectue ses recherches d'implications élémentaires sur les nœuds de O_Q qui sont Faux.

Nous ne donnons pas ici la formulation complète de la partie `Pour` O_Q, mais elle figure en annexe 2.

La Transition de ces objets est exprimée ci-dessous :

```
Transition
Nom : Trans_complex_1
Pour O_D :
    Apply(
        E_D,
        λf <( sup_g,
                  Image(
                        Select(
                            set(f->term->fils_g),
                            λg g->substituable =_0 True
                              ),
                                 λh h
                             )),
                  ( sup_d,
                      Image(
                         Select(
                             set(f->term->fils_d),
                             λg g->substituable =_0 True
                               ),
                                 λh h
                            )
              ) > )
Pour O_Q : Apply(Select(E_Q, λo o->vérité =_0 False), …
```

Figure IV28: Déclaration de la première phase de Transition sur l'exemple complexe.

Les Transitions suivantes

Ces Transitions sont différentes de la précédente par le fait qu'elles utilisent de la connaissance incrémentale (F_{ext_inc}).
Dans le processus décrit ici, les extensions positives du document sont stockées par des objets générés qui représentent les liens gauches et droits supplémentaires par rapport aux liens d'O_D. Au cours d'une nouvelle phase de Transition, pour garder ces liens (c'est-à-dire pour répercuter le fait que ces liens vont toujours exister dans le nouveau monde), il suffit de faire des unions entre les ensembles du monde précédent, et ceux qui sont trouvés par transitivité.

Pour ce qui est de l'extension supplémentaire du document, la partie "extension négative" est la même que dans la première transition (cf. annexe 2). La partie extension positive est proche de ce qui est décrit dans la première Transition, mais on ne va plus rechercher si un nœud est substituable par son fils gauche, mais par ses fils gauches supplémentaires. La Transition est donc la suivante :

```
Transition
Nom : Trans_complex_2
Pour OD :
   Apply(
       Select(ED, λe non-vide(e->sup_g) or non-vide(e->sup_d)),
       λf <(sup_g, Union(
                       f->sup_g,
                              Flatten(Image(f->sup_g, λg g->sup_g))
                                   ) ),
                (sup_d, Union(
                       f->sup_d,
                                Flatten(Image(f->sup_d, λg g->sup_g))
                                   ) ) >
          )
Pour OQ : Apply(Select(EQ, λo o->vérité =0 False), …
```

Figure IV29: Les Transitions suivantes.

V.3.4.5. La déclaration du processus global de correspondance

Ce processus va donc contenir l'ensemble des phases que nous venons de décrire, et nous considérons que la correspondance est Fausse au bout de 5 transitions.

Le processus global est donc décrit dans la figure IV30.

```
Correspondance
Nom : Cor_complex
OD : Nœud_sem
OQ : Nœud_sem
Local : ED : type set(tuple(term : Nœud_sem,
                            sup_g : set(Nœud_sem),
                                     sup_d : set(Nœud_sem)));
          EQ : type set(tuple(term : Nœud_sem,
                                vérité : booleen ))
Génération : Gen_complex
Comparaison : Com_complex
Transition :  [ Nb_transition = 0 | Trans_complex_1]
                     [ Nb_transition > 0 | Trans_complex_2]
Test : Nb_Transition < 5
```

Figure IV30: Déclaration du processus global de correspondance.

V.3.5. Conclusion sur cet exemple

Nous venons de décrire sur un exemple complexe, proche de la réalité d'un système de recherche d'informations existant, comment exprimer les

comparaisons entre deux arborescences dont la sémantique est connue.
La partie décrite ici n'est qu'un élément de la comparaison complète d'un document et d'une requête dans RIME. En effet, nous ne nous sommes pas préoccupés d'éléments comme la généricité entre concepts élémentaires du domaine (par exemple le concept “bras” est un générique du concept “humérus”), et nous n'avons pas décrit ici comment traiter une requête comprenant des connecteurs logiques.
Cependant, malgré les simplifications faites ici, on se rend compte que la liaison logique<->opérationnel est une aide non négligeable pour aborder les problèmes de correspondance entre objets complexes ayant une sémantique donnée.

VI. Simplification du processus

Nous décrivons ici comment, a posteriori, on peut tenter de simplifier la tâche décrite dans les parties précédentes. En effet, on peut se poser la question de l'intérêt des extensions négatives du document, et de savoir dans quelle mesure elles servent réellement.
Au niveau logique, ces éléments se trouvent au niveau de la définition des mondes, et on en tient compte lors de l'implication. Comme nous l'avons indiqué dans la partie opérationnelle de RIME, ces extensions sont utilisées lors de la phase comparaison entre les objets en tant que génératrices de conditions nécessaires à la correspondance. Ces extensions sont générées et modifiées dans les phases de générations et de transitions dans la partie `Pour OQ`. En fait, si on se passe d'extension négative, il suffit de ne pas utiliser les parties `Pour OQ` dans les générations et transitions, et de faire dans le cas d'arborescences un simple parcours récursif en parallèle pour détecter l'inclusion de la requête dans le document étendu.
En fait, étant donné que les éléments traités forment des conditions nécessaires, il est possible de s'en passer et de calculer directement les comparaisons entre objets sans utiliser ces conditions. Nous décrivons ici les avantages (notés +) et les inconvénients (notés −) de l'utilisation de ces extensions négatives :

+ l'apport de ces éléments est de permettre, par test simple, de réaliser la phase de comparaison (récursive dans le cas des arbres) seulement quand on sait que l'on a une bonne chance de trouver effectivement qu'il y a correspondance.

+ dans certains cas, par exemple lors de l'utilisation de propositions qui représentent un objet dont l'évaluation est coûteuse, il est plus intéressant de noter qu'un élément de la requête est vrai, et de ne plus se préoccuper de le réévaluer à chaque monde. Lors de l'évaluation de la formule de la requête, on se sert alors de la valeur vraie associée, sans aller plus loin. Dans ce cas, cet objet sera évalué tant qu'il n'aura pas été trouvé vrai, mais ensuite il ne sera plus évalué, ce qui est intéressant.

− dans le cas de RIME tel que nous l'avons exprimé dans ce chapitre, il est difficile de quantifier a priori si l'on gagne ou non de la vitesse avec l'utilisation d'extensions négatives, en particulier à cause du fait que l'on teste de moins en moins de nœuds au fur et à mesure que ceux-ci correspondent. Le problème vient aussi du fait que, même si la condition nécessaire est vérifiée, l'évaluation de la comparaison peut très bien fournir un résultat faux. De plus, l'utilisation de ces extensions doit tenir compte de la taille (profondeur des arbres, …) des objets comparés. Si nous revenons à RIME, dans lequel les arbres ne dépassent que rarement une profondeur de 5, le temps perdu à faire cette évaluation sera probablement prohibitif.

− comme on peut le remarquer dans les descriptions des phases de générations et de transitions dans le cas de RIME, les expressions les plus complexes proviennent des expressions des extensions négatives (cf. annexes 1 et 2). Il semble donc que le langage que nous avons décrit s'applique difficilement à ces expressions.

En restant objectif, on peut dire que l'intérêt des extensions négatives telles que nous les avons décrites ne sera démontrable qu'en faisant des comparaisons sur des expérimentations. Si un programmeur veut exprimer simplement des comparaisons d'objets, il est préférable pour lui de ne pas se préoccuper des extensions négatives.

VII. Conclusion

Nous avons décrit dans ce chapitre notre approche pour la comparaison d'objets complexes selon des critères prédéfinis et liés à ce que l'on pense de ces objets. Pour cela, nous proposons un outil qui permet de déclarer de telles correspondances. Nous voulons permettre de déclarer des processus de

correspondance qui, quand on donne un objet A et un objet B, permettent de dire si l'objet A peut être considéré comme "répondant" bien à ce que représente l'objet B.

Le processus sous un nouvel angle

Pour placer le travail que nous avons fait sous un autre angle, nous pouvons résumer notre démarche au niveau des objets en disant que notre processus est composé de deux grandes parties :

- une partie de création de liens **entre les objets de la base**, pour réaliser l'extension du document,
- et une partie de calcul d'"inclusion" de l'objet requête dans l'objet document.

Ces deux points utilisent une sémantique des objets comparés non uniquement structurelle.

-> La partie "création de liens" :
Dans le cas de l'exemple simple qui nous a servi à décrire le processus de correspondance, on relie l'objet qui représente le document à des objets de la base. Dans l'exemple initial, il s'agit d'un thésaurus préexistant, comme nous le montrons dans la figure suivante :

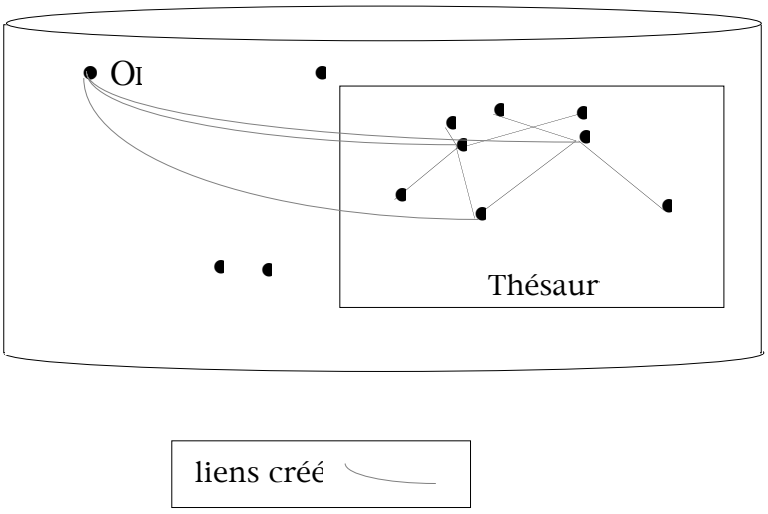


Figure IV31 : Les liens créés dans l'exemple initial.

Dans l'exemple de RIME, les nouveaux liens que nous avons créés ne font en fait que relier des composants d'O_D, mais l'idée est toujours de relier des éléments de la base de données en se servant de connaissance, comme nous le montrons dans

la figure suivante :

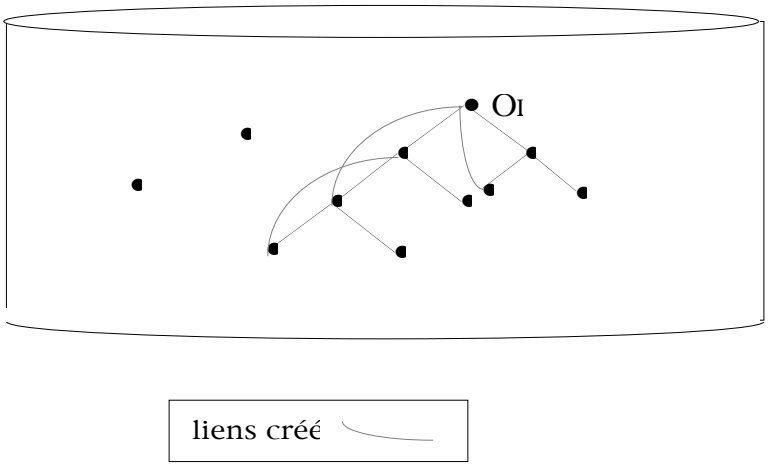


Figure IV32 : La création des liens dans l'exemple de RIME.

Dans les deux cas que nous venons de décrire, nous créons des liens supplémentaires entre des objets de la base, ce qui correspond bien à une extension du document.

-> Dans la seconde partie, on exprime le processus d'inclusion de l'objet requête dans l'objet document étendu par ces nouveaux liens. Dans cette partie, nous formulons des "équivalences" entre les liens *initiaux* de la requête et les liens *initiaux* et *supplémentaires* du document.
Dans le cas simple, on se contente de regarder si l'objet requête O_Q est 1-égal à l'un des objets reliés à O_D, ce qui est assez simple à exprimer :

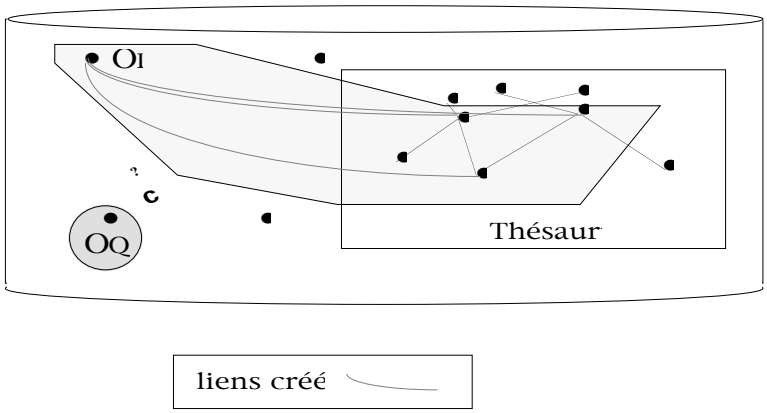


Figure IV33 : L'inclusion dans l'exemple simple.

Dans le cas de RIME, cette inclusion est plus complexe, dans la mesure où il est nécessaire de réaliser des recherches sur tous les fils possibles d'un nœud, via ses liens initiaux et ses liens supplémentaires, mais on recherche toujours une inclusion, comme nous le montrons dans la figure suivante :

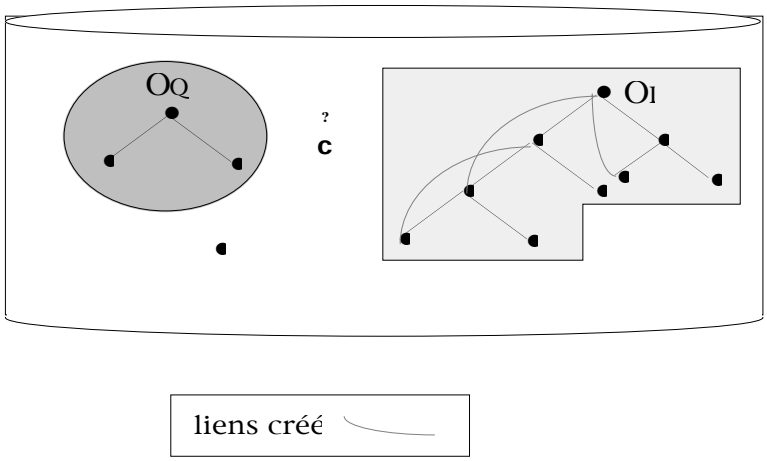


Figure IV34 : L'inclusion dans le cas de RIME

Récapitulatif

Pour revenir à ce que nous avons décrit dans ce chapitre, pour réaliser un processus de correspondance nous déclarons 3 phases distinctes qui sont directement inspirées de la décomposition sous forme logique du modèle logique de recherche d'informations proposé par C.J. Van Rijsbergen.

En utilisant la logique pour représenter la sémantique des objets comparés et la logique modale pour modéliser le traitement de correspondance, on peut déclarer chacune des phases opérationnelles nécessaires à la réalisation effective de la correspondance dans le SGBD Orienté Objet.

Le processus que nous suivons peut donc être décrit pas le schéma de la figure IV35.

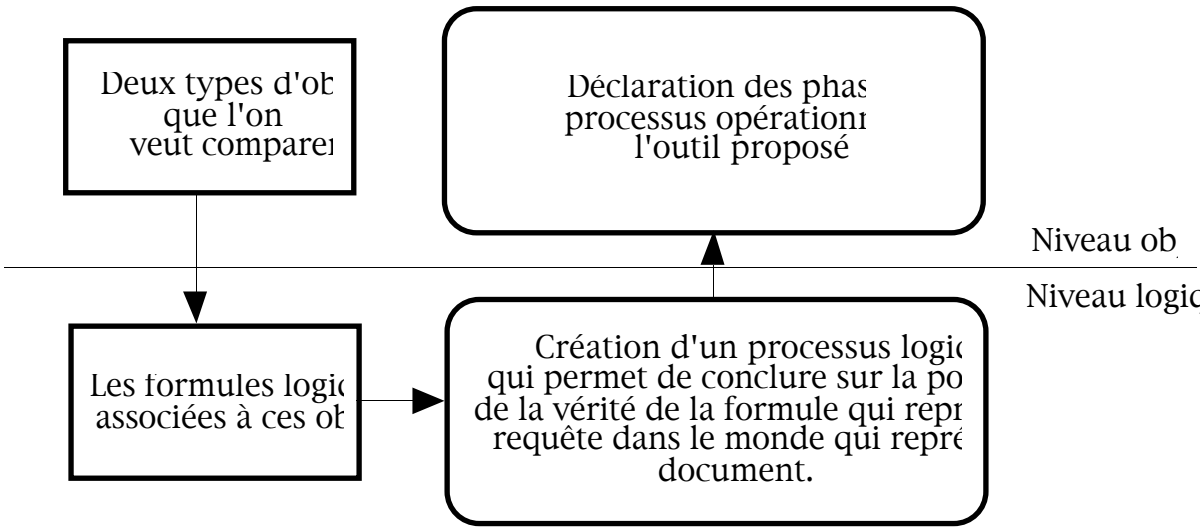


Figure IV35: Les étapes de l'élaboration d'un processus opérationnel de correspondance.

Notre approche permet en fait de comparer principalement des objets de même type. Ce postulat nous permet de ne comparer que des objets qui ont au moins un certain nombre de points communs.

La commande *Apply* est donc réservée à la phase de Transition, car c'est au cours de cette phase que nous réalisons des modifications sur les objets comparés. L'opérateur de projection sera plutôt employé au niveau de la génération, pour créer les objets qui vont être utiles au processus. Les autres éléments, comme la sélection, l'applatissement, le quantificateur *existe*, ..., peuvent être utilisés indifféremment par toutes les phases.

Comme nous l'avons remarqué dans la partie VI, les extensions négatives qui permettent de déclarer des conditions nécessaires pour que les objets correspondent sont à utiliser avec parcimonie.

L'outil que nous proposons est en fait composé d'une interface entre l'utilisateur et le SGBD sous-jacent pour déclarer les différentes phases du processus opérationnel et d'une partie interface entre une application et le SGBD sous-jacent, pour "détourner" les appels vers un processus de correspondance de façon à exécuter les méthodes qui correspondent effectivement à ce processus.

Parallèle avec le système VAGUE

Pour faire un parallèle avec ce que nous avons décrit dans le chapitre précédent, nous pouvons comparer notre approche avec celle du système VAGUE de Motro basé sur le relationnel.
Rappelons que dans VAGUE, le concepteur de la base définit un certain nombre de comparaisons possibles sur des éléments de la base, ou bien sur des n-uplets. Ces définitions sont réalisées par l'intermédiaire d'outils et/ou de fonctions qui permettent de décrire facilement les correspondances. Une fois cette tâche effectuée, l'utilisateur n'a plus qu'à choisir le type de correspondance qu'il veut utiliser lors du traitement d'une requête.
Dans notre cas, nous allons plus loin, dans la mesure où nous proposons un outil et des opérateurs dont l'emploi est régi par une approche logique de la sémantique des objets comparés. De plus, nous avons intégré la notion d'objet complexe qui n'existe pas dans VAGUE.

CHAPITRE V

Correspondance d'objets dans le cadre des domaines BD et RI

CHAPITRE V

Correspondance d'objets dans le cadre des domaines BD et RI

I. Introduction

Dans le chapitre précédent, nous avons indiqué comment un programmeur d'application exprime et réalise des correspondances d'objets complexes. Rappelons que le but de notre travail est d'initialiser le rapprochement des bases de données Orientées Objet et les Systèmes de Recherche d'Informations, en permettent d'intégrer des correspondances basées sur la logique, mais sans nécessiter de moteur d'inférence ou de techniques sophistiquées de déductions, car notre contexte le permet. En effet, nous avons supposé ici que les implications possibles qui permettent d'étendre un document sont élémentaires, c'est-à-dire de la forme "A => B", et que de plus les valeurs de vérité des parties de la requêtes étaient simples à déterminer. Cependant, on peut estimer que les éléments que nous avons utilisés ne sont probablement plus applicables en cas de déductions plus complexes à évaluer.
Nous voulons, dans cette partie, situer clairement des éléments qui "englobent" notre travail, et qui sont :

i) Comment notre approche s'intègre dans la conception d'une application bases de données,
ii) Comment utiliser notre outil pour réaliser des systèmes de recherche d'informations.

Pour le point i), nous ne nous préoccuperons pas réellement de l'inclusion de notre travail dans une méthode de conception, comme MOSAIC [Bri93] par exemple.

La méthode MOSAIC permet de concevoir un système d'applications programmé à l'aide d'un SGBDOO, en privilégiant une décomposition dirigée par les objets au détriment de l'habituelle séparation entre la base de données et les programmes d'application. Nous allons donc nous concentrer uniquement dans la partie II à situer notre outil par rapport aux autres parties de codage de l'application.

Le second point, quant à lui, nous servira à délimiter nos propositions dans le cadre Recherche d'Informations.
Nous insisterons sur les éléments supplémentaires qui doivent être proposés et étudiés pour que l'objectif final soit atteint et que l'on ait l'ensemble des outils pour faciliter l'écriture d'un SRI, qui est vu comme l'écriture d'une application dans un SGBDOO.

II. La Correspondance d'Objets dans une Application BD

Nous allons baser la description qui suit dans le cadre d'une intégration *faible* du processus de correspondance dans le SGBD; ceci veut dire que nous considérons que le processus décrit par le programmeur va être traduit en méthodes, comme pour une application "habituelle".

II.1. La description fonctionnelle de la partie opérationnelle

Notre prototype, que nous nommerons Opérateur pour la Correspondance d'Objets Complexes (OCOC), est donc découpé en deux parties qui sont :

• Le Générateur de Correspondance d'Objets Complexes (GCOC) :
il va être utilisé pour générer, à partir des déclarations des différentes correspondances que nous écrivons, les méthodes, objets … dont nous avons besoin pour formuler ces correspondances par l'intermédiaire de SGBD.

• Le Traducteur d'Appel de Correspondance d'Objets Complexes, ou TACOC:
il va être chargé d'"intercepter" les appels à l'outil de correspondance, et d'appeler les méthodes qui vont effectuer le traitement de correspondance.

L'architecture de l'intégration du prototype est donc celui de la figure V1.

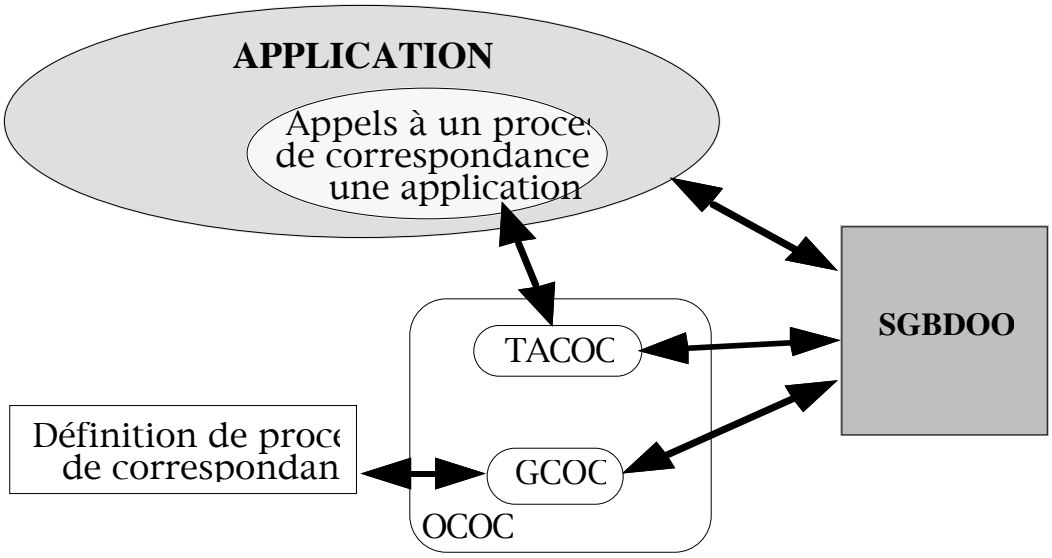


Figure V1: Architecture de la partie opérationnelle de notre démarche.

II.2. La description théorique dans le cadre BD

Resituons-nous dans le cadre propositionnel, c'est-à-dire que la requête et le document sont représentés par une formule logique d'ordre 0.
L'idée de base est de dire que si une proposition P de la requête n'est pas vérifiée dans un monde *mi*, elle peut être vérifiée dans un autre monde possible pour *mi*. Il faut donc bien définir l'univers des mondes W dans lequel nous nous situons, ainsi que les liaisons entre ces mondes.
Si nous définissons un monde comme étant un ensemble de propositions qui sont vraies à un certain stade. Si P est l'ensemble de toutes les propositions, alors W est un sous-ensemble de l'ensemble de tous les sous-ensembles de P, noté 2^P. Cet univers sera défini de la façon suivante :
• le monde *m0* défini à partir du document est élément de W,
• en posant l'utilisation d'un ensemble de connaissances K (exprimées sous la forme d'implications élémentaires), tout monde m' résultat de l'application des implications élémentaires de K à un monde m de W

$$m' \in W \text{ ssi } \exists m \in W \text{ tq } (\forall pci \in m', \exists pr \in m \text{ tq } (pr \Rightarrow pci) \in K \vee (pci \in m))$$

La fonction δ de changement de monde sera la suivante, dans le cas de correspondances strictes :

$\delta(m,m') = 1$ si $\forall$ pc $\in$ m' ($\exists$pr $\in$ m tq (pr => pc) $\in$ K) et m $\subset$ m'
$\delta(m,m') = 0$ sinon

Notre recherche sera donc en fait de savoir si un monde possible à partir de *m0* permet de rendre vraie la formule logique qui correspond à la requête.

Le système de bases de données sous-jacent utilisé n'étant pas capable de réaliser seul les vérifications d'implications logiques de la connaissance K, il faudra donc exprimer ces éléments explicitement, de même que les processus à réaliser pour les changements de mondes et pour le calcul de la valeur de vérité de la requête.

II.3. Description de la démarche

Dans un premier temps, il convient de bien définir les éléments utilisés pour réaliser de telles correspondances d'objets.

Avant d'utiliser le langage de définition mis à la disposition du programmeur d'application, celui-ci doit, comme nous l'avons décrit dans la partie précédente, vérifier un certain nombre d'éléments qui sont :

a) Choisir, parmi les deux objets comparés, lequel correspond à la *requête* et lequel correspond au *document*. Ce choix détermine donc quel est l'objet de référence (la requête), et quel est l'objet que l'on cherche à caractériser comme proche ou non de cette requête.
b) Exprimer sous forme logique les objets comparés,
c) Exprimer sous forme logique la façon d'étendre le document pour trouver si la requête est valide dans le monde représentant le document étendu, et en particulier fixer la part de la connaissance externe au processus lors des phases de Transition.

Une fois ces éléments obtenus, le programmeur se sert du GCOC pour exprimer les différentes phases de son processus de correspondance, pour cela, il doit faire

les liens entre la partie logique et la partie opérationnelle, à savoir :

d) Déterminer le lien entre les structures des objets comparés et leur sémantique,
e) Déterminer les supports des extensions du document et les éléments qui stockent les valeurs élémentaires de la requête,
f) Réaliser l'équivalent de l'implication logique au niveau des objets, pour tester les inclusions,
g) Exprimer de façon opérationnelle les transitions,
h) Choisir les conditions d'arrêt du Test.

Ces points sont réalisés dans la partie *1* de la figure V2.

Parallèlement, dans le point *2* de cette figure, le programmeur réalise son application, en considérant que le processus de correspondance est une *partie élémentaire* de son application. Pour nous, un appel à un processus de correspondance sera en fait, au niveau du programmeur, réalisé comme des appels à une fonction booléenne prédéfinie, appels que l'on peut réaliser aussi bien au niveau d'une expression (à gauche du symbole affectation), qu'au niveau de l'expression d'une requête dans le langage de requête du SGBD utilisé.
Ce dernier choix vient du fait que nous définissons en fait des prédicats (comme l'*i-égalité* d'ENCORE ou le *similar_to* de VAGUE). Ces prédicats ont toute leur utilité dans le langage de requête du SGBD, pour par exemple trouver tous les éléments d'un ensemble d'objets qui correspondent à un objet *référence*, en effectuant une sélection utilisant le prédicat de correspondance.

Le GCOC traduira chacun des appels à un processus de correspondance par l'application d'une méthode à un objet prédéfini. Ce choix peut bien entendu être différent, si les processus de traduction se situent à un niveau plus bas dans le SGBD, c'est-à-dire au niveau du noyau du SGBD.

Nous décrivons donc dans la figure suivante plus en détail les éléments de la figure V1 :

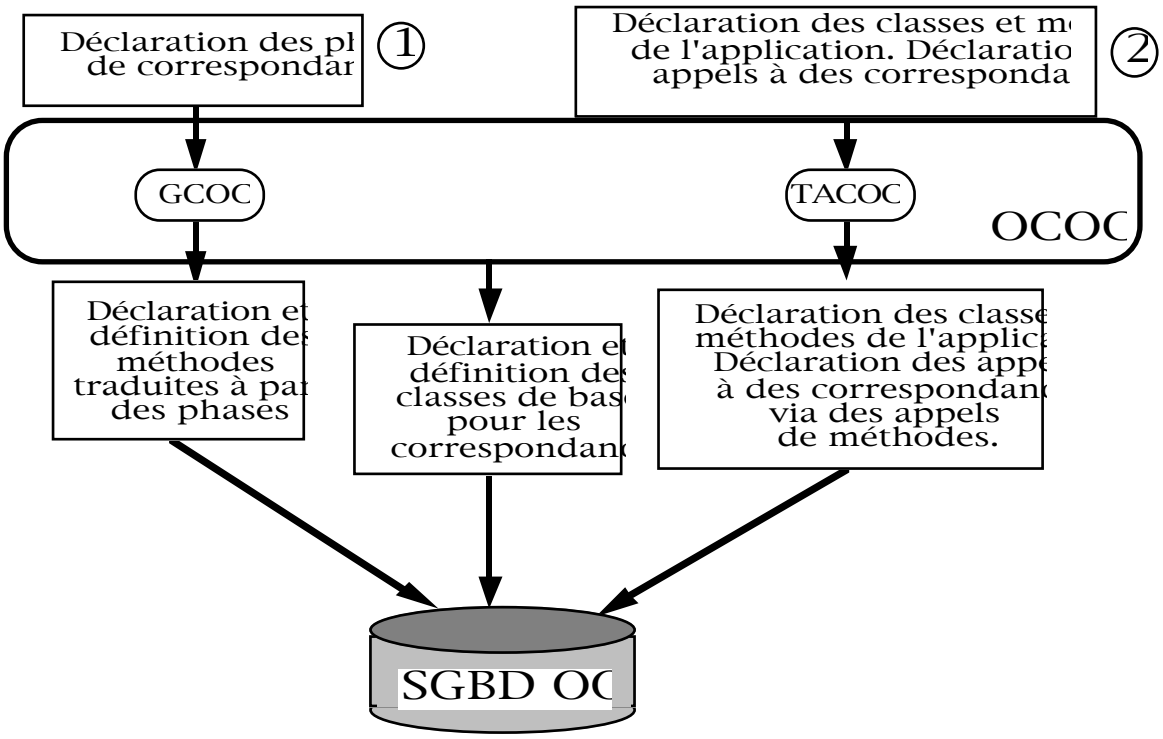


Figure V2 : Description de l'utilisation de l'outil.

En fait, comme nous le montrons par une flèche entre l'OCOC et le SGBD, il faut définir les classes et les objets qui vont être utilisés pour réaliser les correspondances au niveau du système. Ce seront ces classes et ces objets qui seront utilisés lors de l'appel des correspondances.

II.4. Utilisation des correspondances

En fait, on peut voir, dans le cadre des bases de données, deux possibilités d'utilisation de correspondances d'objets complexes, qui sont les suivantes :

• sans utiliser de fonctionnalités supplémentaires, on peut trouver par une requête habituelle un objet intéressant, et rechercher les objets d'un ensemble qui sont susceptibles de correspondre à cet objet,

• en utilisant une fonctionnalité supplémentaire, qui est un "générateur" d'objet requête basé sur une analyse de l'expression d'une requête qui crée un objet requête de référence. Ensuite, il ne reste plus qu'à comparer la requête générée à des objets susceptibles de lui correspondre.

Il faut ici bien souligner le fait que, ci-dessus, nous utilisons à dessein l'expression "susceptible de correspondre". En effet, il est nécessaire de faire

opérer les correspondances sur des ensembles relativement limités d'objets, car un processus de correspondance met en jeu un processus plus lourd que pour de simples requêtes strictes. Ceci veut dire que l'ensemble d'objets susceptibles de répondre doit être d'une cardinalité relativement faible. Dans le cas où nous sommes, ce sous-ensemble de la base est contraint par le type des objets, mais peut aussi se baser sur des caractéristiques des objets qui permettent de filtrer la base pour ne retenir que les objets qui ont une chance de répondre à la requête. Tout le problème consiste donc à formuler des conditions nécessaires à la correspondance des objets comparés.
Par exemple, dans le cas où on veut trouver les documents écrits par "DURANT" en 1980 qui correspondent par le prédicat "Doc_match" à un objet de référence, on commence par filtrer les objets candidats sur les prédicats stricts, c'est-à-dire par exemple “auteur = "DURANT" **et** année = 1980”, puis seulement on compare les objets retrouvés à l'objet de référence.

Dans le premier cas, c'est-à-dire celui où l'on obtient par une requête l'élément de référence qui nous intéresse, la démarche est décrite ci-dessous.
Une succession de requêtes permet de trouver un seul objet qui va servir de référence, et qui correspondra à O_Q. Si une requête est considérée comme une expression dans le langage de requête utilisé, on peut utiliser cette requête comme paramètre pour une correspondance. Si l'objet de référence écrit par "DURANT" en 1980 est titré "La pêche aux coques", et que l'on veut les documents de l'ensemble Documents écrit en 1980 par "DUPONT" qui correspondent par le prédicat Doc_match, la requête serait la suivante :

```
select d
from d in (select j from j in Documents
                where j->auteur = "DUPONT"
                   and j->année = 1980 )
where d Doc_match element( select k from k in Documents
                                      where k->auteur = "DURANT"
                                         and k->année = 1980
                                           and k->titre = "La pêche aux coques" )
```

Dans le cas d'une génération d'un objet requête, on se rapproche ici de ce qui est proposé dans le chapitre IV de [Lem93], avec les opérateurs *parse_by* et *reparse_by*. En fait, cette approche est d'une certaine manière plus limitée que la

précédente. En effet, dans le cas des documents, on ne peut pas demander à l'analyseur de créer un document complet, avec les chapitres, etc, le plus simple dans ce cas est donc la première solution. Par contre, dans un cas comme celui d'arbres sémantiques comme ceux de RIME, cette approche est intéressante. En effet, on ne cherche pas à retrouver un objet de référence qui est pré-existant, mais à générer un nouvel arbre qui correspond à la requête. Dans ce cas, une analyse de la langue naturelle permet de créer l'arbre de référence. Par exemple on peut rechercher les arborescences qui correspondent, via l'opérateur Cor_complex (cf. chapitre IV partie V3.4.5.), à "opacité sur le poumon due à une tumeur". Dans ce cas, on peut faire un premier filtre sur les feuilles des arbres, en recherchant ceux qui ont une intersection non-vide de feuilles, et ensuite faire la correspondance entre l'objet généré et les objets candidats. La recherche des feuilles utilise un parcours récursif des objets Nœud_sem, en filtrant ceux qui n'ont pas de fils, et en projetant sur le nom de ces objets. Nous appelons *analyse_arbre* le processus de création de l'arbre sémantique représentant la phrase.
L'expression de cette requête est donc donnée ci-dessous :

```
select i
from i in ( select j
                from j in Les_arbres
                 where (select k->nom
                            from k in j->desc(0,Ref,Noeud_Sem)
                             where k->fils_g =0 nil ) inclus { "opacité", "poumon",
                                                                "tumeur" } )
where i Cor_complex
                    analyse_arbre("opacité sur le poumon dues à une tumeur")
```

II.5. Conclusion

L'approche logique pour réaliser des correspondances d'objets a le mérite de différencier les éléments mis en œuvre lors de l'expression des correspondances.
L'approche "génération de correspondances" présente des avantages qui sont les suivants pour un programmeur d'application :

• tout d'abord, grâce à une mise en évidence de chacune des connaissances nécessaires à la réalisation de telles correspondances. En fait, on différencie clairement la connaissance interne de la connaissance externe, et les éléments qui sont à rattacher au document ou à la requête. De plus, on sait aussi à quel niveau ces connaissances sont appliquées.
• dans le cas sur lequel nous nous sommes penchés, les correspondances sont en fait des tests d'inclusion qui utilisent des éléments non limités à la structure des objets. Les inclusions traitées sont applicables à des objets ayant des structures arborescentes.
• l'utilisation de ce prototype dans le cadre orienté objet est simple dans la mesure où les formulations qui apparaissent dans la déclaration de phases sont peu différentes du langage de requête d'un tel système. Cet avantage serait perdu dans le cas d'utilisation dans un SGBD relationnel par exemple.

Une contrainte de cette approche vient du fait qu'il faut toujours trouver, ou bien générer, l'objet de référence qui va servir de requête avant de réaliser des correspondances. De plus, trouver les éléments qui contiennent les parties nécessaires à la correspondance n'est pas toujours simple à réaliser.

Le modèle de données des systèmes orienté objet permet la représentation de graphes, dont les arbres sont un cas particulier. Les éléments de base que nous avons proposés dans la partie précédente sont bien applicables au cas des arbres, mais pour proposer une démarche qui traite des graphes, une extension de nos opérateurs serait à analyser puis à effectuer.

III La Correspondance d'Objets dans les Applications RI

III.1. La position et les limites de notre travail dans un SRI

Nous commençons par préciser sur le schéma de la figure V3 où se situent nos propositions par rapport au schéma général d'un SRI qui a été donné dans le chapitre IV.

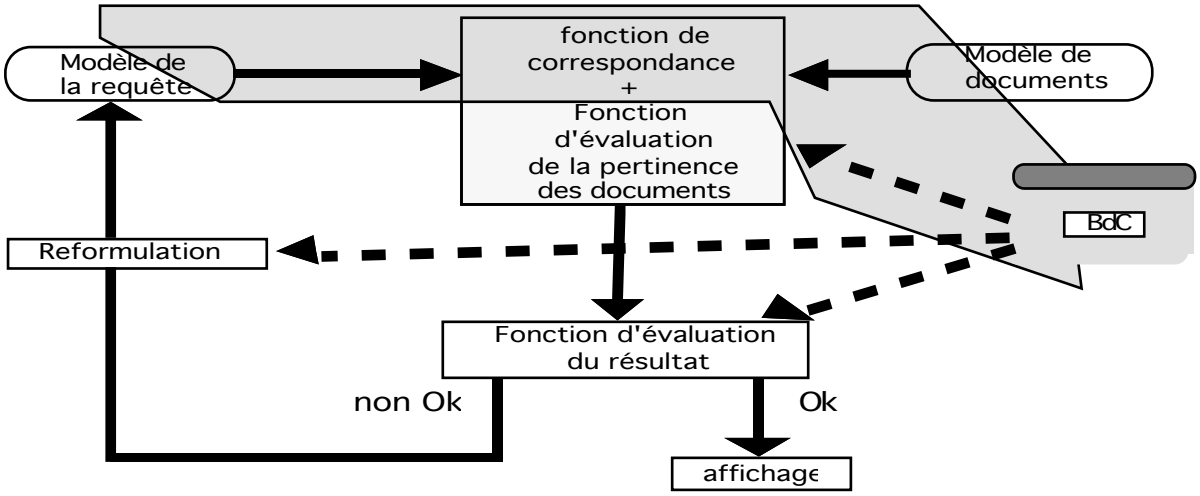


Figure V3 : Les éléments visés par notre travail.

Grâce à ce schéma, on voit que le travail effectué dans cette thèse se situe principalement au niveau de la fonction de correspondance stricte. Cependant, on remarque que nous n'avons exprimé que la comparaison de deux termes d'indexation, et non pas un document et une requête totale, c'est pourquoi sur le schéma ci-dessus on n'englobe qu'une partie du modèle du document et de la requête. Des deux limitations que nous venons d'exprimer, on se rend donc compte que pour définir un SRI complet, il faut :
1) intégrer les modèles des requêtes et des documents en utilisant notre processus,
2) étendre l'approche suivie pour intégrer des valuations, en vue de fournir à l'utilisateur un ordonnancement des réponses.

III.2. Application à des requêtes booléennes

III.2.1. Description générale

Nous allons ici nous intéresser à une représentation **classique en recherche d'informations** : une requête booléenne qui contient des éléments terminaux traités uniformément, et un ensemble d'objets *comparables* aux terminaux de la requête par un processus de correspondance. La requête est donc une requête booléenne, et le document a son contenu sémantique représenté par un ensemble de termes d'indexation.

Considérons que l'on cherche à savoir si un document répond à une requête.

Prenons une requête exprimant le fait que l'on cherche à trouver les éléments d'un ensemble E_Docs qui parlent de "l'agriculture en France ou de la vigne en Bretagne". Les éléments de E_Docs sont en fait des ensembles de termes, dont nous supposons, comme dans le cadre des bases de données ci-dessus, qu'il est en fait le résultat d'une présélection d'objets candidats. Supposons que nous avons une correspondance qui opère sur les éléments des objets de E_Docs appelé *correspond_à*, la requête posée ressemblerait à (en utilisant une syntaxe O_2SQL) celle de la figure V4.

```
select O
from O in E_Docs
where (exists( i in O : i correspond_à TIagri)
       and exists( i in O : i correspond_à TIfrance))
     or (exists( i in O : i correspond_à TIvigne)
        and exists( i in O : i correspond_à TIbreta))
```

Figure V4: Un exemple de requête.

Dans cette requête, nous considérons que les objets TIagri, …, sont des objets-termes prédéfinis.
En fait, l'expression même de cette requête présuppose une connaissance sur la sémantique du document qui est la suivante : un ensemble de termes a une représentation logique qui est la conjonction de ces éléments. L'utilisateur connaît la sémantique des éléments de la requête, ainsi que les éléments qu'il recherche, ce qui lui permet de traduire sa demande sous la forme d'une expression du langage de requête.

En fait, on se rend compte qu'une telle expression possède deux désavantages qui sont : i) celui d'"éclater" les éléments de la requête (les TI...), et de ne les relier que par la requête (un peu comme des jointures relationnelles), alors que l'on peut sans difficulté exprimer cette structure par l'intermédiaire d'un objet requête sous la forme d'un arbre, ii) celui de ne pas situer clairement les différentes connaissances utilisées pour réaliser cette requête car tous les éléments sont regroupés sous la forme d'une seule expression booléenne.

Pour répondre à ces problèmes, on peut utiliser le processus de correspondance. Pour cela, il est donc nécessaire de représenter la requête par un objet O_Q. O_Q ne représente pas la requête globale de la figure V4, mais seulement la partie `where` de cette requête, et la requête au niveau du SGBD serait :

```
select O
from O in E_Docs
where ( O proche_de OQ )
```

En effet, un processus de correspondance ne compare que des couples d'objets. O_Q doit être en fait une sorte "d'idéal" dont la formule associée est “TIagri Ê TIfrance ª TIvigne Ê TIbreta”. Une représentation simple d'une telle formule peut être un arbre binaire (figure V5).

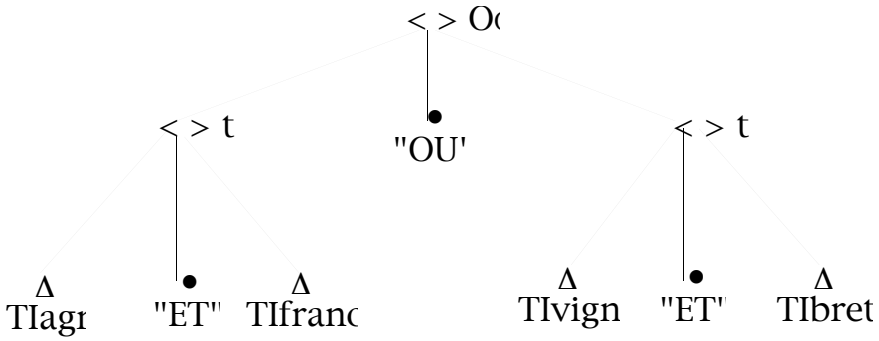


Figure V5 : L'expression arborescente de la requête.

Dans la figure ci-dessus, les notations “Δ” représentent les objets qui sont "de base" pour la requête, c'est-à-dire des éléments dont on n'a à connaître que les identificateurs; en effet, c'est un autre processus de correspondance qui les gère. Nous avons choisi de représenter les opérateurs booléens de l'objet requête **et** et **ou** par les chaînes de caractères "ET" et "OU", ces chaînes seront bien sûr interprétées au moment de l'évaluation de la requête.

L'objet O_D représentant le document est un ensemble d'objets de base. Nous représentons un tel objet qui contient 4 termes d'indexation par la figure V6. On ne se préoccupe que des identificateurs des éléments de bases, nommés *t1*, *t2*, *t3* et *t4*.

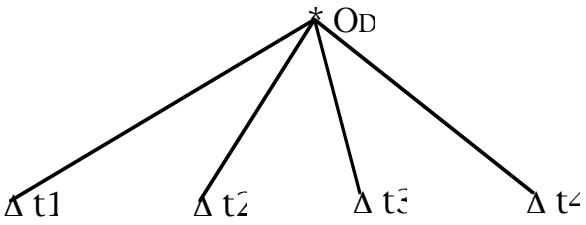


Figure V6 : Un document composé de 4 termes.

Au niveau où nous nous situons, la formule logique F_D qui correspond au document est : "t1 Ê t2 Ê t3 Ê t4".

III.2.2. Le processus logique

En fait, pour réaliser effectivement le test sur la validité de la formule F_Q dans le modèle de F_D, nous allons suivre une démarche, identique au chapitre précédent, qui consiste à étendre le document par "TIagri" si cette proposition correspond à l'une du document, sinon on étend le document par "°TIagri", etc. On crée ainsi le monde *m0*. On étendra donc la formule du document par F_{ext}^0 définie par :

$$F_{ext}^0 = \bigwedge_{p \in F_{K:0}(E_Q,E_D)} p \quad \bigwedge_{p \in G_{K:0}(E_Q,E_D)} \neg p$$

Dans cette expression, E_D et E_Q représentent les ensembles de propositions des formules F_D et F_Q, respectivement.
Les fonctions $F_{K:0}$ et $G_{K:0}$ sont les suivantes:

$$F_{K:0} : \{\text{proposition}\} \times \{\text{proposition}\} \rightarrow \{\text{propositions}\}$$
$$F_{K:0}(E_Q,E_D) = \{ p \mid p \in E_Q \wedge p \textit{ correspond\_à } q \wedge q \in E_D \}$$

$$G_{K:0} : \{\text{proposition}\} \times \{\text{proposition}\} \rightarrow \{\text{propositions}\}$$
$$F_{K:0}(E_Q,E_Q) = \{ p \mid p \in E_Q \wedge \neg(p \textit{ correspond\_à } q \wedge q \in E_D) \}$$

En fait, un seul monde va être généré, *m0*, car nous n'avons pas d'autre connaissance externe à notre disposition que cette notion de *correspondance*, et selon le résultat de l'évaluation de la requête dans ce monde, on saura si le document répond ou non à la requête.

La différence la plus importante par rapport à ce que nous avons décrit à la fin du chapitre IV, est que les opérateurs de la requête sont interprétés pour évaluer l'implication.
En effet, dans le chapitre IV, la structure des objets sous-entendait une conjonction des éléments qui composaient les objets, et ici ce n'est plus le cas. Si le document ne contient qu'un terme qui correspond à l'objet qui représente

“TIagri”, le document est inclus dans la requête, mais ce document ne répond pas à celle-ci car l'interprétation du "ET" logique nous dit que l'objet représentant “TIfrance" doit aussi être dans le document pour qu'il soit pertinent.
Cette approche est donc plus générale qu'une simple inclusion.

Pour en revenir aux phases du processus que nous avons expliquées dans ce document, les extensions du document se feront dans la phase de Génération, puisqu'aucune Transition ne sera faite.
Ensuite, l'évaluation de la requête se fera par l'expression d'une Comparaison récursive qui exprimera explicitement la façon d'évaluer les opérateurs de la requête. Si cette évaluation rend Vrai, alors l'objet O_D correspondra à l'objet O_Q, sinon il ne correspondra pas.

III.2.3. Les applications dans cette direction

L'intérêt de ce que nous venons de décrire ne réside pas dans l'unique fait de réaliser ce que propose le langage de requête d'un SGBD. On constate que l'approche que nous avons suivie s'applique dans d'autres cas, et qu'elle permet de mieux situer les différents éléments qui entrent en jeu dans une correspondance.

De plus, dans les systèmes de recherche d'informations comme RIME, les opérateurs booléens sont intégrés à la requête, comme des relations sémantiques, et donc le traitement de ces opérateurs est à prendre en compte.
Enfin, le fait de traiter les opérateurs booléens comme les autres éléments de requêtes va permettre, avec l'utilisation de correspondances valuées, de définir comment traiter des conjonctions et disjonctions valuées, ce que les langages de requêtes des SGBDOO ne font pas actuellement.

Il faut cependant remarquer une limitation importante de ce que nous proposons, et qui est qu'elle nécessite un découpage précis des parties "correspondances de termes" des parties "opérateurs de requêtes". Nous allons revenir dans la partie III.4. de ce chapitre sur le cas de RIME, dont nous allons représenter la partie évaluation de requête basée sur des correspondances non-strictes.

III.3. Les correspondances valuées basées sur des propositions

Nous avons choisi ici de rester dans le même cadre que l'exemple simple de la partie III.2.2. du chapitre IV, et donc de nous limiter à une utilisation de la logique des prédicats pour cette première analyse.

III.3.1. Situation du problème

Le processus de correspondance doit prendre en considération des correspondances valuées, c'est à dire indiquer *dans quelle mesure* un document répond à une requête, ou plus généralement dans quelle mesure deux objets peuvent être considérés comme similaires.
Par l'utilisation de correspondances valuées, on se rapproche beaucoup plus de l'esprit des formulations données dans la partie II.1 de ce chapitre, sur *l'idée de base du modèle logique*, dont nous rappelons ici le second principe d'incertitude:

"Etant donnés deux ensembles d'informations x et y, la mesure d'incertitude de x→y relative à un certain ensemble K de connaissances donné est déterminé par l'extension nécessaire de x en x' et la certitude de la vérification de x'→y".

Dans le cas où nous attendons des correspondances valuées, l'idée de *l'extension nécessaire* est beaucoup plus cruciale que dans les cas non-valués.
En fait, la réalisation de correspondances valuées peut être séparée en deux cas :
- soit les connaissances utilisées pour étendre le document sont valuées,
- soit ces connaissances ne sont pas valuées.

Si l'on se remémore la façon dont se déroule un processus de correspondance, on se rappelle que pour générer un monde on applique TOUTE la connaissance possible en étendant le document une seule fois, sans réappliquer la connaissance au résultat que l'on obtient.
Dans le cas de correspondances strictes, cette démarche ne pose pas de problème particulier, car on génère en fait des extensions qui peuvent être suffisantes, et non pas nécessaires comme le préconise C.J. Van Rijsbergen, mais qui dans ce cas permettent de conclure sur l'implication.
Dans le cas de correspondances valuées, la première question à se poser est de savoir si notre démarche permet de déterminer, et de valuer, les extensions nécessaires à la validation de la requête.
Pour valuer ces correspondances, nous allons utiliser la logique modale floue.

III.3.2. La logique modale floue des propositions

Nous nous limitons ici à décrire les éléments de cette logique modale qui nous intéressent, en rappelant ce qui a été dit dans le chapitre IV à ce sujet. Nous nous basons ici très fortement sur [Che92].

Une logique modale floue est définie par une syntaxe décrivant l'ensemble des formules F sur un ensemble de propositions atomiques P et un système d'interprétation $S = (W,\delta,\Delta,C,V)$. Les éléments du système S ont le sens suivant:

- W est l'ensemble des mondes possibles,
- δ est une fonction de $W \times W$ dans [0:1], elle indique la certitude de passage d'un monde à un autre,
- Δ est une fonction de $[0;1]\times[0;1]$ dans [0;1]. Elle est utilisée pour le calcul de V en combinant la certitude de passage d'un monde à l'autre avec la valeur de certitude d'une formule V,
- C est une fonction de $P \times W$ dans [0;1]. Elle associe une valeur de validité d'une proposition atomique par rapport à un monde,
- V est une fonction de $F \times W$ dans [0;1]. Cette formule donne la valeur de validité d'une formule par rapport à un monde, en utilisant les mondes possibles d'un monde et les valeurs de certitudes de cette formule dans les mondes possibles.

III.3.3. Utilisation de connaissances valuées

III.3.3.1. Valuation sur des propositions

Dans ce cas, le document est une conjonction de propositions, et la requête est une conjonction et/ou disjonction de propositions. L'exemple de la partie IV.2.2 est un cas particulier de telles expressions.

Si nous revenons brièvement sur ce que nous avons expliqué dans le chapitre IV, ce que nous cherchons à évaluer est : $V_{m0}(F_Q)$, avec le monde *m0* déterminé par le document D (dont la formule correspondante est F_D).

Si F_Q est de la forme $F_1 \wedge F_2$, alors

$$V_{mi}(F_Q) = MIN(V_{mi}(F_1), V_{mi}(F_2)) \quad \forall mi \in W$$

Si F_Q est de la forme $F_1 \vee F_2$, alors

$$V_{mi}(F_Q) = MAX(V_{mi}(F_1), V_{mi}(F_2)) \quad \forall mi \in W$$

W est basé sur la même approche que dans la partie II de ce chapitre.

Comme nous nous situons dans le cadre de la logique propositionnelle, cette décomposition de F_Q nous amène à rechercher les $V_{m0}(P_q)$, pour tous les P_q de E_Q (l'ensemble des propositions de F_Q). Pour calculer ces valeurs, il va falloir utiliser les valeurs des propositions du document, c'est-à-dire les $V_{m0}(P_d)$.

Cette valeur dépend de $C_{Pd}(m0)$, qui donne la validité d'une proposition atomique P dans le monde *m0*, et de $V_{m0}(\Diamond Pd)$, qui donne la valeur de certitude de Pq dans les mondes possibles à partir de *m0*.
Cette formule peut être la suivante :

$$V_{mi}(Pd) = MAX (C_{Pd}(mi), V_{mi}(\Diamond Pd)), \quad \forall mi \in W$$

avec $$V_{mi}(\Diamond Pd) = MAX (\Delta(\delta(mi, mj), V_{mj}(Pd))), \text{ avec } mj \in W, \forall mi \in W$$

Cette formule est donc valable pour le monde m0.

La valeur V_{m0}(Pqd dépend donc à la fois de la valeur de passage entre deux mondes notée δ, et de la valeur $C_{Pd}(m0)$. Si la fonction qui permet de manipuler les valeurs de vérité dans un monde avec les changements de monde appelée Δ doit posséder les caractéristiques suivantes, où c1, c2 et c sont des valeurs de certitudes quelconques de [0,1] :

$\Delta(c1,c2) \leq c1$, $\Delta(c1,c2) \leq c2$,

$c1 > 0$ et $c2 > 0 \Rightarrow \Delta(c1,c2) > 0$ (propriété d'idempotence)

La multiplication (entre autres) vérifie ces propriétés. Cette fonction associe la même importance à ces deux opérandes, et ce choix nous semble a priori intéressant. Dans ce cas, $V_{m0}(\Diamond Pd)$ est la fonction suivante :

$$V_{mi}(\Diamond Pd) = MAX (\delta(mi, mj) * V_{mj}(Pd)), \text{ avec } mi \in W, \forall mi \in W$$

De plus, suivant une approche similaire à celle des correspondances strictes, nous allons utiliser des générations linéaires de monde, ce qui veut dire qu'il n'existe qu'un seul monde mi+1 possible à partir d'un monde mi (c'est-à-dire un seul monde mi+1 tel que $\delta(mi,mi+1) \neq 0$), ceci nous donne la formule suivante :

$$V_{mi}(\Diamond Pd) = \delta(mi, mi+1) * V_{mi+1}(Pd) \text{ , avec } mi+1 \in W, \forall mi \in W$$

En fait, la valuation entre deux mondes n'est pas utilisable dans le cas où l'on cherche à évaluer l'extension nécessaire du document, tout simplement car cette extension est réalisée en bloc, c'est-à-dire en étendant le document au maximum, sans se préoccuper si cette extension est nécessaire ou non. Nous choisissons donc de ne pas valuer dans ce cas les passages inter-mondes, où plutôt de tous les valuer à 1, pour qu'ils n'influent pas sur les valeurs des propositions de F_Q. La formule $V_{mi}(\Diamond Pd)$ est donc la suivante :

$$V_{mi}(\Diamond Pd) = V_{mi+1}(Pd) \text{ , avec } mi+1 \in W, \forall mi \in W$$

Ce qui nous intéresse en fait, c'est de générer des mondes mi qui nous permettent, en se limitant à ces mondes (comme le cas des correspondances strictes), de donner la valeur $V_{mi}(F_Q)$ en utilisant les valeurs élémentaires des propositions de cette formule F_Q.
En fait, cette valeur va être calculée de manière incrémentale, ce qui veut dire que, quand on est arrivé à un monde mi, on va considérer qu'il n'existe aucun monde mi+1 possible. Ceci va nous amener à calculer des maxima locaux, en sachant que le maximum global est atteint lorsqu'on a passé en revue tous les mondes possibles. Nous allons noter ces valeurs locales du monde m0 juqu'au monde mi $V_{m0}^{mi}(Pd)$. Suivant les expressions ci-dessus, pour une proposition P_d on a la formule **(A)**:

$$V_{m0}^{mi}(Pd) = MAX(C_{Pd}(m0), MAX(C_{Pd}(m1), ...MAX(C_{Pd}(mi-1), C_{Pd}(mi))\ldots)$$

Nous cherchons à évaluer cette valeur en ne se basant que sur des éléments de mi. Les seuls éléments sur lesquels nous pouvons jouer sont les $C_{Pd}(mj)$,avec $0 \leq j \leq i$.
On peut considérer que les propositions du document sont certaines, ce qui s'exprime par :

$$\forall P_r \in E_D, C_{Pr}(m0) = 1$$

avec E_D l'ensemble des propositions de F_D.

On pose que toutes les propositions P de F_Q qui ne sont pas dans E_D ont par défaut leur valeur $C_P(m0)$ à 0.

Rappelons que nous utilisons une connaissance valuée notée K, qui va nous servir à étendre le document. On peut la représenter par un ensemble d'implications "$p_k =>_v p_m$", $0 \leq v \leq 1$ représentant la force de l'implication entre une proposition p_k et une proposition p_j. On peut considérer que la valeur $C_{Pd}(mj)$ dépend donc des implications possibles "$x_m =>_{vm} P_d$" de K, c'est-à-dire des $C_{xm}(mj-1)$ et des vm (il peut bien entendu exister plusieurs xm dans ce cas). Notre but étant de garder les meilleurs résultats au fur et à mesure, $C_{Pd}(mj)$ dépendra aussi de $C_{Pd}(mj-1)$. On obtient donc la formulation suivante:

$$C_{Pd}(mj) = MAX(MAX(f^t(C_{xm}(mj-1), vm), C_{Pd}(mj-1))$$

avec les xm et vm tq "$x_m =>_{vm} P_d$" â K, et f^t étant une fonction à déterminer.

Dans ce cas, on a pour un i quelconque :

$$C_{Pd}(m0) \leq C_{Pd}(m1) \leq \ldots C_{Pd}(mi-1) \leq C_{Pd}(mi)$$

On en déduit une expression de la formule **(A)**, notée **(A')**, qui est la suivante :

$$V_{m0}^{mi}(Pd) = C_{Pd}(mi)$$

Une fois ces choix faits, on peut savoir si F_D a une valeur de validité nulle ou non en calculant l'implication d'après les valeurs de validité du monde m0.

Revenons sur la fonction f^t qui permet de déduire une valeur d'une proposition Pd d'après la force vm de l'implication d'une proposition xm exprimée par la connaissance K. Cette fonction n'est pas définie a priori, mais nous considérons qu'elle doit vérifier certaines contraintes qui sont :

- $f^t(C_{xm}(mj-1), 1) = C_{xm}(mj-1)$, ce qui veut dire que si l'implication est certaine, on garde la même valeur de validité,
- $f^t(C_{xm}(mj-1), 0) = 0$, ce qui veut dire que si l'implication est nulle, la valeur de Pd est 0.

L'opérateur de multiplication "*" sur les réels vérifie ces conditions, ce qui permet d'exprimer $C_{Pd}(mj)$ par

$$MAX(MAX(C_{xm}(mj-1) * vm), C_{Pd}(mj-1))$$

avec les xm et vm tq "$x_m =>_{vm} P_d$" $\in$ K.

Dans un monde mi, on a donc, par la valeur $C_{Pd}(mi)$, le maximum d'extensions que l'on a pu tirer entre le monde m0 et mi de l'application des connaissances fixées. Le calcul de F_Q dans ce monde nous donne bien une valeur des extensions nécessaires au document pour répondre à la requête, en utilisant les $C_{Pq}(mi)$ pour calculer $V_{mi}(F_Q)$.

Si nous utilisons cette description sur l'exemple simple inspiré de celui de la partie V.1 du chapitre IV, en considérant que "france" est un générique de "bretagne" à 0,8, (c'est à dire "france $=>_{0,8}$ bretagne") et que "europe" est un générique de "france" à 0,7 (i.e. "europe $=>_{0,7}$ france"), et en posant que l'utilisateur cherche un document sur "l'Europe et la Bretagne" et que nous comparons sa requête à un document traitant de la Bretagne.

$C_{bretagne}(m0) = 1$
$C_{europe}(m0) = 0$

Dans le monde m0, V_{m0}(bretagne ∧ europe) est donc égal à 0, ce qui indique que, pour l'instant, le document ne correspond pas à la requête.
Après une première extension, générant le monde m1, utilisant le fait que france est un générique de bretagne avec une valeur de 0,8, on obtient :

$C_{bretagne}(m1) = 1$
$C_{europe}(m1) = 0$
$C_{france}(m1) = 0,8$

Dans ce monde V_{m1}(bretagne ∧ europe) est toujous égal à 0.
Et après la génération du monde m2, on obtient :

$C_{bretagne}(m2) = 1$
$C_{europe}(m2) = 0,56$
$C_{france}(m2) = 0,8$

Si on s'arrête après trois extensions, on trouve que V_{m2}(bretagne ∧ europe) est égal à 0,56 (qui est le minimum de 1 et de 0,56), et donc le document répond avec une valeur de 0,56 (=0,8*0,7) à la requête.

Les éléments qui permettront de gérer cette nouvelle fonction seront à intégrer à

différentes places de notre processus :

• Dans la phase de Génération, pour le document on devra initialiser les structures qui stockeront les valeurs de validité des éléments et des extensions du document.
• Dans la phase de Comparaison, nous allons calculer la valeur de validité de la requête globale, et il faudra donc intégrer le fait que cette phase renverra un résultat valué et non plus booléen.
• La phase de Transition mettra à jour les valeurs de validité des différents éléments du document.
• L'étape de Test, quant à elle, devra également être en mesure de manipuler des valeurs numériques, pour pouvoir fixer des seuils de correspondance. En effet, une notion de seuil pour la valeur du résultat de la phase de comparaison pourra être utilisée.

Le processus de correspondance valué n'est plus alors considéré comme un prédicat, mais plutôt comme une fonction qui renvoie une valeur numérique.

III.3.3.2. Valuation sur des prédicats

La démarche, dans le cas d'utilisation de prédicats, est la même que dans le cas basé sur les propositions, avec cependant une difficulté supplémentaire qui vient bien entendu de l'emploi des variables. La différence sera en fait à traiter au niveau de l'évaluation de la formule de la requête, comme nous allons le voir.

Reprenons pour étayer notre affirmation un exemple proche de RIME, en utilisant des extensions valuées, avec "r_rel(x,y,poumon)" (p_1) un prédicat du document, et "r_rel(b,c,poumon)" (p2) un prédicat de la requête.
Si nous considérons que $C_{p1}(m0)$ est égal à 1, il en découle que lors de l'évaluation de la requête $C_{p2}(m0)$ devra être 1. Le problème supplémentaire vient donc du fait qu'il faut tenir compte des variables.

Au niveau opérationnel, il y a peu de différences entre les propositions et les prédicats, il faudra exprimer l'"unification" entre les prédicats.

III.3.4. Utilisation de connaissances non valuées

Un exemple de ce cas est en fait RIME tel que nous l'avons décrit dans le chapitre IV. Dans ce cas, il semble impossible par notre approche d'intégrer des valuations qui fournissent une valeur exprimant la connaissance suffisante qui a permis de rendre l'implication vraie. En effet, nous appliquons lors du passage d'un monde à un autre toute la connaissance possible, on pourra juste, en évaluant la requête, savoir si l'extension faite est suffisante ou non, sans déterminer si elle est nécessaire ou non. Comme nous le montrons maintenant, la seule technique utilisable se rapproche plus d'un comptage qui évalue une approximation de la correspondance, cette approximation n'ayant pas de "sens autonome", mais pouvant servir uniquement à comparer le résultat de plusieurs comparaisons entre une requête et plusieurs documents, en fournissant une base pour un ordonnancement des réponses.
Pour intégrer une valuation, en nous ramenant à ce que nous venons de décrire dans le cas de connaissances valuées, une solution simple est de poser, quand on passe d'un monde mi à un monde mi+1, que toutes les implications "$p_k \Rightarrow p_j$" sont en fait des "$p_k \Rightarrow_{vi} p_j$". Cette solution à le mérite d'être simple. Si nous posons qu'en fait tous les v_i ont même valeur v, et en se basant sur les formules vues dans les deux points précédents, le document correspondra à la requête avec une valuation comprise dans l'ensemble $\{1, v, v^2, \ldots, v^n, 0\}$, si on a des mondes de m0 à mn.
Si un document D_1 correspond à la requête au bout de deux extensions, il possèdera une valeur de v^2, alors qu'un document D_2 qui correspond au bout d'un monde à une valeur de v. Le problème vient du fait que ces deux valeurs ne veulent pas dire que D_1 correspond beaucoup moins à la requête que D_2, il faut donc se limiter ici à interpréter la relation d'ordre sur les réels entre 0 et 1, en disant que D_1 correspond moins à la requête que D_2 selon nos critères.

IV. Conclusion

Nous avons défini dans cette partie comment nous situons notre approche au cours de la génération d'une application bases de données. Nous avons aussi montré les limites de ce qui a été fait pour être réellement utilisable pour la création de systèmes de recherches d'informations (ou d'application ayant des composantes RI).

De cette étude, nous déduisons les remarques suivantes :
• Le fait de pouvoir traiter des requêtes booléennes par des comparaisons d'objets nous a montré que l'outil et la démarche que nous proposons peut s'appliquer à d'autres cas que le contexte initial,
• l'introduction de correspondances valuées ne remet pas fondamentalement en cause les éléments que nous proposons.
Nous avons expliqué dans ce chapitre comment intégrer des connaissances valuées pour offrir un résultat pondéré, et comment fournir un ordre sur des correspondances en utilisant des connaissances non pondérées. Dans ce cas, nous utilisons la logique modale floue.

Cependant, notre approche qui consiste à faire des générations linéaires de mondes, avec un passage d'un monde à un autre déterminé par toutes les extensions que l'on peut réaliser à un moment donné pose des problèmes dans l'utilisation de connaissances non valuées, dans la mesurer où on n'est plus capable de discriminer ce qui est suffisant de ce qui est nécessaire pour trouver la requête vraie.

La démarche que nous proposons permet d'exprimer des correspondances sans avoir besoin de toutes les fonctionnalités de déductions habituelles d'un SGBD déductifs. Ceci nous permet, sans avoir à créer un système déductif, de réaliser des fonctionnalités simples suffisantes dans notre cadre. De plus, grâce à l'intégration de valuations, on va permettre réellement la réalisation d'applications Recherche d'Informations, ce que les systèmes déductifs stricts ne proposent pas.

CHAPITRE VI

Expérimentation

CHAPITRE VI

Expérimentation

I. Introduction

Nous allons décrire dans ce chapitre l'implantation de l'Opérateur de Correspondance d'Objets Complexes (OCOC), tel qu'il a été défini au chapitre V. Cette implémentation est faite en ayant comme cible le système O_2 [Lec88, Lec89a, Lec89b, Deu91], dans sa version 3.3.1.

Ce que nous avons défini au chapitre IV, au niveau opérationnel, est en fait **un langage** qui permet de décrire des correspondances. Nous avons donc programmé un compilateur, dont le code source est la description des différentes phases et étapes d'une correspondance, et dont le code cible est en fait une union des langages de manipulation de schéma et de programmation d'O_2, car nous déclarerons des méthodes et nous définirons les corps de ces méthodes.

Le résultat de l'OCOC sera donc une partie de l'application que réalise le programmeur. Nous avons choisi une approche générateur similaire à [Mar91]. Ce choix est une solution qui nous permet d'implémenter des correspondances sans descendre au niveau du noyau du système, ce qui n'était pas possible. En particulier, la version utilisée ne permet pas la manipulation du schéma à l'intérieur d'une application, ce qui nous est nécessaire. La démarche suivie est donc la suivante :

• niveau du GCOC[4] : une fois que le programmeur aura écrit une correspondance, il la *précompilera*, à l'extérieur d'O_2, de manière à obtenir un appel aux éléments qui réalisent effectivement l'exécution de la correspondance. Il ne restera plus

[4] Générateur de Correspondance d'Objets Complexes

qu'à compiler les éléments générés dans O_2,
• niveau du TACOC[5] : un appel à une correspondance sera lui aussi traduit en un appel de méthode sur un objet prévu à cet effet.

Cette démarche nous oblige à bien différencier ce qui fait partie du langage de OCOC de ce que le système O_2 propose. Pour les éléments qui sont similaires, il y a très peu de chose à faire lors de la compilation, mais pour les éléments qui n'existent pas dans O_2, nous allons devoir les "émuler".
Pour cela, dans la partie II de ce chapitre, nous allons décrire les caractéristiques d'O_2, pour bien situer ce qu'il propose. Dans la partie suivante, nous donnerons des exemples de traductions faites pour générer les correspondances. La partie IV expliquera comment s'effectue l'appel à une correspondance. Dans la partie V, nous ferons un court récapitulatif de l'ordre à suivre pour réaliser des générations et des appels à des processus de correspondance. Nous conclurons en distinguant les éléments que nous considérons comme devant "descendre" au niveau du noyau d'un SGBDOO des éléments qui ne constituent pas une extension nécessaire *a priori* pour ces systèmes.

II. Les caractéristiques du système O_2

O_2 est un système qui permet de définir des schémas de bases de données orientés objet, ainsi que les méthodes et les applications qui vont utiliser des objets et des valeurs de ces schémas.
Nous allons examiner les caractéristiques d'O_2 que nous allons utiliser :

• La notion d'identificateur d'objet est supportée par le système. On peut déclarer des objets nommés ou des valeurs nommées qui sont des racines de persistance.
• Le langage de programmation avec lequel sont écrits les corps des méthodes et des programmes d'applications est l'O_2C, qui est une extension du C pour intégrer les notions spécifique au contexte objet.
• Dans O_2, la notion ensembliste de classe décrivant un ensemble d'instances d'un même type n'existe pas, en fait une classe dans la terminologie O_2 définit un *type d'objets*, alors qu'un type O_2 définit un *type de valeurs*. O_2 ne propose que les liens de référence simple entre objets (cf. chapitre II).

[5] Traducteur d'Appel de Correspondance d'Objets Complexes

• Le langage de requête d'O_2 donne comme résultat des valeurs qui sont des ensembles ou des listes, ces valeurs contenant soit des références à des objets, soit des valeurs.

Nous allons dans cette partie nous intéresser à chacun des points ci-dessus les uns après les autres, en nous basant sur le manuel utilisateur [O2U92].

II.1. Objets, Valeurs et Objets nommés

Un objet O_2 est un couple (identificateur, valeur). L'identificateur de l'objet est indépendant de la valeur qu'il contient. Soit une variable correspondant à un objet d'une classe (la classe `Document`, par exemple) de nom `nom` qui à été déclarée. La correspondance entre la variable `nom` et un objet est effectuée explicitement par l'instruction suivante :

```
nom = new Document;
```

Tout objet créé à l'intérieur du corps d'une méthode est par défaut non-persistant.

La notion de valeur en tant qu'entité existe en O_2. Une valeur, du fait qu'elle ne possède pas d'identificateur est en fait non partageable. Une valeur peut être utilisée telle quelle, mais peut aussi composer des objets ou d'autres valeurs. Une valeur est par défaut non persistante, mais elle persiste si elle compose un élément persistant. On atteint par la notation pointée un attribut *A* d'une valeur n-uplet *v* : “v.A”.

La gestion de la persistance dans O_2 se fait au travers de l'introduction de racines de persistances, des objets ou des valeurs, qui sont persistants.
Dans la terminologie O_2, ces racines de persistances sont des objets ou des valeurs nommées. Nous définissons dans l'exemple suivant un objet nommé qui est un Document, et une valeur nommée qui est un ensemble de documents:

```
name Doc : Document;
name Les_Docs : set(Document);
```

La règle de persistance est la suivante : tout objet (ou toute valeur) référencé par

une valeur ou un objet persistant est persistant à son tour, et tout objet (ou toute valeur) non référencé par une valeur ou un objet persistant est non persistant.

II.2. Méthodes, Programmes et Applications

Une méthode est une fonction ou une procédure applicable aux objets d'un certain type (une classe selon la terminologie O_2), et qui décrit une partie du comportement des objets.
Le principe d'encapsulation est respecté, car par défaut la structure des objets est privée, donc les initialisations, modifications et lectures de la valeur d'un objet ne sont réalisables qu'en utilisant les méthodes associées à cet objet. Il est possible de déclarer publique toute la structure (ou simplement une partie) d'une classe, ce qui rompt l'encapsulation.

Une méthode peut être déclarée au moment de la création d'une classe (cf. partie suivante), mais peut aussi déclarée de la manière suivante:

```
method [public|private] <nom de méthode>
            [(<liste de parametres avec leurs types>)]
            [: <type ou nom de classe>] in class <nom de classe>;
```

Une fois qu'une telle méthode est déclarée, on écrit le corps de cette méthode de la manière suivante:

```
method body <nom de méthode>[<liste de paramètres avec leurs types>]
                    [: <type ou nom de classe>] in class <nom de classe>
{
  <corps O2C>
};
```

L'appel d'une méthode *m* sur un objet *o* se fait par la notation fléchées “o->m”.

On peut écrire des applications, qui sont composées de programmes. Une application peut posséder des variables locales utilisables par tous ses programmes. La déclaration d'une application ressemble à celle d'une classe (cf. ci-dessous), et la déclaration de programmes et du corps des programmes ressemble à ceux des méthodes d'une classe O_2 :

```
program [public|private] <nom de programme>
            [(<liste de parametres avec leurs types>)]
            [: <type ou nom de classe>] in application <nom d'application>;
method body <nom de programme>[<liste de paramètres avec leurs types>]
            [: <type ou nom de classe>] in application <nom d'application>
{
  <corps O2C>
};
```

II.3. Types, Classes et Collections

Un objet qui est d'une certaine classe est composé d'une partie structure et d'une partie “comportement”. Lors de la définition d'une classe, on déclare le type de la structure des objets, par application récursive des opérateurs n-uplets, ensembles et listes. Il faut remarquer qu'O_2 fait une différence entre les `unique set` et les `set`, les premier ne permettant pas de double, et le second les permettant (comme les `bag` de SMALLTALK [Bai87]). On peut aussi donner les signatures des méthodes applicables aux objets de la classe. La déclaration de la classe TI (pour Terme d'Indexation) dans le chapitre précédent (partie IV.3.2) est donc la suivante, si l'on considère que l'on veut avoir tous les éléments des objets TI publics:

```
class TI
       public type tuple(terme : string),
       method public generique : set(TI)
end;
```

Dans O_2, l'ensemble des objets d'une même classe C n'est pas accessible. Si l'on veut gérer une telle collection persistante, il faut déclarer une racine de persistance qui est par exemple une valeur du type `set(C)`, dans laquelle nous allons insérer explicitement tout nouvel objet de C par l'intermédiaire d'une méthode.

II.4. Le langage de requête O_2SQL

Comme son nom l'indique, le langage de requête que propose O_2 est fortement inspiré de SQL. Cependant, des éléments supplémentaires ont été introduits pour prendre en compte les objets et leurs spécificités :

- Par exemple, en considérant que les objets TI, sont regroupés dans un

ensemble, valeur nommée (c'est à dire une racine de persistence qui est une valeur), appelée Les_TI, on peut trouver les TI qui ont comme générique un TI contenant la chaîne de caractères "france" en écrivant la requête suivante:

```
select i
from i in Les_TI
where exists g in i->generique :
      ( g->terme = "france" )
```

Le résultat de cette requête sera un ensemble valeur qui contiendra des objets TI qui vérifient la condition.

• Si nous voulons faire une projection de ces objets sur un attribut, comme par exemple sur le `terme` des objets du résultat, il suffit d'écrire:

```
select i->terme
from i in Les_TI
where exists g in i->generique :
      ( g->terme = "france" )
```

Si l'attribut projeté est une valeur (comme c'est le cas ici), le résultat sera donc un ensemble de valeurs. Si, par contre, le cas d'un ensemble d'objets, le résultat sera un ensemble d'objets.

• Si on veut projeter ces objets sur plusieurs attributs et/ou méthodes fonctions, le résultat est un ensemble de n-uplets (valeurs) que l'on déclare de la façon suivante (si on veut le terme du TI et ses TI génériques):

```
select tuple(te : i->terme, gen : i->generique)
from i in Les_TI
where exists g in i->generique :
      ( g->terme = "france" )
```

Dans ce cas, le résultat est un n-uplet à deux attributs, nommés *te* et *gen*. L'attribut *te* contient une valeur chaîne de caractères, et l'attribut *gen* une valeur ensemble d'objets TI.

• L'opérateur flatten existe dans O_2SQL, et il a la caractéristique de réduire d'une unité le nombre d'imbrications d'un ensemble imbriqué. Formulons une requête suivante:

```
select i->generique
from i in Les_TI
where exists g in i->generique :
      ( g->terme = "france" )
```

Cette requête va en fait donner comme résultat un ensemble d'ensembles de TI, car on obtient un ensemble de TI par élément de Les_TI, et le résultat est un ensemble qui englobe ces résultats élémentaires. Si on veut poser d'autres requêtes sur ce résultat, nous allons devoir *aplatir* cet ensemble, par un flatten. Ce qui nous donne la requête suivante:

```
flatten (select i->generique
         from i in Les_TI
         where exists g in i->generique :
              ( g->terme = "france" )   )
```

• Le prédicat "=" sur les objets correspond à la 0-égalité, mais il existe des méthodes-fonctions prédéfinies qui permettent de réaliser :

* la 1-égalité sur des objets o1 et o2, `o1->equal(o2)`, et

* l'i-égalité au niveau le plus profond, par `o1->deep_equal(o2)`.

• A l'intérieur de méthodes, une requête est appelée par l'intermédiaire de l'instruction `o2query(<nom de variable>, <chaîne de caractère>[,<nom de variable>]*)`. Le premier paramètre est le nom de la variable (préalablement déclarée) qui contiendra le résultat de la requête. La chaîne de caractère représente la requête écrite en O_2SQL avec des éléments particuliers qui sont des “$n”, avec n≥1. Ces “$n” permettent de relier les variables locales et les paramètres d'une méthode à une requête O_2SQL. Si nous considérons `res` est un ensemble de TI et que `ch` correspond au générique que nous cherchons, un appel à O_2SQL dans une méthode ou dans un programme sera:

```
o2query(res,"select i->generique \
             from i in Les_TI    \
             where exists g in i->generique : \
      ( g->terme = $1 )",ch);
```

Nous voyons donc qu'O_2SQL propose une grande partie des éléments qui nous sont utiles. Nous allons donc utiliser en grande partie ces éléments pour traduire nos processus sous forme de méthodes. Cependant, nous allons rajouter des éléments qu'O_2 ne propose pas dans la partie suivante.

III Implantation du processus de correspondance

Nous allons dans cette partie nous intéresser à la façon dont nous avons choisi d'implanter la correspondance d'objets complexes. Cette partie s'intéresse donc à la partie “Générateur de Correspondance d'Objets Complexes”.

Comme nous l'avons déjà dit, nous proposons un langage de description de correspondances. Nous avons donc utilisé, pour réaliser la compilation de ce langage en code compréhensible par le système O_2, les utilitaires UNIX *lex* et *yacc*, qui sont respectivement des analyseurs lexicographiques et syntaxiques.

Dans cette partie, nous allons insister sur trois points qui sont :
- le langage de requête utilisé dans les phases, en différenciant les opérateurs des autres éléments qui entrent en jeu,
- la déclaration d'un processus de Correspondance,
- la déclaration d'une phase de Génération.

III.1. Le langage de requête

Les éléments dont nous avons besoin (cf. V.2.3 du chapitre IV) sont les suivants :
- les opérateurs sur les ensembles Select, Image, Project, Flatten, Apply, ce dernier opérateur permettant des modifications du contenu des objets d'un ensemble,
- les opérateurs sur un objet *recurse* et *desc* (pour descendants),
- les prédicats d'i-égalité sur les objets,
- la facilités d'écriture “‡…‡‡…” (“‡ c_1 ‡‡ c_2” renvoie le résultat de l'expression booléenne c_2 uniquement si l'expression booléenne est vraie, sinon renvoie faux), et *reverse* qui permet de retrouver l'objet qui, dans un ensemble, référence un certain objet.

O_2 propose des équivalents pour, les opérateurs sur les ensembles sauf le Apply. De plus, il ne propose pas de *recurse* ou de *desc*. Il permet un sous-ensemble des prédicats d'i-égalité (cf. ci-dessus), pas de méthode *desc*.

La notation “->” permet d'atteindre les attributs d'un objet n-uplet et pour atteindre les méthodes d'un objet, et la notation “.” est utilisée pour atteindre un

attribut d'une valeur n-uplet. O_2 ne propose pas de notation “‡…‡‡…”, ni de *reverse*.

Nos choix sont les suivants pour le prototype que nous proposons:
• nous tentons d'utiliser au maximum ce que propose O_2. L'expression des opérateurs sera donc celle d'O_2. L'intérêt de cette approche est de ne pas nécessiter pour le programmeur l'apprentissage d'un nouveau langage de requête dans le cas d'utilisation de O_2. L'inconvénient par contre est d'être lié à O_2.
• les éléments qui ne sont pas proposés par O_2 vont être traduits en méthodes.

Nous, décrivons maintenant les compilations de *desc* et de *recurse*, puis de la notation “‡…#…” et enfin de l'opération Apply, car ce sont les éléments qui nous manquent.

III.1.1. La méthode *desc*

Dans la version d'O_2 qui a été utilisée, nous ne pouvons pas créer, en étant dans une application O_2, de nouvelles classes ou de nouvelles méthodes. Nous avons donc choisi de générer la déclaration de cette méthode à l'extérieur du système O_2. Nous avons fait les choix suivants:
• on va se limiter à suivre des liens qui rendent un objet de la même classe que le receveur,
• les liens suivis seront monovalués,
• on suppose que tous les liens sont du même type, `Ref`.

Si nous reprenons l'exemple de la partie IV.3.3.1 du chapitre précédent qui décrivait les objets "arbre sémantique", ces objets seraient déclarés par:

```
class Noeud_sem
public type tuple ( nom : string,
                    fils_g : Noeud_sem,
                    fils_d : Noeud_sem ),
method substituable : boolean
end;
```

Figure VI1 : Déclaration en O_2 des objets des arbres de RIME.

Rappelons ce que doit faire cette méthode, avec les simplifications que nous venons de faire:

• elle doit permettre, pour un objet quelconque *o*, de trouver ses descendants de sa classe au niveau n. Si n vaut zéro, alors on veut tous les descendants de l'objet de départ via les liens fixés, si n est supérieur à 0, on ne cherche les objets que jusqu'à la profondeur n.

Le résultat de la génération de la méthode *desc* sera celle décrite Figure VI2 pour la classe Noeud_sem.

```
method public desc(n:integer): unique set(Noeud_sem) in class Noeud_sem;
method body desc(n:integer): unique set(Noeud_sem) in class Noeud_sem
{
  o2 unique set(Noeud_sem) res;

  res = set();
  if (n==0)
  {
    if (self->fils_g != (o2 Noeud_sem) nil)
    {
       res += set(self->fils_g) + self->fils_g->desc(0);
    };
    if (self->fils_d != (o2 Noeud_sem) nil)
    {
       res += set(self->fils_d) + self->fils_d->desc(0);
    };
  };
  if (n>1)
  {
    if (self->fils_g != (o2 Noeud_sem) nil)
    {
       res += set(self->fils_g) + self->fils_g->desc(n-1);
    };
    if (self->fils_d != (o2 Noeud_sem) nil)
    {
       res += set(self->fils_d) + self->fils_d->desc(n-1);
    };
  };
  if (n==1)
  {
    if (self->fils_g != (o2 Noeud_sem) nil)
    {
       res += set(self->fils_g);
    };
    if (self->fils_d != (o2 Noeud_sem) nil)
    {
       res += set(self->fils_d);
    };
  };
};
```

Figure VI2: Un exemple de méthode *desc* générée

Cette génération utilise le nom de la classe et les chemins à suivre pour obtenir le résultat. Le résultat de la compilation tel qu'il est exprimé ici, se base sur le

fait que nous utilisons des arbres. En effet, nous remarquons que nous ne nous sommes pas intéressés aux tests de "bouclage" : si on se situe dans le cas d'un graphe avec cycle pour un objet, cette méthode ne s'arrêtera pas!

III.1.2. La méthode recurse

Nous avons choisi de représenter par une méthode, l'opérateur *recurse* tel que nous l'avons défini dans le chapitre IV. Une méthode, en O2C ne pouvant pas avoir un nom d'attribut en paramètre, nous avons choisi de générer une méthode `recurse_nom_attribut` pour chaque recherche récursive utilisée sur un attribut de nom `nom_attribut`.
Cette méthode peut en fait être considérée comme une version simplifiée de la méthode *desc*, car on suit un seul chemin, et on ne s'arrête que lorsque le chemin se termine (remarquons que, comme dans le cas précédent, on ne se préoccupe pas d'objets qui contiennent des cycles dans cette première version).

Si nous devons utiliser une méthode recurse sur un objet de la classe `Noeud_sem` en suivant le lien `fils_g`, la méthode générée sera celle de la Figure VI3.

```
method public recurse_fils_g: unique set(Noeud_sem) in class Noeud_sem;
method body recurse_fils_g: unique set(Noeud_sem) in class Noeud_sem
{
  o2 unique set(Noeud_sem) res;

  res = set();
  if (self->fils_g != (o2 Noeud_sem) nil)
  {
    res += set(self->fils_g) + self->fils_g->recurse;
  };
};
```

Figure VI3: Un exemple de méthode *recurse* générée

III.1.3. La notation “‡…#…”

Comme dans les cas précédents, nous devons représenter un *traitement* par cette notation, nous avons donc choisi de regrouper ces traitements dans une classe prédéfinie que nous appellerons `fonc_#` définie comme suit:

```
class fonc_#
;
```

On remarque que cette classe n'a pas de structure propre, car nous allons nous servir de son nom pour regrouper uniquement des méthodes.

En fait, nous utiliserons effectivement la notation “$…#…”, car le caractère "‡" n'est pas dans le jeu de caractère standard de la machine sur laquelle nous implémentons.
Une telle notation est toujours définie à l'intérieur de la définition de l'une des phases d'un processus de correspondance. Comme chacune de ces phases a un nom unique, on peut donc définir le nom d'une méthode de la classe fonc_# dérivé de ce nom, nous avons choisi d'utiliser le nom de la phase suivi de “_#” suivi d'un nombre. En effet, on peut utiliser plusieurs expressions de ce style dans la déclaration d'une seule phase.

Prenons un exemple. Considérons, la définition de la phase de comparaison suivante:

```
Comparaison
Nom : Comp_v0
Expression : ($ OQ->fils_g != nil # OQ->nom == "")
```

Figure VI4 : Un exemple de déclaration de “$...#” dans une Comparaison.

Dans ce cas, le code généré pour le “$…#…” sera la suivant:

```
method public Comp_v0_#1(OQ : Sem_node): boolean in class fonc_#;
method body Comp_v0_#1 in class class fonc_#
{
  o2 boolean res;

  res = (OQ->fils_g != (o2 Noeud_sem) nil);
  if (res == true)
  {
    res = (OQ->nom == "");
  };
  return(res);
};
```

Figure VI5 : Le code O_2C correspondant à la déclaration de la Figure VI4.

En fait, cette génération, qui est simple, va être compliquée par le fait que le langage de requête d'O_2 n'est pas intégré au langage de programmation sous forme d'expression, comme nous l'avons dit précédemment. Ceci veut dire que, quand un opérateur de requête est utilisé, il faut utiliser l'instruction *o2query*.

La formulation "$…#…" sera donc transformée par un appel de la méthode générée sur un objet de la classe `fonc_#` appelé `Obj_fonc_#`.

III.1.4. Le "Apply"

Rappelons que l'opérateur Apply permet de déclarer des **modifications** que l'on veut appliquer aux objets d'un ensemble.
Commençons par bien situer les choix que nous faisons.
Tout d'abord, nous posons que cette opération `apply` ne s'adressera qu'à des ensembles de valeurs dans O_2. Cet opérateur jouera sur les éléments locaux à une correspondance, et nous nous limiterons ici à des valeurs locales. Cette opération sera traduite par une fonction, qui renverra l'ensemble modifié.

Nous décrivons ici la génération associée à un opérateur "Apply(Select(E, Cond), Exp)", cette expression indiquant que l'on cherche à modifier seulement un sous-ensemble de E, en posant E un ensemble de **valeurs** :
-> l'ensemble *E* sur lequel on veut modifier les éléments qui vérifient une condition,
-> le filtre que l'on applique à cet ensemble E (le `select`…), basé sur une condition *Cond*, et qui détermine le sous-ensemble des éléments à modifier.
-> enfin l'expression Exp qui décrit les modifications des éléments.

En fait, au cours de la traduction, on va étendre *Exp* de manière à exprimer explicitement que des attributs des objets de l'ensemble de départ ne sont pas modifiés.
Considérons que nous voulons modifier seulement l'attribut A de n-uplets ayant deux attributs A et B, dans ce cas, l'expression étendue indiquera que B reprend la valeur qu'il avait précédemment. L'expression en O_2SQL serait donc

```
select tuple(A:…, B : i.B)
from i in …
```

Nous appellerons cette expression étendue *Exp_étendue(i)*, le paramètre i indiquant que i représente un élément de l'ensemble de départ.

En fait pour garder les valeurs de l'ensemble de départ qui sont inchangées, nous allons faire l'union des valeurs qui peuvent être modifiées (c.à.d. celles qui

vérifient la condition *Cond*) auxquelles on va appliquer *Exp_étendue*, et des valeurs de l'ensemble de départ qui ne vérifient pas la condition *Cond*.
En O_2SQL, l'expression serait la suivante:

```
(select Exp_étendue(i)
 from i in E
 where Cond )
+
(select i
 from i in E
 where not(Cond) )
```

Ce résultat sera utilisé pour valoriser *E*.

Nous utilisons la même démarche que dans le cas précédent de la notation "‡…#", qui est de définir une méthode d'une classe réservée pour les exécutions des opérateurs *apply* appelée `Apply_#`, méthode qui aura pour nom le nom de la phase dans laquelle est défini le *Apply* suivi de "_#n", avec n≥1. Ce *n* permet de différencier les opérateurs, dans une transition qui appelle deux apply, l'un pour O_D l'autre pour O_Q.

La notation qui va être utilisée pour le `apply` se rapproche de la syntaxe O_2SQL:

```
apply tuple([<nom> : fonction]*)
from <variable> in <variable>
where <condition de filtrage>
```

Considérons la transition de la figure suivante (dans laquelle le type de E_Q est "set(tuple(terme:Noeud_sem, vérité : boolean))" et E_D un ensemble de Noeud_sem):

```
Transition
Nom : Trans_v0
Pour OD : rien
Pour OQ : apply tuple(vérité : OQ in ED)
          from i in EQ
          where i->vérité =0 false
```

Figure VI6 : Un exemple de déclaration de l'opérateur apply dans une Transition.

Le résultat de la compilation de l'opération `apply` va être la méthode suivante:

```
method public Trans_v0_#1 (OQ : Noeud_sem,
                        EQ : set(tuple(terme: Noeud_sem, vérité:boolean))):
                  set(tuple(terme: Noeud_sem, vérité:boolean))
                                                  in class Apply_#;
method body Trans_v0_#1 (OQ : Noeud_sem,
                       EQ : set(tuple(terme: Noeud_sem, verite:boolean))):
                  set(tuple(terme: Noeud_sem, vérité:boolean))
                                                  in class class Apply_#
{
  o2query(EQ,"(select (terme:i->terme, verite : $1 in $2) \
               from i in $3 where i->verite == false)
             + (select i \
                from i in $3 where not(i->verite == false)", OQ, ED, EQ);
  return(EQ);
};
```

Figure VI7 : Le code O_2C généré pour le apply de la Figure VI6.

Une telle méthode sera donc appelée dans une phase de Transition, en l'appelant sur l'objet appelé `Obj_Apply_#`.

III.2. La Traduction de la déclaration du processus

Reprenons la déclaration de la correspondance simple du chapitre IV partie IV.2.7. Cette déclaration est la suivante:

```
Correspondance
Nom : Cor_exemple
OD : TI
OQ : TI
Local : set(TI) ED ;
        tuple(vérité : boolean) Oref
Génération : Gen_exemple
Comparaison : Com_exemple
Transition :  [ Nb_transition = 0 | Tra_exemple_1]
                    [ Nb_transition > 0 | Tra_exemple_2]
Test : Nb_transition < 4
```

Figure VI8 : La déclaration du processus.

Nous allons regrouper l'ensemble des traitements qui correspondent à des Correspondances dans une classe appelée Corres_#.

En fait, chaque nouvelle correspondance va être représentée par une méthode fonction qui renvoie une valeur qui contient les éléments (valeurs) susceptibles d'être modifiées par les phases, ainsi que les éléments locaux au processus, et qui va contenir les appels aux phases de cette correspondance, qui seront elles

aussi des méthodes.

Cette méthode va être traduite en O_2C en exprimant le schéma fonctionnel du processus décrit dans la figure IV4 du chapitre IV.

La méthode générée est décrite dans la figure suivante:

```
method public Cor_exemple (OD : TI, OQ : TI )
                                                : boolean in class Corres_#;
method body Cor_exemple (OD : TI, OQ : TI )
                                                : boolean in class Corres_#
{
  o2 boolean res1, res2;
  o2 tuple(ED : set(TI), Oref : tuple(vérité : boolean)) Olocal
  o2 integer Nb_transition;


  Nb_transition = 0;

  Olocal = Obj_Gen_#->Gen_exemple(OD, OQ, Olocal);
  res1 = Obj_Comp_#->Com_exemple(OD, OQ, Olocal,Nb_transition);
  res2 = Obj_Test_#->Cor_exemple_T(OD, OQ, Olocal,Nb_transition);
  while ((res1 == false) || (res2 == true))
  {
    if (Nb_transition == 0)
      Olocal = Obj_Trans_#->Tra_exemple_1(OD, OQ, Olocal, Nb_transition);
    else if (Nb_transition > 0)
        Olocal = Obj_Trans_#->Tra_exemple_2(OD, OQ, Olocal, Nb_transition);
    res1 = Obj_Comp_#->Com_exemple(OD, OQ, Olocal, Nb_transition);
    res2 = Obj_Test_#->Cor_exemple_T(OD, OQ, Olocal, Nb_transition);
    Nb_transition++;
  };
  return(res1);
};
```

Figure VI9 : Le résultat de la traduction du processus de correspondance.

Les objets Obj_Gen_#, Obj_Comp_# et Obj_Trans_# correspondent à des objets nommés des différentes classes nécessaires au processus de correspondance.
L'objet Obj_Test_# joue le même rôle que les objets ci-dessus, mais pour l'étape de Test qui sera en fait traitée comme une phase du processus.

Nous choisissons de passer à chacune des méthodes qui joue le rôle d'une phase l'ensemble des éléments locaux en les regroupant dans une valeur appelée `Olocal`.

III.3. La traduction de la déclaration d'une phase de Génération

Reprenons la génération de l'exemple simple de la partie V.2.4 du chapitre IV. En la traduisant en une syntaxe proche de O_2SQL, cette déclaration est la suivante:

```
Generation
Nom : Gen_exemple
Pour OD : ED = set();
Pour OQ : Oref = tuple(verite: OD == OQ)
```

Figure VI10 : Déclaration d'une Génération.

Nous avons déjà expliqué le traitement effectué pour une méthode `desc` (cf. partie III.1.1 de ce chapitre. La méthode de la classe `Gener_#` (qui regroupe les générations) sera la suivante :

```
method public Gen_exemple_# (OD : C_sem_doc, OQ : Obj_req,
            Olocal : tuple(ED : set(TI), Oref : tuple(vérité : boolean))):
    tuple(ED : set(TI), Oref : tuple(vérité : boolean)) in class Gener_#;
method body Cor_exemple-# (OD : C_sem_doc, OQ : Obj_req,
            Olocal : tuple(ED : set(TI), Oref : tuple(vérité : boolean))):
     tuple(ED : set(TI), Oref : tuple(vérité : boolean))  in class Gener_#
{

  Olocal.ED = set();
  Olocal.Oref = tuple(vérité : OD == OQ );
  return(Olocal);
};
```

Figure VI11 : Le code O_2C généré pour la génération de la Figure VI11.

Nous voyons donc que le seul souci durant cette déclaration est de déterminer les éléments qui sont des paramètres de ces méthodes, de manière à les intégrer sous forme de "$n" (avec n≥1) dans la chaîne de caractère qui est le second paramètre de l'instruction `o2query`.

IV. Les appels à un processus de correspondance

Dans cette partie, Nous allons nous intéresser à la partie "Traducteur d'Appel de Correspondance d'Objets Complexes" ou TACOC.

En fait, nous avons choisi une approche très simple pour ces appels. En effet, un

processus de correspondance est en fait représenté par une méthode de la classe `Corres_#`.

A l'intérieur du code d'une méthode d'une classe ou d'un programme d'une application, l'appel à un processus de correspondance sera fait de la façon suivante:
on utilisera une fonction appelée `Correspondance`, qui contiendra le nom de la correspondance et les paramètres qu'utilise le processus, c'est à dire les objets O_D et O_Q comparés. Cette fonction renverra un résultat booléen.

Prenons un exemple basé sur l'exemple simple. Si nous avons dans le corps d'une méthode les éléments suivants:

```
{
  o2 C_sem_doc d;
  o2 Obj_req r;
  o2 boolean match;

…

  match = Correspondance(Cor_exemple(d,r));
…
};
```

L'appel sera transformé en:

```
…
  match = Obj_Corr_#->Cor_exemple(d,r);
…
```

Ce traitement est donc très simple.

V. Schéma des générations et appels de correspondances

Nous avons décrit dans la partie précédente les éléments qui entrent en jeu, mais sans les placer dans le processus de génération d'une application.
Le déroulement est donné dans le schéma de la figure suivante :

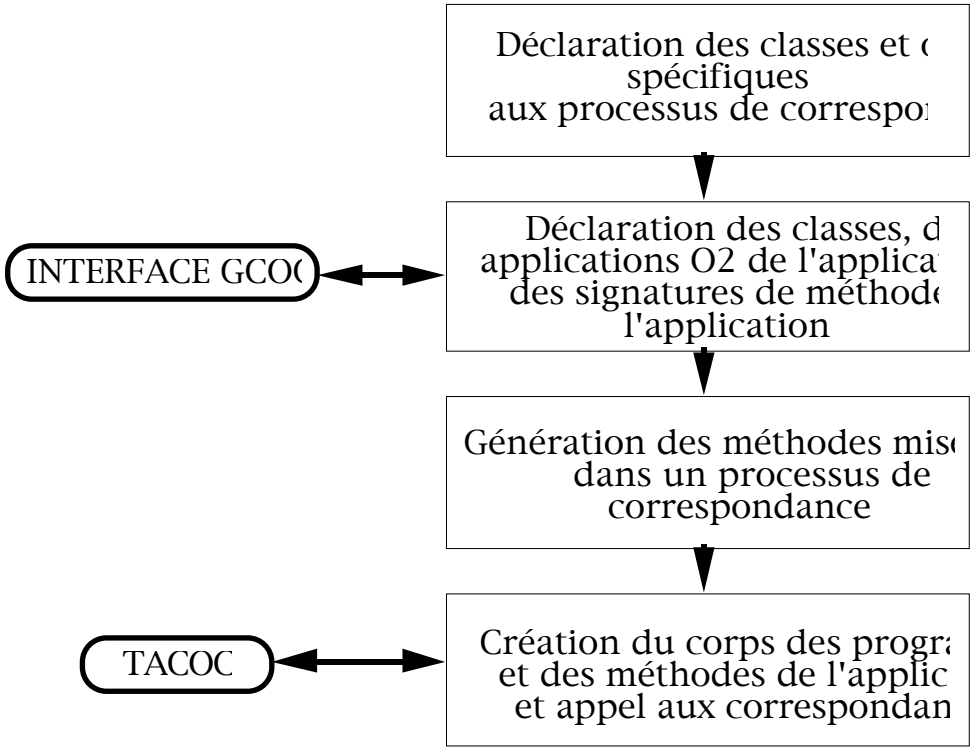


Figure VI12 : Déroulement de la déclaration de Correspondances.

VI. Conclusion

Nous avons donc décrit dans ce chapitre comment réaliser une implantation sur le système O_2 de l'opérateur de comparaisons pour objets complexes.

Cette expérimentation est en fait la réalisation d'un compilateur, qui fournit comme résultat des déclarations de méthodes qui sont compilées en O_2, comme le reste de l'application que développe le programmeur. Pour cette implantation, nous avons choisi certaines contraintes qui facilitent l'expression de la compilation, comme par exemple décider que les éléments locaux à un processus de correspondance sont des valeurs et non pas des objets.

De plus, le choix de définir ces correspondances comme étant au même niveau que le reste de l'application nous amène à une contrainte que nous n'avons pas décrite précédemment et qui est la suivante :

Les parties des objets qui entrent en jeu dans des correspondances d'objets doivent être déclarées publiques, sans quoi on ne pourra pas y accéder pour réaliser les correspondances.

L'élément ci-dessus est en fait un problème inhérent à la correspondance telle que nous l'avons décrite. La solution, dans le cas où un processus de correspondance est considéré comme à part de l'application, serait de permettre aux parties d'une correspondance de briser l'encapsulation de tout objet. A ce moment, il y aurait bien deux niveaux différents, celui des correspondances où il n'y a plus d'encapsulation, et le niveau de l'application où le concept d'encapsulation est vérifié.

Nous allons maintenant nous intéresser aux éléments du langage de requête que nous avons défini pour les correspondances, en cherchant à évaluer lesquels sont susceptibles de descendre au niveau du noyau d'un SGBD, et ceux qui sont à laisser au niveau application.
Les opérateurs tels que le Select, l'Image, le Project, le Flatten sont déjà proposés, sous une version ou une autre, par les systèmes Orientés Objet.
Il serait intéressant d'intégrer le *recurse* et le *desc* aux opérateurs de requêtes. En effet, ces deux éléments prennent en compte la partie structure des classes (pour desc), ou bien la structure et les méthodes d'une classe (pour recurse). Ceci permet à un programmeur d'application de ne pas avoir à les programmer.
L'opérateur *Apply*, dont nous avons décrit une implémentation dans ce chapitre, se rapproche en fait de l'instruction UPDATE du langage SQL . La question à se poser à propos de cet opérateur est de savoir si *on a le droit* de modifier ou non les objets. En fait, comme on se situe dans un contexte très précis, dans lequel cet opérateur s'applique à des éléments locaux au processus de correspondance, ces modifications sont acceptables.

A partir des expressions des différentes phases, on peut se demander s'il est possible de les optimiser. En fait, une solution serait d'examiner les travaux de Zdonik et Shaw [Zdo89b] sur les optimisations algébriques de requêtes. Ces recherches utilisaient des notions d'*équivalence* de requêtes, en indiquant quelles combinaisons fournissaient des résultats i-égaux. A priori, comme les expressions que nous générons sont assez complexes, des optimisations de ce type seraient probablement possibles, mais ce point reste à étudier.

CHAPITRE VII

Conclusion

CHAPITRE VII

Conclusion

Dans cette thèse, nous sommes intéressés à une manière d'exprimer la correspondance d'objets complexes. Notre approche s'est inspirée de travaux réalisés dans le domaine de la Recherche d'Informations.

Ce problème de comparaisons de données complexes apparaît de façon criante dans les systèmes à objet. De tels systèmes sont capables maintenant de gérer des données complexes inter-reliées représentées par des objets. Mais, parallèlement à des recherches sur des problèmes précis, comme les gestions d'objets en mémoires principales et secondaires, les transactions, la cohérence, …, le domaine des requêtes a peu évolué, dans la mesure où les langages de requêtes proposés sont des extensions plus ou moins simples de SQL. Une telle approche est-elle réaliste, ou doit-on repenser ces langages dans le contexte objet? Le problème dans le second cas est que ce choix est une solution de "destruction et reconstruction", auquel seraient probablement opposées les personnes qui utilisent depuis une dizaine d'années SQL, et qui constituerait un frein à l'utilisation des SGBDOO.

Notre solution est située à la croisée des deux options, dans la mesure où nous définissons un langage qui change la vision du processus de requêtes : on réalise un processus de correspondance d'objets dans un premier temps, puis on réalise des appels à ce processus à l'intérieur des expressions habituelles des langages de requêtes des SGBDOO. L'approche proposée n'est pas simplement un système

de requêtes nommées, car l'expression d'une correspondance met en jeu des éléments qui ne sont pas limités au strict langage de requête d'un SGBD.

Un premier pas a été fait par S. Zdonick et G. Shaw dans [Zdo90], pour intégrer des comparaisons d'objets par des prédicats d'*i-égalité* dans des requêtes sur des objets. Cependant, ces correspondances sont prédéfinies et non-modifiables, et ne "jouent" que sur la connaissance que le système a de la structure des objets.
Quant à nous, nous considérons que la structure des objets sert de support à leur sémantique, et que cette sémantique possède des caractéristiques spécifiques que l'on peut utiliser. Plus précisément, nous considérons dans cette thèse que la logique aide à exprimer cette sémantique, ainsi que la correspondance des objets. Nous touchons ici du doigt un problème cher à la communauté "Recherche d'Informations", qui est la comparaison entre une requête et un document représentés par leur sémantique respective.
La démarche à suivre pour utiliser nos propositions est de définir la sémantique des objets comparés sous forme logique, l'un des objets servant de base de référence pour la comparaison. Le concepteur décrit l'utilisation des règles d'extensions logiques qui permettent de réaliser ensuite les comparaisons rendues possibles par l'utilisation de ces règles. Il dispose du langage que nous avons défini pour implémenter la comparaison. L'utilisateur voulant simplement faire des requêtes sur des objets, il peut utiliser un certain nombre de correspondances définies par un concepteur d'application.

Cette approche originale permet donc d'intégrer des connaissances à un processus de correspondance d'objets. Notre parti pris a été de rendre explicite l'utilisation de connaissance et de ne pas employer de mécanisme de déduction comme dans les DOOD (Deductive Object Oriented Databases), qui recherchent explicitement ce genre d'intégration. Nous pensons qu'au niveau des comparaisons d'objets telles que nous les exprimons, ces mécanismes sont trop lourds. En effet, il ne se justifient pas dans le cas où la connaissance est exprimable par des implications élémentaires.

L'utilisation de correspondances valuées permet l'emploi réel de ces propositions pour réaliser des ordonnancements d'objets qui correspondent plus ou moins à un objet de référence.

Un autre aspect intéressant de l'utilisation de la sémantique des objets serait

d'exprimer que l'on désire obtenir la "meilleure partie" d'un objet correspondant à un objet de référence.
Situons-nous par exemple dans le cas d'une base de données textuelles, où les documents structurés en chapitre, sous-chapitre, etc, sont représentés par des objets complexes. La structure des objets, enrichie de certaines connaissances indiquant comment utiliser cette structure, pourrait permettre au système de fournir un objet-livre composé de 5 chapitres comme réponse à une requête plutôt que les 5 chapitres de ce livre indépendamment. Cette notion a été abordée dans [Mul92a], uniquement dans un cadre pratique.

Dans une direction plus typiquement bases de données, une étude sur la manière de représenter des requêtes imprécises (cf. [Mot90]) serait à faire. Une requête imprécise se rapproche des requêtes évoluées du chapitre 2. On peut, par l'expression de telles requêtes, exprimer une certaine imprécision sur les valeurs des attributs des éléments de la base que l'on recherche.
Une telle requête peut être :"Je cherche les personnes qui sont jeunes". En fait, le terme jeune exprime une imprécision sur l'âge des personnes. Dans cet exemple, *jeune* est un générique pour un intervalle d'entiers, par exemple [0,20]. On peut donc rechercher les personnes qui ont un âge dont le générique est "jeune". Ceci revient à exprimer le fait qu'une connaissance nous permet de dire que "jeune" est une extension valide pour une personne, et que dans ce cas la personne "étendue" par les termes qui caractérisent les valeurs de ses attributs correspond à la requête.

Dans cette thèse, nous ne nous sommes pas intéressés à des optimisations de ce processus. Une étude préalable serait nécessaire, pour déterminer les éléments spécifiques des processus de correspondances sur lesquels des optimisations diverses sont possibles. Des travaux comme par exemple ceux de Shaw et Zdonick [Zdo89b] sur des optimisations algébriques seraient à étudier.

L'approche que nous avons décrite dans cette thèse nous semble donc un point de départ intéressant pour voir sous un jour nouveau le processus d'interrogation dans un SGBDOO. Notre approche étant générale, on peut raisonnablement penser qu'elle est applicable à d'autres contextes que celui de la Recherche

d'Informations. De plus, par l'intégration de correspondances valuées indiquant dans quelle mesure deux objets correspondent, notre approche fournira une base à une réelle utilisation des systèmes de gestion de bases de données orientés objets pour réaliser des systèmes de recherche d'informations, ce qui était le but de cette thèse.

Références Bibliographiques

Références Bibliographiques

[Adi92] M. Adiba, C. Collet, P. Dechamboux et B. Defude, "*Integrated Tools for Object Oriented Persistent Application Development*", DEXA'92, Valence, Espagne, Septembre 1992.

[Ala89] A. Alashqur, "*OQL: A Query Language for Manipulating Object-Oriented Databases*", 15ème VLDB, Amsterdam, Pays-Bas, 1989.

[And88] T. Andrews, C. Harris, K. Sinkel, "*The Ontos Object Database*", ?, 1988.

[Ando92] E. Andonoff, C. Mendiboure, C. Morin, V. Rougier et G. Zurfluh, "*Une Interface Graphique Evoluée pour l'Interrogation d'une Base de Données Orientée Objet*", VIIIèmes Journées Bases de Données Avancées, Trégastel, France, Septembre 1992.

[Ari83] H. Arisawa, K. Moriya et T. Miura, "*Operations and the Properties on Non-First-Normal-Form Relational Databases*", VLDB '83, Florence, Italie, Octobre 1983.

[Atk89] M. Atkinson, F. Bancilhon, D. De Witt, K. Dittrich, D. Maier, S. Zdonick,"*The Object-Oriented Database System Manifesto*", Rapport Technique Altaïr 30-89, Août 1989.

[Bai87] C. Bailly, J-F. Challine, Paul Gloess, H-C. Ferri et B. Marchesin, "*Les Langages Orientés Objet*",Techniques Avancées de l'Informatique, Cepadues Editions, Toulouse, 1987.

[Ban88] F. Bancilhon, "*Object-Oriented Database Systems*", 7ème ACM SIGART-SIGMOD-SIGACT, Symposium on Principles of DataBase Systems, Austin, USA, Mars 1988.

[Ban89] F. Bancilhon, S. Cluet et C. Delobel, "*A Query Language for the O2 Object-Oriented Database System*", 2nd Intl. Workshop on Database Programming Languages, juin 1989.

[Bane87] J. Banerjee, H-T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou et H-J. Kim, "*Data Model Issues for Object-Oriented Applications*", ACM Transactions on Office Information Systems, Vol.5, N° 1, Janvier 1987.

[Bau91] J. Baumgardner, "*OOPs defined*", article paru dans le *newsgroup* comp.object, en Janvier 1991.

[Ber91] E. Bertino et L. Martino, "*Object-Oriented Database Management Systems: Concepts and Issues*", IEEE Computer, Avril 1991.

[Berr88] C. Berrut, "*Une méthode d'indexation fondée sur l'analyse sémantique de documents spécialisés. Le prototype RIME et son application à un corpus médical*", Thèse de doctorat de 3ème cycle de l'Université Joseph Fourier, Grenoble, Décembre 1988.

[Bha92] B. Bhargava, "*Transition From a Relation to Object Model Implementation*", 1er CIKM, Baltimore, USA, Novembre 1992.

[Bos91] P. Bosc et O. Pivert, "*Quelques Approches d'Interrogation Flexible de Bases de Données*", VIIèmes Journées Bases de Données Avancées, Lyon, France, Septembre 1991.

[Bos91b] P. Bosc et O. Pivert, "*About Equivalences in SQLf, A Relational Language Supporting Imprecise Querying*", FUZZ-IEEE/IFES 91', Yokohama, Japon, Novembre 1991, pp 309-320.

[Bri93] F. Brissaud, *"Contribution à la conception logicielle de systèmes d'applications. La méthode MOSAIC dans le projet Aristote"*, thèse de doctorat de 3ème cycle de l'Université Joseph Fourier, Grenoble, Mai 1993.

[Bru89] M-F. Bruandet, "*Outline of a Knowledge Base Model for an Intelligent Information Retrieval System*", Information Processing & Management, Vol. 25, N° 1, 1989.

[Car86a] M. Carey, D.DeWitt, J. Richrdson and E. Shekita, "*Object and File Management in the EXODUS Extensible Database System*", VLDB '86, Kyoto, Japon, Août 1986.

[Car86b] M. Carey, D. DeWitt, "*The Architecture of the EXODUS Extensible DBMS*", Intl. Workshop on Object-Oriented Database Systems, Pacific Grove, USA, Septembre 1986.

[Car87] M. Carey et D. DeWitt, "*An Overview of the EXODUS Project*", Database Engineering '87, Juin 1987.

[Cha83] A. Chan, U. Dayal, S. Fox et D. Ries, "*Supporting a Semantic Data Model in a Distributed Database System*", VLDB 83, 1983.

[Che92] J-P. Chevallet, "*Un modèle logique de Recherche d'Informations appliqué au formalisme des Graphes Conceptuels. Le prototype ELEN et son expérimentation sur un corpus de composants logiciels*", Thèse de doctorat de 3ème cycle de l'Université Joseph Fourier, Grenoble, Mai 1992.

[Chi87] Y.Chiaramella, C. Berrut et P. Cinquin, "*A Conceptual Model for Medical Reports in a Multimedia Environment*", A.I. and Neuroscience, Demongeot Ed., Manchester Union Press, Fevier 1987.

[Cro90] B. Croft, R. Krovetz et H. Turtle, "*Interactive Retrieval of Complex Documents*", Information Processing & Management, Vol. 26, N° 5, 1990.

[Cro92a] B. Croft et H. Turtle, "*Retrieval of Complex Objects*", EDBT 92, 1992.

[Cro92b] B. Croft, L. Smith et Howard Turtle, "*A Loosely-Coupled Integration of a Text Retrieval System and an Object-Oriented Database System*", ACM SIGIR 92, Copenhague, Danemark, 1992.

[Cuz92] R. Cuzin, "*Visualisation d'Objets Multimédia. Application aux Dossiers médicaux*", Note Technique Aristote n°14, Octobre 1992.

[Day89] U. Dayal, "*Queries and Views in an Object-Oriented Data Model*", 2nd Workshop on Database Programming Languages, 1989.

[Dec93] P. Dechamboux, "*Gestion d'Objets Persistants : Du Langage de Programmation au Système*", Thèse de doctorat de 3ème cycle de l'Université Joseph Fourier, Grenoble, Février 1993.

[Def86] B. Defude, "*Etude et Réalisation d'un Système Intelligent de Recherche d'Informations : le prototype IOTA*", Thèse de doctorat de 3ème cycle de l'Institut National Polytechnique de Grenoble, Juillet 1986.

[Deu91] O. Deux et al., "*The O2 system*", CACM, Vol. 34, N° 10, Octobre 1991.

[Elm89] R. Elmasri et S. Navathe, "*Fundamentals of Database Systems*", Benjamin/Cumming Publishing Company, 1989.

[Gar90] G. Gardarin et P. Valduriez, "*SGBD AVANCES. Bases de données objets, déductives, réparties*", Editions Eyrolle, 1990.

[Har92] D. Harper et A. Walker, *"ECLAIR : An Extensible Class Library for Information Retrieval"*, Computer Journal, Vol. 35, N° 3, Janvier 1992.

[ING91] "*INGRES Database Administrator's Guide for the UNIX operating SYSTEM. Release 6.4*", Decembre 1991.

[Ish93] H. Ishikawa, F. Suzuki, F. Kozakura, A. Makinouchi, M. Miyagishima, Y. Izumida, M. Aoshima et Y. Yamane, *"The Model, Language, and Implementation of an Object-Oriented Multimedia Knowledge Base Management System"*, ACM TRansactions on Database Systems, Vol. 18, N° 1, Mars 1993.

[Jar90] S. Jarwah et M-F. Bruandet, "*An Object-Oriented model for Hypertext Database: Document Management in Software Engineering*", Rapport de recherche Aristote - RAP004, LGI, Janvier 1990.

[Jar92] S. Jarwah, "*Un Modèle Générique pour la Gestion des Informations Complexes et Dynamiques. Gestionnaire d'Objets du prototype ELEN Dédié au Génie Logiciel*", Thèse de doctorat de 3ème cycle de l'Université Joseph Fourier, Grenoble, Avril 1992.

[Kam90] M. Kamel, B. Hadfield et M. Ismail, "*Fuzzy Query Processing Using Clustering Techniques*", Information Processing & Management, Vol. 26, N° 2, 1990.

[Kim88a] W. Kim, N. Ballou, H-T. Chou, J. Garza, D. Woelk et J. Banerjee, "*Integrating an Object-Oriented Programming System with a Database System*", ACM OOPSLA '88, Septembre 1988.

[Kim89a] W. Kim, E. Bertino et G. Garza, "*Composite Objects Revisted*", ACM SIGMOD on Management of Data, Chicago, USA, 1989.

[Kim89b] W. Kim, "*A Model of Queries for Object-Oriented Databases*", VLDB '89, Amsterdam, Pays-Bas, 1989.

[Kim89c] W. Kim, E. Bertolino et F. Rabitti, "*A Query Language for Object-Oriented Databases*", Rapport Technique MCC n°ACT-OODS-337-89, Austin, USA, Septembre 1989.

[Kle71] S. Kleene, "*Logique Mathémathique*", Collection U, Armand Colin, 1971.

[Lam91] C. Lamb, G. Landis, J. Orenstein et D. Weinreb, "*The ObjectStore Database System*", CACM, Vol. 34, N° 10, Octobre 1991.

[Lec88] C. Lécluse, P. Richard et F. Velez, "*O2, an Object-Oriented Data Model*", EDBT '88, Venise, Italie, Mars 1988.

[Lec89a] C. Lécluse et P. Richard, "*Langages orienté-objet et bases de données: l'expérience O2*", Vèmes Journées Bases de Données Avancées, Genève, Suisse, Septembre 1989.

[Lec89b] C. Lécluse et P. Richard, "*The O2 Database Programming Language*", VLBD '89, Amsterdam, Pays-Bas, 1989.

[Lem89] J. Le Maitre et O. Boucelma, "*LIFOO, un Langage Fonctionnel de Requêtes pour Bases de Données Orientées Objet*", Techniques et Sciences de l'informatique, Hermes, Vol. 11, n° 5, 1992, pp 67-94.

[Lem93] J. Le Maitre et O. Boucelma, "*LIFOO, un Langage d'Interrogation Fonctionnel pour une Base de Données Orienté Objet*", Vèmes Journées Bases de Données Avancées, Genève, Suisse, Septembre 1989.

[Lem 93] J. Le Maitre, *"Construction, Programmation et Interrogation de Bases de Données"*, Habilitation à diriger les Recherches de l'Université d'Aix-Marseille II, faculté des Sciences de Rumilly, Février 1993.

[Loh91] G. Lohman, B. Lindsay, H. Pirahesh et K. Bernhard Schiefer, "*Extensions to STARBURST: objects, types, functions, and rules*", CACM, Vol. 34, N° 10, Octobre 1991.

[Man90] M. Mannino, J. Choi et D. Batory, "*The Object-Oriented Functional Data Language*", IEEE Transactions on Software Engineering, Vol. 16 N° 11, 1990.

[Mar91] H.Martin, *"Contrôle de cohérence dans les bases objets : Une approche par le comportement"*, Thèse de doctorat de 3ème cycle de l'Université Joseph Fourier, Grenoble, Janvier 1991.

[Mei92] C. Medeiros et M. Andrade, "*Implementing Integrity Control in Active Databases*", CIKM-92, Baltimore, USA, Novembre 1992.

[Mot88] A. Motro, "*VAGUE: A User Interface to Relational Databases that permits Vagues Queries*", ACM TRansactions, on Office Information Systems, Vol. 6, N° 3, Juillet 1988.

[Mot90] A. Motro, *"Accomodating Imprecision in Database Systems: Issues and Solutions"*, ACM SIGMOD Records, Vol. 19, N° 4, Décembre 1990.

[Mul91] P. Mulhem, *"Systèmes de Recherche d'Informations et Requêtes dans des SGBD"*, Rapport Aristote RAP-016, Laboratoire de Génie Informatique, Grenoble, France, Octobre 1991.

[Mul92a] P. Mulhem et M-F. Bruandet, *"Tool for Information Retrieval Applications using Object Oriented Database Management Systems"*, VIIIèmes Journées Bases de Données Avancées, Trégastel, France, Septembre 1992.

[Mul92b] P. Mulhem et M-F. Bruandet, *"On Objects Matching"*, International Symposium on Computer and Information Sciences VII, Antalya, Turquie, Novembre 1992.

[Nie90a] J. Nie,"*Un modèle logique général pour les systèmes de Recherche d'Informations. Application au prototype RIME*", Thèse de doctorat de 3ème cycle de l'Université Joseph Fourier, Grenoble, Juillet 1990.

[Nie90b] J. Nie et Y. Chiaramella, "*A Retrieval Model based on an Extented Modal Logic and its Application to the RIME Experimental Approach*", ACM SIGIR 90, Bruxelles, Belgique, 1990.

[O2U92] "*The O_2 User Manual, Version 4.1*", Octobre 1992.

[ORA88] "*ORACLE PL/SQL, Users Guide and Reference, part n°800-v6.0*", 1988.

[Pas88] S. Pasleau, "*SQL. Langage et SGBD Relationnels*", Editions PSI, 1988.

[Pis86] P. Pistor et F. Andersen, "*Designing A Generalized NF2 Model with an SQL-Type Language Interface*", VLDB '86, Kyoto, Japon, Août 1986.

[Rab90] F. Rabitti, "*Retrieval of Multimedia Documents by Imprecise Query Specification*", LNCS n° 416, Advances in Databases Technologies, EDBT '90, Venise, Italie, Mars 1990.

[Raj88] K. Raju et A. Majumdar, *"Fuzzy Functionnel Dependancies and Lossless Join Decomposition of Fuzzy Relational Database Systems"*, ACM Transactions on Database Systems, Vol. 13, N° 2, Juin 1988.

[Rij79] C. Van Rijsbergen, "*Information Retrieval*", seconde édition, Butterworth Londres, 1979.

[Rij90] C. Van Rijsbergen, "*Toward an Information Logic*", 1ère Ecole d'été en Recherche d'Information, Bressanone, Italie, Juillet 1990.

[Ros89] A. Rosenthal, S. Chakravarthy, B. Blaustein et J. Blakeley, "*Situation Monitoring for Active Databases*", 15ème VLDB, Amsterdam, Pays-Bas, 1989.

[Sal71] G. Salton, "*The SMART Retrieval System, Experiments in Automatic Document Processing*", G. Salton Editeur, Prentice-HALL, 1991.

[Sal85] G. Salton, "*A note on Information Retrieval Models and Theories*", Recherche d'Informations Assistée par Ordinateur, Grenoble, France, Mars 1985.

[Sme88] A. Smeaton et C. Van Rijsbergen, "*Experiments on Incorporating Syntactic Processing of User Queries into a Documental Retrieval Strategy*", ACM SIGIR 88, Grenoble, France, Juin 1988.

[Sow84] J. Sowa, "*Conceptual Structures: Information Processing in Mind and Machine*", Addison-Weyley publishing company, 1984.

[Sri92] J. Srinivasan, Y-H. Jiang, Y. Zhang et B. Bhargava, "*Experiments of O-Raid Distributed Object-Oriented Database System*", 1er CIKM, Baltimore, USA, Novembre 1992.

[Sto76] M. Stonebraker, E. Wong, P. Kreps et G. Held, "*The Design and Implementation of INGRES*", ACM Transaction on Database Systems, Vol. 1, N° 3, Septembre 1976.

[Sto86] M. Stonebraker et L. Rowe, "*The Design of POSTGRES*", ACM SIGMOD '86, Washinton, USA, Mai 1986.

[Su88] S. Su, V. Krishnamurphy et H. Lam, "*An Object-Oriented Semantic Association Model (OSAM*)*", A.I. in Industrial Engineering and Manufacturing: Theoretical Issues and Applications, Edité par S. Kumar, A. Soyster et R. Kashyap, American Institute of Indutrial Engineering, Chapitre 17, 1988.

[Swa81] M. N. Swamy et K. Thulasiraman, *"Graphs, Networks and Algorithms"*, Wiley Interscience, 1981.

[Tha90] "*Multimedia Office Filling: The MULTOS Approach*", C. Thanos Ed., Elservier Science (Noth-Holland), 1990.

[Tsu86] S. Tsur, C. Zaniolo, "*LDL: A Logic-Based Data-Language*", 12ème VLDB, Kyoto, Japon, Août 1986.

[Tur90] H. Turtle et B. Croft, "*Inference Networks for Document Retrieval*", ACM SIGIR 90, 1990.

[Vel89] F. Velez, G. Bernard et V. Darnis, "*The O_2 Object Manager: An Overview*", VLDB '89, Amsterdam, Pays-Bas, 1989.

[Weg87] P. Wegner, "*Dimensions of Object-Based Language Design*", OOPSLA '87, Octobre 1987.

[Won90] M. Wong et K. Leung, "*A Fuzzy Database Query Language*", Information Systems, Vol. 15, N° 5, 1990.

[Zdo86] S. Zdonik et P. Wegner, "*Language and Methodology for Object-Oriented Database Envoronment*", 17ème Hawaii International Conference on System Sciences, 1986.

[Zdo89a] S. Zdonik et G. Shaw, "*A Object-Oriented Query Algebra*", 2nd Intl. Workshop on Database Programming Languages, juin 1989.

[Zdo89b] S. Zdonik et G. Shaw, "*Object-Oriented Queries: Equivalence and Optimization*", DOOD '89, Kyoto, Japon, 1989.

[Zdo90] S. Zdonick et G. Shaw, "*A Query Algebra for Object-Oriented Databases*", 6th Intl. Conf on Data Engineering, IEEE 1990.

Annexes

Pour O_Q :

E_Q = **Project**(
Select(**Union**(O_Q->desc(0,Nœud_sem, Ref), set(O_Q)),
λo o->fils_g <>$_0$ nil),
λd <(term,d),
(verite, (d->fils_g->fils_g =$_0$ nil
and d->fils_d->fils_d =$_0$ nil
and **existe**(
Project(E_D, λo o->term),
λe d->nom =$_0$ e->nom
and e->fils_g->fils_g =$_0$ nil
and e->fils_g->nom =$_0$ d->fils_g->nom
and e->fils_d->fils_g =$_0$ nil
and e->fils_d->nom =$_0$ d->fils_d->nom
))
or (d->fils_g->fils_g =$_0$ nil
and d->fils_d->fils_g <>$_0$ nil
and **existe**(
Project(E_D, λo o->term),
λe d->nom =$_0$ e->nom
and ((e->fils_g->fils_g =$_0$ nil
and e->fils_g->nom =$_0$ d->fils_g->nom)
or e->fils_g->fils_g <>$_0$ nil)
and e->fils_d->fils_g <>$_0$ nil))
or (d->fils_d->fils_g =$_0$ nil
and d->fils_g->fils_g <>$_0$ nil
and **existe**(
Project(E_D, λo o->term),
λe d->nom =$_0$ e->nom
and ((e->fils_d->fils_g =$_0$ nil
and e->fils_g->nom =$_0$ d->fils_g->nom)
or e->fils_d->fils_g <>$_0$ nil)
and e->fils_g->fils_g <>$_0$ nil))
or (d->fils_g->fils_g <>0 nil
and d->fils_d->fils_g <>0 nil
and **existe** (
Project(E_D, λo o->term),
λe d->nom =$_0$ e->nom))) >)

Pour O_Q :

```
Apply(
        Select( EQ, λo o->vérité = False),
        λd <(term,d),
            (verite,  (     d->fils_g->fils_g =0 nil
                        and d->fils_d->fils_d =0 nil
                        and existe( ED,
                                λf   d->nom =0 f->term->nom
                                and existe( Union( f->sup_g,
                                                   set(f->term->fils_g),
                                    λe   e->fils_g =0 nil
                                    and e->nom =0 d->fils_g->nom )
                                and existe( Union( f->sup_d,
                                                   set(f->term->fils_d),
                                    λe e->fils_g =0 nil
                                    and e->nom =0 d->fils_d->nom )
                                )  )
                    or  (     d->fils_g->fils_g =0 nil
                        and d->fils_d->fils_g <>0 nil
                        and existe( ED,
                                λf d->nom =0 f->term->nom
                                and existe( Union( f->sup_g,
                                                   set(f->term->fils_g),
                                    λe    ( e->fils_g =0 nil
                                            and  e->nom =0 d->fils_g->nom )
                                          or e->->fils_g <>0 nil )
                                and existe( Union( f->sup_d,
                                                   set(f->term->fils_d),
                                          λe e->fils_g <>0 nil ) )
                     or (     d->fils_d->fils_g =0 nil
                        and d->fils_g->fils_g <>0 nil
                        and existe( ED,
                                λf d->nom =0 f->term->nom
                                and existe( Union( f->sup_d,
                                                   set(f->term->fils_d),
                                    λe   ( e->fils_g =0 nil
                                           and e->fils_g->nom =0 d->fils_d->nom )
                                         or e->fils_g <>0 nil )
                                and existe( Union( f->sup_g,
                                                   set(f->term->fils_g),
                                          λe e->fils_g->fils_g <>0 nil ) )
                                ) > )
```

Résumé : Le point de départ de notre travail a été de doter un système de ge bases de données d'une composante Recherche d'Informations. Notre objectif permettre l'expression de requêtes spécifiques à la recherche d'informations e une requête aux documents de la base.

Un élément qui permet de traiter une part de ce problème est de proposer pour décrire les correspondances entre les structures représentant la séma documents et la requête sur ces documents. Actuellement, les SGBD les plus ap cette démarche sont les SGBD à Objets, nous avons donc choisi d'aider à la con correspondances entre objets complexes.

Nous avons découpé la conception de telles correspondances en deux part théorique utilisant la logique modale appliquée au modèle logique de r d'informations, et la seconde opérationnelle permettant par l'utilisation d' algébrique l'expression des éléments de la partie théorique. La dichotomie entr théoriques et opérationnelles permet la portabilité des concepts théoriques, et base pour la validation des correspondances réalisées.

Ce travail permet donc, pour un programmeur d'application sur un système d d'avoir une aide qui va de la formulation logique des correspondances au la autorise l'expression de ces correspondances.

Mots-clés : base de données, recherche d'informations, langage de req correspondance d'objets, logique modale.

Abstract : The starting point of our work was to provide a DataBase Manag System with a Information Retrieval capability. Accordingly, our goal is to a expression of Information Retrieval queries by comparing a query to the docun base.

We can solve a part of this problem by allowing facilities to describe the r between the structures of the documents and queries semantics. For now, w matching design into two parts: a theoretic one using the logic model of I n Retrieval instanciated with the modal logic, ans an operational one based on a language expressing the elements of the theoretical part. The dichotomy be theoretical part and the operational one allows the portability of the theoretical well as it gives a base for the validation of the matching designed.

This work helps an application programmer at a theoretical level to expre implications between objects as weel as at an operational level by providing a l define matching between objects. The use of such matching as query predicates the query langugaes of these systems, by at last takig into account the objects' s a whole in queries.

Key-words : database, information retrieval, query langage, matching of o modal logic.