Modélisation et interrogation de bases de données lexicales

Jacques Le Maitre*—Nancy Ide—Jean Véronis*****

**Groupe Représentation et Traitement des Connaissances, CNRS*
31, chemin Joseph Aiguier, 13402 Marseille Cedex 20

***Department of Computer Science, Vassar College*
Poughkeepsie, New York 12601, USA.

****Laboratoire Parole et Langage, URA 261 du CNRS*
Université de Provence
29, avenue Robert Schuman, 13621 Aix-en-Provence Cedex 1, France

RÉSUMÉ. *Les bases de données textuelles constituent un domaine d'application encore peu exploré pour les SGBD à objets. L'objectif de cet article est de montrer quels sont les apports mais aussi les limites de ces systèmes pour gérer une catégorie particulière de bases de données textuelles : les bases de données lexicales construites à partir des informations contenues dans les dictionnaires. Nous décrivons deux des plus récents modèles proposés pour les dictionnaires et étudions l'utilisation de O2, un SGBD à objets, pour implanter et interroger des dictionnaires représentés selon ces deux modèles. Notre étude met en évidence les progrés qui restent à accomplir tant sur la modélisation des textes que sur la manipulation des objets complexes.*

ABSTRACT. *Textual databases are a domain of application of object oriented DBMS which is still not very explored. The aim of this paper is to show the contribution as well as the limitations of object oriented systems for the management of a particular class of databases : the lexical databases which are built from information contained in dictionaries. We describe two of the most recent models proposed for dictionaries then we study the use of O2, an object oriented DBMS, for implementing and querying lexical databases represented in accordance with these two models. Our study outlines progress which have to be made on text modelization as well as on complex object manipulation.*

MOTS-CLÉS : *bases de données lexicales, bases de données textuelles, dictionnaires électroniques, systèmes de gestion de bases de données à objets, objets complexes.*

KEY WORDS : *lexical databases, textual databases, electronic dictionaries, object oriented database management systems, complex objects.*

1. Introduction

Une des raisons invoquées pour justifier le développement des systèmes de gestion de bases de données post-relationnels était de répondre aux besoins des nouvelles applications parmi lesquelles les plus citées étaient la CAO ou la géomatique. Il semble que cet objectif commence à être atteint. Il reste cependant pour ces nouveaux systèmes des domaines d'applications encore peu explorés parmi lesquels les bases de données textuelles. Le travail présenté dans cet article concerne une catégorie particulière de base de données textuelles : les dictionnaires[1] .

Depuis quelques années, des dictionnaires de plus en plus nombreux sont disponibles sous forme électronique. Ce fut d'abord le cas auprès d'un public restreint de chercheurs, philologues ou linguistes. Ceux-ci ont vu dans les dictionnaires informatisés une ressource de premier ordre pour l'étude du langage et de son histoire, ainsi que pour la création de lexiques et bases de connaissances pour le traitement automatique du langage naturel. Plus récemment, des dictionnaires ont commencé à être commercialisés auprès du grand public, sous forme de CD-ROM pour un usage sur ordinateur individuel, ou sur consoles spécialisées. Tout porte à croire que ce type de produit est amené à se développer massivement dans les années qui viennent.

Bien que souvent pourvus d'interfaces graphiques assez conviviales, les logiciels d'interrogation de ces dictionnaires électroniques sont pour l'instant relativement sommaires quant aux fonctionnalités offertes. S'ils constituent un progrès par rapport à la consultation manuelle des dictionnaires papiers (en offrant par exemple des fonctions de recherche inverse : extraire tous les mots contenant le mot *x* dans leur définition), ils sont bien loin de permettre d'exploiter pleinement toute la richesse des informations contenues dans les dictionnaires. Une des raisons fondamentales pour lesquelles des fonctions de recherche complexes sont difficiles à implémenter tient à la nature même des données lexicales. Les dictionnaires sont en effet des objets composites, à la fois *textes* et *bases de données.* Ce sont bien des textes : ils sont linéaires, font appel à un large registre de conventions typographiques et d'abréviations, contiennent des définitions ou des exemples rédigés dans une langue naturelle, etc. Mais, en même temps, personne ne lit un dictionnaire de façon linéaire, de A jusqu'à Z. L'accès à l'information (même dans le cas d'une recherche manuelle dans un dictionnaire papier) se fait par l'intermédiaire d'une *clé* (le mot défini), et on cherche typiquement des *valeurs* associées à certains *attributs* (orthographe, prononciation, définition, etc.). On peut voir ces deux aspects, texte et base de données, comme une *structure de surface* (la forme) et une *structure profonde* (le contenu informationnel).

La discordance entre ces deux structures est parfois très importante. Une des raisons principales de cette discordance tient aux contraintes d'espace et de lisibilité propres aux dictionnaires. Ainsi dans l'exemple donné figure 1, l'information "v. i."

---

[1] Ce travail a été réalisé dans le cadre d'un projet conjoint entre le GRTC et le Vassar College aux Etats-Unis, soutenu par un accord collaboratif CNRS/National Science Foundation.

(verbe intransitif) s'applique aux deux premiers sens, et l'information "v. t." (verbe transitif) s'applique aux deux derniers[2] . Ces informations ne sont pas répétées pour chacun des sens. Etant donnée la structure hiérarchique des entrées, qui peut contenir jusqu'à cinq niveaux d'imbrication de sens, la factorisation des informations est parfois complexe, et demande clairement de la part du lecteur une *interprétation* de la structure de surface.

LOGER v.i. **1.** Avoir sa résidence permanente ou provisoire quelque part, habiter. **2.** Trouver place. ♦ v.t. **1.** Procurer un lieu d'habitation, un abri à. **2.** Faire entrer, faire pénétrer, mettre.

Figure 1. Exemple de factorisation

Il n'est donc pas étonnant que la construction de bases de données lexicales selon des modèles relationnels, classique [NAK 88], [BLA 92] ou non-normalisé [NEF 88] se soit heurtée à de gros problèmes (pour une discussion plus détaillée voir [IDE 93a]). Les articles de dictionnaires sont en fait des *objets complexes*. et donc les *systèmes de gestion de bases de données à objets* dont le premier objectif annoncé est la manipulation d'objets complexes [ATK 89] devraient être mieux adaptés aux bases de données lexicales que les systèmes relationnels.

L'objectif de cet article est de montrer quels sont les apports mais aussi les limites des systèmes à objets pour la gestion de bases de données lexicales. Pour cela nous avons choisi deux des plus récents modèles proposés pour les dictionnaires et nous avons étudié l'implantation en O_2, un système de gestion de bases de données à objets [DEU 91] [ADI 93], et l'interrogation de bases de données lexicales construites selon ces deux modèles. Ces deux modèles sont respectivement des modèles de la structure de surface et de la structure profonde des dictionnaires. Le premier modèle choisi [GON 87] repose sur une idée séduisante : représenter un texte par la grammaire qui le structure. Les auteurs de ce modèle ont aussi proposé un langage et un jeu d'opérateurs spécifiques. Le second modèle [VER 92] est basé sur un formalisme inspiré des théories linguistiques, les *structures de traits*. Son intérêt est de permettre la représentation des factorisations complexes des articles sans perte du contenu informationnel, c'est-à-dire qu'il permet de garder l'équivalence informationnelle d'articles présentés différemment dans leur structure de surface. Quant au système O_2 nous l'avons choisi car il apporte en plus des concepts classiques d'identité d'objet, d'encapsulation et d'héritage, la possibilité de construire des valeurs structurées ainsi qu'un langage de requêtes puissant : O_2SQL.

---

[2] Tous les exemples de ce papier ont été tirés de l'édition 1992 du *Petit Larousse.* Les articles ont parfois été quelques peu simplifiés (par exemple, par la suppression des exemples).

2. Modèle basé sur les grammaires

2.1. Description

Dans [GON 87] Gonnet et Tompa, présentent le modèle et les opérateurs qu'ils ont défini pour pour traduire en base de données et manipuler, le "Oxford English Dictionary". Ce modèle repose sur deux idées :

1) Les articles d'un dictionnaire sont des textes composés selon des règles qui peuvent être traduites sous forme d'une grammaire. Le schéma de la base de données peut être exprimé par cette grammaire et les articles peuvent être décrits par leurs arbres syntaxiques.
2) Les arbres syntaxiques étant de profondeur variable, les opérateurs qui les manipulent doivent pouvoir, dans un arbre, identifier les sous-arbres auxquels ils s'appliquent soit par leur position dans cet arbre soit par leur contenu indépendamment de cette position.

2.1.1. Le modèle de données

Le schéma d'une base de données lexicale est une grammaire exprimée sous forme EBNF (*Extended Backus Naur Form*) c'est à dire une grammaire non contextuelle dans laquelle la partie droite des règles est composée d'expressions régulières[3]. La figure 2 présente une grammaire très simplifiée du Petit Larousse. Les symboles *orth*, *gram*, *pron*, *etym*, *num*, *def* et *ex* sont les abréviations respectives de : orthographe, catégorie grammaticale, prononciation, étymologie, numéro de sens, définition et exemple et les symboles + ? * et | sont employés dans leurs acceptations habituelles (une ou plusieurs répétitions pour +, présence facultative pour ?, zéro, une ou plusieurs répétitions pour *, disjonction pour |). Les règles de grammaire devraient descendre jusqu'au niveau du caractère et prendre en compte la typographie mais pour des raisons de simplification seules les règles les plus hautes ont été données.

```
dictionnaire := article+
article := orth gram pron? etym? sens
sens := (num sens)+ | def ex*
```

Figure 2. *Grammaire (très) simplifiée du Petit Larousse*

---

[3] Pour des raisons pratiques d'analyse (voir ci-après l'opérateur *parsed_by*), Gonnet et Tompa imposent que la grammaire génère un langage régulier. Il y a toutefois un flou dans l'article cité, dans la mesure où cette question est indécidable pour les grammaires non contextuelles. Selon nous, on peut certainement accepter des grammaires LR(1) ou LALR(1), ces propriétés étant, elles, décidables.

Une instance est représentée par son arbre syntaxique (appelée *p-string* dans le modèle de Gonnet et Tompa). L'arbre syntaxique représentant l'article "définir" est donné par la figure 3. Remarquons que ces arbres syntaxiques sont des structures originales par rapport à celles qui sont habituellement traités dans les bases de données. Chaque nœud est étiqueté par un symbole de la grammaire (*etym* par exemple) et peut avoir plusieurs fils de même étiquette. Pour manipuler ces arbres, Gonnet et Tompa ont élaboré un langage fonctionnel basé sur un jeu d'opérateurs spécifiques.

DÉFINIR v.t. (lat. *definire*) **1**. Donner la définition de. *Définir le triangle*. **2**. Préciser, fixer ; cerner. *Définir une politique. Un état difficile à définir.*

```
article +— orth ———————————— "définir"
        +— gram ———————————— "v. t."
        +— etym ———————————— "(lat. definire)"
        +— sens +— num ————————— "1"
                +— sens +— def —— "Donner la définition de."
                |       +— ex ——— "Définir le triangle."
                +— num ————————— "2"
                +— sens +— def —— "Préciser, fixer ; cerner."
                        +— ex ——— "Définir une politique."
                        +— ex ——— "Un état difficile à définir."
```

Figure 3. Arbre syntaxique ou *p-string*

2.1.2. Les opérateurs

Deux groupes d'opérateurs ont été définis. Les premiers assurent la traduction entre textes et arbres syntaxiques, les seconds sont des extensions aux arbres des opérateurs de l'algèbre relationnelle. Dans les exemples suivants nous supposerons l'existence de deux variables : *definir* et *Petit_Larousse* auxquelles sont affectés respectivement l'arbres syntaxique de l'article "définir" et la liste de tous les arbres syntaxiques du Petit Larousse, générés par la grammaire de la figure 2.

Les opérateurs *parsed_by* et *string* permettent de convertir une chaîne de caractères en arbre syntaxique et réciproquement. Par exemple:

```
"Donner la définition de. Définir le triangle." parsed_by sens
= sens +— def —— "Donner la définition de."
       +— ex ——— "Définir le triangle."
```

L'opérateur *reparsed_by* permet de transformer un arbre syntaxique en appliquant une nouvelle grammaire à certaines parties de la chaîne qu'il représente. Cet opérateur a pour arguments un arbre syntaxique *S* et une grammaire *G* et produit l'arbre syntaxique obtenu en remplacant chaque sous-arbre *T* de *S* ayant pour racine un symbole non terminal *L* de *G*, par (*string T*) *parsed_by L*. Cette idée d'intégrer l'analyseur syntaxique aux opérateurs de manipulation est un point fort du modèle de Gonnet et Tompa, car cela permet de raffiner ou de restructurer une définition selon les besoins d'une application. Supposons que l'on soit intéressé par les mots contenus dans un exemple on pourra réanalyser celui-ci par la grammaire :

```
Phrase = {ex := (mot sep)+}.
```

Par exemple :

```
(ex ── "Définir le triangle.") reparsed_by Phrase =
= ex +─ mot ── "Définir"
     +─ sep ── " "
     +─ mot ── "le"
     +─ sep ── " "
     +─ mot ── "triangle"
     +─ sep ── "."
```

L'opérateur *in* a pour arguments une étiquette *L* et un arbre syntaxique *S* et produit le premier sous-arbre de racine *L* obtenu en balayant *S* en "profondeur d'abord". L'opérateur *every...in* a lui pour valeur la séquence de tous les sous-arbres de racine *L* ainsi rencontrés. Par exemple :

```
def in definir = def ── "Donner la définition de."

every mot in (definir reparsed_by Phrase)
= vector +─ mot ── "Définir"
         +─ mot ── "le"
         +─ mot ── "triangle"
         +─ …
```

où le symbole *vector* introduit une séquence de sous-arbres.

L'opérateur *where* est semblable à l'opérateur de sélection du modèle relationnel. Il a pour arguments un arbre syntaxique *S* et une fonction boooléenne *p* et il produit l'arbre syntaxique obtenu en supprimant chaque sous-arbre *T* de la racine de *S* tel que $p(T) = faux$. Par exemple, le programme suivant extrait tous les articles du Petit Larousse qui sont des verbes transitifs.

```
verbe_transitif =
   proc(x) (root x = orth) and (gram in x) = "v. t." ;

Petit_Larousse where verbe_transitif()
```

sachant que l'opérateur root a pour argument un arbre syntaxique et en produit la racine et que $f(x_1, \ldots, x_i\text{-}1, , x_{i+1}, \ldots, x_k)(x)$ est équivalent à $f(x_1, \ldots, x_i\text{-}1, x, x_i+1, \ldots, x_k)$.

L'opérateur *transduced_by* est une extension aux arbres de l'opérateur de projection du modèle relationnel. Il a pour arguments un arbre syntaxique S et une grammaire G et produit l'arbre syntaxique obtenu en transformant S de la façon suivante. Pour chaque règle $L := R_1, \ldots, R_n$ de G, chaque sous-arbre T de racine L est remplacé par un arbre dont la racine est L et a pour sous-arbres $T_1, \ldots, T_n$ où $T_i = (R_i \text{ in } T)$ si R_i est un symbole non-terminal de G et $T_i = R_i$ si R_i en est un symbole terminal. Aprés cette opération il peut être nécessaire de supprimer certains nœuds devenus inutiles. C'est ce que permet l'opérateur *suppressing* qui a pour arguments un arbre syntaxique S et un ensemble d'étiquettes E et qui produit l'arbre syntaxique obtenu en supprimant de S tous les nœuds dont l'étiquette appartient à E, les nœuds fils de chaque nœud supprimé étant directement raccordés à leur nœud grand-père. Par exemple la fonction qui extrait du Petit Larousse l'ensemble des vedettes (les mots définis) peut s'écrire :

```
(Petit_Larousse transduced_by {article := orth})
   suppressing {article, orth}
```

et a pour valeur :

```
dictionnaire +— "a"
             +— "à"
             +— "abaca"
             +— …
             +— "définir"
             +— …
```

L'opérateur *partitioned_by* est semblable à l'opérateur *group_by* de SQL. Il s'applique à un arbre syntaxique et à une fonction f et produit l'ensemble de tous les vecteurs $[v, S \textit{ where } f() = v]$ où v appartient à l'ensemble des $f(T)$ tels que T est un sous-arbre de la racine de S. Le programme suivant, par exemple, réalise une partition de l'ensemble des articles du Petit Larousse par catégorie grammaticale :

```
Gram = proc(x) string (gram in x) ;

((Petit_Larousse partitioned_by Gram())
   transduced_by {article := orth})
      suppressing {article, orth}.
```

Il produit l'arbre syntaxique suivant :

```
set +— …
    +— vector +— "v. t."
              +— "dictionnaire" +— …
                                +— "définir"
                                +— …
```

sachant qu'un ensemble d'arbres syntaxiques $\{S_1, \ldots, S_n\}$ est représenté par un arbre syntaxique dont la racine est *set* et a pour sous-arbres $S_1, \ldots, S_n$.

D'autres opérateurs ont été définis mais nous avons décrit les principaux.

2.1.3. Discussion

L'apport du modèle de Gonnet et Tompa est double. Le premier apport est l'idée de schématiser une base de données textuelle par la grammaire qui structure les textes qu'elle contient et d'intégrer l'analyseur syntaxique aux opérateurs de manipulation de la base de données. Le second apport est d'avoir proposé des opérateurs spécifiques pour la manipulation de données arborescentes. Dans la suite de cet article ces opérateurs seront appelés "opérateurs en profondeur".

Ce modèle est particulièrement intéressant vue l'importance grandissante des bases de données textuelles, et du développement rapide de SGML (*Standard Generalized Markup Language*) [ANS 86] comme standard d'encodage des documents informatisés dans toutes sortes de domaines (lexicographie, bases terminologiques, bureautique, documentation, textes littéraires et données linguistiques), comme en attestent des actions internationales telles que la *Text Encoding Initiative* [IDE 93b]. Or, un document SGML est défini par une *Document Type Definition* qui n'est ni plus ni moins qu'une grammaire EBNF définissant la structure du document. Des opérateurs du type de ceux définis par Gonnet et Tompa pourraient donc avoir un grand intérêt pour la consultation et la manipulation de documents en ligne.

Toutefois un problème sémantique se pose : la grammaire qui décrit un texte en traduit la structure de surface et non la structure profonde. Elle n'est donc pas toujours le schéma le plus approprié de ce texte comme nous le montrerons section 3. Il est vrai que l'opérateur *reparsed_by* permet d'obtenir de multiples représentations d'un même texte. Un bel exemple en est d'ailleurs donné dans [GON 87] avec le traitement des étymologies dans l'"Oxford English Dictionary". Toutefois, il n'est pas certain qu'une grammaire non ambiguë puisse être trouvée pour toute les ré-analyses nécéssaires d'un texte. Ce problème de discordance entre la

structure de surface et la structure profonde d'un texte est clairement mis en évidence dans un travail récent sur la manipulation de textes structurés à l'aide d'outils "bases de données" [ABI 93]. La solution proposée consiste à associer au texte un schéma de bases de données qui traduise la structure profonde de ce texte et à spécifier le lien entre le texte et la base de données en annotant la grammaire qui traduit la structure de surface de ce texte.

Une autre critique que l'on peut faire concerne le langage de manipulation des données. Pour les opérations classiques de sélection, de projection et de groupement il est moins pratique qu'un langage de requêtes tel que SQL. Il est d'autre part incomplet car il lui manque un opérateur de jointure. Il est donc intéressant d'étudier l'intégration de ce modèle au sein d'un système de gestion de bases de données car si l'expérience s'avérait positive cela permettrait de faire l'économie du développement, toujours coûteux, de systèmes et de langages spécifiques. C'est l'objet du paragraphe suivant.

2.2. Implantation

2.2.1. Schéma

Nous proposons de traduire une grammaire en un schéma O_2 en associant à chaque partie droite d'une règle, un type ayant pour nom la partie gauche de cette règle. Cette traduction peut être réalisée en trois étapes. La première étape consiste à transformer chaque règle de grammaire $s := d$ en une déclaration de la forme ***type*** $s = t$ où t est le type O_2 obtenu en appliquant à d la fonction *type* définie de la façon suivante :

1) si d est un symbole terminal, alors *type*(d) est le type atomique correspondant dans le modèle O_2,
2) si d est un symbole non terminal alors *type*(d) = d,
3) si d est de de la forme δ? alors *type*(d) = *type*(δ),
4) si d est de la forme δ+ ou δ* alors *type*(d) = *list*(t) où t = *type*(δ),
5) si d est une liste de termes δ_1, …, δ_k alors *type*(d) = *tuple*(n_1: t_1, …, n_k: t_k) où pour i = 1 à k, t_i = *type*(δ_i) et n_i est un nom unique donné au terme δ_i,
6) si d est une disjonction de termes $\delta_1 \mid \ldots \mid \delta_k$ alors *type*(d) = *union*(t_1, …, t_k) où pour i = 1 à k, t_i = *type*(δ_k),
7) si d est de la forme (δ) alors type(d) = type(δ).

En appliquant cette transformation à la grammaire du Petit Larousse (figure 2), on obtient la liste de déclarations suivante :

```
type Dictionnaire = list(Article)
type Article =
   tuple(orth: Orth, gram: Gram, pron: Pron,
         etym: Etym, sens: Sens)
type Sens = union(
   list(tuple(num: Num, sens: Sens)),
   tuple(def: Def, ex:list(Ex)))
```

La deuxième étape consiste à simuler les types *union*, qui n'existent pas en O_2, au moyen de l'héritage. A chaque type *union*(T_1, …, T_k) on associe une classe C (de même nom si ce type est nommé) et k classes C_1, …, C_k sous-classes de C et de types T_1, …, T_k. Puis l'on ajoute la déclaration de ces classes aux déclarations obtenues à la fin de la première étape. Dans le cas du Petit Larousse les déclarations suivantes seront ajoutées :

```
class Sens
class Sens_compose inherit Sens type
   list(tuple(num: Num, sens: Sens))
class Sens_simple inherit Sens type
   tuple(def: Def, ex: list(Ex)).
```

Dans un schéma O_2 les types ne sont utilisés que pour définir les valeurs des objets d'une classe. La seule façon de définir un type est donc de définir une classe qui l'encapsule. La troisième et dernière étape consiste donc à remplacer chaque déclaration ***type*** $T = t$ par la déclaration ***class*** T ***type*** t. La déclaration en O_2 de la grammaire du Petit Larousse est finalement celle de la figure 4.

```
class Dictionnaire type list(Article)
class Article type
   tuple(orth: Orth, gram: Gram, pron: Pron,
         etym: Etym, sens: Sens)
class Sens
class Sens_compose inherit Sens type
   list(tuple(num: Num, sens: Sens))
class Sens_simple inherit Sens type
   tuple(def: Def, ex: list(Ex))
```

Figure 4. *Traduction en O_2 de la grammaire du Petit Larousse*

La transformation qui vient d'être décrite a pour effet d'associer à chaque symbole non terminal d'une grammaire, et donc à chaque règle de grammaire, une classe dont les instances sont les arbres syntaxiques générés par cette règle de grammaire. Il faut remarquer qu'appliquée à une grammaire purement BNF (i. e. sans expressions régulières dans les parties droites de règles) cette transformation produirait un schéma

trés inefficace car profondemment récursif : les ensembles et les listes y seraient représentés sous forme d'arbres. La même constatation est faite dans [ABI 93]. Toutefois un tel schéma pourrait être amélioré en appliquant des transformations semblables à celles qui permettent de traduire les récursions terminales en itération.

Les opérations *parsed_by* et *string* du modèle de Gonnet et Tompa seront définis de la façon suivante. Soit *G* une grammaire et sa classe associée et *String* la classe dont les objets ont pour valeur les chaînes de caractères apparaissant dans les articles d'un dictionnaire, les deux méthodes suivantes seront déclarées :

```
method to_G: G in class String
method to_string: string in class G
```

Ces deux méthodes seront programmées de façon à ce que : appliquée à une chaîne *s* instance de *String* la méthode *to_G* analyse *s* et génère son arbre syntaxique *a* instance de *G* et inversement appliquée à *a* la méthode *to_string* retourne la valeur de *s*. Par exemple, à la grammaire {*Phrase* := (*mot sep*)+} seront associées les classes et les méthodes suivantes :

```
class Phrase type list(tuple(mot: Mot, sep: Sep))

method to_Phrase in class String
method to_Mot in class String
method to_Sep in class String
method to_string in class Phrase
method to_string in class Mot
method to_string in class Sep.
```

Une façon de programmer la méthode *to_G* serait de faire appel à l'analyseur syntaxique Yacc dont une interface avec O_2, nommée O_2Yacc, a déjà été réalisée [ABI 93].

2.2.2. Interrogation

Occupons nous maintenant de l'interrogation de la base de données ainsi obtenue. Nous supposerons pour cela que le Petit Larousse est un objet de la classe *Dictionnaire* accessible par le nom *Petit_Larousse*.

Toutes les opérations qui s'appliquent à des sous-arbres dont la position est donnée s'expriment facilement en O_2SQL et souvent de façon plus concise qu'avec le langage proposé par Gonnet et Tompa. Ainsi la fonction qui donne accés à l'article associé au mot "définir" s'écrira :

```
define definir as element(
   select a
   from a in Petit_Larousse
   where a.orth.to_string = "définir")
```

où *element* est l'opérateur qui sélectionne l'unique objet d'un ensemble. On remarquera dans cette requête et les suivantes, l'utilisation de la méthode *to_string* pour convertir en chaîne de caractères un arbre syntaxique. La requête qui donne la liste des articles qui sont des verbes intransitifs s'écrira :

```
select a.orth.to_string
from a in Petit_Larousse
where a.gram.to_string = "v. t."
```

et celle qui regroupe les mots par catégorie grammaticale s'écrira :

```
group a in Petit_Larousse
by (gram: a.gram.to_string)
with (mots: select p.orth.to_string from p in partition).
```

L'opérateur *group x in e by* t_1 *with* t_2 où t_1 et t_2 sont des fonctions à valeurs "n-uplets", réalise une partition de l'ensemble *e* qui a pour valeur $\{k \text{ conc } t_2(\textit{partition}) \mid k \in \{t_1(x) \mid x \in e\}\}$ où $\textit{partition} = \{x \in e \mid t_1(x) = k\}$.

Par contre les opérations en profondeur dans l'arbre syntaxique s'expriment beaucoup plus difficilement que dans le langage de Gonnet et Tompa, car O_2 n'offre pas d'opérateurs en profondeur tels que *in*, *every…in*, *transduced_by* ou *suppressing*. Même si ceux-ci pouvaient être programmés à bas niveau, ils ne pourraient pas être utilisés dans O_2SQL car le jeu d'opérateurs offert par ce langage n'est pas extensible. Les opérations en profondeur devront donc être programmées de façon récursive ce qui, comme le montrera l'exemple suivant, nuit à la concision des requêtes et certainement aussi à leurs performances. Considérons par exemple la requête permettant d'obtenir l'ensemble des mots contenus dans les exemples cités dans les différents sens du mot "définir". Dans le langage de Gonnet et Tompa elle s'écrit de façon concise :

```
every mot in definir reparsed_by Phrase
```

grâce à l'opérateur *every…in* qui recherche en profondeur les sous-arbres dont la racine est donnée. Pour l'exprimer en O_2SQL il faut tout d'abord créer deux méthodes : l'une dans la classe *Sens_simple* qui retourne la liste des mots des exemples d'un sens simple et l'autre dans la classe *Sens_compose* qui retourne la liste des mots de tous les sens d'un sens composé. Le corps de ces méthodes pourra lui être écrit avec une requête O_2SQL. La réponse à la requête ci-dessus sera alors obtenue par le programme suivant :

```
method body mots: set(Mot) in class Sens_simple {
   query := o2query(
      mots,
      "(select ms.mot
        from e in $1.ex, ms in e.to_string.to_Phrase)",
      self) ;
   return(mots) ;
}

method body mots: set(Mot) in class Sens_compose {
   query := o2query(
      mots, "(select m from s in $1, m in s.mots)", self) ;
   return(mots) ;
}

select m.to_string from a in Petit_Larousse, m in a.sens.mots
```

(où la fonction *o2query*(r, q, p_1, …, p_n) évalue dans r la requête q aprés y avoir remplacé chaque paramètre formel $\$i$ par sa valeur effective p_i).

Une autre façon de traduire en O_2 le modèle de Gonnet et Tompa aurait consisté à définir un arbre syntaxique comme un objet de la classe :

```
class Arbre_S
   type tuple(racine: string, feuilles: list(Arbre_S))
```

En ce cas les opérateurs en profondeur auraient pu être définis comme des méthodes de la classe *Arbre_S* car les étiquettes des sous-arbres à sélectionner auraient été des chaînes de caractères et auraient donc pu être des arguments de ces méthodes. Une telle représentation aurait eu l'inconvénient de supprimer tout typage puisque les symboles non terminaux qui structurent une grammaire auraient été traités comme des valeurs d'attributs alors qu'elles sont traités comme des types dans la traduction que nous avons proposée.

A notre connaissance aucun des langages de requêtes offerts par les systèmes de gestion de bases de données à objets actuellement disponibles ne disposent d'opérateurs spécifiques pour la manipulation d'arbres ou plus généralement de graphes. Une évolution de ces langages dans cette direction serait donc particulièrement souhaitable. Remarquons que ces opérateurs pourraient être définis en O_2SQL si ce langage, qui est un langage fonctionnel, traitait les fonctions comme des objets à part entière. Malheureusement ce n'est pas le cas. La construction d'un langage de requêtes ainsi extensible est étudiée dans [LEM 92].

3. Un modèle basé sur les structures de traits

3.1. Description

La grammaire qui décrit un article d'un dictionnaire ne donne qu'une caractérisation de surface de cet article. En effet pour des raisons de lisibilité et de compactage de l'information les articles d'un dictionnaire sont présentés selon une structure sémantique hiérarchisée qui permet de factoriser les caractéristiques communes à plusieurs sens. Les règles de factorisation varient d'un dictionnaire à l'autre mais aussi à l'intérieur d'un même dictionnaire. Considérons par exemple les définitions des mots "orange" et "rose" dans le Petit Larousse (figure 5).

ORANGE n. f. (de l'ar. *narandj*). **1.** Fruit comestible de l'oranger, d'un jaune mêlé de rouge. **2.** *Orange amère* : bigarade ♦ adj. inv. et n. m. De la couleur de l'orange (mélange de rouge et de jaune).

1. ROSE n. f. (lat. *rosa*) **I.1** Fleur du rosier. …
2. ROSE adj. **1.** Qui a la couleur pourpre pâle de la rose commune. …
3. ROSE n. m. Couleur rose. …

Figure 5. *Factorisations différentes*

Une constatation immédiate s'impose : les sens associés aux catégories grammaticales "n. f.", "adj." et "n. m.", bien que de même nature, ne sont pas placés de la même facon pour chacun de ces deux mots. Pour le mot "orange" ils sont regroupés dans le même article qui est découpé en deux variantes alors que pour le mot "rose" ils font l'objet d'articles séparés (1, 2 et 3). Choisir comme schéma de la base de données lexicale la grammaire de surface qui traduit une telle organisation conduirait à générer des instances de structures irrégulières et rendrait donc difficile l'interrogation de cette base de données.

C'est ce qui a amené Véronis et Ide à définir un nouveau modèle basé sur les structures de traits [VER 92]. Ce modèle permet de représenter aussi fidèlement que possible les articles d'un dictionnaire et propose une forme canonique de cette représentation, sous laquelle les articles peuvent être comparés.

3.1.1. Définition d'une structure de traits

Les structures de traits ont été développées comme un formalisme de représentation des grammaires [SHI 86]. Une *structure de traits* a la forme générale suivante :

$[T_1, \ldots, T_n]$

où chaque T_i est :

- soit une paire $n : t$ dans laquelle n est un nom de *trait* et t sa valeur éventuellement identifiée,
- soit un identificateur de valeur de trait (les structures de traits peuvent donc être réentrantes),
- soit une disjonction de structures de traits appelée *disjonction générale*.

La valeur d'un trait peut être :

- soit atomique,
- soit une structure de traits,
- soit une disjonction de traits, appelée *disjonction de valeurs*.

Une disjonction peut être soit ordonnée : en ce cas elle est notée <…>, soit non ordonnée : en ce cas elle est notée {…}.

Les deux structures suivantes :

```
[n1: a1, n2: {a21, a22}, <[n3: a31, n4: a41], [n3: a32]>]
[n1: a11,
 n2: (1) [n1: a12, n4: a41],
 n3: [n1: a13, n2: (1), n3: [n1: a14, n4: a42]]]
```

dans lesquelles chaque n_i est un nom de trait, chaque t_i est une valeur atomique et (1) est un identificateur de valeur de trait, sont des structures de traits.

3.1.2. Représentation d'un dictionnaire en structures de traits

Dans le modèle de Véronis et Ide, chaque article d'un dictionnaire est décrit par une structure de traits. Ces structures de traits sont moins générales que celles décrites dans le paragraphe précédent :

1) elles ne sont pas réentrantes,
2) elles sont typées [POL 87],
3) chaque structure possède au plus une disjonction générale formée par une disjonction de structures de traits de même type qu'elle (i. e. possèdant les mêmes attributs),
4) les disjonctions sont ordonnées, afin de préserver l'organisation séquentielle d'un article.

Une structure de traits est la représentation, factorisée ou non, d'une disjonction de traits d'un type donné. Un trait peut être soit atomique, soit composé. Un trait

atomique est une chaîne de caractères, un trait composé est un tuple [n_1: t_1, …, n_k: t_k] où n_1, …, n_k sont des noms de traits et t_1, …, t_k des traits atomiques ou composés.

Chaque trait possède un type. Soient A_1, …, A_k des ensembles de traits atomiques et N un ensemble de noms de traits, un type de trait est défini récursivement par :

- A_1, …, A_k sont des types de traits,
- soient n_1, …, n_k des éléments de N, et T_1, …, T_k des types de traits alors [n_1: T_1, …, n_k: T_k] est un type de trait.

Une structure de traits peut avoir l'une des cinq formes suivantes :

1) soit A un type de trait atomique et a un trait de type A alors a est une structure de traits de type A.
2) soit C = [n_1: T_1, …, n_k: T_k] un type de trait composé et s_1, …, s_k des structures de traits de types respectifs T_1, …, T_k alors [n_1: s_1, …, n_k: s_k] est une *structure de traits non factorisée* de type C.
3) soit C = [n_1: T_1, …, n_k: T_k] un type de trait composé, s_1, …, s_k des structures de traits de types respectifs T_1, …, T_k et d_1, …, d_m des structures de traits de type C, alors [n_1: s_1, …, n_k: s_k, <d_1, …, d_m>] est une *structure de traits factorisée* de type C. Dans cette structure [n_1: s_1, …, n_k: s_k] est appelé le *facteur commun* et <d_1, …, d_m> la *partie disjonctive*.
4) soit T un type de trait et d_1, …, d_m des structures de traits de type T, alors <d_1, …, d_m> est une *structure de traits disjonctive* de type T.
5) un *trait nul* (i. e. un trait non attesté) est une structure de traits. Dans les exemples qui suivent les traits nuls d'une structure de traits seront omis dans l'écriture de celle-ci. Un trait nul est une valeur polymorphique qui appartient à chaque type de trait.

Dans ce paragraphe et le suivant nous supposerons que les types de traits atomiques suivants ont été définis :

```
Orth, Pron, Etym, Cat, Genre, Num, Def, Ex
```

ainsi que les types de traits composés suivants :

```
Article: [forme: Forme, etym: Etym, gram: Gram, num: Num,
          def: Def, ex: Ex]
Forme: [orth: Orth, pron: Pron]
Gram: [cat: Cat, genre: Genre].
```

La figure 6 montre la structure de traits représentant l'article *agnelet*. Les traits *pron*, *etym*, *num* et *ex* y sont nuls.

AGNELET n. m. Petit agneau.

```
[forme: [orth: "agnelet"],
 gram:  [cat: "n.", genre: "m."],
 def: "Petit agneau"]
```

Figure 6. *Représentation de l'article "agnelet"*

La figure 7 montre la structure de traits représentant l'article "orange" (voir figure 5). Dans cette représentation il faut remarquer :

1) L'utilisation de structures de traits factorisées pour représenter l'imbrication de sens : ceux de fruit et ceux de couleur. Les traits *form* et *etym* sont mis en facteur pour ces trois sens et le trait *gram* pour les deux sens de couleur.
2) L'utilisation d'une structure de traits disjonctive pour représenter les deux catégories grammaticales possibles pour une couleur : adjectif et nom.

```
[form: [orth: "orange"],
 etym: "(de l'ar. narandj).",
 <[gram: [cat: "n.", genre: "f."],
   <[num: "1", def: "Fruit comestible de l'oranger, …"],
    [num: "2", def: "Orange amère: bigarade"]>],
  [gram: <[cat: "adj.", genre: "inv."],
          [cat: "n.", genre: "m."]>,
   def:  "De la couleur de l'orange…"]>]
```

Figure 7. *Représentation de l'article "orange"*

Toute structure de traits possède une forme canonique qui est la *forme normale disjonctive* obtenue en développant toutes les factorisations dont elle fait l'objet. Elle se trouve alors sous la forme d'une disjonction de traits. Le développement d'une structure de traits *s* est réalisé par l'opération *unfac* définie de la façon suivante :

1) si *s* est un trait atomique alors :

   unfac(s) = <s>,

2) si *s* est une structure de traits non factorisée $[n_1: s_1, \ldots, n_k: s_k]$ alors :

   unfac(s) = $<[n_1: u_1, \ldots, n_k: u_k] \mid u_1 \in \text{unfac}(s_1), \ldots, u_k \in \text{unfac}(s_k)>$

3) si s est une structure de traits factorisée $[n_1: s_1, \ldots, n_k: s_k, <d_1, \ldots, d_m>]$ alors :

   unfac(s) = <[n_1: u_1, …, n_k: u_k] unify d |
   $u_1 \in$ unfac(s_1), …, $u_k \in$ unfac(s_k), d ∈ unfac(<d_1, …, d_m>)>

4) si s est une structure de traits disjonctive $<d_1, \ldots, d_m>$ alors :

   unfac(s) = <u | u ∈ unfac(d_1) ou … ou u ∈ unfac(d_m)>

5) si s est un trait nul alors :

   unfac(s) = <>.

Dans le cas 3 l'unification d'un trait du facteur commun avec un trait de la partie disjonctive ne peut être une simple concaténation car il peut arriver que ces deux traits aient des attributs communs de valeur non nulles. L'unification de deux traits est définie de la façon suivante :

1) L'unification de deux traits composés $[n_1: t_1, \ldots, n_k: t_k]$ et $[n_1: t'_1, \ldots, n_k: t'_k]$ a pour valeur :

   [n_1: (t_1 unify t'_1), …, n_k: (t_k unify t'_k)].

2) L'unification d'un trait nul avec un trait non nul (et vice versa) a pour valeur le trait non nul.
3) L'unification de deux traits atomiques non nuls a_1 et a_2 doit être définie pour chaque type de trait. La plupart du temps on aura $(a_1 \textit{ unify } a_2) = a_2$ en considérant que, dans un article, le trait le plus interne est une redéfinition du trait le plus externe. Les numéros de sens, par contre, seront concaténés afin de conserver la numérotation hiérarchisée des articles d'un dictionnaire (I.1, I.2, …).

La figure 8 donne la forme normale disjonctive de l'article "orange". Remarquons que les quatre sens du mot orange ont été énumérés : les deux de fruit et les deux de couleur (adjectif et nom). Notons aussi comment a fonctionné le développement : les traits f et e ont été distribués sur chacun des quatre sens, la catégorie grammaticale g_1 a été distribuée sur les deux premiers sens et la définition d a été distribuée sur les deux derniers.

```
<[form : f, etym: e, gram: g1, num: "1", def: "Fruit comest….“],
 [form : f, etym: e, gram: g1, num: "2", def: "Orange amère….“],
 [form : f, etym: e, gram: g2, def: d],
 [form : f, etym: e, gram: g3, def: d]>
```

où :

```
f =  [orth: "orange"]
e =  "(de l'ar. narandj)."
g1 = [cat: "n.", genre: "f."]
g2 = [cat: "adj.", genre: "inv."]
g3 = [cat: "n.", genre: "m."]
d =  "De la couleur de l'orange, ….".
```

Figure 8. *Forme normale disjonctive*

La forme normale disjonctive joue un rôle fondamental pour interroger une base de données lexicale car elle permet de définir une relation d'équivalence entre structures de traits. Deux structures de traits deviennent équivalentes si elles ont la même forme normale disjonctive, quelle que soit leur factorisation originale. C'est le plus souvent sous cette forme que les contenus informationnels de deux articles peuvent être comparés. Il est par exemple facile de vérifier que sous forme normalisée les articles "orange" et "rose" deviennent comparables : chacun des termes de leurs formes normales disjonctives associe (entre autres) une orthographe, une catégorie grammaticale et une définition.

Le modèle de Véronis et Ide a lui-aussi une forte relation avec l'encodage de documents en SGML. Il a servi de base au développement du standard d'encodage des dictionnaires informatisés proposé par la *Text Encoding Initiative* [IDE 93c]. Dans ce standard, la DTD est une grammaire qui définit la forme profonde du dictionnaire.

3.2. Implantation

3.2.1. Schéma

Comme cela a été dit au paragraphe précédent une structure de traits peut avoir cinq représentations différentes : atomique, non factorisée, factorisée, disjonctive ou nulle. Le schéma O_2 que nous proposons est donc le suivant. A chaque type de trait (*Form*, *Etym*, *Gram*, …) est associée une classe qui a autant de sous-classes que de représentations possibles. A chaque type de trait atomique *T* sont associées les trois classes suivantes :

```
class T
class T_nonfac inherit T type string
class T_disj inherit T type list(T_nonfac),
```

et à chaque type de trait composé T = [n_1: T_1, …, n_k: T_k] sont associées les quatre classes suivantes :

```
class T
class T_nonfac inherit T type tuple(n1: T1, …, nk: Tk)
class T_fac inherit T type
   tuple(common: T_nonfac, disj: T_disj)
class T_disj inherit T type list(T).
```

La figure 9 montre le schéma O_2 du Petit Larousse modélisé en structure de traits.

```
class Dictionnaire type list(Article)

class Article
class Article_nonfac inherit Article type
   tuple(forme: Forme, etym: Etym, gram: Gram,
         num: Num, def: Def, ex: Ex)
class Article_disj inherit Article type list(Article)
class Article_fac inherit Article type
   tuple(common: Article_nonfac, disj: Article_disj)

class Orth
class Orth_nonfac inherit Orth type string
class Orth_disj inherit Orth type list(Orth_nonfac)

name Petit_Larousse: Dictionnaire
```

(Les classes *Form* et *Gram* et leurs sous-classes sont définies de façon similaire à la classe *Article* et les classes *Pron*, *Etym*, *Cat*, *Genre*, *Num*, *Def*, *Ex* et leurs sous-classes sont définies de façon similaire à la classe *Orth*)

Figure 9. *Schéma O_2 du Petit Larousse*

Dans cette base de données l'article associé au mot "orange" sera représenté par un objet de la classe *Article* comme le montre la figure 10 dans laquelle : i_1, …, i_n sont des identifiants d'objets, la notation $i = C(v)$ indique que l'objet d'identifiant i appartient à la classe C et a la valeur v, et enfin les attributs ayant une valeur nulle n'ont pas été représentés.

```
i1 = Article_fac(tuple(common: i2, disj: i6))
i2 = Article_nonfac(tuple(forme: i3, etym: i5))
i3 = Forme_nonfac(tuple(orth: i4))
i4 = Orth_nonfac("orange")
i5 = Etym_nonfac("(de l'ar. narandj.)")
i6 = Article_disj(list(i7, i19))
i7 = Article_fac(tuple(common: i8, disj: i12))
i8 = Article_nonfac(tuple(gram: i9))
i9 = Gram_nonfac(tuple(cat: i10, genre: i11))
i10 = Cat_nonfac("n.")
i11 = Genre_nonfac("f.")
i12 = Article_disj(list(i13, i16))
i13 = Article_nonfac(tuple(num: i14, def: i15))
i14 = Num_nonfac("1.")
i15 = Def_nonfac("Fruit comestible de de l'oranger, ….")
i16 = Article_nonfac(tuple(num: i17, def: i18))
i17 = Num_nonfac("2.")
i18 = Def_nonfac("Orange amère: bigarade")
i19 = Article_nonfac(tuple(gram: i20, def: i27))
i20 = Gram_disj(list(i21, i24))
i21 = Gram_nonfac(tuple(cat: i22, genre: i23))
i22 = Cat_nonfac("n.")
i23 = Genre_nonfac("m.")
i24 = Gram_nonfac(tuple(cat: i25, genre: i26))
i25 = Cat_nonfac("adj.")
i26 = Genre_nonfac("inv.")
i27 = Def_nonfac("De la couleur de l'orange, ….")
```

Figure 10. *Objets représentant l'article "orange"*

3.2.2. Interrogation

Une base de données lexicale décrite selon le schéma précédent peut être facilement interrogée en O2SQL à condition de mettre les articles sous forme normale disjonctive et donc de disposer de l'opération *unfac* définie au paragraphe 3.1.2. Nous montrerons dans ce paragraphe comment programmer cette opération et donnerons quelques exemples de requêtes l'utilisant.

Le développement d'une structure de traits est réalisé par la méthode *unfac* définie sur chaque classe de traits *T* par :

```
method unfac: list(T_nonfac) in class T
method unfac: list(T_nonfac) in class T_nonfac
method unfac: list(T_nonfac) in class T_disj
method unfac: list(T_nonfac) in class T_fac.
```

Appliquée à un objet de la classe *T* la méthode *unfac* retourne une liste de traits atomiques ou composés de la classe *T*.

La méthode *unfac* utilise la méthode *unify* qui doit être définie sur chaque classe *T* de traits non factorisés par :

```
method unify(t: T_nonfac): T_nonfac in class T_nonfac
```

Appliquée à un trait non factorisé t_1 avec pour argument un trait t_2 de la même classe, la méthode *unify* devra retourner un nouveau trait de cette classe dont la valeur est l'unification des traits t_1 et t_2 selon les règles définies au paragraphe 3.1.2.

La valeur de la méthode *unfac* dépend du type d'objet auquel elle s'applique : trait atomique, structure de traits non factorisée, factorisée ou disjonctive, ou trait nul :

1) appliquée à un trait atomique *a* la méthode *unfac* retourne *list*(*a*),
2) appliquée à un objet de la classe *T_nonfac* de valeur *tuple*(n_1: s_1, …, n_k: s_k) la méthode *unfac* retourne :

   list(u | $u_1 \in$ unfac(s_1), …, $u_k \in$ unfac(s_k)),

   où *u* est un nouvel objet de la classe *T_nonfac* de valeur :

   tuple(n_1: u_1, …, n_k: u_k),

3) appliquée à un objet *s* de la classe *T_fac* la méthode *unfac* retourne :

   list(unify(c, d) | c ∈ unfac(common(s)), d ∈ unfac(disj(s))),

4) appliquée à un objet *s* de la classe *T_disj* la méthode *unfac* retourne :

   list(u | d ∈ valeur(s), u ∈ unfac(d)),

5) appliquée à un trait nul la méthode *unfac* retourne :

   list().

Disposant de la méthode *unfac* la plupart des requêtes à une base de données lexicale s'expriment de façon simple car il n'est plus besoin de se préoccuper de la façon dont sont factorisés les articles du dictionnaire interrogé. Par exemple, pour

obtenir tous les articles du Petit Larousse qui sont des adjectifs il suffira d'écrire en O_2SQL :

```
select u.form.orth
from a in Petit_Larousse, u in a.unfac
where u.gram.cat = "adj."
```

Les mots "rose" et "orange" (fig. 5) répondront à cette requête bien que le mot "rose" sous forme d'adjectif fasse l'objet d'un article à part entière alors que pour le mot "orange" cette forme n'est qu'un sous-article de l'article unique consacré à ce mot. Autre exemple, les noms de fleurs qui sont employés comme adjectifs de couleur pourront être obtenus par la requête :

```
select f.form.orth from
   af in Petit_Larousse,
   ac in Petit_Larousse,
   f in af.unfac,
   c in ac.unfac
where
   f.gram.cat = "n." and
   f.def = "?fleur?" and
   c.gram.cat = "adj." and
   c.def = "?couleur?" and
   f.form.orth = c.form.orth.
```

en supposant qu'un nom de fleur possède le mot "fleur" dans sa définition et qu'un adjectif de couleur possède le mot "couleur" dans la sienne.

3.2.3 Développements futurs

La suite de ce travail va consister à définir un opérateur de *factorisation* (qui ne peut être traduit directement en un opérateur de groupement car la profondeur de la mise en facteur n'est pas connue à priori), puis à implanter à grande échelle ce modèle et ces deux opérateurs et enfin à réaliser des mesures de performances.

On peut d'ores et déjà dire que le problème sera celui du coût de la "défactorisation". Il semble peu raisonnable, en effet, de développer les factorisations de tous les articles d'un dictionnaire (plusieurs dizaines de milliers) à chaque requête ! Par ailleurs, un stockage sous forme normale disjonctive n'est ni pratique (il entraîne un accroissement important de la taille des données, comme c'est visible, même dans l'exemple simple ci-dessus), ni souhaitable (car on perd alors l'information sur la structure hiérarchique des articles, qui peut faire l'objet, elle-même, de certains types de requêtes). Deux solutions semblent possibles : (1) déterminer les types de requêtes qui peuvent directement être posées sur la structure factorisée et ne se servir de la

forme développée que si nécessaire, (2) développer les factorisations des articles lors de leur création afin de créer des index de chemin [DEL 91].

4. Conclusion

Nous avons présenté deux modèles pour les données lexicales, et nous avons décrit leur implantation en O_2. Aucun des deux modèles proposés ne résout définitivement la question de la représentation des données lexicales. Comme il a été mentionné, le premier est un modèle de la structure de surface et le second, un modèle de la structure profonde. Le premier propose un jeu d'opérateurs permettant de manipuler les structures textuelles. Il rencontre cependant un certain nombre de difficultés pour rendre comparables des données informationnellement équivalentes mais structurées différemment, par exemple factorisées d'une autre façon. Le second modèle permet de résoudre cette difficulté, et rend comparables, grâce à une relation d'équivalence, des données factorisées différemment, mais de même contenu informationnel. En revanche, il permet moins facilement d'en manipuler les caractéristiques textuelles. Il semble donc que le modèle "idéal" reste encore à définir; il devra sans doute faire la fusion des notions présente dans les deux modèles exposés.

Ce travail montre qu'un système de gestion de bases de données tel que O_2 permet d'obtenir rapidement une maquette à partir de modèles de données particuliers, en faisant l'économie du développement coûteux de systèmes et de langages spécifiques. Parmi les fonctionalités propres aux systèmes à objets, c'est évidemment leur aptitude à manipuler des objets complexes qui a été la plus utile pour construire et interroger des bases de données lexicales. Toutefois nous avons montré les limites de ces systèmes, dans leur version actuelle, pour la manipulation des données arborescentes. On a vu en effet (paragraphe 3.2) qu'il était difficile de programmer les opérateurs "en profondeur" définis par Gonnet et Tompa. L'identité d'objet, qui permet le partage de structures, n'était pas nécesaire pour les modèles que nous avons choisi puisqu'ils sont basés sur des structures hiérarchiques. Elle le serait par contre pour prendre en compte l'aspect "hyper-texte" d'un dictionnaire ou bien pour construire des structures de traits réentrantes. Enfin le rôle de l'héritage s'est limité à la simulation des types disjonctifs dont il serait préférable qu'ils soient directement pris en compte par le modèle O_2.

Il semble donc que les caractères tout-à-fait spécifiques des données lexicales, en particulier, leur factorisation libre, entraînent de nouvelles directions de recherche intéressantes en ce qui concerne la représentation et la manipulation des objets complexes. Il reste donc sans doute encore des domaines d'applications susceptibles de poser des défis aux systèmes post-relationnels, et, peut-être, de les faire avancer, comme ce fut le cas de la CAO en son temps par rapport aux systèmes relationnels.

5. Références

[ABI 93] S. Abiteboul, S. Cluet, T. Milo, "Querying and updating the file", *9ièmes Journées Bases de Données Avancées*, Toulouse, 1993.

[ADI 93] M. Adiba, C. Collet, *Objets et bases de données le SGBD O2*, Hermes, 1993.

[ANS 86] ANSI, International Standard ISO 8879: Information Processing Text and Office Systems Standard Generalized Markup Language (SGML), ANSI, 1986.

[ATK 89] M. Atkinson, F. Bancilhon, D. De Witt, K. Dittrich, D. Maier, S. Zdonick, "The Object Database Manifesto", *1st Conference On Deductive and Object Oriented Databases (DOOD'89)*, Tokyo, 1989.

[BLA 92] B. Bläser, U. Schwall, A. Storrer, "A reusable lexical database tool for machine translation", *Proceedings of the 15th International Conference on Computational Linguistics (COLING'92)*, Nantes, 1992, p. 510-516.

[DEL 91] C. Delobel, C. Lécluse, P. Richard, "Bases de données : des systèmes relationnels aux systèmes à objets", Interéditions, 1991.

[DEU 91] O. Deux et al., "The O_2 Story", *Communications of the ACM,* vol. 34, n° 10, p. 34-48.

[GON 87] G Gonnet, F. W. Tompa, "Mind your grammar: a new approach to modelling text", *Proceedings of the 13th Conference on Very Large Data Bases (VLDB'87)*, Brighton, 1987, p. 339-346.

[IDE 93a] N. Ide, J. Le Maitre, J. Véronis, "Outline of a Model for Lexical Databases", *Information Processing and Managment*, vol. 29, n° 2, p. 159-186.

[IDE 93b] N. Ide, J. Véronis (Eds.), The Text Encoding Initiative, Special Issue of *Computers and the Humanities* , to appear.

[IDE 93c] N. Ide, J. Véronis, "Encoding machine-readable dictionaries", in [IDE 93b].

[LEM 92] J. Le Maitre, O. Boucelma, "LIFOO, un langage fonctionnel de requêtes pour bases de données orientées objets", *Technique et science informatiques*, vol. 11, n° 5, 1992, p. 67-94.

[NAK 88] J. Nakamura, M. Nagao, "Extraction of semantic information from an ordinary English dictionary and its evaluation", *Proceedings of the 12th International Conference on Computational Linguistics (COLING'88)*, Budapest, 1988, p. 459-464.

[NEF 88] M. S. Neff, R. J. Byrd, O. A. Rizk, "Creating and querying lexical databases", *Proceedings of the Association for Computational Linguistics Second Applied Conference on Natural Language Processing*, Austin, 1988, p. 84-92.

[POL 87] C. Pollard, I. A. Sag, *Information-based Syntax and Semantics*, CSLI Lecture Notes Series, University of Chicago Press, 1987.

[SHI 86] S. M. Shieber, *An Introduction to Unification-based Approaches to Grammar*, CSLI Lecture Notes Series, University of Chicago Press, 1986.

[VER 92] J. Véronis, N. Ide, "A feature-based model for lexical databases", *Proceedings of the 15th International Conference on Computational Linguistics (COLING'92)*, Nantes, p. 588-594.

Jacques Le Maitre *est chargé de recherche au CNRS. Il a tout d'abord travaillé au Centre de Recherches Archéologiques sur le développement de systèmes de gestion de bases de données documentaires. Il a ensuite rejoint le Groupe Représentation et Traitement des Connaissances (GRTC) où ses recherches actuelles concernent les langages fonctionnels de programmation de bases de données et la construction de bases de données textuelles. Il participe au projet européen MULTEXT.*

Nancy Ide *et* **Jean Véronis** *sont depuis 1988 co-directeurs d'un projet de recherche conjoint CNRS/Vassar College dans le cadre duquel cette étude a été réalisée. Ils sont co-fondateurs de la Text Software Initiative et ont publié en commun de nombreux articles dans le domaine de la linguistique computationnelle.*

Nancy Ide est directrice du Département d'Informatique du Vassar College aux Etats-Unis. Elle est la présidente de l'Association for Computers and the Humanities. Elle a fondé la Text Encoding Initiative en 1987, et elle en préside le comité directeur. Elle fait également partie du groupe d'experts européen EAGLES et participe au projet européen MULTEXT.

Jean Véronis est Maître de Conférences en Informatique à l'Université de Provence. Aprés plusieurs années passées au GRTC il vient de rejoindre le Laboratoire Parole et Langage du CNRS. Il fait partie de la Text Encoding Initiative et du groupe d'experts européen EAGLES. Il est enfin coordinateur du projet européen MULTEXT.