OMEGA : un SGBD multimedia oriente objet pour les applications geographiques

Christophe Damier

THESE

présentée par

Christophe DAMIER

pour obtenir le titre de **DOCTEUR**

de l'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

(arrêté ministériel du 5 juillet 1984)

spécialité : **INFORMATIQUE**

OMEGA :
Un SGBD multimédia orienté objet pour les applications géographiques

thèse soutenue le 5 Juillet 1989

COMPOSITION DU JURY :

Président :	M. Yves CHIARAMELLA
Rapporteurs :	M. Claude DELOBEL
	M. Michel SCHOLL
Examinateurs :	M. Michel ADIBA
	M. Jacques MOSSIERE
Invitée :	Mme Estrella GOULPEAU

THESE PREPAREE AU SEIN DU LABORATOIRE DE GENIE INFORMATIQUE
A L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

Je tiens à remercier :

Monsieur Y.Chiaramella, Professeur à l'Université Joseph Fourier et Directeur du Laboratoire de Génie Informatique pour avoir bien voulu présider le jury de cette thèse.

Monsieur C.Delobel, Professeur à l'Université d'Orsay, et Monsieur M.Scholl, Directeur de recherche à l'INRIA, qui ont accepté de rapporter sur cette thèse. Leurs remarques et leur commentaires ont largement contribué à la qualité de ce document.

Monsieur M.Adiba, Professeur à l'Université Joseph Fourier, qui a dirigé mes travaux de thèses : au cours des trois années passées, il a bien voulu s'intéresser à mon travail et y apporter de nombreuses remarques constructives.

Monsieur J.Mossiere, Professeur à l'INPG, Directeur de l'ENSIMAG, qui me fait l'honneur de participer à ce jury.

Madame E.Goulpeau, responsable base de données à MS2I, qui m'a acceuilli au sein de son équipe. Je la remercie de sa disponibilité et de la confiance qu'elle m'a accordé pour la réalisation de cette étude.

Je dois aussi beaucoup à Bruno Defude, qui a participé à l'étude conduisant à cette thèse et dont les conseils et les encouragements ont toujours été judicieux.

Je remercie aussi tout le personnel de MATRA-MS2I de val de reuil et plus particulièrement tous ceux qui m'ont cotoyé durant ces trois années. Je peux citer par exemple : Eric, Nathalie, Michel, Philippe, Alain, Pierre, Chou, Yvette, Rémy, Nicole, Patrick, Martine, Christian, Jean François, Stephane, ...

J'ai aussi beaucoup apprécié l'ensemble des membres du LGI, tout particulièrement l'équipe base de données.

Je remercie enfin toutes les personnes qui ont bien voulu apporter quelques corrections à cette thèse : Bruno, Isabelle, Bénédicte, ...

TABLES DES MATIERES

CHAPITRE 1

INTRODUCTION

MS2I (branche de MATRA.S.A filialisée le 1er Janvier 1989) en collaboration avec le LGI (Laboratoire de Génie Informatique) a démarré une étude pour uniformiser et structurer la gestion des données dans ses diverses applications. Elle concerne la définition et la réalisation d'un gestionnaire de données adapté aux besoins informatiques d'un département de la société : la DPMAC (Direction de Programme des Moyens d'Aide au Commandement).

Ce département se compose d'une centaine de personnes, son activité est centrée autour du marché militaire. Il est chargé des études sur l'imagerie et la cartographie. Il produit des SIG (Systèmes d' Information Géographique [Smith 87]) basés sur la reconnaissance aérienne, la préparation de missions, la cartographie numérique et la télédétection.

Depuis sa création, ce département développe des logiciels ad-hoc où les données sont gérées de façon spécifique à l'aide de systèmes de gestion de fichier. L'analyse du coût de ces produits a mis en évidence le besoin d'un gestionnaire de données adaptable et souple, évitant un développement systématique pour chaque nouvelle application, facilitant les opérations de maintenance et permettant la gestion et le partage d'une grande quantité de données. Ces fonctionnalités sont celles qui définissent le domaine des bases de données.

MS2I s'est tout d'abord intéressé à la possibilité d'acquérir un SGBD commercialisé. Parmi ceux-ci, les SGBD relationnels sont reconnus comme les plus souples et les plus performants. Ils constituent la dernière étape d'un processus de recherche et de développement initialisé au début des années 70 [Codd 70] dont le but était de permettre la gestion, l'organisation et le partage d'une grande quantité de données. Ces SGBD relationnels répondent parfaitement aux applications de gestion et de comptabilité.

Par contre ils ne satisfont pas complètement les besoins des applications MS2I. Ils ne gèrent que des types alphanumériques mais ne supportent pas les données volumineuses. Le modèle relationnel ne permet pas d'exprimer de façon simple des liens sémantiques comme la composition d'objets [Chen 76] ou l'héritage [Goldberg 83].

En effet les données manipulées à la DPMAC ont les propriétés suivantes:

- les données sont multimédia : il s'agit de cartes numériques, d'images satellites, de compte-rendus d'observation . . .
- les données sont de grande taille et fortement liées : par exemple, une image du satellite SPOT panchromatique représente une zone de 60 sur 60 kilomètres et un volume de 36 mégaoctets. Elle peut être décomposée en un grand nombre d'objets de plus petite taille,
- les données sont multifacettes, en fonction de l'application visée, le

même objet peut être considéré différemment (par exemple, en reconnaissance aérienne un pont est vu sous son aspect géométrique, alors que dans un système d'aide à la circulation routière, il est vu sous son aspect "débit de véhicules",
- les données sont liées à des référentiels du terrain (coordonnées terrestres),
- les objets évoluent dans le temps (les images satellites météo par exemple).

En plus de ces caractéristiques, il faut respecter des contraintes liées à l'environnement industriel, c'est à dire l'utilisation du nouveau SGBD dans des logiciels déjà opérationnels, ou l'intégration dans le SGBD d'opérations élémentaires sur des types multimédia (filtrages simples d'images par exemple).

La gestion des informations géographiques est une des nouvelles applications des bases de données. On peut déjà trouver plusieurs exemples de modèles ou de systèmes (pas forcément relationnels) qui répondent à certains de ces besoins [Mac Keown 86]. En effet, depuis les années 80 de nouveaux besoins sont venus stimuler la recherche en base de données:
- l'émergence de nouvelles applications demandant (comme les applications MS2I) des fonctionnalités "base de données" : la CAO (Conception Assistée par Ordinateur), la bureautique, le génie logiciel, ou l'intelligence artificielle ...
- la demande de nouvelles fonctionnalités de la part des applications nouvelles ou traditionnelles:
    - outils de modélisation plus sophistiqués que pour les applications comptables (types abstraits, objets complexes, données spatiales, historique. . .),
    - opérateurs spécialisés (récursivité, fermeture transitive, opération sur les polygones. . .),
    - déclencheurs ou règles de production (des données sont automatiquement produites à partir d'une action sur la base de données),
    - transaction longue (la mise à jour d'un document en bureautique peut durer plusieurs heures, il s'agit pourtant d'une opération atomique) . . .
- la difficulté de communication entre les langages de programmation et les Langages de Manipulation de Données (LMD) : langage de programmation impératif manipulant les objets un par un et LMD déclaratif manipulant comme SQL, des ensembles d'objets,
- la modification des contraintes "systèmes" : augmentation de la taille des mémoires centrales et amélioration des transferts avec les disques.

Le monde de la recherche en base de données a poursuivi son effort dans de nombreuses directions:

- la définition de modèles de données plus adaptés aux nouvelles applications : les modèles relationnels NF2 [Abiteboul 86], les modèles sémantiques [Chen 76][Codd 79][Shipman 81] avec notamment les modèles TIGRE [Velez 84] et CADB [Rieu 85] développées au sein du LGI, et les modèles orientés objets [Bancilhon 88b],
- l'intégration entre les langages de programmation et les langages de manipulation de données [Bancilhon 88d],
- l'utilisation des techniques de l'intelligence artificielle et de la logique dans les bases de données : BD déductives [Gardarin 87] [Gallaire 87],
- l'extension des SGBD existants: types abstraits, SGBD extensibles et générateur de SGBD [Carey 86][Dayal 86][Stonebraker 86a], index spécifiques ou nouveaux algorithmes [Freston 87][Sellis 87].

Dans le cadre des applications géographiques multimédia, deux voies principales se sont dégagées :

- les SGBD Orientés Objets [Bancilhon 88a],
- les SGBD extensibles [De Witt 88].

L'étude menée par MS2I en collaboration avec le LGI a conduit à la définition d'un nouveau modèle de données appelé ESTRELLA, à la définition d'un langage de définition et de manipulation de données appelé SQLOMEGA, et à la réalisation d'un prototype de SGBD appelé OMEGA. OMEGA se place dans le courant des SGBD Orientés Objets. Cette étude a été partiellement financée par un contrat DRET [Damier 88c].

Dans la première partie de cette thèse, nous présentons l'état de l'art concernant la gestion de données dans les applications géographiques. Nous nous intéressons aux applications MS2I puis plus généralement aux systèmes d'information géographiques. Nous étudions les solutions proposées par les bases de données images et les bases de données spatiales.

Le chapître suivant (chapitre 3) est consacré à l'étude des possibilités de gestion des données géographiques dans les nouveaux SGBD. Il fait le point sur les SGBD orientés objets et les SGBD extensibles.

Les chapitres suivants sont consacrés à la solution adoptée par MATRA. Cette solution repose sur une modélisation orientée objet adaptée pour la géographie en introduisant de nouvelles fonctionnalités comme la gestion de versions, la gestion de l'intégrité et l'utilisation de types abstraits. Le chapitre 4 contient la définition du modèle de données ESTRELLA conçu pour intégrer tous ces concepts.

Le chapitre 5 est consacré aux interfaces entre une base de données ESTRELLA et ses utilisateurs. Il propose une interface avec le langage C et

une interface interactive basée sur un langage de définition et de manipulation des données, extension de SQL, appelé SQLOMEGA. Nous présentons les objectifs et les fonctionnalités de ces interfaces en insistant sur leur utilisation dans le cadre des application MS2I.

Les résultats de cette étude sont présentés dans les chapitres 6 et 7. Le chapitre 6 spécifie un prototype de SGBD (appelé OMEGA) qui doit mettre en oeuvre les fonctionnalités du modèle ESTRELLA. Ce prototypage est basé sur l'utilisation du SGBD relationnel ORACLE. Certaines fonctionnalités du modèles ESTRELLA sont déjà opérationnelles : une partie importante du langage de définition et de manipulation de données et du langage de requête peut être montrée (voir chapitre 6.8).

Le chapitre 7 dresse un bilan de notre action. Ce bilan discute de plusieurs aspects :

- l'intérêt de la modélisation ESTRELLA dans les applications MS2I,
- le développement et l'efficacité du prototypage.

Ce chapitre 7 conclut cette thèse en présentant un ensemble de perspectives futures.

CHAPITRE 2

ETAT DE L'ART
TRAITEMENT DES INFORMATIONS GEOGRAPHIQUES

chapitre2: état de l'art : traitement des informations géographiques

2.1 LES APPLICATIONS GEOGRAPHIQUES

2.1.1 Présentation générale

La géographie est la science qui décrit la réalité du terrain. Son objectif est de fournir des informations utilisables au niveau de l'homme. Ces informations géographiques ne sont que des descriptions approximatives de la réalité géographique. Elles se manifestent sous des formes diverses : descriptions littéraires, mesures de terrains, photographies, ou cartes. . .

L'information géographique peut aussi être le résultat d'un traitement. Il a pour but de répondre à des questions sur l'état du terrain et fait appel à des notions de topologie [David 88] :

quelle est l'altitude du mont blanc ?

la RN14 coupe-t-elle le département de la Somme ?

quel est le plus court chemin ferroviaire de Paris à Grenoble ?

Quelle que soit sa source, une caractéristique importante de l'information géographique est la persistance des sous-informations élémentaires invariables qui vont permettre son calcul : la réalité géographique est en effet peu modifiable. Même dans le cas d'une modification, on souhaite presque toujours conserver ses anciennes versions.

Une application informatique peut être qualifiée de géographique lorsqu'elle manipule des informations et qu'elle leur applique des traitements, de nature géographique. Les informations géographiques sont traduites dans l'application informatisée par des données :

- les mesures de terrains deviennent une simple liste de réels,
- les descriptions littéraires sont des chaînes de caractères ou des documents structurés,
- les photographies et les cartes peuvent être représentées par des images numériques ou des graphiques.

Une application géographique manipule donc des données de nature multimédia :

- des données alphanumériques,
- des images sous format raster (matrice de pixels),
- des graphiques, c'est-à-dire des ensembles de figures constituées à la base par certains types de dessins élémentaires (en général points, lignes, segments et polygones),
- des textes sous forme de documents volumineux structurés suivant la décomposition classique en chapîtres, sections et paragraphes.

Elle doit la plupart du temps en gérer la persistance : pour des raisons d'efficacité, on ne digitalise pas une carte ou une image à chaque exécution d'une application. . .

La modélisation géographique se traduit par l'utilisation de structures de données destinées à saisir la sémantique de l'application. Par exemple, les types "Point" et "Mesures" peuvent être déclarés comme suit :

```
type point = record x: real, y: real, z: real end;
type mesures = array [1..100] of point;
```

Ils modélisent l'ensemble des mesures gérées par l'application, et signifient que ces mesures sont ordonnées et que chacune se divise en trois coordonnées x, y et z.

Les traitements géographiques sont de natures très variables, ils correspondent, par exemple, à des algorithmes répondant aux questions précédemment citées. Parmi ceux-ci, on met en valeur la notion de requêtes spatiales [Laurini 88] [Orenstein 88][Scholl 88]. Il s'agit de répondre à des questions sur la géométrie des objets :

R1 - calcul de distances ou de surfaces,
R2 - recherche des objets contenant un point ,
R3 - recherche des objets contenus dans une région.

Ce type de requêtes intervient pratiquement dans toutes les applications géographiques. Ce sont des commandes fréquentes et interactives qui conditionnent l'efficacité des applications.

2.1.2 Les applications MATRA

Le département T2I de MS2I a de nombreuses activités dans le domaine de l'imagerie et de la cartographie. Ces activités donnent lieu à des manipulations de données multimédia à l'intérieur d'applications géographiques. En effet, ces applications mêlent les traitements de nature géographique avec des traitements spécifiques à chaque type de données. Elles font notamment appel au traitement d'images et à la recherche sur le contenu des textes.

On recense quatres types d'applications caractéristiques :
- la photointerprétation,
- la cartographie,
- la préparation de mission et l'aide au commandement,
- la télédétection.

2.1.2.1 La photointerprétation

Le travail de photointerprète consiste à étudier des photographies

(généralement aériennes) prises au cours d'une reconnaissance du terrain. Il cherche alors à y reconnaître certains objets pour en déduire des éléments d'information.

Un poste de photointerprétation informatisé doit mettre à la disposition du photointerprète toutes les informations géographiques qui sont rattachées soit au lieu de la prise de vue soit aux objets à reconnaître:

- les coordonnées de la photographie à interpréter,
- le type et la précision du capteur,
- des cartes (pour avoir une vision ergonomique de la zone photographiée),
- d'autres photographies de zones voisines ou de la même zone à des instants précédents,
- le MNT (Modèle Numérique de Terrain : cartes donnant l'altitude du sol) centré sur la zone photographiée,
- les silhouettes et des descriptions littérales des objets à reconnaître,
- un compte-rendu contenant le résultat de l'interprétation.

Les traitements géographiques sont relativement simples : recherche des cartes correspondant aux coordonnées de la photographie, projection sur les différents systèmes de coordonnées terrestres, mesure de distance sur les cartes ou les photographies. Par contre, cette application peut demander d'autres traitements importants : rotation, zoom ou corrélation sur image, superposition ou union d'images, représentation tri-dimensionnelle à l'aide du MNT, ou recherche par contenu sur des compte-rendus. . .

L'application prend fin avec la validation du compte-rendu de la photographie puis l'archivage et la datation du couple compte-rendu, photographie.

2.1.2.2 La cartographie

Intuitivement une cartographie est la transcription complète et cohérente des détails du terrain sur des supports utilisables par des opérateurs humains. "Complète" signifie qu'aucun objet n'a été oublié dans cette transcription et "cohérente" signifie que les proportions entre les objets ont été respectées. Une cartographie est cependant toujours approximative, elle dépend des facteurs d'échelle et de thème.

La cartographie informatique a pour but de reconstituer des cartes à partir d'informations géographiques préalablement enregistrées. Cette reconstitution peut être plus ou moins complexe : elle dépend de la nature des informations qu'elle utilise et du type de cartes demandées par l'utilisateur.

Par exemple, à partir d'un ensemble de cartes numérisées recouvrant la

France (format raster : obtenu en digitalisant au scanner des cartes "papier"), une application simple de cartographie consiste à produire d'autres cartes numérisées de dimensions paramétrables, centrées sur un point quelconque du territoire national (figure 1).

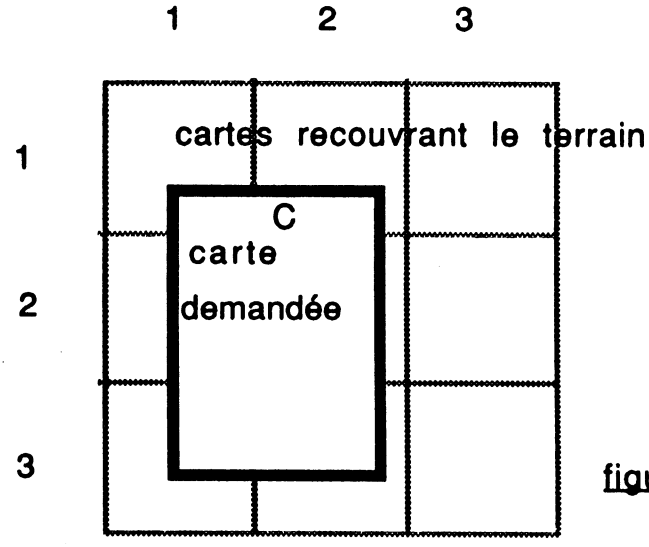


figure 1

L'application n'a alors besoin que de quelques fonctionnalités géographiques : recherche des cartes recoupant une zone centrée sur un point. Par contre elle utilise certains traitements d'image : extraction de sous-image, union, lissage . . .

Une autre application plus complexe modélisera les informations géographiques de façon moins élémentaire : chaque objet significatif est géré séparément avec sa représentation cartographique et des attributs de description. On peut alors produire un très large éventail de cartes en interrogeant le système selon la nature des objets. Dans ce cas les traitements géographiques sont plus importants : changement d'échelle, projection, recherche du contenu d'une zone géographique. . .

2.1.2.3 La préparation de mission et l'aide au commandement

La préparation de mission et l'aide au commandement sont des activités difficiles à appréhender en raison du grand nombre de paramètres pouvant intervenir à chaque instant. On peut cependant dégager des caractéristiques géographiques primordiales : l'espace de travail peut être recouvert par une cartographie et tous les objets manipulés ont une position sur ces cartes. Un utilisateur, doit prendre rapidement des décisions en fonction d'un état du terrain (variable dans le temps, sur le chemin à suivre pour la préparation de mission).

Le problème de la modélisation de ce type d'application n'est pas simple.

Les objets peuvent être liés indépendamment de leur relations topologiques. Par exemple, une entreprise est divisée en départements mais ceux-ci peuvent être situés dans différents lieux géographiques. . .

Ces applications font intervenir des démarches intuitives où chaque personne applique les régles issues de son expérience personnelle : il s'agit avant tout d'un travail d'expert. Un système informatique d'aide au commandement a donc tout lieu de se matérialiser dans un système expert. Ce système n'en reste pas moins une application géographique au sens où il met en oeuvre des données géographiques :

- cartes décrivant l'espace de travail,
- coordonnées ou positions d'objets sur ces cartes.

Un système de préparation de mission demande de nombreux traitements géographiques :

- accès aux cartes contenant ou se superposant avec un objet,
- objets se situant dans une zone,
- calcul de distance ou de surface,
- calcul de voisinage d'une zone, d'un objet ou d'une trajectoire. . .
- restitution de cartes avec les objets superposés.

Une application d'aide au commandement met donc en oeuvre une importante composante géographique même si le système principal est d'une autre nature.

2.1.2.4 La télédétection

La télédétection consiste à gérer et à exploiter des images satellites. L'image satellite brute (telle qu'elle est reçue depuis l'espace) n'est pas utilisable. Elle doit être traitée pour réparer les déformations subies à la fois à cause de l'imprécision de son capteur et à cause des phénomènes géographiques comme la rotation de la terre. Pour pouvoir effectuer ces traitements, on doit faire intervenir de nombreuses informations géographiques concernant cette image [Rogala 85] : altitude et coordonnées du capteur, grille de déformation. . .

Depuis l'image brute jusqu'à l'image exploitable il faut donc appliquer une série de traitements coûteux :

- rectifications géométriques,
- projections,
- filtrages,
- lissages.

La phase d'exploitation de ces images fait également intervenir de

nombreuses informations géographiques lors de divers traitements :

- recherche des cartes recoupant la zone photographiée,
- superposition avec ces cartes,
- corrélation entre plusieurs images satellites,
- mesure d'altitude et génération de MNT à partir d'images stéréoscopiques.

L'exploitation des images satellites réside essentiellement dans la production d'autres données géographiques (images ou MNT) qui seront à leur tour utilisées dans d'autres applications géographiques.

2.1.3 Gestion des données dans ces systèmes

Ces applications sont exécutées chacunes sur un poste de travail mais doivent pouvoir communiquer entre elles : le poste de télédétection envoie des images rectifiées au poste de photointerprétation, le résultat de la photointerprétation est pris immédiatement en compte par le système d'aide au commandement. . . Ces systèmes ont donc, en général, des fonctionnements parallèles et communiquent entre eux via un réseau.

Pour effectuer la gestion de ces données dans ses applications, MS2I s'appuie actuellement sur les systèmes de gestion de fichiers des disques de ses postes de travail. Elle utilise également des équipements serveurs sur le réseau pour stocker les gros volumes : un dérouleur de bande magnétique pour images et un gestionnaire de DON (Disque Optique Numérique).

Les structures de données et les algorithmes, destinés aux opérations géométriques sont développés séparément pour chaque application. Des systèmes de traitements d'images et de recherche documentaire sont directement intégrés selon les besoins.

Quelque soit l'application, MS2I s'est heurté aux mêmes problèmes de gestion des données géographiques :

- persistance de données multimédia : certaines données doivent survivre à la fin d'une application notamment les images, les cartes, les graphiques ou les textes,
- présence d'un gros volume de données : le volume de données varie selon les applications. Cependant il reste très important. Il faut environ 160 gigaoctets pour enregistrer la cartographie de la France au 50000éme,
- importance des relations entre les données : la modélisation des applications en terme de structures de données n'est pas simple. Elle nécessite de nombreuses possibilités :
    - la définition d'objets complexes : par exemple, une région

contient une liste de départements, chaque département est lui même composé d'une préfecture et d'une liste de circonscriptions. . .

  - la gestion d'historique : on désire presque toujours garder une trace des modifications appliquées aux objets géographiques,
  - la gestion de relations spatiales entre les objets : tous les objets sont liés à leur position dans l'espace et l'accès à ces objets s'effectue presque toujours suivant des requêtes spatiales (distance, objets contenant un point, et objets contenus dans une région).

- partage de ces données entre une communauté d'utilisateurs : les données géographiques sont communes à toutes les applications. De plus certaines d'entre elles peuvent fonctionner en mode multi-utilisateurs,
- fonctionnement en mode dégradé et reprise après pannes : un des postes de travail peut être momentanément en panne; les autres applications du réseau doivent être en mesure de fonctionner partiellement en préservant l'intégrité et la cohérence des données. Lors de la reprise du poste arrêté, ce dernier doit avoir perdu un minimum du travail effectué avant la panne et le réseau d'application ne doit pas être trop perturbé.

Pour apporter un début de réponse aux problèmes de la gestion de données dans ces applications, MS2I a effectué une étude sur les gestionnaires de données existants adaptés aux problèmes de l'information géographique [Goulpeau 86]. Cette étude a mis en évidence l'existence de plusieurs domaines de la recherche sur la gestion des données géographiques :

- un domaine issu des besoins informatiques du milieu géographique : il a conduit à l'apparition des Systèmes d'Information Géographique (SIG),
- un domaine issu de la recherche en base de données qui s'est spécialisé dans les applications géographiques. Il a conduit à l'apparition d'une nouvelle génération de bases de données appelées bases de données géographiques,
- un domaine des bases de données cherchant à étendre les possibilités des SGBD traditionnels pour s'intégrer par exemple dans les applications de CAO, d'intelligence artificielle, ou de géographie. . . : les SGBD généralisés.

2.2 LES SYSTEMES D'INFORMATIONS GEOGRAPHIQUES

Le concept de SIG est apparu dans les années 60. Il est le résultat de l'utilisation de l'outil informatique en géographie. A l'époque, il s'agissait d'exploiter les techniques récentes de photogrammétrie, de cartographie et de production de données numériques. L'objectif était la gestion et l'exploitation de données provenant de sources multiples (photogrammétrie numérique, numérisation de cartes, statistiques, images satellites. . .).

Le premier système CGIS fut implanté en 1964 pour aider à planifier l'utilisation du sol au Canada. Depuis le nombre des SIG n'a cessé de croître. On trouve une grande variété de systèmes adaptés soit à l'utilisation des sols ou des ressources naturelles, soit à la planification urbaine, régionale ou nationale. Certains systèmes généraux sont commercialisés, le plus répandu au monde est ARC/INFO [Schaller 87].

2.2.1 Fonctionnalités d'un SIG

Le but des SIG est de faciliter le travail des utilisateurs pour la recherche, la gestion et la planification géographique. En phase d'analyse, un SIG met en oeuvre une suite d'actions : sélection, observation et étude des données. D'autres phases de fonctionnement sont aussi importantes : saisie des données, restitution des résultats. . . Dans [Calkins 77] on trouve une présentation synthétique des premiers SIG. Ils sont définis comme des systèmes d'information devant accomplir cinq tâches :

- saisie et formatage des données,
- organisation et gestion des données,
- accès rapides aux données,
- analyse et traitement sur les données,
- sortie et restitution des données et des résultats.

[Smith 87] montre l'insuffisance d'une telle définition et décrit les futurs SIG comme des outils géographiques munis des fonctionnalités suivantes :

- la gestion de données géographiques dans une base de données,
- la possibilité d'interrogation de ces données suivant des critères géographiques (requêtes spatiales),
- l'exécution efficace de ces interrogations pour pouvoir travailler interactivement,
- une souplesse d'adaptation du système aux opérations de maintenance et à la modification des schémas des applications,
- des possibilités d'apprentissage sur les objets spatiaux, leur signification et leur implantation durant le vie du système.

[Smith 87] montre qu'un SIG répondant à ces fonctionnalités assure

obligatoirement les tâches décrites par [Calkins 77].

Il est cependant évident que les SIG actuels ne fournissent pas la totalité de ces fonctionnalités. Plusieurs approches ont été suivies pour le développement de ces SIG. Elles se distinguent suivant la modélisation de l'espace géographique et les techniques d'implantation mises en oeuvre.

2.2.2 modélisation géographique et implantation dans les SIG

La plupart des concepteurs de SIG ont défini leur système à partir des spécifications des applications qui allaient l'utiliser. De ce fait, ils n'ont pas fait de distinction entre le modèle de représentation des objets géographiques et leur implantation. Contrairement à l'un des souhaits des concepteurs de base de données, il n'y a pas d'indépendance entre la description logique et la représentation physique des objets.

Ces systèmes utilisent les techniques classiques de décomposition géographique que l'on peut répartir en deux grandes catégories :

- le modèle topologique (ou format vecteur),
- le modèle à base de tessélation

2.2.2.1 Le modèle topologique

Une décomposition topologique du plan modélise les objets qui le composent à l'aide de leur frontière. Il s'agit d'enregistrer les relations de proximité entre les objets géographiques en gérant le réseau de lignes contenu dans le plan.

Une première méthode peut être employée : un objet (considéré comme un polygone) est représenté par un ensemble de lignes. Les relations de proximités sont définies implicitement par le partage de la frontière séparant deux objets.

```
type objet = record
                nom : string;
                   frontière : array [1..100] of pointeur_de_lignes;
             end;

type ligne = array [1..100] of point;
type pointeur_de_ligne = ^ligne;
```

Une seconde méthode consiste à définir les relations de proximités dans les lignes : une ligne est définie par ses points et pointe vers les objets de gauche et de droite. Un exemple d'une telle structure est celle définie par l'IGN (Institut Géographique National) pour la gestion du projet BD TOPO

(Base de Données TOPOgraphique) :

```
type arc = record
             nombre_de_points : integer;
             objet_gauche : string;
             objet_droit : string;
             sens_de_parcours : boolean;
             liste_de_points : array [1..100] of point;
           end;
```

Ces types de structures permettent une grande précision dans la représentation des objets. Elles permettent les vérifications de l'intégrité (adjascence et inclusion) et de la cohérence (un objet à droite ne peut être aussi à gauche...). Par contre les algorithmes de requêtes spatiales sont complexes et coûteux.

2.2.2.2 La tessélation

Une tessélation est un ensemble de cellules (par exemple les rectangles de la figure 1) qui forment une partition régulière du plan. La cellule de base de la tessélation constitue une unité d'espace à laquelle on associe l'ensemble des objets qui la chevauchent.

exemple 1 : tessélation horizontale T de la figure 1 (9 cellules)

T = [(1,1:C) (1,2:C) (1,3:) (2,1:C) (2,2:C) (2,3:) (3,1:C) (3,2:C) (3,3:)]

Les algorithmes et les opérateurs sur les tessélations ont été abondamment étudiés, classés et optimisés :

- algorithmes de compressions (suppression des cellules vides, unification des cellules voisines identiques, codage de leur contenu sur une chaîne de bits...),
- algorithmes de mise à jour (redécoupage de cellules trop pleines. . .),
- algorithmes exprimant des opérations spatiales (reconstitution d'objets, intersection de zones. . .).

La tessélation a l'avantage de gérer implicitement les relations spatiales entre les cellules. Par contre le choix de la taille de la cellule est capital : trop petite, le volume de données augmente rapidement, trop grande les objets de faible taille ne sont plus représentés correctement.

Pour lever cette difficulté, des modèles de tessélation imbriquée ont été proposés. Ils reposent sur un découpage récursif du plan. On trouve par exemple les traditionnels quadtrees et k-d trees ou plus récemment les Grid-files, et R+trees [Sellis 87] [Samet 88]. L'apport obtenu dans la

représentation des objets est en général compensé par l'augmentation de la complexité des algorithmes et de leur temps de réponse. Ils sont malgré tout indispensables lorsqu'une grande précision est nécessaire.

2.2.3 Les fonctionnalités base de données dans les SIG

Les problèmes d'implantation des SIG traditionnels ont mis en valeur le besoin d'une gestion de données plus cohérente. [Smith 87] et [Frank 88] insistent sur la nécessité d'inclure une base de données dans les nouveaux SIG. Ils notent que les futurs SIG seront demandeurs de fonctionnalités abondamment étudiées et maitrisées par la communauté base de données :

- manipulations élémentaires sur les données : saisie, requête sur des attributs, restitution de résultats. . .
- intégrité : certaines contraintes entre les objets doivent être vérifiées et maintenues au cours du fonctionnement du SIG (par exemple, les contraintes d'inclusion ou d'adjascence entre objets),
- concurrence : gestion des accès multiples et concurrents des utilisateurs sur des données,
- confidentialité : mécanisme permettant d'autoriser ou d'interdire l'accès aux données,
- sécurité de fonctionnement : les SIG doivent permettre le retour après une panne dans un état cohérent.

[Peuquet 84] montre l'importance de la modélisation des données dans les SIG. Il propose de construire un SIG comme implantation d'un modèle de données spatial.

La communauté des chercheurs sur les SIG reconnait l'intérêt des concepts base de données pour leur discipline. Plusieurs expériences ont été tentées pour construire des SIG à l'aide de SGBD relationnels. Là encore on trouve deux approches :

(1) La première approche repose sur la gestion séparée entre les données classiques dans le SGBDR et la localisation des objets dans un système de stockage spécifique [Chang 81]. Il faut alors définir un lien entre les objets décrits dans le SGBDR et leur localisation : un pointeur, un nom de fichier ou des adresses physiques. Cette approche a été adoptée notamment par ARC/INFO. Elle pose plusieurs problèmes en particulier des problèmes de cohérence entre les deux sous-systèmes.

Pour limiter au maximum les risques, certains SIG ont adoptés une démarche intermédiaire : ils gèrent une localisation approchée dans le SGBDR (IMAID [Chang 81], GRAIN [Chang 77]. . .).

Par exemple, dans le système GRAIN (Graphic-oriented Relationnal Algebric INterpreter) trois tables gèrent la description des objets dans le

SGBDR :

POT est la table de définition des objets : elle contient le nom et les attributs qui caractérisent chaque objet, en particulier le nom du propriétaire (dans le cas de l'inclusion dans un autre objet), les coordonnées d'un point d'appui, et l'angle par rapport au propriétaire.

PCT est la table des contours. Chaque objet de POT a un contour dans PCT. Le nuplet de PCT contient le nom de l'objet, le format de stockage du contour et dans un champ LONG, le contour.

PPT fait le lien avec les supports externes. Elle donne le nombre de pages, leur taille et l'adresse pour accéder à chaque objet.

Les utilisateurs disposent des opérateurs de l'algébre relationnelle et de quelques outils sur la localisation approchée (affichage des contours dans GRAIN) pour déterminer les objets intéressants. Ensuite, ils peuvent faire intervenir les systèmes de stockages externes.

A l'aide de ce mécanisme hybride, ces SIG diminuent les risques mais augmentent la complexité de leur implantation, notamment pour gérer la cohérence entre les deux localisations des objets.

(2) L'autre approche repose sur le codage complet de la structure de données spatiales dans le SGBDR [Lorie 84]. Il s'agit de préserver la cohérence en évitant plusieurs supports séparés. Les structures simples du modèle relationnel (nuplet, relation), peuvent être facilement utilisées pour stocker un modèle topologique du plan :

exemple 2 : relation gérant un modèle topologique

POINT (identificateur, x, y)
LIGNE (identificateur de ligne, numéro du point, identificateur du point)
OBJET (nom, numéro de ligne, identificateur de ligne, position par rapport à la ligne)

Ce type de stockage est inefficace, c'est pourquoi [Lorie 84] et [Rogala 85] proposent d'utiliser les champs LONG des SGBDR pour stocker les listes de points ou les images. Cependant cette solution n'est pas entièrement satisfaisante puisque ces champs LONG ne sont plus pris en compte par l'algèbre relationnelle, on perd donc la puissance des opérateurs d'un

modèle de haut niveau.

Tous ces systèmes sont opérationnels mais souffrent de la faiblesse des SGBDR dans la gestion d'index spatiaux, la modélisation d'objets fortement structurés et le support de données multimédia.

C'est pourquoi, la plupart des chercheurs sur les SIG [Frank 88] [Cowen 88] [Calkins 87] demandent la définition de nouveaux SGBD qui, tout en conservant les fonctionnalités traditionnelles des SGBDR, offriront de nouvelles possibilités :

- modélisation de structures spatiales,
- extension du langage de requêtes par des opérateurs spatiaux,
- utilisation d'index spatiaux pour exécuter efficacement ces requêtes.

2.3 LES SGBD GEOGRAPHIQUES

Les SGBD géographiques sont nés de la recherche en base de données. Dans le début des années 80, les chercheurs se sont intéressés à l'extension des SGBD pour de nouvelles applications dans de nombreux domaines (intelligence artificielle, génie logiciel, bureautique, CAO). Parmi les applications géographiques, les systèmes de traitement d'image ont été les premiers demandeurs de fonctionnalités base de données. On a assisté à l'éclosion d'une génération de bases de données appelées bases de données images.

Comme on l'a vu dans le paragraphe précédent, les SIG sont devenus depuis peu, des demandeurs de SGBD. Ils spécifient un produit qui a priori, est voisin de celui mis en oeuvre pour les systèmes de traitement d'images (puisqu'ils manipulent les mêmes types de données) mais qui se révéle assez différent au niveau de la modélisation et des traitements demandés. On appelera ces systèmes, les bases de données spatiales.

2.3.1 Les SGBD images

La recherche et la conception des SGBD images des années 80, a été fortement influencée par les SGBD relationnels qui arrivaient juste sur le marché. Les SGBD images reprennent presque tous les concepts du modèle relationnel en y incluant des extensions au niveau de types de données (le type "image" ou le type "graphique". . .) et au niveau des opérateurs (traitements spécifiques aux images). La plupart de ces systèmes sont d'ailleurs implantés à l'aide d'un SGBD relationnel.

La gestion des images reste très pragmatique : beaucoup d'efforts concernent le stockage ou l'accès mais il y a peu de tentatives de modélisation par exemple, pour définir des opérateurs ou pour donner un

sens aux prédicats relationnels sur le domaine "image". Ces systèmes qui sont essentiellement orientés vers le traitement d'images, mettent en oeuvre les fonctionnalités traditionnelles de cette activité :

- saisie et formatage de données images,
- structures de stockage,
- méthodes de compression sur la structure,
- algorithme performant de traitements suivant la structure,
- restitution de résultats.

On retrouve les principes de SIG tels que les définissait Calkins. Pour représenter les images, on retrouve les deux formats de gestion classiques de stockage et de gestion :

- le format vecteur,
- le format raster.

Le but de ces systèmes n'est plus de gérer la précision de l'information géographique par rapport à l'efficacité des requêtes spatiales, mais de comparer le volume de stockage des images avec l'efficacité des algorithmes de traitements dans chacun des formats. Pour plus de précision, on présente dans ce qui suit, quelques exemples de SGBD images tirés de [Munoz-Baca 87].

2.3.1.1 Le système ADM

(Aggregate Data Manager) [Takao 80]

Le système a été développé au centre scientifique IBM (Japon). L'idée principale est d'offrir un outil qui permet une manipulation intégrée et homogène de données "image" et de données conventionnelles. Le modèle relationnel est utilisé avec une extension sur les types de données à gérer. Le type agrégat est introduit, il représente un terme général pour désigner toute donnée non structurée (images, graphiques, textes, ensembles. . .). Les données de type agrégat sont d'abord stockées dans un espace de travail, elles sont ensuite transférées dans la base de données où elles sont associées à des données classiques (entiers ou chaînes de caractères) dans des relations.

Pour le moment, le type agrégat concerne uniquement les images et les ensembles, mais sa portée peut être facilement étendue aux autres données non structurées (textes, graphiques. . .). Les images sont divisées en deux classes : les images binaires (noir et blanc) et les images à plusieurs niveaux de gris. Un ensemble est considéré comme un groupe non ordonné de données atomiques qui servent à décrire des données de type agrégat (un ensemble de mots décrit une image).

Les images binaires sont comprimées alors que les images à plusieurs

niveaux de gris sont stockées sans compression. Un tuple image contient un identificateur interne, une chaîne de bits de longueur variable contenant l'image et la taille de cette chaîne. Les ensembles sont gérés dans des relations binaires. Dans chaque tuple, on trouve l'identificateur interne de l'ensemble et un de ses éléments.

ADM repose sur quatre outils :

(1) un système interactif qui dialogue avec les utilisateurs, enregistre les commandes et restitue les résultats.

(2) un système d'édition : pour chaque type de données agrégat, il existe un éditeur qui permet d'effectuer des opérations spécifiques. Un éditeur d'images accomplit outre l'affichage, des traitements élémentaires comme l'extraction, la suppression, la superposition, l'inversion, et le zoom.

(3) un espace de travail est donné à chaque utilisateur. Il contient les données de type agrégat qui sont retenues temporairement pendant la session de travail, soit pour des insertions dans la base de données, soit pour des manipulations de données déjà insérées.

(4) la base de données contrôle les accès. Elle constitue une interface homogène entre les utilisateurs et les données, quels que soient leurs types.

ADM offre deux modes d'interrogations : un mode basé sur des formulaires et un mode interactif avec des commandes. Celles-ci sont classées en quatre groupes : (1) les commandes "scroll", utilisées pour obtenir une liste des tables et choisir les images à afficher, (2) les commandes utilisées pour gérer l'espace de travail contenant les données agrégats, (3) la commande d'édition EDIT suivie de l'espace de travail (c'est une commande générique : le type d'éditeur (image ou texte) est déterminé automatiquement à partir du type de l'objet à éditer), (4) les commandes de la base de données qui sont celles du langage SQL sans extension.

Les opérateurs de comparaison qui peuvent se trouver dans une requête sur des images sont l'égalité et la non-égalité. Ils agissent seulement sur l'identification interne de l'image. Les opérateurs autorisés sur les ensembles sont plus nombreux : égalité, différence, inclusion stricte, chevauchement, disjonction. Les requêtes SQL contenant des opérations sur les ensembles ont besoin d'une transformation avant d'être exécutées puisque les ensembles sont projetés sur des relations.

2.3.1.2 Le système ELF

(Extended relationnal model for Large, Flexible picture database) [Yamagushi 80]

· Ce système a été développé à l'université de Tokyo (Japon) pour la gestion de clichés photographiques. Chaque cliché est considéré comme la représentation instantanée d'un ensemble d'objets. ELF cherche à gérer les relations entre ces objets en fonction de leurs apparitions sur les clichés. Les objets ont des propriétés qui sont dérivées des connaissances physiques de l'objet ou des images elles-mêmes (mesures, nombre. . .). Ils sont classés dans des unités logiques qui représentent le monde réel et forment des classes génériques appelées aussi ensembles d'objets. Par exemple, "Maison", "Hopital", "Mairie" sont des objets appartenant à la classe générique "BATIMENT".

Le processus de description d'une image consiste à associer les objets avec des classes génériques, à déterminer les relations qui existent entre eux et à déterminer un cadre sémantique de validité de ces relations. On obtient finalement un monde d'objets constitué par une énumération de l'ensemble des objets, de l'ensemble des relations, et par la définition du cadre sémantique dans lequel l'ensemble des relations est valide.

Le modèle relationnel sert de base pour représenter ce modèle d'images et d'objets. Deux types de relations sont utilisées pour établir une correspondance entre les deux modèles :

(1) Les relations "OBJETS" décrivent les propriétés des objets qui composent les images. Le nom de la relation détermine un objet. Un identificateur est affecté à chaque tuple dans chaque relation pour déterminer l'image à laquelle appartient l'objet.

(2) Les relations "ASSOCIATIONS" décrivent les relations qui existent entre les objets. Il existe une relation "ASSOCIATION" pour chaque ensemble de relations définies.

La figure 2 présente un monde d'objets pour trois images identifiées par les numéros &1, &2 et &3. Ces images contiennent des objets (une mairie, une école, des routes. . .) qui sont décrits dans les relations OBJETS. Les relations "ASSOCIATIONS" décrivent les classes génériques et les relations entre les objets suivant les clichés.

La structure sémantique de validité des relations entre les objets est exprimée à travers des contraintes d'intégrité sur ces deux types de relations.

RELATIONS OBJETS :

MAIRIE

image	couleur	position
&1	noire	456,908
&2	grise	341,1009

ECOLE

image	couleur	position
&1	blanche	450,907
&2	noire	1041,209

RN15

image	longueur
&2	18
&3	14

RELATIONS ASSOCIATIONS :

BATIMENT

nom
MAIRIE
ECOLE

ROUTE

nom
RN15

EST_PRES_DE

obj1	obj2	cliché
MAIRIE	ECOLE	&1
ECOLE	RN15	&2

figure 2

Une originalité du système ELF est d'autoriser un nom de relation comme domaine d'un attribut. Ceci implique une extension du modèle relationnel, pour pouvoir accepter des noms de relations comme valeur d'attributs. En conséquence le calcul relationnel doit être aussi étendu afin de pouvoir manipuler les noms des relations indifféremment avec les autres valeurs de la base de données. L'interrogation se fait à travers le langage QBE-LIKE, qui est basé sur le langage QBE avec les extensions mentionnées ci-dessus.

2.3.1.3 Le système IBIS

(Image-Based Information System) [Zobrist 80]

C'est un produit des laboratoires de "Jet Propulsion" de l'institut technologique de Pasadena, Californie (USA). Le but du système est d'établir une communication directe entre les informations obtenues par des capteurs et les processus qui réalisent le traitement de données. C'est-à-dire qu'il offre un outil qui facilite le transfert de données aux processus qui en ont besoin. Deux facteurs sont cruciaux : la manipulation d'une grande quantité de données et les opérations de traitement d'images. Le système IBIS est couplé avec le système de traitement d'images VICAR, qui supporte les deux types de format d'images (matriciel et vectoriel).

Dans IBIS, trois types de données sont utilisés : (1) les données tabulaires, (2) les données graphiques et (3) les données images. Chaque type de données est enregistré dans un fichier. Les données tabulaires et les données images sont générées à partir des données graphiques grâce à un module formé par un ensemble de routines Fortran. Ce module transforme les données graphiques (Polygones) en images des régions ou images des surfaces digitalisées, et crée ainsi les données images. Ensuite il réalise une série de mesures sur ces dernières et génère les données tabulaires. Les images sont reçues par VICAR. Les données tabulaires sont utilisées pour générer des rapports.

2.3.1.4 Le système IDMS

(Integrated Database Management System) [Tang 81]

Le système IDMS a été développé à l'université de New York, Buffalo (USA), il est assez général puisqu'il permet de travailler avec n'importe quelle image numérisée (photographie, radiographie, signature ou carte. . .). Le système traite de la même façon les données alphanumériques et les données images. Cela repose sur une extension du modèle relationnel avec deux nouveaux types de données : le type "picture" et le type "device". Les attributs de type "picture" contiennent une image matrice de pixels. Ils sont définis par trois paramètres (M, N, B) où N.M donne la taille de la matrice et B le nombre d'octets du pixel. Les opérations classiques d'égalité, de différence, de somme, d'inclusion, et de négation sont supportées par le système sur le domaine "picture". Il les effectue en exécutant l'opérateur correspondant, sur chaque pixel des matrices participant au calcul.

Le type "device" spécifie les outils nécessaires à l'affichage d'une donnée. Un attribut sur le domaine "device" prend sa valeur parmi les entrées-sorties connues du système. Le but de ce type "device" est d'offrir plus de simplicité en assurant la compatibilité entre les types de données et les entrées-sorties.

L'interrogation est réalisée à l'aide du langage SQL. Il a été modifié pour permettre la manipulation des nouveaux types et les opérations matricielles. D'autres extensions ont été ajoutées au langage. Elles portent sur les problèmes de stockage pour les données de type "picture". Les utilisateurs peuvent invoquer des procédures de compression ou de migration de données à travers trois nouvelles commandes : "STORE PICTURE", "STORE RELATION", et "CODE OF ... IS ..." . "STORE" indique le support de stockage et "CODE" la méthode de codage.

2.3.2 Les bases de données spatiales

Le concept de la base de données spatiales est relativement récent. Il est issu de la cartographie et des fonctions de traitement de l'espace. Il repose sur une démarche qui prend sa source dans l'étude des besoins des SIG et se conclut par la définition d'un SGBD complet : interfaces spécifiques à la géographie, modélisation de cartes et méthodes d'implantation pour les opérateurs spatiaux (notamment pour les requêtes spatiales).

Les objets géographiques sont décrits à travers trois composants :

- un ensemble de propriétés générales sur leur nature pouvant s'exprimer simplement avec des données alphanumériques,
- une représentation graphique,
- une représentation spatiale : localisation, surface, espace occupé. . .

Les deux derniers points constituent de nouveaux types de données encore peu pris en compte par les bases de données.

Les objets géographiques complexes (par exemple des cartes) sont modélisés par des ensembles de sous-objets géographiques. Leur représentation graphique n'est pas stockée, elle est obtenue par union des représentations graphiques des sous-objets. De la même façon leur représentation spatiale peut être le résultat d'un calcul sur les représentations spatiales de leurs sous-objets.

Les modèles de données de bases de données spatiales peuvent être considérer comme des extensions du modèle relationnel.

[David 88] et [Güting] proposent des extension du modèle relationnel et définissent de nouveaux domaines qui ont pour valeur des représentations spatiales. [Scholl 89b] définit une modèle relationnel étendu pour la gestion d'objets géographiques complexes et leur utilisation à l'aide d'un algèbre de régions.

Contrairement aux SGBD images essentiellement tournés vers le stockage et l'accès aux données, les bases de données spatiales définissent les mêmes opérateurs sur leurs nouveaux types que sur les types traditionnels (entier, caractère. . .).

Les méthodes mises en oeuvre dans les bases de données spatiales rejoignent celles des SIG. Elles ont pour but d'implanter efficacement les requêtes spatiales (algorithme optimisé de produit spatial [Scholl 88] [Güting 88], ou jointure de Peano [Laurini 88]).

2.3.2.1 Le modèle SPATIAREL

SPATIAREL [David 88] est un bon exemple des bases de données spatiales. Le modèle de données relationnel est étendu par un nouveau domaine "spatial" (représentation spatiale et graphique). Les opérations classiques sont définies sur le domaine "spatial" :

- la somme A + B de deux représentations spatiales correspond à l'union de ces représentations,
- la différence A - B calcule le complémentaire de B dans A,
- l'intersection A B,
- l'extension d.A produit l'extension de la représentation A suivant le facteur réel d.

Le prédicat d'égalité est utilisable sur le domaine "spatial". Cependant ce n'est pas le plus intéressant. [David 88] définit de nouveaux prédicats de comparaisons :

- l'inclusion (au sens large) de A dans B notée $A \leq B$,
- le chevauchement de A et B noté A <> B,
- l'adjacence de A et B (A et B ont certains points communs à leur frontière),
- la connexité d'une représentation spatiale A,

L'algèbre relationnelle est étendue avec les opérateurs de fusion et de décomposition (opérateurs s'appliquant sur une relation dont un des attributs est de type spatial). La fusion d'une relation sur un attribut "spatial" permet de regrouper les nuplets ayant mêmes valeurs pour les attributs non spatiaux en réalisant l'union de leur représentation spatiale. La décomposition d'une relation suivant un attribut "spatial" remplace les nuplets dont la représentation spatiale était composée de N espaces disjoints, par N nuplets dont chaque représentation spatiale est un espace connexe.

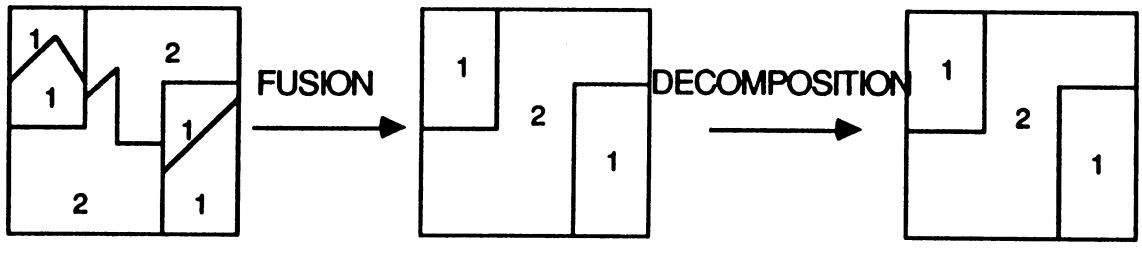


FIGURE 3 : opérateurs spatiaux

Les opérateurs classiques de sélection, projection, différence, union et produit conservent une signification sur des relations avec des attributs

"spatiaux". David introduit la jointure par chevauchement (aussi appelée produit spatial) qui correspond aux produits de deux cartographies.

Dès lors le langage SQL permet d'exprimer la plupart des opérations géographiques souhaitables. Par exemple les requêtes spatiales sont traduites par des requêtes faisant intervenir le prédicat de chevauchement ou d'inclusion :

exemple 3 : recherche des régions contenant la tour Eiffel

région (nom : chaîne, espace : spatial)
point (nom : chaîne, espace : spatial)

```
SELECT r.nom
FROM   région r , point p
WHERE  p.espace < r.espace and p.nom = "tour Eiffel"
```

2.3.2.1 Le modèle GEO-relationnel

[Güting 88] adopte une démarche parallèle. Il définit le modèle géo-relationnel comme suit :

- les types "POINT", "LIGNE" et "REGION" sont ajoutés à la liste des types autorisés. Les attributs des relations peuvent être définis sur ces types géométriques. Seuls de nouveaux opérateurs permettent la manipulation des types géométriques, les attributs géométriques ne sont pas directement accessibles par l'utilisateur,
- un n-uplet d'une relation contient à la fois des attributs de valeur géométrique et non géométrique,
- Une relation est un ensemble homogène de n-uplets.

exemple 4 : quelques opérateurs sur les types géométriques
(* indique la possibilité d'exécuter l'opérateur avec un ensemble de valeurs)

=	POINT x POINT -> BOOLEAN
inclus_dans	LIGNE x REGION -> BOOLEAN
intersection	LIGNE* x LIGNE* -> POINT*

L'algèbre associée à ce modèle comprend des opérateurs traditionnels sur les relations (sélection, projection sur une relation, produit et téta-produit. . .) et des opérateurs sur les types géométriques. La plupart des requêtes spatiales peuvent être transformées en une suite d'opérations de l'algébre géo-relationnelle.

exemple 5 : recherche des villes de plus de 30000 habitants de l'Isère

VILLE (nom : chaîne, nb_habitants: entier, centre : POINT)
DEPARTEMENT (nom : chaîne, population : entier, contour : REGION)

têta_produit (
VILLE select(nb_habitants>30000) project [nom, centre]
DEPARTEMENT select(nom="Isère") project [contour]
[centre **inclus_dans** contour]
) project [nom]

La requête est exprimée comme une suite d'opérations algébriques. Il s'agit simplement d'une jointure entre les relations "VILLE" et "DEPARTEMENT" suivant un prédicat géométrique d'inclusion. Güting définit d'autres opérateurs plus complexes, comme le prédicat de chevauchement et l'opération de recouvrement.

Le principal intérêt de ces systèmes est de traduire les requêtes spatiales en une succession d'opérateurs définis sur les nouveaux domaines. Par contre, leur interface avec les applications est mal définie. D'autre part, ces systèmes ne sont pas conçus pour la gestion et le traitement de données atomiques volumineuses pourtant très utiles dans les applications géographiques : image sous format matriciel, texte. . .

2.4 CONCLUSION

Nous avons tout d'abord vu dans ce chapître, les différentes applications géographiques développées par MS2I. Il en ressort des problèmes de modélisation, de cohérence, de partage, de volume et d'efficacité dans la gestion des données. L'étude préliminaire sur les gestionnaires de données existants et réutilisables par MS2I dans les applications géographiques informatisées, a permis d'évaluer différentes solutions.

Les SIG n'apportent pas de solutions. Ils se heurtent aux mêmes problèmes de modélisation, de cohérence et de partage. Par contre, ils mettent en oeuvre des algorithmes efficaces dans le traitement des requêtes spatiales.

Les bases de données images ont des objectifs plus restreints. Elles gèrent peu les traitements géographiques et portent leurs efforts sur la gestion des gros volumes de données et de leur cohérence.

Enfin les bases de données spatiales apportent une solution aux problèmes des SIG. Leurs fonctionnalités sont intéressantes mais ces

systèmes (encore au stade de la recherche) n'intègrent pas de gestion de données multimédia.

MS2I situe son SGBD à l'intersection de ces systèmes puisqu'il doit permettre à la fois la gestion de données multimédia, la modélisation d'applications géographiques et l'exécution de requêtes spatiales. Le bilan de cet état de l'art montre qu'il n'existe pas de solution immédiatement utilisable. Les applications MS2I sont très variées et ne se satisfont pas de ces systèmes trop spécialisés. On peut donc s'intéresser aux SGBD généralisés dont le but est d'être utilisables pour le plus large éventail possible d'applications.

CHAPITRE 3

LES NOUVEAUX SGBD ET LES DONNEES GEOGRAPHIQUES

chapitre3 : Les nouveaux SGBD et les données géographiques

On entend généralement sous le terme de "données géographiques", toutes données représentant des informations de nature géographique : cartes, images, légendes, coordonnées. . .

Pour effectuer une étude sur la gestion des données géographiques dans les nouveaux SGBD, on va prendre l'exemple des applications cartographiques. On se place dans le cadre d'une cartographie évoluée (voir 2.2.2) où les données sont organisées de la façon suivante : l'espace de travail est décrit par un ensemble de cartes. Chaque carte contient un ensemble d'objets définis par leur localisation sur la carte et par des attributs de type quelconque : entier, réel, chaîne, texte, image, carte ou objet.

La manipulation de ces objets repose soit sur les opérateurs classiques des SGBD portant sur leurs attributs, soit sur des opérateurs spatiaux portant sur leur localisation.

Par rapport aux concepts des SGBD relationnels, on distingue trois nouveaux problèmes :

P 1 - La modélisation de ce type d'organisation n'est pas facilement exprimable avec le modèle relationnel : composition d'objet, héritage,

P 2 - Le SGBD doit gérer des données multimédia,

P 3 - Les opérateurs spatiaux doivent être intégrés aux outils de manipulation et doivent être implantés efficacement.

Ces problèmes ont tous été étudiés lors de la spécification des nouveaux SGBD. Deux principaux courants de pensée ont essayé d'y apporter une réponse : les SGBD orientés objets et les SGBD extensibles.

3.1 LES SGBD ORIENTES OBJETS

Les SGBD orientés objets constituent les résultats de la fusion entre les Systèmes Orientés Objets (SOO) (par exemple, SIMULA 67, SMALLTALK 80 [Goldberg 83]) et les bases de données.

Historiquement ce sont les recherches sur l' "impedance mismatch" (séparation entre les langages de programmation et les LMD) qui ont conduit à s'intéresser aux SOO : le formalisme objet intègre sous une même entité la donnée, sa structure et les programmes (appelés méthodes) qui la manipulent. De plus la modélisation objet semble plus appropriée que le relationnel pour les nouvelles applications des bases de données.

3.1.1 Présentation générale

A l'heure actuelle la recherche a produit de nombreux prototypes de SGBDOO, on peut par exemple citer ENCORE [Bloom 87], GEMSTONE [Maier

86], IRIS [Fishman 87], O2 [Lecluse 87] ou ORION [Banerjee 87a]. Afin de décrire ce vaste domaine d'expérimentation, on va dans un premier temps présenter quelques principes communs à la plupart des SGBDOO, et par la suite on détaillera certains systèmes comme GEMSTONE, IRIS et O2.

En effet, contrairement au modèle relationnel, il n'y a pas de support théorique unanimement reconnu pour la modélisation orientée objet. On peut simplement dégager un certain nombre de points de reflexion, communs à l'ensemble des modèles orientés objet [Bancilhon 88a] :

- **l'identité d'objet,**
- **les objets complexes,**
- **les types et les classes,**
- **l'encapsulation,**
- **l'héritage,**
- **les méthodes génériques.**

3.1.1.1 L'identité d'objet

Un objet est désigné de façon unique dans le SGBDOO par son identificateur. L'identité de l'objet est indépendante de sa valeur. Un objet O d'une classe A est défini par son identificateur i et sa valeur v:

$$O = (i,v).$$

- L'objet peut être composé d'autres objets : la valeur v de O peut contenir des identificateurs d'objets,
  v = [5,i1,"le chat est là"].
- L'objet peut même être cyclique : v = (5,i,"le chat est là"),
- Un objet peut être partagé par d'autres objets :
  O' = (i',v') et v' = [10,i1,"j'aime le pain"]
  L'objet identifié par i1 est partagé par O et O'

La mise à jour de O altère la valeur v mais ne modifie pas i, toutes les relations de composition d'objets sont donc conservées. Cette mise à jour n'est pas équivalente à une suppression suivie d'une insertion : la suppression puis l'insertion entraîne le remplacement de (i,v) par (i',v'). Les objets qui contenaient O en utilisant son identificateur i, ne le référencent plus.

On dispose d'une notion d'identité d'objet et d'une notion d'égalité de valeur. Dans un programme d'application, les objets de type A sont désignés par des variables:

declare variable a,b **de type** A

a représente l'objet (i1,v1) et b l'objet (i2,v2). Au cours de l'éxécution du programme le test a = b peut prendre deux significations:

- a et b ont même identificateur: a = b est exécuté par i1 = i2,
- a et b ont même valeur: a = b est exécuté par v1 = v2.

3.1.1.2 Les objets complexes

La valeur des objets correspond à une structure définie à l'aide de types simples d'objets et de constructeurs comme le n-uplet, l'ensemble ou la liste. Cette structure est donc complexe. Ses attributs prennent leur valeur parmi les objets de la base.

exemple 1 : structure complexe des objets e1, e2 et d1

e1 = [nom : "martin",age : 28,salaire : 10000]
e2 = [nom : "dupont",age : 24,salaire : 6000]
d1 = [nom : "R&D", employés : { e1 , e2 }]

e1 et e2 ont la même structure S1, d1 a la structure complexe S2 :
S1 = [nom : **string**, age : **int**, salaire :**int**]
S2 = [nom : **string**, employés : { **S1** }]

Les structures complexes permettent une modélisation plus fine et plus proche des types manipulés dans les langages de programmation. Ils permettent de décrire facilement les objets à plusieurs niveaux: textes (document puis chapître, section, paragraphe...), circuits (puce, cellule, transistor, masque) ou même des organisations hiérarchiques (entreprise, secteur, département, section. . .).

Les SGBDOO offrent un certain nombre de constructeurs pour définir des objets complexes: nuplet, liste, ensemble, tableau, choix, arbre. . . Ces constructeurs sont redondants: un arbre peut être défini à l'aide de nuplets et d'ensembles. Le choix des constructeurs dépend des applications visées par le SGBDOO: bureautique (ensemble, nuplet et liste pour décrire des documents), IA (nuplet, ensemble et arbre). . .

3.1.1.3 Les types et les classes

Les objets de structure semblable et sur lesquels s'appliquent les mêmes opérations sont regroupés dans des classes. Leur structure et leurs méthodes définissent un type. Les types représentent le schéma conceptuel de la base de données et sont utilisés pour optimiser la compilation des méthodes.

exemple 2 : types et classes de e1, e2 et d1

e1 et e2 sont de type EMPLOYE et appartiennent à la classe de même nom:
EMPLOYE = [nom : **string**, age : **int**, salaire :**int**]
d1 est dans la classe DEPARTEMENT, son type est :
DEPARTEMENT = [nom : **string**, employés : { **EMPLOYE** }]

Tous les SGBDOO ne font pas la distinction entre types et classes. Cette distinction subtile sépare le type (notion déclarative) de la classe (notion exécutive): un type décrit le comportement des objets, la classe est un objet qui les contient. Les méthodes définies sur les classes exécutent des opérations élémentaires (création, destruction d'objets).

3.1.1.4 L'encapsulation

La donnée, sa structure et les programmes qui la manipulent sont rassemblés dans la notion d'objet. Le programmeur ne peut utiliser que les méthodes pour accéder à l'objet. Il ignore la structure et l'implantation des méthodes de l'objet et ne connait que l'interface des méthodes. Dans la plupart des SGBDOO l'activation des méthodes s'effectue par passation de messages.

exemple 3 : méthodes définies sur les classes EMPLOYE et DEPARTEMENT

declare méthode
augmenter_salaire(plus : int) **sur** EMPLOYE

declare méthode
ajouter_employé(new : EMPLOYE) **sur** DEPARTEMENT

La méthode "ajouter_employé" ne peut s'appliquer qu'à des objets de type département. Elle met à jour le champ "employés" défini dans l'exemple 2, cette mise à jour ne peut pas être effectuée directement sur les objets sans appel de méthodes. Les méthodes sont activées par passation de messages:

exemple 4 : ajouter un employé à un département puis l'afficher

declare variable d **de type** DEPARTEMENT
declare variable e **de type** EMPLOYE

d <- ajouter_employé(e) <- afficher

Des SGBDOO comme O2 [Lecluse 87] ou GEMSTONE [Maier 87] mettent en œuvre ce principe d'encapsulation. Cependant comme les opérations de manipulation des structures sont fréquentes dans les SGBD, la plupart des SGBDOO offrent en standard des méthodes pour manipuler les champs des types. D'autre part l'utilisation exclusive des méthodes sur les objets se prête mal à l'accès associatif (qui est une des fonctionnalités principales des SGBD relationnels et qui demande une opération sur les classes elles-mêmes).

3.1.1.5 L'héritage

Les SGBDOO et les SOO autorisent une relation d'ordre partiel (appelée **"est-sous-classe-de"** ou **"isa"**) entre les classes. Une classe A définie comme sous-classe de B vérifie par définition les propriétés suivantes :

- tout objet de A est aussi un objet de B.
- toutes les méthodes de B sont applicables sur les objets de A,
- la structure des objets de A spécialise celle de la classe B.

A est dite sous-classe de B et B superclasse de A. La sous-classe peut disposer de méthodes supplémentaires et peut avoir une structure plus large que B :

exemple 5 : définition de la classe INGENIEUR sous-classe d'EMPLOYE avec les attributs supplémentaires collaborateurs et projet

```
INGENIEUR est-sous-classe-de EMPLOYE
type(INGENIEUR) = [nom : string, age : int, salaire :int
                   collaborateurs : { EMPLOYE }, projet : string]

i1 = [nom= "Charpin", age=27, salaire= 18000,
      collaborateur= { e1, e2}, projet= "turbo" ]
```

L'objet i1 appartient à la classe INGENIEUR, c'est aussi un objet de la classe EMPLOYE: on peut lui appliquer la méthode augmenter_salaire.

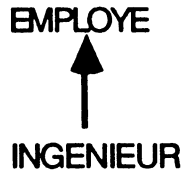
exemple 6 : programme augmentant de 1000, l'objet i1

```
declare variable i de type INGENIEUR pour l'objet i1

i <- augmenter_salaire(1000)
```

L'héritage permet la réutilisation des codes des méthodes des superclasses. Il améliore donc les conditions de programmation.

La relation d'ordre "est-sous-classe" permet de décrire une hiérarchie entre les classes. Selon les SGBDOO, cette relation est simple (une classe ne peut avoir qu'une seule super-classe) ou multiple (plusieurs super-classes).

exemple 7 : hiérarchie de classes



L'héritage simple peut être représenté par un arbre alors que l'héritage multiple donne un graphe orienté sans cycle. Ces organisations hiérarchiques permettent une meilleure compréhension du schéma conceptuel, elles apportent donc un important pouvoir de modélisation .

3.1.1.7 Les méthodes génériques

Il s'agit de définir une seule méthode pour une fonctionnalité du système (exemple l'affichage). Cette méthode accepte donc des types différents, le choix du code à exécuter s'effectue lors de son exécution.

Les méthodes génériques sont des facilités de programmation: un seul nom de méthode doit pouvoir s'appliquer à un grand nombre de types. Par exemple la méthode "afficher" accepte tous les types de la base de données.

Dans les SGBDOO les méthodes génériques utilisent les concepts de surcharge sémantique et de résolution tardive. La surcharge sémantique donne la possibilité de redéfinir des méthodes héritées d'une superclasse:

exemple 8 : la classe objet_affichable admet deux sous-classes photo et texte.

declare méthode afficher **sur le type** objet_affichable
méthode afficher **sur** objet_affichable **est** ( )
méthode afficher **sur** photo **est** (- - - code C - - -)
méthode afficher **sur** texte **est** (- - - autre code C - - -)

La méthode "afficher" est définie sur objet_affichable, elle est héritée par photo et texte qui sont des sous-classes de objet_affichable. Sur les classes photo ou texte, on peut soit réutiliser le code déclaré sur objet_affichable (nul dans ce cas), soit reprogrammer cette méthode.

Après surcharge sémantique, un même nom de méthode représente alors plusieurs codes différents. La résolution tardive (qui implique une liaison dynamique) en est une conséquence : lors de l'exécution de la méthode "afficher", le SGBDOO doit faire dynamiquement le lien avec le code voulu.

exemple 9 : afficher tous les objets d'un plan graphique géré par un SGBDOO

declare variable o **de type** objet_affichable

pour tout o du plan graphique: o <- afficher

Selon le type de l'objet, afficher exécute la méthode définie sur photo ou sur spot. Ce choix est fait à l'exécution car on ignore à priori les objets du plan graphique.

3.1.1.7 Gestions des données géographiques

Les SGBDOO sont des systèmes essentiellement tournés vers la programmation. Ils ont pour but de résoudre l'"impedance mismatch", donc de faciliter la gestion de la persistance dans les applications.
A priori ce ne sont pas des systèmes orientés vers la géographie. Cependant ils offrent des mécanismes suffisamment puissants pour permettre la gestion des données géographiques.

exemple 10 : modélisation des cartes dans un SGBDOO

type(CARTE) = [nom: **string**, échelle: **int**, objets: { **OBJET** }]
type(OBJET) = [nom: **string**, localisation: **POSITION**,
dessin: { **GRAPHIQUE** }]
type(POSITION) = [x: **real**, y: **real**]
type(GRAPHIQUE) = [origine: **POSITION**, couleur: **real**]

POLYGONE sous-classe-de GRAPHIQUE
type(POLYGONE)= [origine: **POSITION**, couleur: **real**, liste: < **POSITION** >]

· CERCLE sous-classe-de GRAPHIQUE
type(CERCLE) = [origine: **POSITION**, couleur: **real**, rayon: **real**]

Les modèles orientés objets sont suffisamment riches pour répondre au problème **P1** (modélisation d'une organisation géographique) : ils offrent la possibilité de définir des objets complexes ainsi que la puissance de l'héritage.

Par contre les SGBDOO n'offrent pas de solution immédiate pour gérer les données multimédia. Le système permet la définition de n'importe quels types de données mais laisse aux programmeurs des applications le soin de les implanter. Le SGBDOO peut néanmoins les utiliser grace au mécanisme d'encapsulation : il utilise l'interface de méthodes programmées pour l'application.
De la même façon, les opérateurs spatiaux ne sont pas intégrés aux SGBD, ils doivent être programmés sous forme de méthodes attachées à une classe représentant les données spatiales.

exemple 11 : opérateurs sur les graphiques

declare méthode superposer(g: graphique) -> graphique
sur le type graphique

declare méthode intersection(g: graphique) -> graphique
sur le type graphique

méthode superposer **sur** graphique **est** (- - - code C - - -)
méthode intersection **sur** polygone **est** (- - - code C - - -)
méthode intersection **sur** cercle **est** (- - - code C - - -)

L'exemple 12 montre l'utilisation des méthodes définies dans l'exemple 11 pour la programmation d'une application géographique.

exemple 12 : utilisation des opérateurs dans un programme d'application

declare variable g0, g1, g2 **de type** GRAPHIQUE

g0 <- intersection(g1 <- superposer(g2))

L'utilisation des SGBDOO pour la gestion de données géographiques est donc possible. Elle reste cependant difficile puisque le système n'offre en standard que la gestion des constructeurs de base (nuplet, liste, ensemble...) et qu'il faut implanter soit-même les outils géographiques.

D'autre part, peu de SGBDOO disposent encore d'un langage de requêtes interactif. C'est pourtant un outil indispensable dans une application géographique où les requêtes ne peuvent être entièrement connues à l'avance. Dans le cas contraire, les SGBDOO (une fois développées toutes les méthodes géographiques) offrent un bon potentiel en tant que couche basse d'une application géographique.

3.1.2 QUELQUES EXEMPLES DE SGBD ORIENTES OBJETS

Dans cette section, nous présentons trois prototypes de SGBDOO. Ces systèmes ont adopté la plupart des principes exposés dans les paragraphes précédents. Ils diffèrent généralement par leur implantation et leurs interfaces. On peut cependant noter quelques nuances dans la définition de leur modèle orienté objet.

3.1.2.1 GEMSTONE : un SGBD commercialisé

GEMSTONE est un produit commercialisé par SERVIO LOGIC depuis deux ans [Maier 86]. Il est présenté comme un SGBD adapté à une très grande gamme d'applications : bureautique, fabrication ou conception assistée par ordinateur, cartographie et système d'information géographique. . .

GEMSTONE met en oeuvre un modèle de données typique des SGBDOO. Il intègre les notions d'identité d'objets, de type et de classe, des constructeurs variés (ensemble, nuplet, tableau) permettant de gérer des objets complexes, les mécanismes d'encapsulation et d'héritage de la structure et des méthodes.

Il offre une unique interface dans le langage orienté objet OPAL : Il s'agit d'un langage complet permettant d'effectuer à la fois toutes les opérations classiques sur les SGBD (définition ou manipulation de données, requête avec accès associatif) et toutes les instructions d'un langage de programmation. Ce nouveau langage vise donc à résoudre l' "impedance mismatch". GEMSTONE propose pour les applications développées en OPAL, un environnement de programmation complet (bibliothéque mathématique...) appelé OPE (OPAL Programming Environment).

exemple 13 : accès associatif avec OPAL sur une base de données, dans l' exemple 10, recherche des cartes de la France à grande échelle.

```
carteSET select { [:nom, :échelle] | carte.nom = 'France'
                                     and carte.échelle > 250000 }
```

cette commande OPAL applique la méthode "select" sur carteSET (un ensemble de cartes).

Select sélectionne les cartes de la France dont l'échelle est supérieure au 250000ème, elle leur applique la méthode [:nom, :échelle] qui construit un nuplet contenant le nom et l'échelle de la carte.

GEMSTONE n'a pas d'interface interactive prédéfinie. Il propose une bibliothéque de fonctions basée sur le multifenêtrage qui doit permettre aux utilisateurs de construire en utilisant OPAL leurs propres interfaces interactives.

Les applications désirant utiliser GEMSTONE peuvent cependant être écrites dans des langages de programmations conventionnels. GEMSTONE met à leur disposition des interfaces langage-OPAL : exemple GCI (GEMSTONE-C Interface).

GEMSTONE fonctionne dans un environnement réparti. Une machine serveur de données est connectée par un réseau à des stations de travail (figure 1). Actuellement le serveur est disponible sur VAX, les stations de travail sont soit des Macintosh, des IBM PC ou des SUN.

Sur chaque station de travail on trouve des applications utilisant soit directement OPAL (environnement OPE), soit une interface interactive, soit une interface langage-OPAL.

Le serveur est divisé en deux machines GEM et STONE. GEM effectue les opérations au niveau du modèle de données. Il dispose du schéma conceptuel et gère la compilation ou l'exécution des méthodes OPAL. STONE est chargé de la gestion des accès physiques sur le disque, des protections, de la concurrence, des transactions et de la reprise après panne. Les objets ont un identificateur interne (un surrogate) et STONE gère une table de correspondance entre les surrogates et les adresses physiques.

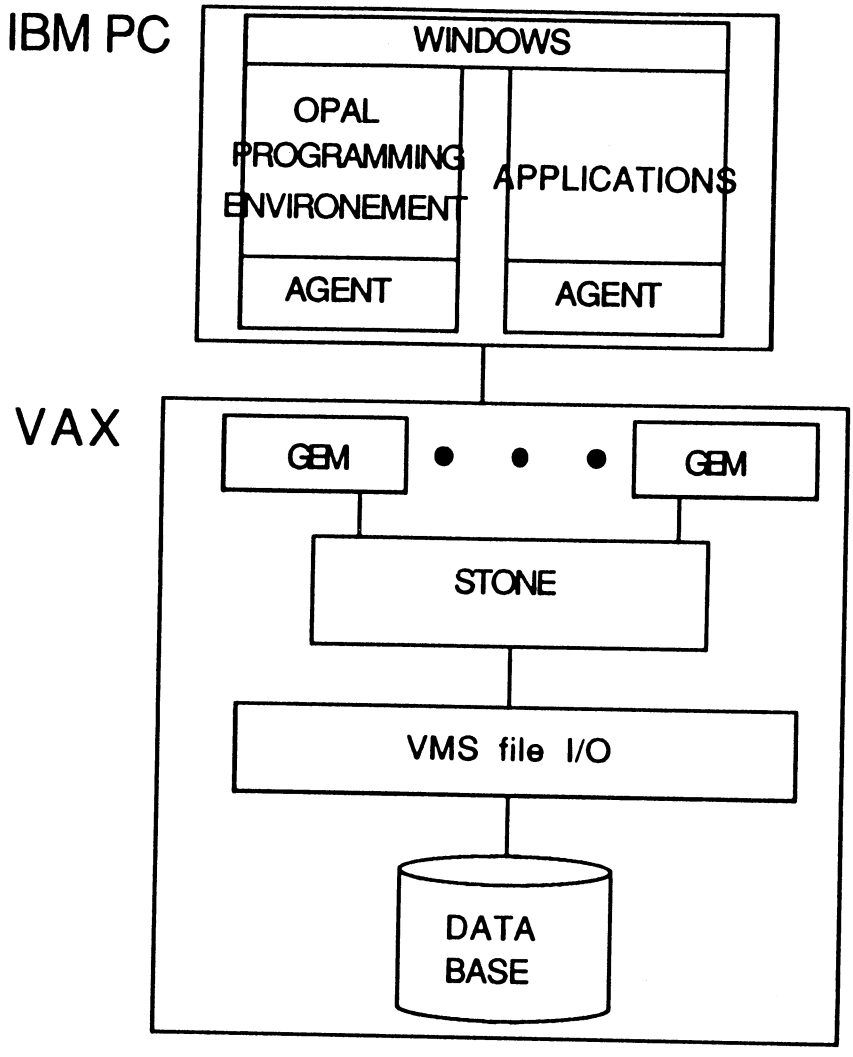


figure 1

GEMSTONE a déjà été utilisé dans le cadre d'applications géographiques notamment pour la production de cartes de navigation aux USA. En effet GEMSTONE supporte les types tableaux et ensembles qui permettent de décrire des objets multimédia (une image est un tableau de pixels, un graphique est un ensemble de segments). Cependant les descriptions de ces application ne mentionnent pas l'utilisation d'opérateurs spatiaux ni de méthodes pour l'accès rapide aux données volumineuses.

3.1.2.2 IRIS : un projet de HP

Ce projet est développé par les laboratoires de Hewlett-Packard en Californie [Fishman 87]. Il vise les nouvelles applications des bases de données : bureautique, base de connaissances, CAO. . . IRIS offre un grand nombre de fonctionnalités nouvelles par rapport aux SGBD traditionnels.

Le modèle de données défini pour IRIS suit une approche fonctionnelle à partir de DAPLEX [Shipman 81]. Les types définissent un ensemble d'objets partageant les mêmes propriétés. Les valeurs des objets sont obtenues par

un ensemble de fonctions qui décrivent leur type. Ces fonctions sont soit stockées en extension soit déduites par une requête sur d'autres objets.

exemple 14 : types et fonctions d'une base de données carte-objet

type carte = (nom: string, echelle: int)
type objet = (nom: string, position: point)
new operation obj(c/carte) = o/objet
new operation sacarte(o/objet) = c/carte

Dans l'exemple 14, on définit deux types "carte" et "objet" et deux fonctions "obj" et "sacarte". La première fonction s'applique sur une carte est retourne un ou plusieurs objet; la seconde retourne une carte pour chaque objet.

IRIS permet le sous-typage et la redéfinition d'opérations dans les sous-types. Les objets des sous-types sont des objets de leurs super-types. Un objet peut avoir plusieurs types sans relation de sous-typage entre eux. Le multitypage des objets augmente la souplesse et la dynamicité des objets. Par contre, elle complique leur utilisation (appel de méthodes et connaissance de l'objet).

IRIS n'effectue pas de liaison dynamique. Il n'admet pas d'opérations génériques. Par contre, IRIS prévoit une gestion de versions, un mécanisme d'inférence ainsi que la possibilité de définir de nouveaux types de données.

Ce SGBD propose des interfaces de haut niveau pour son modèle de données :

- un langage OSQL (extension de SQL) prennant en compte l'héritage et la composition d'objet,
- un utilitaire : l' "inspector" permettant aux utilisateurs LISP d'examiner la base de données,
- un ensemble de méthodes permettant d'interroger IRIS dans un programme LISP,
- une gestion d'objets persistants dans un environnement LISP.

IRIS est implanté au dessus d'un SGBD relationnel : les opérations stockées sont gérées dans des relations. Les nouveaux types de données sont gérés dans le SGBD relationnel (types de données non volumineux) mais peuvent aussi être dans des SGBD spécialisés (notamment pour les types multimédia) à condition qu'ils remplissent des fonctionnalités compatibles avec IRIS: transaction, partage, intégrité, requête... Ces types sont manipulés à l'aide d'opérations non stockées qui sont liées avec IRIS lors de la définition du nouveau type.

La réutilisation de SGBD spécialisés déjà existants pourrait permettre d'intégrer simplement des opérateurs spatiaux et des interfaces multimédia dans IRIS. Cela peut donc contribuer à une meilleure réponse de ce SGBD aux problèmes des données géographiques à condition que ces systèmes répondent aux conditions imposées par IRIS.

3.1.2.3 O2 : le système d'Altaïr

O2 est un projet en cours de développement au sein du GIP Altaïr. Le but de ce GIP (Groupement d'Intérêt Public regroupant principalement IN2, l'INRIA et le LRI) est de commercialiser un SGBDOO répondant aux nouveaux besoins de SGBD. Il s'agit avant tout de résoudre deux problèmes :

- l' "impedance mismatch". En intégrant correctement LMD et langage de programmation, le GIP vise le marché des programmeurs en espérant augmenter leur productivité,
- l'inadéquation entre les SGBD actuels et ce que l'on appelle traditionnellement les nouvelles applications des bases de données.

Le GIP a reconnu l'approche orientée objet comme la plus prometteuse pour les deux problèmes : encapsulation pour résoudre "l'impedance mismatch" et objet complexe pour modéliser les nouvelles applications.

O2 repose sur un modèle orienté objet incluant les notions d'identité d'objet, de type et de classe associée, d'encapsulation, d'héritage et de méthode générique. O2 permet aussi de gérer directement des valeurs (et non seulement des objets).

La gestion de la persistance dépend des utilisateurs : les objets persistants sont nommés. Les autres objets sont eux-mêmes persistants tant qu'ils composent les objets persistants. S'ils ne sont plus rattachés à une racine de persistance, ils sont alors détruits et leur place est réutilisée par un ramasse-miettes.

exemple 15 : base de données employé-département définie avec O2

```
add class adresse type = [ numéro : int, rue : string, ville : string        ]
add class employé type = [ nom : string, age : int, salaire : real,
                           adresse : adresse                                 ]
add class département type = [ nom : string, chiffre : real,
                              lieu : adresse, emps : { employé }             ]

add object mes_employés : { employé }
add object mes_départements : { département }
add object PDG              : employé

add method augmenter ( entier ) in class employé
```

Dans l'exemple 15, on définit trois classes, trois objets persistants ("mes_employés", "mes_départements" et "PDG") et une méthode. Mes_employés contient l'ensemble des employés qui héritent donc de la persistance, Mes_départements contient les départements qui sont donc persistants. PDG est un objet persistant particulier qui peut être géré à part. Les adresses ne sont persistantes qu'à condition d'appartenir à un employé ou à un département.

O2 permet de gérer directement des valeurs sans identité. c'est le cas dans l'exemple 15 pour l'attribut "emps" de "département" qui a pour valeur un ensemble d'employés sans qu'il y ait d'objet de ce type.

Les utilisateurs d'O2 peuvent programmer des applications principalement dans deux langages : C, BASIC. L'intégration des appels de méthodes O2 dans ces langages a été particulièrement étudiée. Elle repose sur la définition des méthodes et leur encapsulation lors de la définition du type et sur la passation de messages dans le langage hôte.

La méthode "*" permet d'accéder directement à la valeur d'un objet en désactivant le mécanisme d'encapulation. Elle identifie la manipulation des objets O2 à la syntaxe C sur les structures.

Le but de cette approche est de faciliter au maximum la programmation :

exemple 16 : utilisation de types O2 dans le langage C

```
employé e;
e = new(employé);
strcpy(*e.nom, "damier" );
[e augmenter(1000) ];
printf("l'employé %s est payé %d \n", *e.nom, *e.salaire);
```

Dans l'exemple 16, la commande "new" crée un objet dans la classe "employé". On manipule directement sa valeur en utilisant "*". On se place dans une approche orientée objet en utilisant la méthode "augmenter".

Le GIP étudie aussi des interfaces interactives pour O2 : un générateur d'applications [Collet 87] et des langages de requêtes basées soit sur SQL [Delobel 88] soit sur l'approche fonctionnelle [Fishman 81].

Un prototype d'O2 est opérationnel, il fonctionne en mode distribué : un serveur de données et des stations de travail. O2 prévoit des directives lors de la passation de messages pour le choix du lieu d'exécution des méthodes (sur le serveur ou sur la station). En régle générale, le serveur transmet des objets vers l'application qui les utilise à l'aide d'un gestionnaire d'objet en mémoire, local au poste de travail (figure 2).

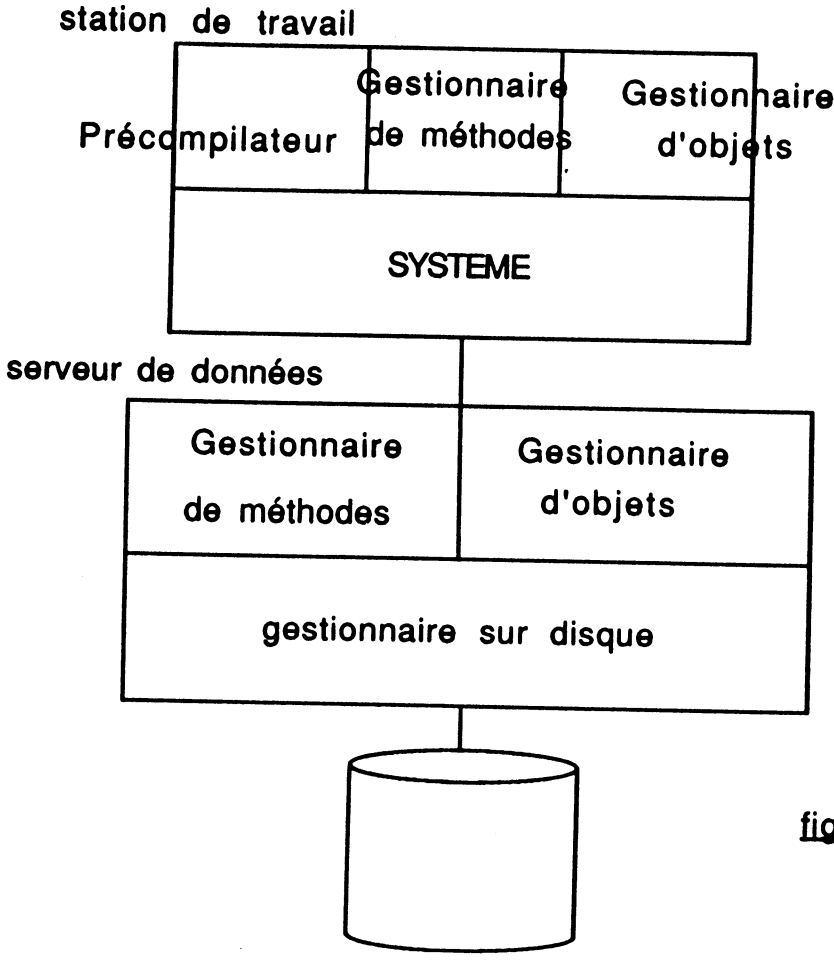


figure 4

Sur le serveur, les objets sont stockés dans des fichiers. Ils sont regroupés suivant des directives fournies par l'administrateur de la base de données. Le gestionnaire sur disque utilise le gestionnaire de transactions et de fichiers WISS. Sur les stations de travail. Un mécanisme d'indexation des classes est à l'étude. Il doit permettre avec le regroupement d'objets d'obtenir de bonnes performances.

Parallèlement à ces développements, le GIP Altaïr étudie les applications possibles pour O2. Il s'intéresse aux applications géographiques. Cependant l'état de leurs travaux ne permet pas de conclure sur l'intérêt spécifique de ce système pour les applications géographiques. Il n'en reste pas moins que leur démarche est intéressante.

3.2 LES SGBD EXTENSIBLES

L'émergence des nouvelles applications pour les bases de données a fait

apparaître de nouveaux besoins:

- modélisation d'applications plus complexes que les applications comptables (objets complexes, données spatiales, historique...),
- opérations spécialisées (déduction, opération sur les polygones),
- structures de données spécialisées et plus efficaces,
- déclencheurs,
- transactions longues.

La définition des SGBD extensibles a été motivée par ces besoins. Les SGBD extensibles reprennent les concepts des SGBD existants et les modifient pour qu'ils puissent être étendus par de nouvelles technologies :

- nouveaux opérateurs (jointure ouverte, récursion. . .),
- nouveaux algorithmes (jointure par hachage),
- nouveaux index (R-tree, Grid file, hachage dynamique. . .).

Les SGBD extensibles sont soit de nouveaux SGBD qui prévoient des extensions selon les applications, soit des boîtes à outil permettant de construire le SGBD spécifique à son application (générateur de SGBD).

Comme pour les SGBDOO, un grand nombre de projets ont vu le jour :

- DASDBS	[Paul 86]	(Université de Darmstadt)
- EXODUS	[Carey 86]	(Université du Wisconsin)
- GENESIS	[Batory 88]	(Université d'Austin)
- POSTGRES	[Stonebraker 86b]	(Université de Berkeley)
- PROBE	[Dayal 86]	(Computer Corporation of America)
- STARBUST	[Lindsay 87]	(IBM Amalden)
- TIGRE	[Velez 84]	(Université de Grenoble)
- VERSO	[Scholl 89]	(INRIA)

3.2.1 Présentation générale

On détaillera par la suite trois exemples caractéristiques PROBE, POSTGRES et EXODUS. Auparavant on va présenter les principales caractéristiques de cette nouvelle génération de SGBD. En général, tous ces systèmes conservent les grandes fonctionnalités des bases de données :

- sécurité de fonctionnement,
- indépendance donnée-programme,
- modélisation par un LDD,
- manipulation à l'aide d'un LMD.

Mais en plus ils intégrent de nombreuses nouvelles fonctionnalités :

- **la définition de nouveaux types de données,**
- **la définition de nouveaux opérateurs,**
- **l'implantation de nouvelles techniques d'accès ou de stockage,**
- **la modification ou la génération d'un modèle de données.**

On peut noter que les SGBD orientés objet offrent certaines de ces possibilités: définition d'objets complexes et de nouveaux opérateurs pouvant utiliser de nouvelles structures de données. Par contre ils n'ont pas de notion de requêtes ou d'optimisation de requêtes (dans GEMSTONE on signale explicitement l'utilisation d'un index). Ce sont des systèmes essentiellement tournés vers les programmeurs.

3.2.1.1 Définition de nouveaux types de données

Les SGBD du commerce ont un ensemble fermé de types (entier, réel, chaîne de caractères et date pour ORACLE) et un constructeur unique : le nuplet. Les SGBD extensibles proposent deux sortes d'extension sur ces types :

- la définition de types d'objets complexes,
- les types abstraits.

Le problème de la gestion des objets complexes a amené trois types de solutions :

- **extension du modèle relationnel** : les plus anciennes extensions du modèle relationnel que sont le modèle entité association [Chen 76] et les modèles NF2 [Schek 86] [Abiteboul 84] ont été choisies par les projets TIGRE [Velez 84] et VERSO [Scholl 89a]. D'autres extensions comme les déclencheurs, les procédures ou l'utilisation du LMD comme donnée sont utilisées dans POSTGRES [Rowe 87].
- **la définition de nouveaux modèles de données** : c'est le cas par exemple de PROBE qui repose sur le modèle fonctionnel DAPLEX [Shipman 81] ou d'ETIC qui utilise le modèle sémantique CADB [Rieu 85]. La plupart de ces nouveaux modèles intégrent la notion de composition d'objet et l'héritage pour modéliser les objets complexes.

- **la génération d'un modèle de données suivant les besoins** : certains SGBD extensibles n'ont pas de modèle de données prédéfini. Il suffit donc de générer un modèle de données adapté aux objets complexes des applications ciblées

par les utilisateurs.
Dans [Carey 88], on montre comment définir un modèle orienté objet pour EXODUS. DASDBS intègre à priori les opérateurs de l'algébre NF2 pour faciliter l'implantation d'un modèle pour objet complexe [Schek 86b]. Ce type de SGBD a l'inconvénient de définir un nouvel intermédiaire pour l'utilisateur : le DBI (Data Base Implementor), le programmeur de la base de données.

Le mécanisme de type abstrait a été conçu notamment pour la manipulation des données multimédia comme les textes ou les images, les cartes. . .

Les types abstraits sont une ouverture sur l'extérieur du SGBD. Ils sont encapsulés : le SGBD ne dispose que d'une interface (ensemble de méthodes) pour les manipuler. Il ignore leur structure, leur implantation et leur stockage.

Le SGBD n'accède donc à des valeurs de type abstrait que par appel de méthodes. Il effectue dynamiquement le lien entre ces méthodes et leurs paramètres.

Ce lien est possible grâce à la définition du type abstrait dans une classe (entité, table ou type selon le modèle) particulière du SGBD qui décrit son interface. Les méthodes et l'implantation du type sont développées dans un langage de programmation interfacé avec le SGBD : C pour POSTGRES et E (extension de C++) pour EXODUS.

Parmi ces méthodes, on distingue les méthodes de base qui permettent d'effectuer des opérations élémentaires obligatoires sur les objets (insertion, suppression. . .) des autres méthodes qui sont les opérateurs définis par les utilisateurs. Pour plus de facilité d'écriture dans les requêtes, les méthodes utilisateurs peuvent être renommées à l'aide d'un symbole.

3.2.1.2 Définition de nouveaux opérateurs

Les SGBD relationnels fournissent un ensemble d'opérateurs correspondant à l'union de l'algèbre relationnelle et de certains agrégats (max, min, moyenne. . .). Les SGBD extensibles étendent cet ensemble soit en prédéfinissant de nouveaux opérateurs, soit en laissant les utilisateurs en rajouter :

- opérateurs déductifs : récursion prédéfinie dans PROBE et STARTBUST, gestion de règles de production dans POSTGRES,
- opérateurs spatiaux : opérations géométriques sur des polygones dans PROBE et DASDBS,
- opérateurs documentaires : recherche d'informations, accès sur le contenu de textes dans DASDBS,

- opérateurs temporels : gestion de versions dans POSTGRES ou PROBE.

Le langage de requête prend aussi en compte les opérateurs portant sur les nouveaux types de données ou des fonctions et des procédures exécutant des traitements plus complexes que les opérations d'un LMD.

Par exemple, on a pu voir au paragraphe précédent que les SGBD extensibles autorisent la définition de méthodes associées aux types abstraits, ces méthodes peuvent être invoquées dans une requête.

3.2.1.3 Implantation de nouvelles techniques d'accès ou de stockage

Les SGBD relationnels du commerce utilisent des techniques générales (B+tree, hashcode), les SGBD extensibles peuvent prendre en compte des techniques spécifiques d'une application (par exemple les R+trees [Sellis 87] ou les Z-order [Manola 87] dans PROBE pour les données spatiales).

Les SGBD extensibles autorisent donc l'utilisation de nouveaux index développés par leurs utilisateurs. Ces index doivent améliorer les performances du SGBD dans des applications spécifiques.

Le domaine des applications géographiques se prête parfaitement à ce genre d'optimisation. Il existe un grand nombre d'index prévus pour les opérations multi-dimensionnelles : R+tree [Sellis 87], Grid File [Nievergelt 84], Bang File [Freston 87] ...

La mise en oeuvre de ces structures demande le développement d'algorithmes, leur insertion dans le SGBD et leur prise en compte dans l'optimisation des requêtes. La plupart des SGBD extensibles proposent leur intégration à l'aide du mécanisme de type abstrait.

exemple 13 : définition d'un type polygone dans POSTGRES

```
define abstract data type polygone is
(
 inputProc = ChartoPolygone  /* procédure d'insertion dans le SGBD */
 outputProc = PolygonetoChar       /* suppression du SGBD */
 Filename = "PolygoneCode"
)
```

Cette déclaration étend l'ensemble des types de POSTGRES en y ajoutant le type abstrait "polygone". Les valeurs de ce nouveau type sont codées puis stockées dans de simples chaînes de caractères. Le fichier PolygoneCode contient les codes des procédures d'insertion, de suppression et d'accès. La déclaration de ces procédures doit contenir une indication de leur efficacité en cas de choix lors de l'optimisation des requêtes.

· Les générateurs de SGBD peuvent aussi autoriser des index spécifiques, ils permettent même de les intégrer directement dans le code du SGBD et donc dans l'optimiseur de requêtes.

3.2.1.4 La modification ou la génération d'un modèle de données

On distingue deux catégories de SGBD extensibles: les SGBD fournissant un noyau de base et des possibilités d'extension (ETIC, PROBE, POSTGRES, STRABUST. . .) et les générateurs de SGBD qui offrent un certain nombre d'outils pour écrire soi-même "son" SGBD (DASDBS, EXODUS, GENESIS...). La modification ou la génération du modèle de données est une fonctionnalité de la seconde catégorie.

Les générateurs de SGBD ont été conçus en admettant le principe qu'il n'existe pas de modèle de données universel convenant à toutes les applications. Ils proposent donc de générer le modèle qu'il convient pour chaque cas.

Certains SGBD extensibles ont adopté une autre démarche (voir PROBE et POSTGRES dans la section 3.2.2) : ils offrent un modèle de données très riche sachant que chaque application n'en utilisera qu'une partie.

3.2.1.5 Gestion des données géographiques dans les SGBD extensibles

La plupart des SGBD extensibles prévoient la gestion des données géographiques. Ils proposent des solutions aux trois problèmes exposés au début du chapître 3.

Ces SGBD visent la définition de nouveau types de données notamment comme on l'a vu au paragraphe 3.2.1, pour la gestion des objets complexes. Ce type de modélisation permet la description des organisations typiques des SIG de la même façon que pour l'exemple 10.

Les autres possibilités de la génération des SGBD extensibles (définition de nouveaux opérateurs et implantation de nouvelles techniques d'accès ou de stockage) permettent la gestion des données multimédia (P2) et l'implantation d'opérateurs spatiaux efficaces (P3).

En règle générale, ces SGBD prévoit la modélisation des applications géographiques, cependant chaque système est un cas particulier dont certaines fonctionnalités diffèrent des grands principes précédemment évoqués. Par la suite on va donc présenter trois SGBD extensibles et préciser leur réponse aux problèmes de la gestion des données géographiques.

3.2.2 Quelques exemples de SGBD extensibles

3.2.2.1 PROBE : une base de connaissances

PROBE [Dayal 86] est à l'origine un projet sur les bases de connaissances. Cependant la technologie et les possibilités offertes le placent parmi les SGBD extensibles. PROBE repose sur le modèle de données fonctionnel DAPLEX [Shipman 81].

Une application est modélisée en terme de types et de classes comme dans les SGBDOO mais PROBE la transforme en modélisation DAPLEX à base d'entités et de fonctions. Elles permettent la représentation d'objets complexes tels que ceux décrits dans l'exemple 10:

exemple 14 : modélisation de cartes et d'objets avec DAPLEX

```
DECLARE carte() -> ENTITY
DECLARE nom(carte) -> string
DECLARE échelle(carte) -> int
DECLARE objet() -> ENTITY
DECLARE nom(objet) -> string

DECLARE obj(carte) ->> objet     (->> signifie fonctions multivaluées)
DEFINE sacarte(objet) -> carte INVERSE OF obj(carte)
```

Les entités "carte" et "objet" représentent les objets complexes, les fonctions "nom", "échelle" sont des attributs simples d'objets alors que "obj" et "sacarte" modélisent la composition d'objets.

DAPLEX permet un sous-typage implicite au niveau des entités à l'aide de la notion de fonction. L'héritage se traduit alors simplement par la composition de fonctions :

exemple 15 : déclaration de "ville" comme sous-entité d' "objet"
héritage de la fonction nom sur "ville"

```
DECLARE ville() ->> objet      plutôt que     DECLARE ville() -> ENTITY
PRINT nom(ville)               sans déclarer  DECLARE nom(ville)-> string
```

L'interrogation de cette base de données utilise la notation fonctionnelle classique. Pour exprimer des requêtes, les utilisateurs doivent cependant manipuler un langage complexe très éloigné des LMD traditionnels comme SQL ou QUEL.

exemple 16 : "liste de noms des objets de la carte de la france au 50000éme"

```
FOR EACH objet SUCH THAT nom(sacarte(objet)) = 'France'
```

```
                                AND  echelle(sacarte(objet)) = 50000
PRINT nom(objet)
```

PROBE conserve les acquis de ce modèle et propose les extensions suivantes :

- gestion des versions des entités,
- définition d'une classe d'éléments géométriques avec un opérateur de jointure spatiale associée,
- possibilité de déduction avec un opérateur de récursion.

La modélisation d'applications géographiques passe donc par l'utilisation de la géométrie fournie par le système. Il s'agit d'une géométrie approchée destinée à fournir rapidement des résultats approximatifs : la jointure spatiale est un opérateur fournissant un sur-ensemble des solutions à l'aide d'un filtre rapide parmi un très grand nombre d'objets.
Une plus grande précision est à la charge de la couche application au dessus du SGBD.

Les entités du schéma carte-objet-graphiques doivent donc être associées à la géométrie définie par le système [Orenstein 88]. Cette association a lieu lors de la définition de l'application en terme de classe et de méthode.

exemple 17 : définition de la géométrie approchée pour les objets

```
X       DIMENSION int
Y       DIMENSION int
map     SPACE x(0), y(0)

CLASS position IS POINT IN map
  (x: X, y: Y)

CLASS objet IS ENTITY IN map
  (nom: string, position: POSITION, dessin: GRAPHIQUE)
```

A chaque entité "objet" on attache une dimension géométrique dans l'espace "map". Dès lors PROBE permet d'appliquer des fonctions de jointure par superposition sur chaque "objet".

exemple 18 : utilisation de la géométrie approchée

```
FOR EACH objet SUCH THAT overlaps(objet,un_rectangle)
PRINT nom(objet)
```

Les entités multimédia comme les images ou les graphiques ne peuvent pas être gérées directement par PROBE. Il propose donc d'utiliser un mécanisme de type abstrait au niveau de l'application.

exemple 19 : application gérant des données géographiques

```
CLASS carte IS ENTITY
  (nom: string, echelle: int , obj: { objet } )
X       DIMENSION int
Y       DIMENSION int
map     SPACE x(0), y(0)
CLASS position IS POINT IN map
  (x: X, y: Y)
CLASS objet IS ENTITY IN map
  (nom: string, position: POSITION, sacarte: carte, dessin: GRAPHIQUE)
ABSTRACT DATA TYPE graphique
```

PROBE implante des opérateurs DAPLEX et donc effectue automatiquement la transformation de la syntaxe orientée objet en commandes DAPLEX. Par contre PROBE n'altère pas les commandes liées aux types abstraits. L'implantation de ces types est à la charge des utilisateurs et dépend des applications (deux applications A1 et A2 peuvent implanter et utiliser le type GRAPHIQUE différemment).

PROBE fournit donc des fonctionnalités minimales pour les données géographiques : il permet leur modélisation et l'utilisation d'opérateurs géométriques. Par contre le SGBD ne gére pas directement de données multimédia et n'assure pas la cohérence de ces types de données d'une application à l'autre.

3.2.2.2 POSTGRES : un SGBD relationnel étendu

POSTGRES [Stonebraker 86b] est le successeur d'INGRES, à partir du modèle relationnel, il vise les applications multimédia et l'intelligence artificielle. Pour cela, il propose plusieurs mécanismes d'extension au modèle relationnel:

- gestion de nouveaux types de données,
- définition de types abstraits,
- l'intégration d'un système de régles de production et de déclencheurs,
- l'héritage et l'identification par clé sur les relations,
- gestion de l'historique.

Il prévoit aussi l'utilisation de nouveaux supports de stockage et une interface souple avec des langages de programmation (C, LISP). Il conserve les grandes fonctionnalités des SGBD: indépendance donnée-programme, langage de définition de données et langage de manipulation de données.

Pour modéliser et manipuler des données géographiques avec POSTGRES, on doit utiliser les mécanismes offerts pour gérer des objets complexes et des types abstraits.

La gestion d'objets complexes reposent sur l'utilisation du type POSTQUEL. Un champ de ce type a pour valeur une expression du langage de manipulation de données (une requête). Cette expression peut être évaluée à l'aide de l'instruction EXECUTE.

Les types abstraits sont définis par la commande DEFINE TYPE. Elle décrit la liaison entre POSTGRES et les méthodes élémentaires. La commande DEFINE OPERATOR permet la définition des méthodes utilisateurs sur les types abstraits.

L'exemple 19 montre l'utilisation de ces moyens pour une gestion de cartes semblables à celle de l'exemple 10. Le principal problème de cette modélisation est de délimiter la séparation entre les types abstraits et les relations sachant qu'il n'y a pas d'héritage possible sur les types abstraits et qu'on ne peut pas définir d'opérateurs sur les nuplets des relations.

exemple 19 : modélisation de cartes dans POSTGRES

```
CREATE TABLE carte ( nom: string, echelle: int, obj: POSTQUEL)
CREATE TABLE objet ( nom: string, localisation: position,
                          dessin : POSTQUEL , sacarte : string     )
CREATE TABLE graphique (sonobjet: string, origin: position )
CREATE TABLE polygone ( liste: array [1,100] of position     )
INHERITS      graphique
CREATE TABLE cercle ( rayon: real )  INHERITS graphique

DEFINE TYPE position IS
    (
        internallength = 12,              (taille occupée dans le SGBD)
        inputProc = ChartoPosition, (procédure d'insertion dans le SGBD)
        outputProc = PositiontoChar, (procédure de suppression)
        Filename = "PositionCode"     (code des procédures)
    )
```

Dans cette modélisation, on ne peut pas définir d'opérateurs sur les graphiques dans le SGBD. Ils devront être écrits dans la couche application en utilisant ceux déclarés pour le type POSITION.

Une autre modélisation aurait consisté dans la définition de trois types abstraits (POSITION, POLYGONE et CERCLE) ayant chacun une implantation séparée et des opérateurs particuliers.

exemple 20 : autres modélisations de cartes dans POSTGRES

```
CREATE TABLE objet ( nom: string, localisation: POSITION,
                     dessin : POSTQUEL , sacarte : string )

CREATE TABLE polygone (sonobjet: string, graphique: POLYGONE )

CREATE TABLE cercle (sonobjet: string, graphique: CERCLE )

DEFINE TYPE position      /* identique à l'exemple 14 */
DEFINE TYPE polygone
DEFINE TYPE cercle
```

Cette modélisation multiplie le nombre des définitions et supprime les liens sémantiques entre les cercles, les polygones et les positions. Par contre elle permet la définition d'opérateurs efficaces sur les graphiques. Il faut cependant noter que ces opérateurs ne seront jamais hérités.

D'autre part l'utilisation du champ POSTQUEL pour représenter les liens entre les nuplets apporte une nouvelle source de difficultés. L'évaluation de ce type de champs rajoute au LMD une instruction EXECUTE peu ergonomique. Mais surtout le résultat de cette évaluation n'est pas typée. Cela entraine une grande complexité d'utilisation et certainement de mise en oeuvre.

exemple 21 : utilisation du type POSTQUEL

```
INSERT INTO carte
VALUES( 'France',250000 ,
            'retrieve objet.all where sacarte = France')
```

utilisations du champ POSTQUEL:

```
(1)     EXECUTE (CARTE.obj) WHERE CARTE.nom = 'France'
(2)     EXECUTE (CARTE.obj) WHERE CARTE.nom = 'Suisse'
```

· CARTE

nom	echelle	obj
France	250000	'retrieve objet.all where sacarte = France'
Suisse	250000	'retrieve objet.all where sacarte = Suisse'
Belgique	250000	'retrieve objet.all where sacarte = Belgique'

OBJET

nom	localisation	dessin	sacarte
Paris	12098453	'retrieve graphique.all where sonobjet = Paris'	France
Lyon	349876534	'retrieve graphique.all where sonobjet = Lyon'	France
Geneve	49479208	'retrieve graphique.all where sonobjet = Geneve'	Suisse
Bruxelle	58960012	'retrieve graphique.all where sonobjet = Bruxelle'	Belgique

figure 3

La commande (1) renvoie un ensemble d'objets { Paris, Lyon }, (2) retourne seulement Genève. On peut même avoir dans un champ POSTQUEL, plusieurs requêtes renvoyant des types différents. . .

On constate donc que la modélisation et la manipulation des données géographiques est possible dans POSTGRES mais qu'elle est plus complexe que dans les exemples précédents (SGBDOO et PROBE). Par contre POSTGRES offre des fonctionnalités BD que ne possèdent pas les SGBDOO (indépendance logique-physique, LDD, LMD) et permet l'utilisation uniforme de types abstraits contrairement à PROBE.

3.2.2.3 EXODUS : un générateur de SGBD

EXODUS [Carey 86] est un générateur de SGBD. Il n'offre pas de modèle de données prédéfini mais seulement un système de stockage et un système de gestion de types.

Pour utiliser EXODUS, il faut développer soi-même son SGBD. Ce travail n'est pas simple bien qu'il soit guidé : EXODUS offre un générateur automatique d'analyseurs syntaxiques et d'optimiseurs de requêtes. Le langage E extension de C++, permet au DBI (Data Base Implementor) d'écrire les autres extensions qu'il souhaite :

- composants fixes d'EXODUS :
  - système de stockage,
  - gestionnaire de types.
- outils du DBI :
  - langage E,
  - générateur d'analyseurs syntaxiques,
  - générateur d'optimiseurs de requêtes.
- librairie disponible :
  - méthodes d'accès sur disque (B+tree, Hashing. . .),
  - opérateurs algèbriques,
  - système de départ avec modèle orienté objet.

Aucun modèle de données n'est défini à priori, le DBI le choisit en fonction de son application. Cependant il existe une version de départ d'EXODUS, elle implante un modèle orienté objet EXTRA[Carey 88] et propose un langage de requêtes EXCESS (extension du langage QUEL défini pour INGRES). Cette version d'EXODUS offre les fonctionnalités classiques des SGBD :

- indépendance données-programme,
- langage de définition de données,
- langage de manipulation de données.

En plus, la version EXTRA/EXCESS permet certaines possibilités des SGBDOO :

- identité d'objet : les objets sont définis par la commande **create** et identifié par un nom,
- objet complexe : les constructeurs ensemble, tableau et nuplet sont offerts. Ils peuvent être utilisés sans aucune restriction,
- type et classe : les objets sont typés et appartiennent aux classes définies par leur type,
- encapsulation : on peut définir des fonctions et des procédures à l'aide du langage EXCESS. Elles sont rattachées à une classe,

- héritage : EXTRA admet l'héritage multiple. Un sous-type hérite de la définition, des fonctions et des procédures de ces super-types.

EXTRA/EXCESS ne prévoit pas les méthodes génériques mais il permet la définition de types abstraits :

exemple 22 : définition du type abstrait texte dans EXODUS

```
define abstract data type texte
(
 interface = /usr/exodus/adts/texte.h
 code = /usr/exodus/adts/texte.e
)
```

Le fichier texte.h contient la liste de méthodes:

```
dbclass texte
(
public
        texte(string)
        texte()
texte change_auteur(string)
string     auteur()
booleancherche_mot(string)
void  afficher()
)
```

Les types abstraits peuvent être utilisés soit pour gérer les données multimédia soit pour implanter de nouvelles méthodes d'accès. Dans l'exemple des textes, on peut par exemple utiliser une méthode de signature pour effectuer une recherche par contenu.

exemple 23 : utilisation du type texte dans EXODUS, recherche des documents traitant des bases de données

```
define type document
        (
         titre     : string
         auteur : emp
         texte     : texte
        )
```

```
define operator "$" for boolean cherche_mot

range of D is document
retrieve D.nom,D.titre where D.texte $ 'base de données'
```

L'utilisation d'EXODUS pour les applications géographiques peut conduire à deux démarches différentes :

- modélisation à l'aide du système EXTRA/EXCESS,
- génération d'un SGBD spécifique de l'application.

La modélisation d'application géographique à l'aide d'EXTRA reprend les problèmes de l'utilisation des SGBDOO (voir exemple 10). Cependant la gestion des données multimédia peut ëtre envisagée à l'aide du mécanisme de type abstrait et l'utilisation du SGBD est facilitée par l'existence d'un LMD étendu à partir d'un langage connu (QUEL).

La génération d'un SGBD à partir des outils offerts par EXODUS pose de nombreux problèmes. Elle comprend plusieurs tâches encore peu étudiées :

- définir les besoins de l'application en termes de gestion de données,
- déduire de ces besoins un modèle de données : descriptions et manipulations des données,
- identifier les méthodes d'implantations les plus efficaces pour ce modèle,
- spécifier ces méthodes par rapport aux utilitaires offert par EXODUS,
- développer ces méthodes à l'aide des générateurs automatiques et du langage E,
- intégrer le logiciel complet et vérifier qu'il répond aux besoins exprimés dans le premier point.

Ce mécanisme de génération est une approche prometteuse, elle soulève encore bien des questions qui seront certainement résolues dans les années futures :

- rôle du DBI : qui sera en mesure de l'assumer ?
- maintenance du logiciel : en plus de la notion de mise à jour du schéma conceptuel, on peut songer à celle de mise à jour du modèle de données...

3.3 L'APPROCHE CHOISIE PAR MATRA

Le bilan de cette revue des nouveaux SGBD est plutôt positif : les problèmes des applications géographiques n'ont pas été ignorés. La plupart des projets permettent une bonne modélisation (dans l'exemple choisi, il s'agit de la cartographie) et supportent des données multimédia. Le

problème de la modélisation et de l'exécution des requêtes spatiales est moins traité. Seul PROBE propose une géométrie approchée intégrée [Orenstein 88] mais ce système est limité dans la gestion des données multimédia.

Devant l'absence de projet de recherche entièrement satisfaisant et suite à la conclusion négative de l'étude des systèmes commercialisés, MS2I a décidé de spécifier son propre SGBD. Il reprend de nombreux concepts des nouveaux SGBD mais reste guidé par des objectifs précis :

- répondre aux besoins des applications MS2I,
- assurer la continuité avec les gestionnaires de données utilisés dans ses applications.

C'est pourquoi les caractéristiques de l'information géographique que nous avons cherché à prendre en compte sont les suivantes :

- les données atomiques sont multimédia, il peut s'agir de données alphanumériques mais aussi de cartes numérisées, d'images satellites, de photographies, de graphiques ou de textes. . .
- ces données atomiques non alphanumériques sont de grande taille (une image SPOT panchromatique qui représente une zone de 60km sur 60km, occupe 36 mégaoctets),
- les données non atomiques ont une structure complexe avec de nombreux niveaux d'imbrication et pouvant contenir des cycles,
- les données sont multifacettes, en fonction de l'application visée, le même objet géographique peut être considéré différemment (par exemple, en reconnaissance aérienne un pont est vu sous son aspect géométrique, alors que dans un système d'aide à la circulation routière, il est vu sous son aspect débit de véhicules),
- les objets géographiques ont une représentation spatiale (volumique, surfacique ou ponctuelle). Ils sont liés à un référentiel terrain et peuvent être identifiés par leur position,
- les objets géographiques évoluent dans le temps (les photographies satellites d'un même point ne sont jamais identiques, elles décrivent l'historique de ce point).

La modélisation des traitements communs à ces applications nous oblige à prévoir :

- certains traitements d'images (extraction, zooming, ou filtrages simples),
- des requêtes spatiales (mesures de distances ou de surfaces, régions contenant un point, objets contenus ou chevauchant une région).

En plus de ces caractéristiques, nous avons dû respecter les contraintes liées à l'environnement MS2I, c'est-à-dire d'une part la prise en compte des interactions entre les applications (fonctionnant sur des postes de travail

reliés par un réseau) et d'autre part l'intégration des fonctions déjà existantes dans les gestionnaires de données dans ce nouveau SGBD (traitements d'images. . .). La plupart de ces fonctions étaient déjà opérationnelles, elles doivent être récupérées sans trop de modification.

Pour répondre au besoin de communication de données entre les applications, le nouveau SGBD a été défini comme un serveur de données sur le réseau. Chaque application agit comme client : demande de données, réception puis traitement et restitution des résultats. La gestion de la concurrence, de la confidentialité et de l'intégrité est donc assuré par le SGBD.

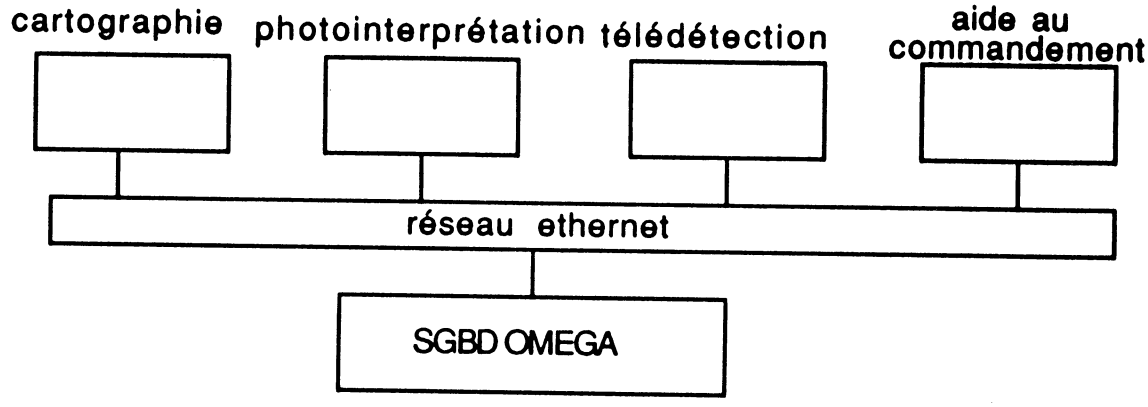


figure 4

Notre approche concernant la modélisation de ces applications, repose sur la définition d'un modèle de données orienté objet et d'un langage de définition et de manipulation de données supportant les possibilités du modèle. Un langage de requêtes permet l'interrogation de la base de données, il est conçu comme l'extension de SQL supportant entre autres les requêtes spatiales et des commandes de traitements d'images.

Les caractéristiques principales de ce modèle (appelé ESTRELLA) sont :

- c'est un modèle multimédia, un certain nombre de types sont prédéfinis (images, textes, graphiques. . .), de nouveaux types de données peuvent être ajoutés dynamiquement avec un minimum de modifications,
- il reprend les principaux concepts de modèles orientés objets : identité d'objet, typage, objet complexe, héritage. . .
- Il prévoit la gestion du temps et des contraintes d'intégrité.

Nous avons porté notre effort sur la définition d'un langage de requêtes simple pour l'interrogation de la base de données. Il est conçu comme une extension de SQL supportant les requêtes spatiales et des commandes simples de traitement d'images.

- Le système implantant ces fonctionnalités conserve les acquis des SGBD traditionnels : gestion de la concurrence, intégrité et confidentialité. Contrairement au SGBDOO, nous souhaitons conserver le principe d'indépendance entre l'organisation logique des données et leur représentation physique. Cependant un mécanisme d'ouverture vers des sous-systèmes implantant des traitements d'images ou des requêtes spatiales efficaces est spécifié.

Deux interfaces entre les SGBD et ses utilisateurs ont été définies :

- une interface interactive basée sur l'entrée des commandes du LDD, du LMD et du langage de requêtes. Cette interface supporte les données multimédia et les objets complexes,
- une interface programmable permettant d'invoquer la plupart des fonctionnalités du modèle.

Dans le chapitre suivant, nous définissons le cadre formel du modèle de données ESTRELLA. On s'inspire des travaux effectués sur les modèles orientés objets comme [Lecluse 87], [Abiteboul 87b] ou [Bancilhon 87a].

CHAPITRE 4

LE MODELE DE DONNEES ESTRELLA

chapitre 4 : LE MODELE DE DONNEES ESTRELLA

4.1 INTRODUCTION

L'objectif d'un modèle de données est d'offrir un ensemble de constructeurs permettant d'organiser et de structurer les données manipulées dans des applications. Le modèle relationnel [Codd 70] [Delobel 82] propose une décomposition des applications à l'aide des notions de domaine, de n-uplet, d'attribut et de relation :

- un domaine est un ensemble de valeurs,
- un attribut est caractérisé par son nom et son domaine,
- un n-uplet est une entité constituée d'attributs,
- une relation est un ensemble de n-uplets de même structure.

Le modèle de données ESTRELLA est un modèle orienté objet. Comme dans tous les systèmes orientés objets, toutes les entités et toutes les données d'une application sont des objets : un simple entier est autant un objet qu'une structure complexe représentant un avion ou un satellite.

Dans les SOO (Systèmes Orientés Objets), un objet est en fait un espace mémoire qui contient son état. Cet espace mémoire est composé des valeurs des attributs de l'objet. Ces valeurs sont elles-même des objets et donc disposent de leurs propres espaces mémoires pour leurs attributs.

Les SOO distinguent les objets de base (entier, réel, ...), qui n'ont pas d'attributs et dont l'espace mémoire est limité à leur valeur, et les objets plus complexes qui ont des attributs pouvant contenir à leur tour d'autres attributs. Ainsi deux objets peuvent avoir des attributs qui référencent le même objet. Par exemple, l'attribut "capteur" d'un cliché aérien référence un objet qui représente un satellite et ce même satellite peut être la valeur d'un attribut "charge-utile" d'un objet représentant une fusée.

En général, le comportement des objets est encapsulé par un ensemble de méthodes. L'appel des méthodes et l'interface des objets ont lieu par passation de messages.

Pour des raisons d'abstractions et de compressions de places occupées, les objets sont regroupés dans des classes. Tous les objets d'une même classe ont les mêmes attributs et les mêmes méthodes. Ils réagissent aux mêmes messages.

Le modèle de données ESTRELLA conserve les grands principes de la modélisation des SOO. Il apporte cependant certaines nuances aux niveaux de la définition des objets. En effet, ESTRELLA utilise les notions de valeurs, de clés pour définir la notion d'objets : un objet est un couple (c,v) où c est sa clé et v sa valeur moins la clé.

On suppose l'existence d'un ensemble de classe de base contenant des objets élémentaires (entiers, réels, chaîne de caractères, date et booléens) pour lesquels c est nulle, et v (valeur moins la clé) est leur valeur. Cet ensemble peut d'ailleurs être étendu à l'aide d'un mécanisme de type

abstrait [Stonebraker 86a] (appelé descripteur).

D'autres objets peuvent être créés en utilisant les constructeurs n-uplet, ensemble et tableau. Leur clé c et leurs valeurs v sont construites comme des n-uplets dont les attributs peuvent être multivalués. Si leur clé est nulle alors ils se comportent comme de simple valeur. Sinon, leur clé est unique et invariante, elle est utilisée pour référencer l'objet comme dans les SOO : l'attribut "capteur" du cliché aérien et l'attribut "charge_utile" de la fusée référencent le satellite par sa clé.

Les n-uplets de la clé c et de la valeur v décrivent la structure de l'objet. Comme dans les SOO, les objets de même structure sont regroupés dans des classes. Par contre, ESTRELLA ne gère pas l'encapsulation des méthodes. Comme dans IRIS [Fishman 87], les méthodes (appelées fonctions) sont des opérations définies indépendamment des classes, applicables sur les objets.

ESTRELLA prévoit aussi la gestion des contraintes d'intégrité et la gestion de versions.

Les objets du schéma conceptuel ESTRELLA peuvent être divisés en trois catégories :

- les classes qui contiennent les objets de la base de données,
- les fonctions qui ne sont pas encapsulées,
- les prédicats qui décrivent des contraintes d'intégrité.

4.2 LES CLASSES

4.2.1 Définition des classes

Une classe modélise un ensemble d'objets de même structure, respectant les mêmes contraintes d'intégrité et subissant les mêmes traitements. Les associations et les liaisons sémantiques entre les objets sont modélisées à l'aide d'attributs permettant la composition d'objets.

Comme pour tous les modèles orientés objet, une classe définit automatiquement un type qui correspond à la structure commune de tous ses objets. En générale, on ne fait qu'une très faible différence entre le type (structure des objets) et la classe (ensemble d'objets de même type).

Par exemple, les photographies aériennes sont des objets. Elles constituent une classe dont la structure est définie par des attributs : coordonnées du centre de la photo, date de prise de vue, type de l'appareil, ou image numérisée. Les associations avec d'autres objets sont réalisées par l'intermédiaire de nouveaux attributs : l'attribut "capteur" référence un objet de la classe des appareils, l'attribut "objectif" contient un objet de la classe des objectifs à photographier...

Les opérations applicables sur les photographies aériennes sont communes à toute la classe : affichage, inverse vidéo, filtrage...

4.2.1.1 Les classes de base

Nous avons prédéfini un ensemble de classes (appelées classes de base) qui représente l'ensemble des objets simples couramment utilisés (appelés objets de base). On trouve classiquement les classes suivantes :

- les entiers,
- les réels,
- les chaînes de caractères,
- les dates,
- les booléens.

Ces classes de base sont réutilisées pour définir les nouvelles classes (les classes utilisateurs ou les classes descripteurs).

4.2.1.2 Les classes utilisateurs

Les classes définies par les utilisateurs forment la partie organisation des données du schéma conceptuel. La structure de ces classes est constituée par le n-uplet de leurs attributs. Certains de ces attributs peuvent être multivalués : ensemble ou tableau (à plusieurs dimensions). Chaque attribut est caractérisé par :

- son nom,
- son type (classe de base ou classe déjà définie),
- une valeur par défaut.

Cette valeur par défaut peut être soumise à des contraintes de manipulation :

- elle peut être une valeur immédiate ou un calcul,
- elle peut être modifiable ou constante (c'est alors une propriété génériques de la classe).

exemple 1 : définition de la classe des OBJETS GEOGRAPHIQUES

```
struct(OBJETGEO) = [ nom : chaîne, coordonnées : { réel },
                     longueur : réel, largeur : réel,
                     type_d_objet : chaîne                ]
```

valeur par défaut dans OBJETGEO : type_d_objet = "point"

Les constructeurs introduits dans la structure d'une classe sont les seuls outils de modélisation à la disposition des concepteurs d'applications. Le choix de ces constructeurs est donc capital. Il dépend essentiellement des applications ciblées par le modèle de données.

ESTRELLA se place dans le cadre des applications géographiques où interviennent les notions de n-uplets (voir exemple 1), d'ensembles (une

carte est un ensemble d'objets géographiques), et de tableaux (une image est un tableau à deux dimensions de pixels).

ESTRELLA autorise l'emploi des tableaux et des ensembles dans les n-uplets mais, par contre, ne permet pas l'utilisation libre des constructeurs. Par exemple les ensembles d'ensembles sont interdits. Pour manipuler ces ensembles d'ensembles ou des ensembles de tableaux, il faut utiliser un n-uplet comme structure intermédiaire. Il s'agit d'une contrainte pour les utilisateurs mais ce type de construction reste, malgré tout, peu utilisé dans les applications MS2I.

En contre-partie, le modèle ESTRELLA offre une vue identique de tous ses objets (tous les objets ont une structure racine n-uplet) comme dans IRIS [Fishman 87].

4.2.1.3 Les classes descripteurs

Le modèle de données ESTRELLA prévoit un mécanisme de type abstrait [Stonebraker 86] pour ouvrir la base de données à des types d'objets gérés extérieurement. Ce mécanisme permet la définition de descripteurs qui représentent parmi les objets du modèle ESTRELLA, les valeurs des types abstraits.

Ces descripteurs sont des objets de classes particulières dites classes descripteurs et ils ont le même comportement que les objets des classes de base.

Le mécanisme de descripteur a été conçu notamment pour permettre la gestion des données multimédia de gros volume (images, cartes, textes, ...) qui souvent nécessitent des supports spécialisés.

La structure d'une classe descripteur contient deux attributs particuliers :

- l'attribut "chemin d'accès" (ou "path") permet au SGBD implantant le modèle ESTRELLA d'atteindre le répertoire (ou "directory") où sont gérés les types abstraits,
- l'attribut "identificateur" identifie la valeur du type abstrait dans son répertoire. Il permet au SGBD de faire le lien entre le descripteur et cette valeur.

Lors de l'insertion d'une valeur de type abstrait, ESTRELLA doit effectuer plusieurs opérations :

- il doit déclencher la création de la valeur dans le répertoire de stockage,
- il doit créer un descripteur,
- il doit établir deux liens : un premier entre l'objet utilisant cette valeur et le descripteur, un second entre le descripteur et la valeur du type abstrait.

De la même façon, lors de la suppression d'une valeur de type abstrait,

ESTRELLA doit effectuer les trois opérations inverses : suppression des liens, suppression du descripteur, suppression de la valeur.

En conséquance, la définition d'une classe descripteur doit être accompagnée de fonctions élémentaires réalisant ces opérations :

- fonction de création (appelée "insert"),
- fonction de suppression (appelée "delete").

En fait, les utilisateurs ne sont pas autorisés à manipuler directement une valeur de type abstrait ni son descripteur. Ces données sont encapsulées sous un couche de fonctions comprennant les fonctions élémentaires, plus d'autres fonctions d'accès courants.

Ces fonctions qui gèrent le stockage et l'accès aux types abstraits, sont développées par l'équipe d'administration de la base de données à l'aide d'un langage de programmation traditionnel (langage C) étendu pour ESTRELLA.

exemple 2 : classe descripteur pour des textes

```
struct(TEXTE) = [  path : chaîne, identificateur : chaîne,
                   taille : entier                          ]
```

Dans l'exemple 2, les textes sont gérés comme des types abstraits. Le SGBD détient des informations sur chaque texte dans la classe descripteur (path, identificateur et taille). Ces informations ne sont pas accessibles par les utilisateurs. Elles sont affectées lors de l'appel de la fonction "insert" sur les textes (voir chapitre 6) et sont utilisées par le SGBD pour accéder aux contenus des textes.

4.2.2 Structure et organisation des classes

4.2.2.1 Structure d'une classe

On peut donner une définition récursive de la structure S d'une classe C :

- S = C si C est une classe de base,
- S = [a1 : S1, a2 : S2, ... , an : Sn] où les ai sont des noms d'attributs et les Si sont soit des structures de classe soit des ensembles ou des tableaux de structures de classe.

On a donc des structures complexes, c'est-à-dire des structures non plates contrairement aux n-uplets des SGBD relationnels. Ces structures sont des arborescences composées d'ensembles de n-uplets et de tableaux.

Les structures complexes permettent une modélisation plus fine et plus proche des types manipulés par les langages de programmation : les n-uplets, les ensembles et les tableaux modélisent les record, array et set du langage PASCAL.

Elles permettent de décrire facilement les objets que l'on peut classer à plusieurs niveaux ou hiérarchiquement : textes (documents puis chapitres, sections et paragraphes), zones géographiques (cartes, objets composites et objets élémentaires), organisations hiérarchiques (entreprises, départements et services), ou même décompositions taxonomiques (véhicules, automobiles, poids lourds...).

4.2.2.2 Clé d'une classe

On définit la clé d'une classe comme un sous-ensemble de ses attributs qui identifie de manière unique chaque objet. Formellement la clé d'une classe est un sous-nuplet de sa structure, et la clé d'un objet est la valeur de ce sous-nuplet. Il est important de noter que la clé d'un objet fait partie de la valeur de l'objet.

exemple 3 : clé de la classe OBJETGEO

```
struct(OBJETGEO) = [ nom : chaîne, coordonnées : { réel },
                     longueur : réel, largeur : réel,
                     type_d_objet : chaîne                  ]

clé(OBJETGEO) = [ nom : chaîne, coordonnées : { réel } ]
```

Tout objet o d'une classe C a une clé unique. On appelle clé(C) le n-uplet définissant la clé d'une classe. Si deux objets ont la même clé alors ils sont identiques.

Les clés sont déclarées explicitement lors de la définition de la classe. Une classe avec des clés est appelée une classe identifiée. Une classe sans clé est appelée une classe non identifiée ou une classe imbriquée. Le comportement des objets de ces classes est identique à celui des objets internes de GUIDE [Krakowiak 87] ou des valeurs de O2 [Lécluse 87].

On définit la persistance comme une propriété caractérisant un objet ou une classe dans une base de données ESTRELLA : un objet est persistant si les opérations telles que sa création et sa destruction doivent être explicitées par les utilisateurs.

Les opérations de création et de destruction des objets non persistants sont pris en charge par la base de données.

Par définition, ESTRELLA attribue la persistance comme suit :

- les objets de base ne sont pas persistants,
- les descripteurs ne sont pas persistants,
- les objets non identifiés ne sont pas persistants,
- Seuls les objets identifiés sont persistants.

Cette définition n'est pas altérable.

Par contre, la propriété de persistance est transmise aux valeurs des attributs d'un objet persistant : Les objets d'une classe imbriquée n'ont pas d'existence propre. Ils sont définis comme valeur d'un attribut pour un objet de classe identifié (notion d'entité imbriquée dans le modèle SOCRATE [Delobel 82]) et par conséquant ils deviennent persistants.

4.2.2.3 Relation d'ordre sur les structures

Nous reprennons la relation d'ordre partielle sur les structures définie dans [Bancilhon 87a] et [Lécluse 87].

Soient deux structures S1 et S2, S1 est inférieur à S2 (noté S1 << S2) si S1 spécialise S2.

Formellement, on définit << par les assertions suivantes :

- << est prédéfinie sur les structures élémentaires de classes de bases : entier << réel,

- si S1 et S2 sont des n-uplets alors
  S1 = [a1 : S11, a2 : S12, ... , an : S1n] et
  S2 = [a1 : S21, a2 : S22, ... , am : S2m] avec n >= m,

  S1 << S2 <=> pour tout i de 1 à m, S1i << S2i

- si S1 et S2 sont des ensembles de structures de classe alors
  S1 = { S'1 } et S2 = { S'2 }

  S1 << S2 <=> S'1 << S'2

- si S1 et S2 sont des tableaux de structure de classe alors
  S1 = array [I1..J1, I2..J2, ... , In..Jn] of S'1 et
  S2 = array [I'1..J'1, I'2..J'2, ... , I'm..J'm] of S'2.

  S1 << S2 <=> m = n
  et pour tout k de 1 à n : Ik = I'h et Jk = J'k
  et S'1 << S'2

Cette relation d'ordre sur les structures permet la définition formelle de l'héritage et de la spécialisation de classes.

4.3 L'HERITAGE

L'héritage est une fonctionnalité classique des SGBD orientés objet. Il permet de modéliser des hiérarchies de types en procédant par raffinement

et enrichissement successifs des types pour obtenir une précision maximum dans la description des objets.

Il favorise la ré-utilisation du code : par exemple, on évite la re-programmation de méthodes sur des sous-types en héritant du code développé pour les super-types.

ESTRELLA offre la possibilité de définir une classe à l'aide de l'héritage. On appelle ce mécanisme la spécialisation. Cette spécialisation peut être multiple.

4.3.1 La spécialisation

Une classe A est une spécialisation d'une classe B (on note A sous-classe de B et B super-classe de A) si et seulement si A a au moins les mêmes attributs que B. La classe A peut aussi avoir des attributs supplémentaires.

Par définition les classes A et B vérifient les propriétes suivantes :

struct(A) << struct(B)
objet(A) c objet(B)
clé(A) << clé(B)

La définition de spécialisations déclenche la mise en action d'un mécanisme d'héritage entre les classes : toute sous-classe hérite des propriétés génériques de ses super-classes ainsi que des valeurs par défaut.

La clé des objets d'une classe est constituée de sa clé propre plus des clés des super-classes.

exemple 4 : définition des sous-classes GARE et COMMUNE comme spécialisation de la classe OBJETGEO.

GARE sous-classe de OBJETGEO

structure et clé propre de VOIE_DE_COMMUNICATION :
structure_propre(VOIE_DE_COMMUNICATION) =
[numéro : entier, nb_voies : entier,
villes : COMMUNE]
clé_propre(GARE) = []

structure et clé de GARE avec héritage :
struct(GARE) =
[nom : chaîne, coordonnées : { réel }, longueur : réel,
largeur : réel, type_d_objet : chaîne, numéro : entier,
nb_voies : entier, villes : COMMUNE]

clé(GARE) = [nom : chaîne, coordonnées : { réel }]

valeur par défaut : type_d_objet = "voie de communication"

COMMUNE sous-classe de OBJETGEO

structure et clé propre de COMMUNE :
structure_propre(COMMUNE) = [nb_habitants : entier, sites : {OBJETGEO}]
clé_propre(COMMUNE) = []

structure et clé de COMMUNE avec héritage :
struct(COMMUNE) =
[nom : chaîne, coordonnées : { réel }, longueur : réel,
largeur : réel, type_d_objet : chaîne, nb_habitants : entier,
sites : OBJETGEO]

clé(COMMUNE) = [nom : chaîne, coordonnées : { réel }]

valeur par défaut : type_d_objet = "fleuve"
nb_habitants = 30 000

Les classes GARE et COMMUNE sont des spécialisations de la classe OBJETGEO. Elles héritent des attributs "nom", "coordonnées", "longueur", "largeur" et "type_d_objet"; et définissent de nouveaux attributs respectivement "numéro", "nb_voies" et "ville" pour GARE puis "nb_habitants" et "sites" pour COMMUNE.

Il n'y a pas héritage des objets : les objets de A sont automatiquement des objets de B. Les objets déjà existants dans B avant la création de A ou les objets directement créés dans B ne sont pas propagés dans A. Par exemple, tous les fleuves et toutes les voies de communications sont des objets de la classe OBJETGEO, mais il peut exister des objets d'OBJETGEO qui ne soient ni des voies de communications ni des fleuves.

4.3.2 La spécialisation multiple

Une classe A peut être définie comme spécialisation multiple de plusieurs classes Bi. Elle vérifie alors les mêmes propriétés que pour la spécialisation simple avec chaque super-classe :

pour tout i : struct(A) << struct(Bi)
objet(A) c objet(Bi)

clé(A) << clé(Bi)

Tout objet de A est alors objet de tous les Bi, on note A sous-classe des Bi.

exemple 5 : OBJET_PHOTOGRAPHIE sous-classe d'OBJETGEO et CLICHE

struct(OBJETGEO) = [nom : chaîne, coordonnées : { réel },
longueur : réel, largeur : réel,
type_d_objet : chaîne]
struct(CLICHE) = [numéro : entier, centre : point, qualité : chaîne,
angle : réel, photo : image, capteur : satellite]

struct(OBJET_PHOTOGRAPHIE) << [nom : chaîne, coordonnées : { réel }, longueur : réel, largeur : réel, type_d_objet : chaîne, numéro : entier, centre: point, qualité : chaîne, angle : réel, photo : image, capteur : satellite]

Les objets de la classe OBJET_PHOTOGRAPHIE héritent des attributs de OBJETGEO et CLICHE. Tout objet de cette classe appartient aussi aux classes OBJETGEO et CLICHE. La clé d'OBJET_PHOTOGRAPHIE est constituée des clés d'OBJETGEO et de CLICHE.

4.3.3 Le treillis de classes

On peut définir une relation d'ordre <c sur les classes traduisant la spécialisation simple et la spécialisation multiple.

Soient deux classes C1 et C2,

C1 <c C2 <=> C1 = C2
ou bien
C1 sous-classe de C2, c'est-à-dire
struct(C1) << struct(C2)
objet(C1) c objet(C2)
clé(C1) << clé(C2)

<< étant elle-même une relation d'ordre sur les structures, <c est de façon évidente réflexive, antisymétrique et transitive.

On définit deux classes : UNIVERSAL et NIL telles que toute classe C vérifie :

NIL <c C <c UNIVERSAL

Les classes descripteurs sont définies comme les sous-classes d'une

classe prédéfinie appelée DESCRIPTOR, cette classe étant elle-même une sous-classe directe d'UNIVERSAL.

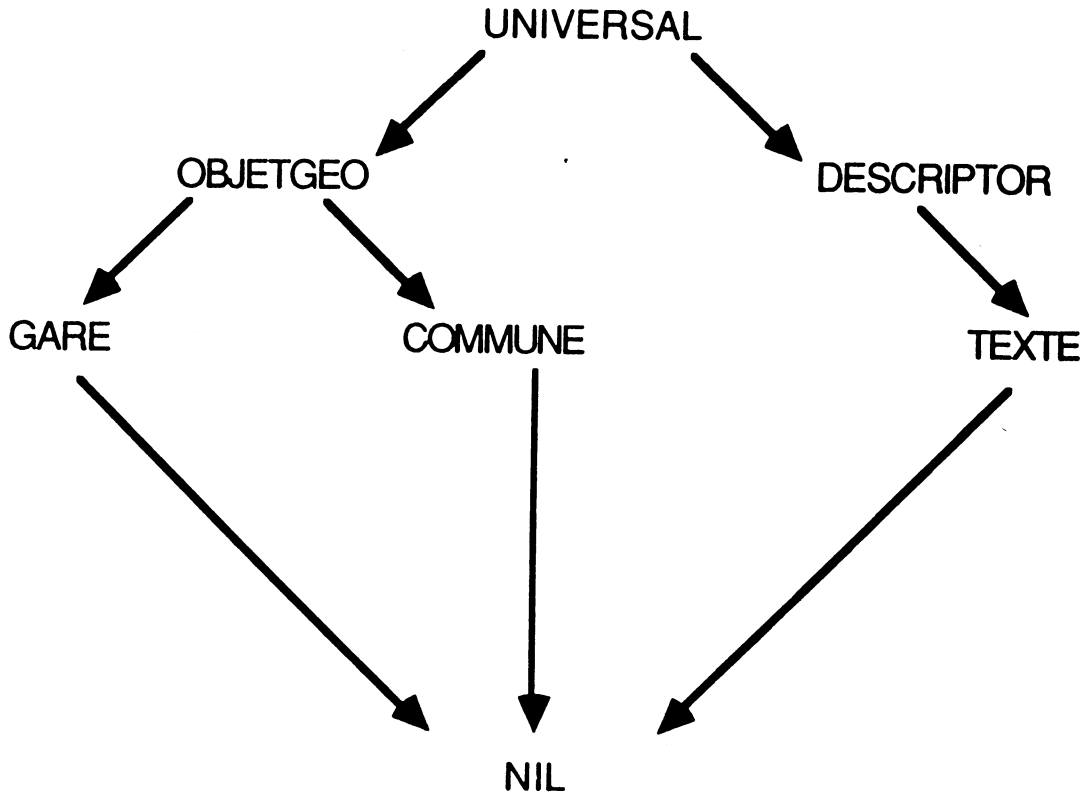
On démontre facilement que l'ensemble des classes muni de la relation <c forme un treillis dont les extrémités sont UNIVERSAL et NIL.

On distingue dans le treillis :

- les classes spécifiques (ou prédéfinies) qui forment quatres groupes :
  - les classes extrémités UNIVERSAL et NIL,
  - les classes descripteurs qui sont sous-classes de la classe DESCRIPTOR et qui permettent la gestion des types abstraits,
  - la classe "FUNCTION" qui gére les descriptions des traitements applicables aux objets,
  - la classe "PREDICATE" qui contient les prédicats décrivant les contraintes d'intégrité.

- les classes de base,

- les classes définies par les utilisateurs.



Les flèches représentent la relation <c

Figure 1: TREILLIS constitué des classes UNIVERSAL, NIL OBJETGEO, GARE, COMMUNE, DESCRIPTOR et TEXTE

4.3.4 Identification des attributs hérités

La spécialisation multiple pose le problème de l'identification des attributs hérités dans une classe. Supposons deux classes B et C ayant un attribut de même nom "x", et A une sous-classe de B et C dans laquelle on redéfinit un attribut supplémentaire "x".

A hérite d'un attribut "x" de B, d'un autre "x" de C et dispose de son propre attribut de même nom. Il faut pouvoir distinguer les trois significations de "x". Ce problème a déjà été résolu par de nombreux systèmes orientés objets, trois types de solutions sont généralement utilisés :

- la redéfinition locales des attributs en conflits [Lécluse 89],
- la redéfinition des noms des attributs hérités [Carey 88],
- l'utilisation de notation préfixée pour désigner l'origine de l'héritage [Fishman 87],
- l'utilisation de règles par défaut [Banerjee 87].

Dans le modèle ESTRELLA, nous avons choisi la troisième solution,

c'est-à-dire l'identification de l'attribut par l'ajout d'un préfixe donnant son origine.

exemple 6 : attributs "x" dans A

A sous-classe de B et C
struct(B) = [b1 : S1, x : S2]
struct(C) = [c1 : S3, x : S4 , c2 : S5]

struct(A) << [b1 : S1, **B.x** : S2, c1 : S3, **C.x** : S4 , c2 : S5, **x** : S6]

Dans l'exemple 6, lors des manipulations d'un des attributs "x" de la classe A, la provenance de "x" sera déterminée grâce à son préfixe : A.x (attribut propre de A), B.x (attribut hérité de B) ou C.x (attribut hérité de C).

Dans le cas d'un héritage sur plusieurs niveaux, le préfixe peut être limité à n'importe quel nom de classe discriminant.

exemple 7 : A sous-classe de B et C, C sous classe de D

struct(D) = [d1 : S1, x : S2]
struct(C) = [d1 : S1, D.x : S2 , c1 : S3]
struct(B) = [b1 : S4, x : S5]

struct(A) = [d1 : S1, C.D.x : S2 , c1 : S3, b1 : S4, B.x : S5]

L'exemple 7 montre l'héritage multiple de x dans A sur plusieurs niveaux. L'attribut "x" hérité de B est déterminé par B.x. L'attribut "x" hérité de D est nommé par C.D.x, on peut aussi utiliser les notations D.x et C.x qui sont parfaitement discriminantes.

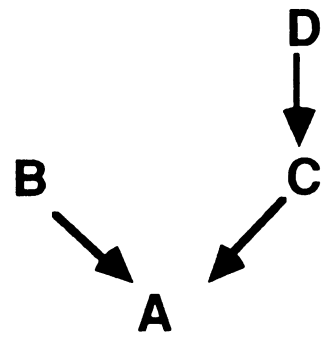


Figure 2 : treillis de l'exemple 7

4.3.5 Chemin multiple dans le treillis

La spécialisation multiple permet la définition de chemins multiples dans le treillis.

exemple 8 : chemins multiples dans le treillis

A sous-classe de B et C
B sous-classe de D
C sous-classe de D

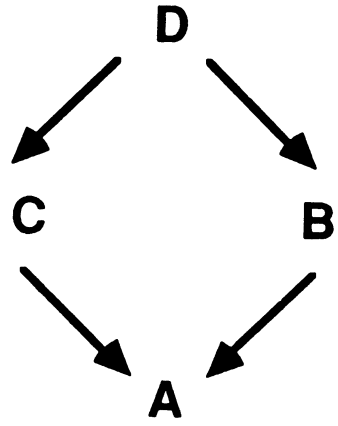


Figure 3 : treillis de l'exemple 8

Ce type de définition conduit à un héritage double d'une classe D dans la classe A. Au sens de la définition de la spécialisation, la clé de A contient la clé de B et celle de C, donc deux fois celle de A : un objet de A représente deux objets de D.

exemple 9 : clé des classes A, B, C et D

clé(D) = [d1 : S1]
clé(B) = [b1 : S2, D.d1 : S1]
clé(C) = [c1 : S3, D.d1 : S1]
d'où clé(A) = [a1 : S4, B.b1 : S2, B.D.d1 : S1, C.c1 : S3, C.D.d1 : S1]

La clé étant une identification unique, les deux objets de D sont distincts alors qu'ils seraient un même objet dans A. Pour supprimer ce paradoxe, on interdit la définition d'héritage multiple conduisant à hériter plusieurs fois d'une même classe (sauf pour la classe UNIVERSAL). Il n'y a donc pas de chemin multiple dans le treillis.

4.3.6 Héritage et identité d'objet

La notion de clé et d'identité d'objet impose certaines restrictions sur l'utilisation de la spécialisation : nous imposons l'uniformité et l'homogénéité des sous-graphes du treillis.

Par exemple, la spécialisation d'une classe imbriquée (classe sans clé) ne peut pas être identifiée : on ne peut pas lui donner une clé. Inversement, la spécialisation d'une classe identifiée est automatiquement identifiée puisqu'elle hérite de la clé de sa super-classe.

exemple 10 : spécialisation identifiée d'une classe imbriquée

POINT sous-classe de UNIVERSAL
  struct(POINT) = [x : réel, y : réel, z : réel]
  clé(POINT) = []

AMER sous-classe de POINT

struct(AMER) = [x : réel, y : réel, z : réel, nom : chaîne]
clé(AMER) = [nom : chaîne]

La définition de l'exemple 10 divise les points entre les points identifiés par un nom appelés les amers et les points sans identité. Les amers étant identifiés sont persistants et ont une existence propre. Les autres points ne subsistent que s'ils sont utilisés comme composants d'un objet identifié.

AMER étant une sous-classe de POINT, tous les amers sont des objets de la classe POINT. Il y a donc une différence de comportement entre objets d'une même classe contrairement au principe de définition des classes.

Cette différence n'est pas admise dans le modèle ESTRELLA, c'est pourquoi nous rejetons ce type de définition.

4.3.7 Opérations dans le treillis

Deux types de mise à jour sur le treillis sont significatifs : les modifications de la définition des classes (ajout ou suppression d'attributs) et les modifications des liens dans le treillis. Ces dernières représentent la création et la destruction des classes ainsi que la redéfinition de l'héritage entre les classes (ajouter ou supprimer une super-classe). On trouve la sémantique de ces opérations pour les modèles orientés objets dans [Kim 88].

ESTRELLA ne retient qu'une partie de ces opérations :

- la création d'une nouvelle classe : cela consiste à définir en extension sa structure, sa clé et ses super-classes,

- la destruction d'une classe : cette opération entraîne aussi la destruction des sous-classes,
- l'ajout d'un attribut non clé,
- la suppression d'un attribut non clé.

Certaines autres opérations comme ajouter ou supprimer une super-classe et ajouter ou détruire un attribut clé, sont impossibles car elles entraînent une modification de l'identité des objets (en modifiant la clé de la classe). On risque soit d'avoir deux objets différents ayant la même identité, soit d'avoir des clés partiellement instanciées ce qui est interdit.

4.3.8 Définition cyclique d'une classe

La définition cyclique d'une classe est possible dans ESTRELLA. Cependant pour des raisons pratiques, ESTRELLA accepte une définition de classe que si toutes ses super-classes et tous les types des attributs de sa structure sont connus. Ce principe interdit les définitions directes comme dans l'exemple 11.

exemple 11 : Structure cyclique de la classe EMPLOYE

```
struct(EMPLOYE) = [  nom : chaîne, prénom : chaîne, age : entier
                     sexe : chaîne, salaire : réel, chef : EMPLOYE ]
```

La définition de la structure de la classe EMPLOYE n'est pas valide car le type de l'attribut "chef" n'est pas encore connu. La définition de la classe EMPLOYE est donc rejetée.

Ce rejet est un handicap pour la modélisation à l'aide d'ESTRELLA car ce type de définition cyclique intervient souvent dans les applications géographiques (par exemple, une figure est un objet composé lui-même d'un ensemble de figures...). C'est pourquoi le rejet direct de ces définitions est compensé par deux mécanismes permettant de retrouver des structures cycliques.

La première méthode consiste à utiliser l'héritage, on la retrouve dans [Rieu 84].

exemple 12 : structures permettant l'utilisation de cycles

```
PERSONNE sous-classe de UNIVERSAL

  struct(PERSONNE) = [nom : chaîne, prénom : chaîne, age : entier ]

EMPLOYE sous-classe de PERSONNE
```

structure_propre(EMPLOYE) = [sexe : chaîne, salaire : réel,
chef : PERSONNE]
struct(EMPLOYE) = [nom : chaîne, prénom : chaîne, age : entier
sexe : chaîne, salaire : réel, chef : PERSONNE]

Dans l'exemple 12, la classe PERSONNE est connue lors de la définition de la structure de la classe EMPLOYE. L'attribut "chef" d'un objet d'EMPLOYE est donc une personne. Cette personne peut ne pas appartenir à la classe EMPLOYE, mais tous les employés sont des objets de la classe PERSONNE.
Lors de la création d'un objet dans EMPLOYE, on peut donc choisir un chef parmi les employés et réaliser ainsi une définition cyclique.

On définit de la sorte, une structure cyclique non stricte, c'est-à-dire dépendante des choix des créateurs des objets.

Plus généralement, on peut énoncer une règle de définition pour les structures cycliques :
Soient les Ai, n classes définies telles que pour tout i, Ai ait un attribut ai de type A i+1.
La définition de An par la structure struct(An) = [... , an : A1 , ...] est rejetée. Par contre on peut définir les classes A0 et An telles que A1 soit sous-classe de A0 et que l'attribut an de la structure de An soit de type A0: struct(An) = [... , an : A0 , ...]

Une autre méthode est possible pour définir une structure cyclique. Elle repose sur la possibilité de modifier la structure d'une classe en ajoutant un attribut.
On procéde en deux étapes :

1- définition partielle de la classe EMPLOYE

struct(EMPLOYE) = [nom : chaîne, prénom : chaîne, age : entier
sexe : chaîne, salaire : réel]

2- ajout d'un attribut de type EMPLOYE

ajouter chef : EMPLOYE à la classe EMPLOYE

La première opération n'est pas rejetée puisque tous les types sont connus. La seconde devient valide lorsque la première a été enregistrée par ESTRELLA. La structure de la classe EMPLOYE devient la suivante :

struct(EMPLOYE) = [nom : chaîne, prénom : chaîne, age : entier
sexe : chaîne, salaire : réel, chef : EMPLOYE]

La structure de la classe EMPLOYE est strictement cyclique. L'attribut "chef" est de type EMPLOYE.

4.4 LES OBJETS

4.4.1 Définitions

Les objets sont les instances des classes. On retrouve les distinctions entre objets telles qu'elles ont été définies pour les classes :

- les objets de base sont les instances des classes de base. Ce sont classiquement des entiers, des réels ou des chaînes de caractères...
- les descripteurs sont les instances des classes descripteurs. Ils représentent les valeurs des types abstraits. Ils répondent au besoin de gestion de données extérieures à la base, notamment des données multimédia,
- les objets des classes utilisateurs sont répartis en deux catégories :
    - les objets identifiés sont directement accessibles par les utilisateurs. Ce sont les instances des classes identifiées.
    - les objets non identifiés sont les instances des classes imbriquées.

La persistance des objets a été définie en fonction de la notion de clé : seuls les objets identifiés sont persistants. Les autres objets ne sont que des valeurs sans identité. Ils ne persistent que s'ils sont contenus dans un objet persistant.

On trouve des définitions différentes dans d'autres systèmes. Dans O2 [Lecluse 87] ou dans EXODUS [Carey 88], les objets persistants sont nommés alors que dans ORION [Banerjee 87], GEMSTONE [Maier 86] et IRIS [Fishman 87], tous les objets sont persistants.

4.4.2 Représentation des associations entre objets

ESTRELLA permet la matérialisation des associations entre deux classes utilisateurs au travers de la composition d'objets : un objet est un n-uplet dont les attributs ont pour valeur d'autres objets.

On distingue deux types de composition : celles liant deux classes de nature indépendante (notion de référence) et celles liant une classe "maître" à une classe "esclave" (notion d'imbrication ou de hiérarchie).

Ce type d'association peut aussi être appliqué sur des descripteurs. Cela permet de lier un objet extérieur à la base de données avec les objets du modèle ESTRELLA.

4.4.2.1 La référence

Soit une classe C contenant un attribut "x" dont le type est B :
struct(C) = [... , x : B , ...]

"x" est une référence de C vers B si et seulement si B est une classe identifiée. On peut dire que "x" référence un objet de B dans les objets de C (notion de pointeur).

Le comportement des objets de C est totalement indépendant du comportement des objets référencés par "x" : par exemple, la destruction d'un objet de C n'affecte pas leur référence. L'identification des classes B et C donnent la propriété de persistance à leurs objets.

Cette persistance ne peut pas leur être otée. Toute association avec une classe identifiée ne peut exister que sous forme de référence.

Les objets référencés peuvent être partagés par d'autres objets :

Soient deux objets O1 et O2 de C, et un objet B1 de B.
O1 = [a1 = V1, ... , x = B1 , ...] et O2 = [a1 = V2, ... , x = B1 , ...]
B1 est partagé par les objets O1 et O2 de C.

La mise à jour de l'objet B1 n'altére pas les objets O1 et O2, elle ne modifie pas la valeur de l'attribut "x" et donc conserve le lien de référence de O1 et O2 vers B1.

Cette mise à jour n'est pas équivalente à une suppression suivie d'une insertion : la destruction de B1 suivie de la création de B'1 crée une "référence folle". Le lien "x = B1" n'est plus cohérent. O1 et O2 ne référencent plus B'1. Ce type d'incohérence peut être combattu par la définition de prédicats (voir la section 6).

L'ensemble des références entre les objets forment un graphe orienté. Ce graphe est complètement indépendant du treillis des classes.

4.4.2.2 L'imbrication (ou hiérarchie)

Soit une classe C contenant un attribut "y" dont le type est A. "y" représente une imbrication d'un objet de A dans C si et seulement si A n'est pas une classe identifiée.

On dit aussi que A est une classe imbriquée : tous les objets de A persistants sont ceux qui sont imbriqués dans un autre objet persistant par l'intermédiaire d'un attribut d'imbrication.

L'existence de chaque objet imbriqué est complétement liée à celle de l'objet qui le possède :

- un objet imbriqué ne peut pas être créé directement mais il est créé lors de la création de l'objet qui le contient,
- Il ne peut être mis à jour qu'à travers les mises à jour de l'objet

le contenant,
- sa destruction correspond à la destruction de l'objet qui lui transmet sa persistance.

Il ne peut pas être partagé par d'autres objets. Par contre, il peut être dupliqué. Cet objet se comporte comme une simple valeur.

On ne peut pas accéder directement aux objets imbriqués, leur valeur n'est accessible qu'à travers celle des objets identifiés qui les contiennent (comme les objets internes de GUIDE [Krakowiak 87]).

4.4.2.3 Les types abstraits

Nous avons cherché à intégrer dans cette notion de type abstrait la gestion des données de gros volume : les images, les textes, et les graphiques. Ces objets sont en effet difficiles à gérer dans un SGBD : ils saturent rapidement la capacité de stockage. On propose donc le mécanisme de type abstrait pour pouvoir ré-utiliser des supports plus adaptés.

Ce mécanisme de type abstrait est traduit par les classes descripteurs. Les descripteurs font le lien entre la donnée et sa sémantique dans les objets ESTRELLA. Ce lien repose sur les principes suivants :
- les données volumineuses sont atomiques,
- elles se comportent comme des objets imbriqués.

Les descripteurs se comportent exactement comme les données qu'ils représentent : ce sont des objets imbriqués non persistants. Chacun dépend donc entièrement de l'objet qui l'utilise.

Cependant contrairement aux objets imbriqués, il n'y a pas de mise à jour possible de descripteurs puisque les données qu'ils représentent sont atomiques.

4.4.3 Composition d'objets et définition de clé

La clé d'une classe a été définie comme un sous n-uplet de la structure. La clé d'un objet est la valeur de la clé de la classe, elle n'est pas modifiable puisqu'elle donne l'identité de l'objet.

La présence d'attributs indirectement modifiables comme les références peut entraîner des incohérences :
- mise à jour indirecte de la clé : en modifiant l'objet référencé, on respecte la règle d'invariabilité de la clé mais on peut altérer la sémantique de la base de données,
- non instanciation de la clé : la suppression de l'objet référencé crée une "référence folle" dans la clé d'un objet,
- impossibilité de création : un circuit de référence dans les clés de

plusieurs classes peut empêcher toute création. Pour instancier complétement une clé, il faut qu'un objet existe déjà. Pour créer cet objet, il aura fallu instancier complétement ses clés : c'est un cercle vicieux.

On interdit donc la définition de clé contenant une référence. Par contre, les imbrications peuvent appartenir à la clé d'une classe.

4.4.4 Egalité entre objets

Soient O1 et O2 deux objets. O1 et O2 sont comparables s'ils appartiennent à la même classe :

- ESTRELLA gère classiquement l'égalité sur les classes de base.
- nous utilisons la fonction "egal" obligatoirement définie entre deux descripteurs,
- si O1 et O2 appartiennent à une classe utilisateur, la comparaison entre O1 et O2 est possible. Elle revient à comparer chacun de leurs attributs. Cet opérateur de comparaison est activé par le symbole "=".

Si O1 et O2 appartiennent à une classe identifiée, ils admettent chacun une clé respectivement c1 et c2. On appelle v1 et v2 leur valeur moins leur clé. La comparaison entre O1 et O2 peut être simplifiée de la façon suivante :

O1 = O2 <=> c1 = c2 et v1 = v2
La clé est un identificateur unique des objets d'une classe donc
c1 = c2 => v1 = v2
d'ou :
O1 = O2 <=> c1 = c2

L'égalité d'objets sur une classe identifiée se confond avec l'égalité des clés de ces objets. Cette simplification facilite la mise en oeuvre du prédicat d'égalité sur les classes identifiées.

Par contre, la comparaison de deux objets de classes imbriquées doit être exécutée de manière systématique sur chacun de leurs attributs.

4.4.5 Opérations sur les objets

Un SGBDOO doit fournir un certain nombre d'opérateurs élémentaires sur les objets. Dans la section précédente, nous avons présenté le prédicat d'égalité, nous abordons maintenant, les manipulations indispensables : création, destruction et mise à jour.

4.4.5.1 Création des objets

L'opérateur de création d'un objet insère une instance dans une classe identifiée. Il la propage dans toutes ses superclasses. Il permet d'affecter tous les attributs définis dans la classe (les attributs propres plus les attributs hérités). Par contre l'affectation d'attributs dans les sous-classes est interdite.

Lors de la création d'un objet, toutes ses clés doivent être instanciées (y compris les clés héritées). Les autres attributs peuvent ne pas être affectés.

La création des objets des classes imbriquées et des descripteurs a lieu pendant la création de l'objet identifié qui les contient.

exemple 13 : création d'objets dans la classe CLICHE

```
o(CLICHE) = [       numéro = 1004,
                    centre = [ x= 12, y = 25.12, z = 0.74 ],
                    qualité = "excellente",
                    photo = [ identificateur = "spot10045" ],
                    capteur = ref(SATELLITE)
              ]
```

Les références ne sont pas directement affectées. Les utilisateurs se servent d'un calcul ou d'une requête pour accéder à un objet déjà créé. Nous symbolisons cet accès par la fonction ref() dans l'exemple 13.

4.4.5.2 Insertion d'un objet dans une sous-classe

Cette opération est définie pour permettre aux utilisateurs de faire coïncider le type de leur objet avec l'évolution de schéma conceptuel. Notamment pour permettre de faire suivre les objets créés avant la définition de sous-classes.

Là encore, nous imposons certaines restrictions qui pourront être levées par la suite :

- un objet n'appartient qu'à une seule sous-classe : il n'y a pas d'objets multitypés sur des classes non liées par la relation d'héritage,
- un objet ne peut être transféré que vers une de ses sous-classes et le retour n'est pas possible.

Ces restrictions n'ont pas de fondement théorique. Elles ne sont que des simplifications facilitant d'une part la suite de la spécification du modèle (le problème du multitypage influence la définition du passage de paramètres aux fonctions) et d'autre part l'implantation rapide d'un prototype.

Ainsi, lors du transfert d'un objet vers une sous-classe, ESTRELLA s'assure de la non-existence de cet objet dans les autres sous-classes. Cette opération permet d'instancier les nouveaux attributs de l'objet, notamment ses nouveaux attributs clés.

4.4.5.3 Mise à jour des objets

L'opération de mise à jour est limitée aux attributs non clés des objets. Les types de ces attributs peuvent être des classes de base, des classes imbriquées, des classes descripteurs ou des classes identifiées. Dans le dernier cas, la mise à jour permet uniquement de changer la référence, dans les autres cas, elle se propage aux sous-objets.

4.4.5.4 Suppression des objets

La suppression d'un objet ne peut être explicite que sur les classes identifiées. Elle entraîne des répercussions dans le treillis et dans le graphe d'objets :

- elle se propage dans les sous-classes et dans les super-classes,
- tous les objets imbriqués et tous les descripteurs sont supprimés,
- les références à cet objet ne doivent pas devenir des "références folles". Elles doivent prendre une valeur "non affectée".

4.5 LES FONCTIONS

4.5.1 Définitions

Une fonction est un opérateur défini par un utilisateur, prenant en entrée une liste d'objets et retournant en sortie un objet. Son créateur la décrit par son nom, par sa signature (liste des types des paramètres d'entrée), par le type de son résultat et par le moyen d'atteindre puis d'exécuter son code.

Comme dans le projet IRIS, les fonctions ne sont pas liées à un type particulier : il n'y a pas d'encapsulation.

Soit f une fonction définie comme suit :

f : s -> r	avec	s = C1 x C2 x ... x Cn	(signature de f)
	et	r = Cr	(résultat de f)

Nous définissons une relation d'ordre partiel notée "<s" sur les signatures à partir de la relation d'ordre "<c" sur les classes :

Soient S1 et S2 deux signatures, S1 et S2 ne sont comparables que si elles ont même taille.

S1 = C11 x C12 x ... x C1n
S2 = C21 x C22 x ... x C2n

S1 <s S2 <=> pour tout i de 1 à n, C1i <c C2i

Cette relation d'ordre sur les signatures nous permet de spécifier de manière formelle l'héritage des fonctions entre les classes bien qu'il n'y ait pas d'encapsulation.

exemple 14 : fonction de reconnaissance d'un objet dans un cliché

fonction = EST_DANS : OBJETGEO x CLICHE -> BOOLEEN
code = est_dans.c (fichier contenant le code de la fonction)

Par définition, toutes les fonctions sont des objets de la classe "FUNCTION". Cette classe est prédéfinie et inaltérable dans ESTRELLA. Sa clé comprend le nom et la signature.

Pour créer une fonction, il faut d'abord disposer d'un code exécutable interfaçable avec le modèle de données ESTRELLA, puis utiliser la procédure de création d'objet (définie dans la section 3) dans la classe "FUNCTION".

FUNCTION est sous-classe de UNIVERSAL
struct(FUNCTION) = [nom : chaîne, signature : array (1..100) of chaîne,
résultat : chaîne, code : chaîne]
clé(FUNCTION) = [nom : chaîne, signature : array (1..100) of chaîne]

4.5.2 Héritage des fonctions

ESTRELLA assure l'héritage des fonctions sur le treillis. Considérons la fonction f définie par f : S -> C, une signature S' et une classe C'. Nous définissons les notions d'applicabilité et de comparabilité des fonctions comme suit :

- f est applicable en S' si et seulement si S' <s S

- f est comparable en C' si et seulement si C' <c C

ESTRELLA autorise donc l'appel des fonctions sur les sous-signatures de la signature de définition S. Les fonctions peuvent donc être employées dans les sous-classes des classes contenues dans S. La relation d'ordre <s

permet de vérifier la validité des passages de paramètres.

ESTRELLA permet les expressions de fonctions sur les sous-classes de leur classe résultat. Les vérifications de type peuvent être effectuées par l'intermédiaire de la relation d'ordre <c sur les classes.

exemple 15 : héritage de la fonction EST_DANS

EST_DANS est définie par OBJETGEO x CLICHE -> BOOLEEN

EST_DANS est aussi applicable sur COMMUNE x CLICHE
ou sur GARE x CLICHE

4.5.3 Fonctions génériques et liaison dynamique

ESTRELLA permet la redéfinition de fonctions de même nom. La clé de la classe "FUNCTION" est composée de deux champs [nom, signature]. Deux fonctions peuvent avoir le même nom à condition d'avoir une signature différente.

exemple 16 : redéfinition de fonctions

Soient deux sous-classes CARTE et SPOT de la classe CLICHE, on définit les fonctions suivantes :

(1) visualiser : CLICHE -> NIL
(2) visualiser : CARTE -> NIL
(3) visualiser : SPOT -> NIL

A la compilation de l'appel d'une fonction plusieurs fois redéfinie, les types des paramètres effectifs sont partiellement connus. Par exemple, si on demande de visualiser un ensemble de clichés, il se peut que certains de ces clichés soient des cartes ou des images spots.

Plusieurs stratégies de compilation sont possibles :
- liaison statique avec la fonction dont la signature est la plus proche (au sens de <s) des types présumés des paramètres effectifs : dans ce cas pour l'ensemble des clichés, la seule fonction utilisée sera la fonction (1),
- liaison dynamique lorsque les types réels de paramètres effectifs seront connus, c'est-à-dire lors de l'exécution : dans ce second cas, les trois fonctions pouront être utilisées pour l'ensemble des clichés.

ESTRELLA effectue la liaison dynamique à l'exécution comme la plupart

des SGBDOO [Bancilhon 88c]. Ce type de liaison permet de raffiner les fonctionnalités d'une fonction selon le type des objets sur lesquels elle s'exécute : par exemple, on utilisera pour les clichés qui sont aussi des cartes, la fonction (2), pour les clichés qui sont des images spots la fonction (3) et pour les autres clichés la fonction (1).

Pour effectuer le choix parmi ces n fonctions, ESTRELLA se base sur la relation d'ordre partielle <s entre les signatures en procédant de la façon suivante :

- on détermine l'ensemble F des fonctions de même nom et de même taille de signature que la signature S des paramètres effectifs,

- parmi les fonctions de F, on calcule l'ensemble des fonctions applicables sur S : F' = { fi / fi in F & fi : Si -> Ri, S <s Si }

- ensuite on recherche la fonction dont la signature est la borne inférieure des signatures de F' au sens de <s :
  fmin : Smin -> R où Smin = min{ Si }

La fonction fmin est la fonction qu'ESTRELLA doit exécuter sur la signature S. Dans le cas de la fonction "visualiser", cette méthode détermine la fonction la plus précise pour chaque cliché.

4.5.4 Identification de fonctions héritées

Le problème du choix de la fonction à exécuter parmi les fonctions applicables sur une signature n'est pas toujours décidable dans le cas de l'héritage multiple. <s n'est qu'une relation d'ordre partiel, de ce fait la borne inférieure d'un ensemble de signatures n'existe pas toujours. Le calcul de fmin n'est donc pas toujours possible.

exemple 17 : définition multiple de la fonction EST_DANS

(1) EST_DANS : OBJETGEO x CLICHE
(2) EST_DANS : GARE x CLICHE
(3) EST_DANS : OBJETGEO x CARTE

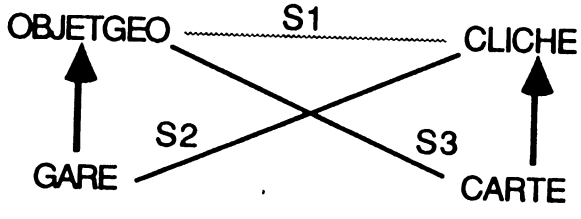


figure 4 : Signatures et Classes de l'exemple 17

Soient g un objet de la classe GARE et c un objet de CARTE.
Le choix parmi 1,2 et 3 pour l'appel EST_DANS(g,c) n'est pas décidable .

Les trois fonctions (1), (2) et (3) sont applicables sur S = GARE x CARTE.
Soient S1 = OBJETGEO x CLICHE,
S2 = GARE x CLICHE,
S3 = OBJETGEO x CARTE.
S2 <s S1 et S3 <s S1 mais S2 et S3 ne sont pas comparables.
Le min { S1, S2, S3 } n'existe pas.

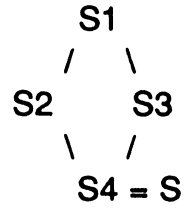


Figure 5 : ordre des signatures

ESTRELLA ne peut pas décider seul quelle fonction employer. Pour résoudre ce problème, les utilisateurs doivent spécifier la fonction EST_DANS sur la signature S4 = GARE x CARTE. Le nouvel ensemble des signatures applicables est Sapp = { S1, S2, S3, S4 }. Il admet une borne inférieure Smin = min{ S1, S2, S3, S4 } = S4.

ESTRELLA doit détecter chaque cas non solvable lors des définitions de fonctions. Il faut rejeter la définition de la fonction qui l'a généré. Dans l'exemple 19, la troisième définition de la fonction EST_DANS est rejetée tant que EST_DANS n'est pas définie sur GARE x CARTE.

4.5.5 Les fonctions sur les descripteurs

Certaines fonctions définies sur les classes descripteurs ont une signification particulière : les opérateurs élémentaires du SGBD (création, suppression, comparaison et affichage d'objets). Ces fonctions sont déclarées comme suit :

créer :	UNIVERSAL -> "classe descripteur"
supprimer :	"classe descripteur" -> NIL
égal :	"classe descripteur" x "classe descripteur" -> NIL
afficher :	"classe descripteur" -> NIL

La fonction "créer" n'est pas directement utilisable. Elle doit être redéfinie avec une signature plus précise que la seule classe UNIVERSAL. D'autres fonctions peuvent être définies sur les classes descripteurs. Elles peuvent, par exemple, implanter des opérateurs spécialisés sur des types

abstraits et mettre en oeuvre de nouvelles méthodes d'accès.

exemple 18 : définition d'un descripteur de rectangle

RECTANGLE sous-classe de DESCRIPTOR
struct(RECTANGLE) = [identificateur : chaîne, nom_fichier : chaîne,
surface : réel, path : chaîne]

valeur par défaut : nom_fichier = "R-tree" non modifiable
path = "usr/spatial_index" non modifiable

La classe "RECTANGLE" est une nouvelle classe descripteur. Tous les rectangles sont stockés dans le même fichier "R-tree" du répertoire "usr/index". On accède à chaque rectangle à l'aide de son identificateur dans le fichier. Les attributs de la structure "rectangle" ne sont pas manipulés par les utilisateurs et ne servent qu'au SGBD pour agir sur les rectangles.

exemple 19 : fonctions élémentaires sur les rectangles

créer :	UNIVERSAL -> RECTANGLE
supprimer :	RECTANGLE -> NIL
égal :	RECTANGLE x RECTANGLE -> NIL
afficher :	RECTANGLE -> NIL

Ces fonctions définissent les opérateurs élémentaires sur les rectangles. La fonction "créer" doit être redéfinie pour être exploitable.

exemple 20 : différentes fonctions pour créer un rectangle

(1) créer :CHAINE	-> RECTANGLE
(2) créer :REEL x REEL x REEL x REEL	-> RECTANGLE
(3) créer :RECTANGLE x REEL	-> RECTANGLE

(1) création d'un rectangle à partir d'une chaîne de caractères
(2) création d'un rectangle à partir de quatre coordonnées : Xmin, Xmax, Ymin, Ymax
(3) création d'un rectangle par homothétie sur un autre rectangle

exemple 21 : autres fonctions sur les rectangles

surface :	RECTANGLE	-> REEL
inclus :	RECTANGLE x RECTANGLE	-> BOOLEEN
intersection :	RECTANGLE x RECTANGLE	-> RECTANGLE

La fonction "surface" n'est en fait que l'accès à l'attribut surface du descripteur (non accessible directement) mais pré-calculé lors de la création du rectangle (voir chapître 6). Les deux autres fonctions sont développées en C, elles peuvent mettre en oeuvre des techniques géométriques ou des index spécifiques comme les R+tree [Sellis 87] ou les Bang File [Freston 87].

4.5.6 Position des fonctions dans le schéma

Dans la plupart des SGBDOO, les fonctions sont encapsulées avec les données dans les structures d'objets ou de classes. Dans ces systèmes, toutes les manipulations de données s'effectuent par exécution de fonctions (appelées méthodes). Elles sont déclenchées par passation de messages. L'encapsulation a l'intérêt de forcer la définition du comportement des objets en même temps que celle de leurs structures et de permettre l'héritage des fonctions simultanément avec celui de ces structures.

Cependant l'encapsulation se prête mal à certaines opérations courantes dans les bases de données. Par exemple, l'encapsulation ne convient pas à l'expression générale de l'accès associatif.

Pour ce faire, il faut définir une méthode de sélection M sur un type d'objets T. Mais elle ne peut pas être appliquée à un ensemble d'objets de type T. Il faut redéfinir une méthode de sélection M', utilisant M, pour un nouveau type T' = { T }.

D'autre part, avec l'encapsulation, il faut rattacher artificiellement des fonctions comme EST_DANS à l'une de ses classes paramètres alors qu'elles jouent toutes un rôle équivalent.

Nous avons choisi de ne pas encapsuler les fonctions. ESTRELLA met en oeuvre l'accès associatif sur toutes les classes dans un langage de requête ainsi que dans une interface avec le langage C. Pour effectuer des manipulations sur les objets, ESTRELLA prévoit soit d'utiliser un LMD (pour des opérations élémentaires), soit d'utiliser des fonctions (pour des opérations moins courantes).

Les fonctions sont des éléments indépendants du schéma conceptuel au même niveau que les objets ou les classes.

4.6 LES PREDICATS

4.6.1 Définition

Un prédicat est défini par un utilisateur pour la vérification d'une contrainte d'intégrité. Il est identifié par son nom comme un objet d'une classe prédéfinie : la classe "PREDICATE" :

PREDICATE sous-classe de UNIVERSAL
struct(PREDICATE) = [nom : chaîne, type : chaîne, corps : chaîne]
clé(PREDICATE) = [nom : chaîne]

Le corps d'un prédicat est de la forme : NULL ou NOT NULL (ensemble), où l'ensemble est le résultat d'une requête sur les classes ou sur d'autres ensembles d'objets.

ESTRELLA interdit la mise à jour de prédicat. Toutes les autres opérations sur les prédicats sont ramenées aux opérations classiques sur les objets de la classe "PREDICATE".

La contrainte d'intégrité est vérifiée tant que l'évaluation du corps du prédicat rend une valeur "vraie".

On distingue trois types de prédicats :

- les prédicats de type "domaine" servant à vérifier les contraintes de domaine,

  exemple 22 : domaines d'attributs dans les classes

  P1 : NULL [{ o.type_d_objet / o in OBJETGEO }
  MOINS
  { "fleuve", "voie_de_communication", "gare",
  "pont", "commune" }]

  P2 : NULL [{ c.qualité / c in CLICHE }
  MOINS
  { "bon", "moyen", "mauvais" }]

- les prédicats de type "intra-classe" permettant de vérifier des contraintes internes à une classe :

  exemple 23 : contrainte sur la taille des objets géographiques

  P3 : NULL [{ o / o in OBJETGEO
  & o.longueur x o.largeur < taille_max(o.type_d_objet) }]

- Le prédicat P3 fait appel à la fonction "taille_max" définie comme suit:

  taille_max : chaîne -> réel

  Ce prédicat ne comporte que des attributs se référant à la classe OBJETGEO.

- les prédicats de type "inter-classe" qui permettent de représenter différentes contraintes comme l'intégrité référentielle ou les partitions sur les sous-classes. Le modèle de données ESTRELLA assure l'intégrité référentielle dans le cas de la spécialisation simple ou multiple ainsi que pour les imbrications. Par contre, une référence peut être non affectée. On peut définir un prédicat qui assure l'existence des références.

  exemple 24 : Intégrité référentielle entre les gares et les communes

  P4 : NULL [{ g / g in GARE }
  MOINS
  { g / g in GARE et g.ville in COMMUNE }]

  P4 calcule l'ensemble des gares dont l'attribut "ville" ne référence pas de commune. P4 est valide si cet ensemble est vide.

  exemple 25 : Partition de la classe CLICHE entre CARTE et SPOT

  P5 : NULL [{ c / c in CLICHE }
  MOINS
  [{ s / s in SPOT} UNION { m / m in CARTE }]
  ]

  P5 calcule l'ensemble des clichés qui ne sont ni des images spots ni des cartes. P5 est valide si cet ensemble est vide, c'est-à-dire s'il y a une partition de la classe CLICHE entre les classes SPOT et CARTE.

  Le type des prédicats tels que P5 et P6 est appelé "inter-classe" puisque leur évaluation demande un calcul sur plusieurs classes.

4.6.2 Sémantique des prédicats

Les prédicats servent à vérifier la validité des opérations sur la base de données. Deux types d'opérations sont à prendre en compte :

- les opérations sur les objets : elles ne sont effectives que lorsque leur

validité a été assurée ce qui consiste à vérifier tous les prédicats concernant l'objet. Lorsqu'un objet n'est pas totalement instancié, les attributs restants ont une valeur théorique appelée "non affecté". Le mécanisme d'évaluation des prédicats tient compte de cette valeur particulière :

t = "vraie", f = "faux", n = "non affecté"

A	B	A&B	AvB	¬A
n	t	n	t	n
n	f	f	n	n
n	n	n	n	

La table de vérité vrai-faux s'accroit de trois lignes supplémentaires qui permettent d'évaluer dans tous les cas les prédicats. Une opération qui conduit à l'évaluation "fausse" ou "non affecté" d'un prédicat est rejetée.

- les opérations sur le schéma : elles ne sont pas soumises à des prédicats. Par contre les prédicats et leurs résultats dépendent de l'évolution du schéma conceptuel des classes et des fonctions.

  exemple 26 : suppression de la classe CLICHE

  les prédicats P2 et P5 n'ont plus de signification dans le schéma conceptuel ESTRELLA. Il faut donc définir un mécanisme qui assure la cohérence des prédicats par rapport au schéma conceptuel.

4.6.3 Cohérence de l'ensemble des prédicats

L'ensemble des prédicats constitue la classe "PREDICATE". Ces prédicats décrivent l'état valide de la base de données. Un certain nombre de contraintes doivent être respectées sur cette classe "PREDICATE" :

- ces prédicats ne doivent pas être contradictoires :

  exemple 27 : deux prédicats incohérents

```
P2 : NULL  [    { c.qualité / c in CLICHE }
                MOINS
                { "bon", "moyen", "mauvais" } ]

P7 : NULL  [    { c.qualité / c in CLICHE }
                MOINS
                { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } ]
```

- ces prédicats doivent être cohérents avec le schéma de la base de données (voir exemple 27).

4.7 LES VERSIONS

La gestion de versions en fonction du temps est un élément important pour les applications des bases de données [Adiba 86] [Adiba 87]. Elle est effectivement capitale pour MS2I. En effet, les objets tels que les cartes, les clichés varient en fonction du temps et représentent les versions de diverses entités géographiques.

4.7.1 Les classes historiques

Nous proposons un modèle simple de versions d'objets. Il s'adresse uniquement aux objets identifiés et consiste en la gestion de l'historique daté de ces objets. Contrairement aux travaux décrits dans [Fauvet 88], nous ne gérons pas d'arbres de versions. En fait les versions d'un objet constituent la liste des états intermédiaires connus de l'objet.

Les utilisateurs décident des objets pour lesquels une gestion des versions est nécessaire. Cependant la gestion des versions n'est pas propre à un objet. Elle est généralisée à toutes les instances d'une classe.

Lors de la définition d'une classe, son créateur doit donc préciser sa nature:

- classe avec gestion de versions : classe historique,
- classe sans gestion de versions : classe simple.

Par défaut, une classe sera gérée sans historique.

La structure d'une classe historique contient un attribut de type "DATE" sur lequel portent les versions. Cet attribut peut être hérité. Il doit faire partie de la clé de classe. S'il n'est pas spécifié lors de la création de la classe, ESTRELLA crée par défaut un attribut "date_instance".

exemple 28 : gestion de versions sur les clichés

```
CLICHE sous-classe de UNIVERSAL
        CLICHE est historique
        struct(CLICHE) = [numéro : entier, centre : point, qualité : chaîne,
                          photo : image, date_de_prise_de_vue : date      ]
        clé(CLICHE) = [numéro : entier, date_de_prise_de_vue : date ]
        attribut de datation : date_de_prise_de_vue
```

Dans le modèle ESTRELLA, les versions sont les objets d'une classe

historique. L'objet représentant l'ensemble des ses versions, appelé objet générique, n'appartient pas à la classe historique car il ne correspond pas à une structure d'objet de classe ESTRELLA :

- une version V est un couple : V = (C,v) avec C = (c,d) où c est la clé de l'objet générique et d la date de la version,
- un objet générique O est l'ensemble de ses versions : O = (c, {di, vi}).

Dans la suite, nous distinguerons les opérations sur les versions des opérations sur les objets génériques. En effet, bien que les objets génériques ne soient pas des objets ESTRELLA, il nous est apparu intéressant de pouvoir les manipuler. De plus, ces manipulations apparaissent comme des extensions simples d'opérations sur les versions : ce sont les mêmes opérations considérées sur un ensemble de versions.

Nous avons défini des opérateurs de manipulations de versions :

- accès à la version courante d'un objet,
- accès aux versions définies sur un intervalle de temps donné,
- accès à toutes les versions (donne un ensemble de versions représentant l'objet générique),
- accès à la version "la plus proche" (par valeur antérieure) d'une date donnée.

Ce dernier opérateur tient lieu de modèle d'interpolation entre deux versions d'un objet.

Une classe historique peut être le type d'un attribut. On peut alors référencer indifféremment n'importe quelle version d'un objet. Par contre, on ne peut pas référencer un ensemble de versions ou l'objet générique lui-même.

4.7.2 Sémantique des classes historiques dans le treillis

Nous allons préciser les notions d'objets, de versions et d'historique par rapport à l'héritage et au treillis des classes.

ESTRELLA permet l'héritage des attributs et des fonctions. Par contre, le modèle de données n'offre pas l'héritage de la gestion du temps. C'est-à-dire que la gestion des versions est locale à la définition propre d'une classe. Elle n'est pas transmise à ses sous-classes.

exemple 29 : nature des sous-classes de CLICHE

SPOT sous-classe de CLICHE

SPOT est historique
structure_propre(SPOT) = [latitude : réel, longitude : réel,

angle_des_capteurs : réel]
clé_propre(SPOT) = []
attribut de datation : CLICHE.date_de_prise_de_vue

CARTE sous-classe de CLICHE

CARTE n'est pas historique
structure_propre(CARTE) = [nom : chaîne, échelle : réel,
thèmes : { chaîne }]
clé_propre(CARTE) = [nom : chaîne, échelle : réel]

La classe CLICHE est historique : la gestion des versions porte sur tous les clichés (notamment sur les cartes et les images spots) mais cette gestion ne concerne que les attributs de la classe CLICHE.

Les objets de la classe CARTE héritent donc partiellement de la gestion de versions de la super-classe CLICHE : ESTRELLA gère les versions de certains de leurs attributs. Par exemple, l'attribut "thèmes" est commun à toutes les versions alors que l'attribut "qualité" dépend de la date de prise de vue.

La gestion des versions est redéfinie sur la classe SPOT. Elle est liée à celle de la classe CLICHE par l'attribut de datation. De ce fait, chaque version d'une image spot correspond à une version d'un cliché.

Les créateurs de classes ont à leur disposition un mécanisme souple de gestion de versions où ils peuvent choisir entre la gestion totale sur tout le treillis, l'absence de version et des gestions partielles selon les attributs.

4.7.3 Opérations sur les classes historiques

Les deux catégories classiques d'opérations sur les bases de données sont concernées par la gestion de versions :

- les opérations sur le schéma conceptuel,
- les opérations sur les objets.

Toutes les opérations sur le schéma conceptuel décrites pour les classes simples peuvent être étendues à des classes avec gestion des versions : création, destruction de classe et ajout ou suppression d'attributs non clés.

Nous définissons deux nouvelles opérations en plus :

1- le passage d'une classe simple à une classe historique,

2- la transformation d'une classe historique en une classe simple.

La première opération prend en entrée l'ensemble des objets d'une classe. et la date D1 de la première version. Elle retourne pour chaque objet un

couple (objet, liste de versions) où la liste de version n'en contient qu'une seule datée de D1. Elle étend la structure de la classe en ajoutant l'attribut de datation.

La seconde opération réalise le traitement inverse. En entrée, elle reçoit les objets et leurs versions ainsi que la date D2 de création d'objets simples. Elle retourne un ensemble d'objets créés à partir des versions de la date D2. Elle supprime l'attribut de datation de la structure de la classe.

4.7.4 Opérations sur les objets

Les opérations sur les objets sont une extension de celles décrites dans la section 4.4.5. Elles ont une signification différente puisqu'elles peuvent s'adresser à des objets génériques ou à des versions :

- création d'un objet générique : l'opérateur de création d'un objet générique est identique à l'opérateur de création de la section 4.4.5.1. Il exécute en plus la création de la première version de l'objet.

- mise à jour d'un objet : la mise à jour d'un objet est encore limitée aux attributs non clés. Elle est propagée à toutes les versions de l'objet.

- création d'une version : une nouvelle version est créée dans une classe à partir de la mise à jour d'une ancienne version avec modification de l'attribut de datation.

- mise à jour d'une version : la mise à jour d'une version est limitée à ses attributs non clés y compris les objets imbriqués. Cette mise à jour n'altére pas la valeur de l'attribut de datation. Cette opération ne peut modifier que les attributs sur lesquels porte l'historique, les autres attributs restent invariants.

- suppression d'un objet : la suppression d'un objet entraîne la suppression de toutes ses versions. Les règles de supression énoncées dans la section 4.4.5.4 restent applicables.

- suppression d'une version : la suppression d'une version d'une classe historique est accompagnée de la suppression des versions identiques dans toutes les sous-classes. Les références à cette version deviennent des "références folles". On peut imaginer des protocoles qui substituent à cette version, une référence vers la version précédente... Dans le modèle ESTRELLA, nous proposons de remplacer la référence par une valeur "non affecté". Si la version supprimée était la version courante, la nouvelle version courante est la version immédiatement précédente.

4.8 CONCLUSION

Dans ce chapître, nous avons présenté le modèle de données ESTRELLA. Il s'agit d'un modèle de données orienté objet défini pour répondre aux besoins exprimés à la fin du chapître 3.

Il reprend les grands concepts des SGBDOO tels que les classes, les objets et l'héritage mais se distingue par l'utilisation de concepts déjà connus en base de données mais non encore adoptés par les modèles orientés objets :

- les objets et les classes sont identifiés par des clés,
- les objets identifiés sont persistants et propagent cette persistence à leurs sous-objets,
- les classes sont organisées en treillis et les conflits d'héritage sont résolus par le préfixage,
- les fonctions ne sont pas encapsulées dans des classes. Elles sont néanmoins utilisables sur les sous-types de leur paramètres formels,
- les contraintes d'intégrité peuvent être exprimées à l'aide de prédicat,
- les versions des objets peuvent être conservées dans des historiques de classes.

Ce modèle de données est en adéquation avec les objectifs fixés à la fin du chapître 3 :

- il permet la gestion de données multimédia à l'aide de la notion de descripteur,
- il favorise la réutilisation des systèmes déjà existants puisque ces descripteurs sont le lien entre le modèle ESTRELLA et d'autres supports informatiques,
- il permet une modélisation simple des applications MS2I : nous donnons, en annexe 1, le contenu d'un schéma conceptuel ESTRELLA pour une application de gestion d'objets planimétriques,
- il est conçu pour prendre en compte l'évolution des objets dans le temps à l'aide d'une gestion simple de versions,
- il permet l'intégration et l'utilisation d'opérateurs géométriques grâce aux descripteurs et aux fonctions qui les manipulent :

Dans les SGBDOO classiques, les opérateurs géométriques ne sont pas directement accessibles. Dans un premier temps, il faut les développer; ensuite ils sont encapsulés avec leur type géométrique et ne sont pas facilement utilisables.

Les opérateurs géométriques ne sont pas directement intégrés au modèle ESTRELLA. Ils doivent aussi être développés. Pour ce faire nous préconisons des techniques spécifiques comme les index spatiaux [Samet 88]. Ensuite, ils sont liés au modèle ESTRELLA par l'intermédiaire des descripteurs (voir

4.5.5).

Dès lors, contrairement aux autres SGBDOO bloqués par l'encapsulation, ils peuvent être appelés dans n'importe quelle situation : directement sur une objet ou sur un ensemble d'objets, via des références ou des imbrications.

Ces appels sont traités à travers les interfaces du modèle ESTRELLA qui sont décrites dans le chapitre suivant et qui sont partiellement mises en oeuvre dans le prototype de SGBD appelé OMEGA.

CHAPITRE 5

LES INTERFACES

chapitre5 : LES INTERFACES

5.1 PRESENTATION GENERALE

L'étude des échanges entre OMEGA et ses applications a amené la définition de deux interfaces :

- une interface interactive avec les utilisateurs,
- une interface avec un langage de programmation (langage C).

L'interface interactive repose sur le langage SQLOMEGA qui supporte les fonctionnalités traditionnelles des langages de bases de données :

- description des données : création, suppression et mise à jour sur le schéma conceptuel (classes + fonctions + prédicats),
- manipulation de données : création, suppression et mise à jour sur les objets,
- interrogation des données : capacité à retrouver des objets suivant leur valeur.

Contrairement à de nombreux SGBDOO, OMEGA offre un langage de requêtes ce qui nous paraît complètement justifié pour plusieurs raisons :

- l'accès associatif est indispensable dans les applications géographiques,
- les requêtes souhaitables dans nos applications ne sont pas toutes déterminées à l'avance,
- la plupart des applications sont interactives.

L'interface interactive utilise un gestionnaire d'écran avec multi-fenêtrage. Une fenêtre de dialogue avec l'utilisateur enregistre ses commandes et affiche uniquement des comptes rendus. La restitution des résultats à l'écran a lieu dans une autre fenêtre créée dynamiquement. Les données sont présentées de façon tabulaire. Les objets complexes et les données atomiques multimédia sont transformés en "boutons". Pour consulter leur valeur, il suffit de pointer le bouton à l'aide d'une souris et ensuite d'appuyer sur la touche de la souris. L'interface interactive crée alors une fenêtre contenant la valeur de l'objet pointé.

La conception du langage SQLOMEGA a été influencée par le désir de faciliter son couplage avec une interface interactive de haut niveau. SQLOMEGA ne constitue pas une interface ergonomique pour les usagers "naïfs" (généralement non informaticiens). Il s'agit plutôt d'une couche au dessus de laquelle on pourra construire des interfaces conversationnelles de type "graphique".

Dans ce type d'interface, les utilisateurs disposent d'une visualisation graphique du schéma conceptuel (au moins en terme de classes et de fonctions disponibles) dans une fenêtre. L'affichage du schéma peut être

affiné de façon variable :

- affichage du treillis de classes,
- liste des attributs des classes,
- affichage du graphe d'objets. . .

A partir de ce schéma, les utilisateurs construisent leur requête. Les notions de variables et trajets disparaissent. Il suffit de pointer sur les classes intéressantes en utilisant le treillis ou le graphe d'objets. Par contre, le mécanisme d'affichage des résultats reste inchangé.

L'interface programmable est destinée aux développeurs d'applications manipulant des données OMEGA. Les classes définies dans le schéma conceptuel sont manipulées en tant que types dans le programme C. L'interface est de type précompilateur. Les opérations OMEGA incluses dans le code C subissent des vérifications de types à l'aide des catalogues OMEGA, puis sont transformées en une liste d'appels de fonctions. Ces fonctions font partie d'une bibliothèque qui permet des opérations sur la base de données ainsi que sur les objets en mémoire d'exécution du programme.

Les sections 2,3 et 4 présentent le langage SQLOMEGA. Les exemples illustrant ces sections sont tirés du schéma conceptuel présenté en annexe 1 (introduit dans le chapître précédent). La section 5 décrit les différentes approches possibles pour l'interface programmable, elle propose une solution simple qui pourra être approfondie dans la suite de cette étude. La syntaxe exacte du langage SQLOMEGA est précisée dans l'annexe 2.

5.2 LE LANGAGE DE REQUETES

Nous avons défini un langage de requêtes pour le modèle de données ESTRELLA. C'est une extension de SQL qui reprend des idées de POSTGRES [Stonebraker 86], LAMBDA [Velez 84] et EXCESS [Carey 88].

Les principales caractéristiques de ce langage sont les suivantes :

- le résultat d'une requête est un ensemble de n-uplets dont les attributs sont des objets ou des ensembles d'objets,

- l'utilisation de variables de classe permet de simplifier les expressions. Les prédicats d'identité d'objets et d'égalité de valeurs sont autorisés. L'interprète assure aussi la vérification des types et la compatibilité en fonction du treillis de classes,

- des opérateurs sur les ensembles et les tableaux sont introduits pour les clauses SELECT et WHERE,

- trois sortes de jointures sont possibles entre les classes : deux jointures implicites exprimées à l'aide de notation pointée (aussi appelée "trajet") représentant la composition d'objets ou l'héritage. Une jointure explicite entre deux classes utilisant des prédicats de comparaison,

- l'appel de fonctions définies dans le schéma conceptuel (classe FUNCTION) est autorisé dans les clauses SELECT et WHERE. OMEGA assure la validité des appels , la liaison dynamique et la vérification des types.
  Les fonctions appliquées sur les descripteurs permettent l'appel d'opérateurs géométriques.

Ces extensions sont nécessaires pour supporter les possibilités du modèle de données. Elles seront illustrées dans les sections suivantes : la section 1 effectue une présentation informelle d'une requête. La section 2 détaille la syntaxe et la sémantique des jointures et des trajets, elle montre aussi l'emploi des ensembles et des tableaux dans les requêtes. La section 3 présente l'utilisation des fonctions.

5.2.1 Expression générale des requêtes

L'expression générale des requêtes reprend la syntaxe et la sémantique du langage SQL :

exemple 1 : requêtes recherchant les noms des gares de Paris

```
R1:  SELECT g.nom
     FROM   gare g, commune c
     WHERE  g.ville = c
     AND    c.nom = "Paris"

R2:  SELECT g.nom
     FROM   gare g
     WHERE  g.ville.nom = "Paris"
```

Ces deux requêtes sont valides. Elles expriment exactement la même sémantique mais ont une syntaxe différente. La première requête effectue une jointure entre les classes "gare" et "commune" suivant le prédicat "g.ville = c". Cette jointure associe une commune à chaque gare. Ensuite R1 sélectionne sur les couples (gare, commune), ceux dont le nom de la commune est Paris.

· R2 ne concerne que la classe "gare". La syntaxe "g.ville.nom" accède directement au nom de la ville de chaque gare. R2 sélectionne les gares dont le nom de la ville est Paris. Dans les deux cas, R1 et R2 ont pour résultat un ensemble de chaînes de caractères : les noms des gares sélectionnées.

On remarque évidemment que la syntaxe de R2 est plus concise. Elle fait apparaître une nouvelle construction "g.ville.nom" appelée un trajet qui permet l'accès aux objets contenus dans un autre objet.

exemple 2 : recherche des objets contenus dans la Picardie

```
SELECT o.numéro, o.type
FROM   objet_planimétrique o, région r
WHERE  r.nom = "Picardie"
AND    est_dans(o,r)
```

L'exemple 2 montre l'utilisation d'une fonction sur des objets pour réaliser une opération géographique. Cette fonction est appelée dans la clause WHERE d'une requête. Elle a été définie dans le schéma conceptuel.

La syntaxe d'appel de fonction est parfaitement naturelle, elle s'inspire de l'utilisation des agrégats dans SQL. La sémantique de cet appel est différente : la fonction "est_dans" s'adresse à un seul couple (objet_planimétrique, région) et non pas à l'ensemble des couples comme dans le cas des agrégats.

exemple 3 : trouver les régions dont la capitale n'est pas référencée par une gare

```
SELECT   r.nom
FROM     région r
WHERE    r.capitale NOT IN ( SELECT g.ville
                             FROM   gare g )
```

L'exemple 3 montre une imbrication de requêtes SQLOMEGA. Trois opérateurs d'imbrication sont possibles : égalité (=), appartenance (IN) et non appartenance (NOT IN).

D'autre part, l'exemple 3 met en évidence une des utilisations de l'héritage dans le langage de requête : l'attribut "numéro" est un attribut hérité dans la classe "objet_dans_cliché" à partir de sa superclasse "cliché".

A partir de ces exemples on peut déduire la forme générale d'une requête SQLOMEGA (la syntaxe complète des requêtes est donnée en annexe 2) :

```
SELECT V1, V2, ... , Vn
FROM   C1, C2, ... , Cp
[ WHERE condition ]
```

La clause SELECT précise l'information que l'on désire obtenir. Les Vi sont soit des trajets, soit des appels de fonctions. La clause FROM donne la liste des classes où se trouve cette information. Chaque Ci désigne une classe du schéma conceptuel, on peut lui associer une variable représentant ses objets. La clause WHERE spécifie les prédicats que doit vérifier l'information sélectionnée. Elle contient une suite de conditions élémentaires liées par les opérateurs logiques traditionnels de négation (NOT), conjonction (AND) et disjonction (OR).

Une condition élémentaire peut être une fonction booléenne, un prédicat de comparaison entre deux valeurs (par exemple g.ville.nom = "Paris"), ou un prédicat ensembliste pouvant contenir une autre requête.

5.2.2 Opérations sur les classes

On définit certains opérateurs génériques sur les classes. Ces opérateurs sont des extensions canoniques des opérateurs de l'algèbre relationnelle. Ils sont définis à partir de travaux déjà réalisés sur les objets complexes comme [Abiteboul 86], [Abiteboul 87b], [Lecluse 87], [Schek 87b] et vont permettre de donner un modèle d'exécution des requêtes SQLOMEGA.

En effet, l'exécution d'une requête peut être interprétée comme l'application d'une suite d'opérateurs sur les classes, même si elle est exécutée réellement à l'aide de mécanismes différents pour être plus efficace. Par la suite, on utilisera les opérateurs suivants :

- sélection sur une classe suivant un prédicat portant sur ses attributs (attributs directs plus les attributs hérités) : C est une classe définie sur l'ensemble d'attributs X et P[X] est un prédicat.

  $C : P[X] = \{ c / c \text{ in } C \ \& \ P[c] \}$

- projection d'une classe sur certains de ses attributs : la classe C est définie sur les ensembles d'attributs X et Y.

  $C(X,Y)\ [Y] = \{ y / (x,y) \text{ in } C \}$

- produit cartésien de deux classes C et R :

  $C \times R = \{ (c,r) / c \text{ in } C \text{ et } r \text{ in } R \}$

- extraction d'un ensemble d'objets homogènes (référencés ou imbriqués) qui sont les valeurs de l' attribut x de type K dans une classe C :

C -> x = { (c,k) / k in K & c in C & c[x] = k }

Ces opérateurs seront étendus à tout ensemble d'objets homogènes (de même type).

On utilisera en plus certaines fonctions définies sur les classes :
- clé(C) retourne l'ensemble des attributs clés de C,
- super(C) retourne la liste des super-classes de C,
- sousclasse(C) retourne la liste des sous-classes de C,
- prédicat(C1,C2,X) construit le prédicat comparant C1 et C2 sur les attributs de l'ensemble X.

5.2.3 Expressions des jointures

Comme on l'a vu dans l'exemple 1, SQLOMEGA permet les jointures explicites entre les classes. Cependant cette jointure va plus loin que la jointure relationnelle, en effet on distingue deux types de jointures :
- la jointure à l'aide de prédicats sur les valeurs,
- la jointure à l'aide de prédicats sur l'identité des objets.

L'exécution des jointures n'est valide que si les classes exprimées dans les prédicats sont compatibles. Cette vérification de type est immédiate pour les classes de base et les classes descripteurs. Pour les autres classes, la vérification de type fait appel à la notion de comparabilité entre classes sur le treillis : A et B sont deux classes.

A et B sont comparables <=> A et B ont une super-classe commune
<=> super(A) n super(B) ≠ {}

exemple 4 : trouver les gares dont le nom est identique au nom d'une ville

```
R3 : SELECT   g.nom
     FROM     gare g , commune c
     WHERE    g.nom = c.nom
```

Dans l'exemple 4, le prédicat P1 : "g.nom = c.nom" exprime une jointure suivant la valeur des attributs "nom" dans les classe "gare" et "commune". La sémantique SQL de cette opération est encore valide : cette jointure correspond à une sélection suivant P1 sur la classe produit de "gare" et

"commune" :

R3 : (gare x commune : P1) [nom]

exemple 5 : trouver les noms des gares des capitales des régions

```
R4 :  SELECT    r.nom, g.nom
      FROM      région r, gare g
      WHERE     r.capitale = g.ville
```

L'exemple 5 montre une jointure sur l'identité d'objets. "r.capitale" et "g.ville" sont des variables représentant les instances de la classe "commune". Le prédicat P2 : " r.capitale = g.ville" exprime une comparaison entre l'identité des deux objets. Cette jointure correspond encore à une sélection sur le produit des classes de chaque objet, par contre le prédicat de sélection n'est pas immédiat. Il doit être calculé comme la conjonction de la comparaison des clés communes :

```
R4 :  C1 et C2 représentent les ensembles de commune extrait de
      "r.capitale" et "g.ville"
      X = clé(commune)    clé(commune)
      P = prédicat( commune , commune , X)
      ( C1 x C2 : P ) [nom,numéro]
```

On a montré dans le chapitre précédent que l'identité de valeurs était équivalente à l'identité d'objets dans le modèle ESTRELLA. En effet, les clés étant inclues dans la valeur, la différence sémantique entre ces jointures est minime. On adopte donc la même syntaxe pour les jointures par identité et pour les jointures par valeur. On ne définit donc pas de nouveau symbole "==" pour l'identité d'objet.

5.2.4 Expression des trajets

SQLOMEGA offre une construction syntaxique unique (appelée un trajet) pour les requêtes portant sur les objets identifiés, imbriqués ou historiques. Il intégre dans cette syntaxe l'utilisation de l'héritage et de la composition d'objets.

Les clauses SELECT et WHERE d'une requête peuvent contenir les expressions de trajets. Un trajet est constitué de pas, il a une origine et une extrémité. Son origine désigne les objets d'une des classes citées dans la clause FROM de la requête.

trajet = origine . pas1 . pas2 ... pasN . pas(N+1) ... extrémité

- On notera aussi :
  trajet = sous-trajet-origine . pasN . sous-trajet-extrémité

En supposant qu'au pas N, le trajet désigne un objet O d'une classe C, le pas N+1 désigne soit le même objet dans une des super-classes de C, soit un objet valeur d'un attribut de O dans C. On distingue donc plusieurs significations possibles pour un pas :

- un pas dans le treillis permet de désigner une superclasse d'un objet. L'utilisation du pas dans le treillis est indispensable pour l'identification d'attributs de même nom hérités de super-classes différentes (voir chapître 4).

  exemple 6 : utilisation de pas dans le treillis

  ```
  SELECT oc.numéro , oc.objet_planimétrique.type, oc.cliché.type
  FROM   objet_dans_cliché oc
  WHERE  oc.date > 01/04/89
  ```

  La requête de l'exemple 6 retrouve le numero, le type et le type de cliché pour chaque objet repéré dans un cliché récent. Les trajets "oc.objet_planimétrique.type" et "oc.cliché.type" contiennent un pas dans le treillis. L'attribut "type" est hérité de deux classes : objet_planimétrique et cliché. Les pas "objet_planimétrique" et "cliché" sont des pas dans le treillis, ils permettent d'identifier la provenance de l'attribut "type" dans la classe "objet_dans_cliché".

- un pas dans le graphe d'objets permet l'accès à un objet référencé ou imbriqué. Il simplifie l'écriture des requêtes en supprimant les jointures sur l'identité des objets à l'aide de la notation pointée.
  Soit E l'ensemble des objets de même type désignés par le sous-trajet-origine jusqu'au pasN-1. PasN est alors un attribut de E de type C. Le franchissement du pasN peut être transformé en une suite d'opérations sur les ensembles E et C :

  ```
  [sous-trajet-origine . pasN] : E -> pasN
                                 ou bien
                                 F = E [pasN]
                                 P = prédicate(F,C,clé(K))
                                 ( F x C : P )
  ```

  Dans le cas où pasN est l'extrémité du trajet, le franchissement du pas devient une simple projection :

[sous-trajet-origine . extrémité] : E [extrémité]

exemple 7 : utilisation de pas dans le graphe d'objets

```
SELECT g.nom , g.ville.nom
FROM   région r , gare g
WHERE  r.capitale.nom = "Lyon"
AND    est_dans(g,r)
```

L'exemple 7 recherche les gares et leurs villes dans la région dont Lyon est la capitale. "z.capitale.nom" et "g.ville.nom" sont deux trajets constitués de pas dans le graphe d'objets. Ces trajets suppriment les jointures avec la classe "ville".

exemple 8 : requête équivalente sans trajet

```
SELECT g.nom , v.nom
FROM région r, commune c, gare g , commune v
WHERE  r.capitale = c
AND    c.nom = "Lyon"
AND    est_dans(g,r)
AND    g.ville = v
```

Cette nouvelle requête est plus complexe que la précédente. La notation pointée est plus naturelle, elle est identique à la syntaxe des langages de programmation comme C ou Pascal.

remarque :

un trajet ne peut accéder aux attributs des classes descripteurs. Ce sont des classes opaques, les utilisateurs ne peuvent pas les manipuler directement.

5.2.5 Manipulations des ensembles et des tableaux

Le modèle de données ESTRELLA autorise la définition d'attributs multivalués dans les classes. Il est possible de définir des ensembles et des tableaux. Lorsque ces attributs sont manipulés globalement, les tableaux seront considérés comme des ensembles dont les éléments sont des couples (position, élément) où position est la position de l'élément dans le tableau.

Dans les requêtes SQLOMEGA, certaines règles sont imposées pour la

manipulation des ensembles et des tableaux :

- un trajet ne contient pas de pas multivalués sauf en son extrémité. Un pas intermédiaire ne peut donc pas être un ensemble,

  exemple 9 : utilisation des attributs multivalués

  ```
  SELECT r.nom , c.nom
  FROM région r , commune c
  WHERE c IN r.grandes_villes
  ```

  L'attribut "grandes_villes" de la classe "région" est un ensemble de cartes. On ne peut pas invoquer directement les éléments de cet attribut dans des expressions quantifiées comme par exemple :
  FOR ALL c IN r.grandes_villes : c.nb_habitants = 50000
  De la même façon, on ne peut pas invoquer l'ensemble des nombres d'habitants des grandes villes d'une région : l'expression z.grandes_villes.nb_habitants est interdite.

- les requêtes imbriquées sont considérées comme des opérations retournant un ensemble d'objets homogènes dont le type est calculé à l'aide du contenu de la clause SELECT,

- SQLOMEGA offre des opérateurs binaires et unaires pour manipuler les ensembles.

Parmi les opérateurs binaires, on distingue les prédicats qui retournent une valeur booléenne :

- appartenance d'un élément à l'ensemble (symbole IN ou NOT IN),
- égalité de deux ensembles (symbole = ou !=),
- inclusion (symbole IN ou NOT IN).

exemple 10 : donner le nom des gares des grandes villes de Picardie

```
R5 :    SELECT    g.nom
        FROM      gare g, région r
        WHERE     g.ville IN r.grandes_villes
        AND       r.nom = "Picardie"
```

```
R5 :    C = (GARE x REGION : [nom = "picardie"] )
        ( C : [ville IN grandes_villes] ) [nom]
```

On définit aussi des opérateurs binaires traditionnels retournant un ensemble :

- l'intersection (|),
- l'union (v),

exemple 11 : trouver les régions ayant des villes communes avec le Lyonnais

```
R7 : SELECT   r.nom
     FROM     région r, région L
     WHERE    L.nom = "Lyonnnais"
     AND      NOT NULL ( r.grandes_villes  |  L.grandes_villes)
```

```
R7 : L = REGION : [nom = "Lyonnais"]
     (REGION x L : [NOT NULL(grandes_villes  |  grandes_villes)] ) [nom]
```

Les opérateurs unaires retournent une valeur simple : comparaison avec l'ensemble vide : NULL ou NOT NULL (ensemble), et agrégat simple sur l'ensemble :

```
COUNT    : compte le nombre d'éléments d'un ensemble,
MAX      : calcule le maximum d'un ensemble d'entiers ou de réels,
MIN      : calcule le minimum d'un ensemble d'entiers ou de réels,
SUM      : calcule la somme d'un ensemble d'entiers ou de réels,
AVG      : calcule la moyenne d'un ensemble d'entiers ou de réels,
COLLAPSE: concatène les éléments d'un ensemble soit de chaînes de
          caractères, soit d'un ensemble d'ensembles (voir exemple 13).
```

exemple 12 : retrouver les régions qui ont plus de 10 villes importantes

```
R7 :   SELECT   r.nom
       FROM     région r
       WHERE    COUNT ( r.grandes_villes ) > 10
```

```
R7 :   K = { (z,r) / z in REGION & r = COUNT(z[grandes_villes]) }
       ( K : [r>10] ) [nom]
```

exemple 13 : retrouver les gares dont les villes sont parmi les grandes villes de certaines régions

```
R8 :    SELECT    g.nom
        FROM      gare g
        WHERE     g.ville IN COLLAPSE ( SELECT r.grandes_villes
                                        FROM   région r          )
```

```
R8 :    E = COLLAPSE ( REGION[grandes_villes] )
        ( GARE : [ville IN E]) [nom]
```

Les exemples 12 et 13 illustrent les opérateurs unaires sur les ensembles. Dans l'exemple 13, le résultat de la requête imbriquée est de type ensemble d'ensembles. La fonction COLLAPSE transforme le résultat en réalisant l'union entre ses éléments.

SQLOMEGA permet de manipuler précisément un élément d'un tableau. Dans une requête on utilise la notation classique entre crochets. L'interprète de la requête fait appel à l'opérateur de sélection sur le tableau en le considérant comme un ensemble dont un attribut est la position des éléments.

exemple 14 : recherche de la troisième ville de Normandie

```
R9 :    SELECT    r.grandes_villles[3].nom
        FROM      région r
        WHERE     r.nom = "Normandie"
```

```
R9 :    (( REGION : [nom = "Normandie"]) -> grandes_villes : [ 3 ]) [nom]
```

5.2.6 Utilisation des fonctions

La notion de fonction est importante dans le modèle ESTRELLA. On distingue plusieurs types de fonctions :

- les fonctions agrégats sur les ensembles définis dans la section précédente,
- les fonctions de manipulation du temps et des historiques,
- les fonctions élémentaires de manipulation des classes descripteurs,
- les fonctions définies par les utilisateurs.

Toutes ces fonctions font partie du schéma conceptuel des applications. Elles sont des objets de la classe FUNCTION.

Les exemples 12 et 13 montrent des appels de fonctions agrégats sur des ensembles. Les exemples 2, 7 et 8 font apparaître des appels de fonctions utilisateurs dans la clause WHERE d'une requête. La clause SELECT peut aussi admettre de tels appels :

exemple 15 : calcul de la surface des gares de Paris

```
SELECT g.nom , SURFACE(g.contour)
FROM gare g
WHERE g.ville.nom = "Paris"
```

L'appel d'une fonction est possible dans une requête sous certaines conditions :

- la fonction est un objet de la classe FUNCTION,
- la fonction n'effectue pas de manipulations élémentaires sur les descripteurs,
- les types des paramètres d'appel sont compatibles avec la signature de la fonction.

En vertu de la possibilité de surcharge dynamique, la liaison entre le code de la fonction et l'interprète de la requête n'est effectuée qu'au moment de son exécution : pour chaque paramètre effectif de la fonction, il faut déterminer le code adéquat. Pour cela, on introduit une nouvelle construction : **for each** dont la syntaxe est la suivante :

for each élément **in** ensemble { liste L de commandes }

A l'aide de cette construction, on peut décrire l'exécution de fonctions sur des ensembles d'objets :

- **for each** applique les commandes L à chaque élément de E,
- on appelle "eval_function" la procédure qui met en oeuvre l'algorithme de calcul de la fonction à exécuter tel que nous l'avons défini dans la section 4.5.3.

Tout calcul de fonction F sur un ensemble E peut être interprété par la commande suivante :

for each e **in** E { eval_function(F,e) }

Dans le cas de l'exemple 15, la requête est exécutée comme suit :

```
R :     E = ( GARE -> ville : [ville.nom = "paris"] ) [gare.nom, contour]
        F = {}
        for each e in E
          {
           f = eval_function(SURFACE, e)
           F = F u [ e[nom] , f(e[contour]) ]
          }
        return(F)
```

La mise en oeuvre des fonctions est donc complexe. L'accès au code exécutable de fonctions est fourni lors de sa définition dans la classe "FUNCTION". Ce code est écrit dans un langage de programmation traditionnel interfaçable avec le modèle ESTRELLA (voir la section 5.5).

Dans la clause WHERE d'une requête, l'évaluation d'une fonction oblige à introduire des étapes intermédiaires pour interpréter son résultat :

exemple 16 : interprétation de la requête de l'exemple 2

```
SELECT o.numéro, o.type
FROM   objet_planimétrique o, région r
WHERE  r.nom = "Picardie"
AND    est_dans(o,r)
```

R :

```
E = ( OBJET_PLANIMETRIQUE  x REGION : [nom = "Picardie"])
F = {}
for each e = (o, r) in E
   {
    f = eval_function( est_dans , o , r )
    F = F u [e, f(o,r)]
   }
( F : [f = true] ) [numéro,type]
```

Dans cet exemple, il faut effectuer une sélection sur le résultat du calcul de la fonction. L'évaluation des conditions de la requête est séparée en deux étapes par le calcul de la fonction. La transformation de la requête vers l'algorithme R n'est pas triviale. Ce type de transformation sera décrit dans le chapitre suivant lors de la description de l'évaluation des requêtes dans le prototype OMEGA.

Les fonctions de manipulation des historiques sont aussi autorisées dans les clauses SELECT et WHERE. Elles sont décrites dans le chapitre sur le modèle de données ESTRELLA et elles subissent les mêmes règles d'appel que les fonctions utilisateurs.

5.2.7 Objets en résultat

Dans tous les exemples précédents, nous avons montré des requêtes SQLOMEGA retournant des valeurs simples : un ensemble de n-uplets dont les champs étaient des objets de base du modèle ESTRELLA. Cependant le résultat d'une requête peut être un ensemble de n-uplets dont les attributs

sont des objets ou des ensembles d'objets.

exemple 17 : requête retournant des objets complexes

```
SELECT g.nom , g.ville
FROM   région r , gare g
WHERE  r.capitale.nom = "Lyon"
AND    est_dans(g,r)
```

Cette requête retrouve les villes de la région Lyonnaise ayant une gare. Le type du résultat est [nom : char(20), ville : commune]. Ce type n'est pas un type existant d'ESTRELLA. Le résultat ne peut donc être exploité dans une autre requête, par contre il peut soit être affiché soit être transmis à un programme d'application.

L'affichage utilise les techniques proposées au paragraphe 5.1. Une fenêtre est ouverte, elle contient un tableau de deux colonnes : la première affiche le nom des gares sélectionnées, la seconde contient un bouton symbolisant l'objet "commune". On peut visualiser cet objet en activant le bouton à l'aide de la souris.

L'utilisation du résultat dans un programme d'application passe par la définition de variables de liaison avec OMEGA. Ces variables sont définies de type "char(20)" et "commune". Elles reçoivent les valeurs du résultat de la requête. Ce type d'interface sera décrit plus précisément dans la section portant sur l'interface avec un langage de programmation.

Le même mode d'utilisation est défini pour les requêtes retournant des descripteurs (objets de gros volume ou implantation particulière).

Il faut remarquer que SQLOMEGA ne permet pas de construire des objets complexes de type quelconque en résultat de requête. Seuls sont autorisés les n-uplets contenant des objets de type déjà existant.

Des systèmes comme AIM [Pistor 87], VERSO [Verso 86] et O2 [Delobel 88] autorisent des résultats de type quelconque (construit à partir de n-uplets et d'ensembles). La syntaxe de leurs requêtes devient rapidement très complexe. Pour la simplifier, certains proposent l'utilisation de fonctions dans la clause SELECT (masque pour la syntaxe).

Les fonctions ESTRELLA ne correspondent pas à cette approche puisque le type de résultat d'une fonction est obligatoirement une classe du schéma conceptuel. Une requête avec fonction se comporte comme une requête simple : son résultat est de type n-uplet contenant des objets des classes

déjà définies. Il s'agit bien sûr d'une restriction par rapport aux systèmes cités précédemment, mais nous pensons qu'elle n'est pas très importante :

- les objets de la base peuvent être retournés par des requêtes SQLOMEGA (nous pensons qu'il s'agit là de la plupart des requêtes),
- les résultats de type quelconque ne sont pas exploitables dans l'interface interactive. Ils devront être reconstruits dans les programmes d'applications.

Pour simplifier l'accès aux objets de la base de données, la liste des attributs d'une classe peut être substituée par "*". L'affichage du résultat reprend alors les attributs propres et proposent des boutons pour les attributs hérités (un par super-classe). On peut cependant demander l'affichage direct des attributs hérités en précisant le niveau de substitution devant le symbole "*".

Un autre symbole "all" permet l'affichage de la totalité des attributs directs ou hérités. Il est équivalent au symbole "*" précédé de la profondeur maximum de l'arbre des superclasses

exemple 18 : affichage des gares de Lyon avec les caractéristiques héritées de la classe "objet_planimétrique".

```
SELECT 1 *
FROM gare
WHERE ville.nom = "Lyon"
```

5.2.8 Conclusion

Nous avons présenté le langage de requête de SQLOMEGA. Il repose sur le bloc de qualification SQL : la clause SELECT définit les résultats, la clause FROM introduit les classes et leurs variables, la clause WHERE contient la condition que doit vérifier la sélection.

Ce langage permet l'utilisation des fonctions du schéma conceptuel. Tous les objets de la base de données peuvent être obtenus. La notion de trajet permet de simplifier la syntaxe.

Ces requêtes sont souvent réutilisées dans d'autres commandes de SQLOMEGA, notamment pour obtenir des références sur des objets identifiés.

5.3 LE LANGAGE DE DEFINITION DE DONNEES

Nous donnons dans ce chapitre une description générale du LDD (Langage de Définition de Données). La syntaxe en BNF est fournie en annexe 2.

Afin de mieux cerner les objectifs du LDD, nous rappelons la nature du

modèle ESTRELLA : c'est un modèle orienté objet. Les objets appartiennent à des classes. Les classes sont organisées en treillis. Parmi les classes, on distingue :

- la classe UNIVERSAL : origine du treillis,
- la classe NIL : extrémité du treillis,
- la classe DESCRIPTOR et ses sous-classes qui permettent de gérer des types abstraits,
- la classe FUNCTION : catalogue des fonctions,
- la classe PREDICATE : catalogue des prédicats,
- les classes utilisateurs : noeuds du treillis.

Un schéma conceptuel ESTRELLA est entièrement décrit par les classes et le treillis. Le LDD reprend donc les opérations sur le treillis décrites dans le chapître 4.

5.3.1 Création sur le schéma conceptuel

Les commandes de création sur le schéma conceptuel permettent la définition de nouvelles classes, de nouvelles fonctions et de nouveaux prédicats.

:5.3.1.1 Création d'une classe

Les classes sont identifiées par leur nom. Elles sont définies en décrivant explicitement leurs super-classes et leurs attributs propres. Les propriétés héritées sont omises puisqu'elles sont déductibles par le SGBD.
Par contre, on peut reprendre un attribut hérité pour lui adjoindre une valeur par défaut (si elle n'existe pas déjà). Cette valeur par défaut peut être simple ou calculée à l'aide d'une requête.

exemple 19 : création de la classe NATION

```
CREATE CLASS HISTORIC nation ISA UNIVERSAL
BEGIN
KEY  nom        : char(20) = "France",
     régions    : ARRAY [1..20] OF région,
     cartes     : SET OF carte,
     population : int = 56 000 000,
     capitale   : commune = SELECT *
                            FROM   commune
                            WHERE  nom = "Paris",
     DATE_INSTANCE = date
END
```

Un certain nombre de constructions permettent de définir des clés (KEY), des attributs multivalués (ARRAY [] OF, SET OF) ou l'historique (HISTORIC, DATE_INSTANCE). Pour plus de précisions, on pourra consulter l'annexe 2.

5.3.1.2 Création d'une vue

Nous introduisons ici la notion de vue. Elle ne remet pas en cause le modèle de données ESTRELLA et ne doit être perçue que comme un outil externe de manipulation des données. Une vue est définie à l'aide d'une requête sur une ou plusieurs classes.

Les opérations possibles sur les vues sont :
- définition d'une vue,
- destruction d'une vue,
- interrogations de son contenu,
- mise à jour sur les objets accessibles à travers une vue.

Nous mettons deux contraintes de mise à jour à travers une vue : elle doit être une restriction projection d'une seule classe [Delobel 83], et elle doit contenir toutes ses clefs.
Une vue n'est pas une classe. On ne peut donc pas hériter d'une vue ou la référencer : elle n'est qu'une requête pré-établie. On peut cependant utiliser une vue pour en définir une autre.

exemple 20 : création d'un vue sur la classe NATION

```
CREATE VIEW capitale (pays, région, ville, cartes)
AS
(     SELECT    N.nom, R.nom, R.capitale, N.cartes n R.cartographie
      FROM      nation N, région R
      WHERE     R in N.régions
)
```

La vue "capitale" retourne des valeurs de type t :

```
t = [pays : char(20), région : char(20),
     ville : commune, cartes : SET OF carte ]
```

Elle est obtenue par jointure entre les classes "nation" et "région", elle fait appel à des opérateurs ensemblistes : l'appartenance et l'intersection. Aucune mise à jour n'est autorisée dans cette vue, elle peut par contre être réutilisée :

exemple 21 : réutilisation d'une vue

```
CREATE VIEW information_capitale
AS
(
        SELECT      c.pays, c.région, c.ville, g.nom, g.dessin
        FROM        capitale c, gare g
        WHERE       g.ville = c.capitale
)
```

Dans l'exemple 19, on ne redéfinit pas les noms des champs du type du résultat. La vue "information_capitale" retrouve les gares des capitales des régions. Le type du résultat est :

```
t = [pays : char(20), région : char(20), ville : commune,
     nom : char(20), dessin : graphique ]
```

5.3.1.3 Création d'une fonction ou d'un prédicat

Les fonctions comme les prédicats sont des objets de classes particulières (FUNCTION et PREDICATE). La création d'une fonction ou d'un prédicat utilise donc la commande du LMD effectuant la création d'objets.

Des exemples de création de fonctions et de prédicats sont fournis en annexe 1. La syntaxe donnant les informations nécessaires à leur création est détaillée dans l'annexe 2. On peut cependant décrire les attributs et les clés de ces deux classes.

La classe FUNCTION a quatre attributs :

- nom : nom de la fonction,
- nb_paramètre : nombre de paramètres,
- paramètre : tableau de chaînes de caractères contenant le nom des paramètres de la fonction,
- résultat : nom de la classe résultat de la fonction.

La création d'une fonction est soumise à une contrainte : l'ensemble des signatures des fonctions de même nom et de même nb_paramètre doit être cohérent et permettre la liaison dynamique avec toutes les signatures applicables (voir chapître 4). Cette contrainte se traduit par l'exécution de vérifications présentées dans la section 4.5.4 à chaque mise à jour dans la classe FUNCTION.

- La classe PREDICATE a aussi quatre attributs :
  - nom : identificateur du prédicat,
  - creator : utilisateur ayant le droit de suppression sur le prédicat,
  - type : catégorie de prédicats parmi "domaine", "interclasse", "intraclasse",
  - corps : requête permettant d'interpréter le prédicat.

L'attribut "corps" des prédicats prend comme valeur une condition d'existence ou de non existence sur le résultat d'une requête. Nous avons choisi de définir de cette façon nos prédicats pour simplifier le langage : il n'y a pas de commande spécifique aux prédicats.

Le pouvoir d'expression à l'aide des requêtes est suffisant pour exprimer un grand nombre de contraintes. Dans l'annexe 1, on montre que des contraintes référentielles, d'inclusion, de partition peuvent être exprimées facilement.

5.3.2 Suppression dans le schéma conceptuel

A l'opposé des commandes de création, les commandes de suppression du LDD permettent de détruire les objets d'un schéma conceptuel ESTRELLA. Elles peuvent donc intervenir sur les classes, les fonctions ou les prédicats.

5.3.2.1 Suppression d'une classe

La suppression d'une classe est une commande syntaxiquement simple. Par contre, elle pose de nombreux problèmes de répercussions dans la base de données :

- une suppression d'une classe C entraine la destruction de toutes ses sous-classes : supprimer C revient à détruire tous ses objets, puisque les objets des sous-classes de C appartiennent aussi à C, il faut les détruire.

  exemple 20 : destruction de la classe OBJET_PLANIMETRIQUE

  DROP CLASS objet_planimétrique

  les trois sous-classes "gare", "commune" et "objet_dans_cliché" d'objet_planimétrique doivent être détruites

- la suppression d'une classe n'est valide que si cette classe n'est pas utilisée par d'autres classes dans le graphe d'objets. Une exception est

toute fois tolérée : lorsque la classe est référencée par ses sous-classes ou par des classes qui seront détruites par la même commande.
La commande de l'exemple 22 ne peut donc être exécutée puisque par exemple, la classe "région" utilise la classe "commune" qui devrait être détruite pendant la destruction de la classe "objet_planimétrique".

- les vues définies à l'aide de classes supprimées sont également détruites.

- les fonctions et les prédicats définis sur des classes supprimées doivent être détruits.

- les valeurs par défaut calculées invoquant des classes détruites sont invalidées.

- les objets imbriqués dans les objets des classes à supprimer doivent être détruits. Les classes imbriquées doivent également être supprimées si elles ne sont plus utilisées par aucune autre classe. Par contre, les classes descripteurs ne peuvent être détruites que par une commande explicite de l'administrateur de la base de données.

5.3.2.2 Suppression d'une vue

La commande de suppression de vue est strictement identique à celle de SQL. Elle entraîne la suppression des vues définies à partir de la vue détruite.

exemple 23 : suppression de la vue CAPITALE

```
DROP VIEW capitale
```

Cette commande détruit la vue "capitale" et entraîne la destruction de la vue "information_capitale".

5.3.2.3 Suppression d'une fonction ou d'un prédicat

De la même façon que pour leur création, la destruction d'une fonction ou d'un prédicat utilise la commande de destruction d'objets du langage SQLOMEGA. Pour que cette commande soit valide, elle doit respecter les contraintes du modèle de données.

Dans le cas des fonctions, la destruction doit conserver la cohérence par rapport aux signatures applicables. Pour cela, on permet de supprimer en une seule commande plusieurs fonctions : suppression de toutes les fonctions appelées EST_DANS ayant deux paramètres ...

D'autre part, les fonctions sont susceptibles d'être utilisées dans les vues, les valeurs par défaut et les programmes d'applications. La suppression d'une fonction doit donc générer une mise à jour de ces objets. SQLOMEGA prévoit un mécanisme automatique d'invalidation de vue et de valeur par défaut mais ne gère pas la cohérence des programmes.

Dans le cas des prédicats, seul son créateur peut détruire un prédicat. Il est détruit en invoquant son nom ou une partie de son nom : par exemple on peut supprimer tous les prédicats commençant par la lettre P.

5.3.3 Autres opérations sur le schéma conceptuel

Outre les opérations de création et de destruction, six autres commandes sont prévues dans le LDD :

- l'ajout d'un attribut à une classe,
- la suppression d'un attribut dans une classe,
- la définition d'une valeur par défaut,
- la suppression d'une valeur par défaut,
- la transformation d'une classe historique en classe simple,
- la transformation d'une classe simple en classe historique.

Ces commandes ne seront pas réalisées dans l'immédiat. Elles sont prévues pour les développements futurs du prototype OMEGA.

5.3.3.1 Ajout d'un attribut à une classe

On peut toujours ajouter un attribut à une classe si le type de cet attribut est connu. Ce nouvel attribut ne peut pas être un attribut clé, par contre il peut avoir une valeur par défaut et une contrainte d'existence ou de généricité.

exemple 24 : définition d'un nouvel attribut pour les régions

```
ALTER CLASS    région
ADD  sous_régions : SET OF région
```

La syntaxe de cette commande reprend la syntaxe standard de SQL. Dans l'exemple 24, le nouvel attribut permet la définition de sous-régions. Pour

assurer la cohérence des régions, on pourra par exemple définir un prédicat empêchant une région d'être sous-région d'elle-même.

Il faut noter que tout nouvel attribut est immédiatement hérité dans le treillis : il est aussi défini sur les sous-classes et prend une valeur "non affecté" (ou bien la valeur par défaut). Il peut y avoir des problèmes d'identification de l'attribut dans les requêtes déjà formulées dans les vues, dans les valeurs par défaut, dans les prédicats ou dans les programmes d'application. Un mécanisme de validation automatique des vues, valeurs par défauts et prédicats doit donc être activé. Par contre, ce sont les utilisateurs qui doivent valider leur application.

5.3.3.2 Suppression d'un attribut dans une classe

La suppression d'un attribut dans une classe est possible. Elle est cependant restreinte aux attributs non clés et non hérités. Un attribut supprimé est aussi détruit dans toutes les sous-classes.

exemple 25 : suppression de l'attribut "qualité" de la classe CLICHE

```
ALTER CLASS cliché
DROP qualité
```

L'attribut "qualité" est supprimé dans la classe "cliché". Il est aussi détruit dans les classes "objet_dans_cliché" et "cliché_par_région".

Un attribut supprimé peut avoir été utilisé dans une vue, une valeur par défaut, un prédicat. Un mécanisme similaire à la validation d'un ajout d'attribut doit répercuter cette suppression sur les autres objets du schéma conceptuel.

5.3.3.3 Définition ou suppression d'une valeur par défaut

Deux commandes permettent de manipuler les valeurs par défaut indépendamment des attributs et des classes. La commande de définition d'une valeur par défaut n'est valide que dans le cas où une valeur par défaut n'est pas héritée. Par contre, cette commande écrase les valeurs déjà définies.

5.3.3.4 Transformation d'une classe simple en classe historique

Cette commande transforme une classe identifiée C en une classe historique H par la définition d'un attribut sur lequel porte la datation. Chaque objet de C devient un couple (version, objet) dans H. Si C était historique sur certains de ses attributs hérités, la nouvelle version dans H

est datée avec la version courante héritée dans C (on fait coïncider le nouvel historique avec l'historique hérité).

exemple 26 : transformation de la classe "objet_planimétrique"

```
ALTER CLASS objet_planimétrique
ADD HISTORY
WITH DATE_INSTANCE = date
FIRST VERSION IS 01/09/89
```

L'exemple 26 montre la transformation d'une classe n'héritant pas d'attribut historique : on définit la datation ainsi que la date de départ.

exemple 27 : transformation de la classe "commune"

```
ALTER CLASS commune
ADD HISTORY
WITH DATE_INSTANCE = objet_planimétrique.date
```

Dans l'exemple 27, l'historique sur les communes est défini pour coïncider avec celui sur les objets planimétriques.

Cette commande entraîne des répercussions dans le schéma conceptuel et parmi les objets :

- toutes les requêtes enregistrées (vues, valeur par défaut, prédicat) doivent être revues,
- dans le graphe d'objets, de nombreuses références sont à modifier : l'identité des objets est modifiée puisque l'on définit un nouvel attribut clé : la date dans le classe H. Il faut donc modifier toutes les références aux objets de C,
- les programmes d'applications ne correspondent plus à la réalité du schéma conceptuel.

Cette commande est coûteuse et donc doit être maniée avec précaution.

5.3.3.5 Transformation d'une classe historique en classe simple

Cette commande transforme une classe historique H en une classe simple C en créant un objet à partir des versions des objets de H.

exemple 28 : transformation de CLICHE

```
ALTER CLASS cliché
```

```
DROP HISTORY
WITH OBJECT AT 01/04/89
```

Dans l'exemple 28, la classe "cliché" devient une classe simple en utilisant les versions les plus proches du 01/04/89.

La suppression de l'historique entraîne la modification de l'identité des objets lorsqu'elle détruit un attribut de type "date". Cette suppression est d'ailleurs refusée lorsque cette date est réutilisée dans les sous-classes.

exemple 29 : commande de transformation invalide

```
ALTER CLASS objet_planimétrique
DROP HISTORY
WITH OBJECT AT 01/04/89
```

La commande de l'exemple 29 n'est pas valide car elle supprime l'attribut date qui sert à dater les versions de la classe "commune".

La modification de l'identité génère de nombreuses "références folles" dans le graphe d'objets. La commande DROP HISTORY permet de réparer ces incohérences en précisant les versions utilisées pour créer les nouveaux objets avec la clause WITH OBJECT AT <date> :

- les références aux versions antérieures à <date> deviennent des valeurs "non affectées",
- les références aux versions postérieures à <date> prennent la valeur du nouvel objet.

Si aucune date n'est précisée, toutes les références doivent être remplacées par les nouveaux objets.

L'algorithme qui met en oeuvre ces substitutions dans le graphe d'objets est très couteux : il oblige à un parcours presque complet de la base de données.

D'autres répercussions sont à craindre dans le schéma conceptuel, elles correspondent à l'union des problèmes connus pour la création de classes, la suppression de classes et la suppression d'attributs.

Les commandes de manipulation des définitions historiques sont donc très sensibles !

5.4 LE LANGAGE DE MANIPULATION DE DONNEES

SQLOMEGA offre trois commandes distinctes dans le LMD (Langage de Manipulation de Données). Ces commandes sont des extensions des commandes traditionnelles de SQL (INSERT, UPDATE et DELETE). Chacune de ces commandes a une syntaxe unique avec plusieurs significations possibles selon la nature des objets sur lesquels elle s'applique.

- INSERT : création ou spécialisation d'objets,
- UPDATE : mise à jour d'objets ou de versions ou bien création de versions,
- DELETE : suppression d'objets ou de versions.

Nous donnons dans ce chapitre une description sémantique du LMD, la syntaxe exacte en BNF étant fournie en annexe 2.

5.4.1 Création d'un objet

La commande de création d'objets est locale à une classe. Pour les classes historiques, on crée à la fois un objet et une version.

Cette commande est réalisée par une liste d'affectations d'attributs comprenant tous les attributs clés. Les autres attributs peuvent être omis. La liste d'affectations peut contenir des attributs hérités, l'identification des attributs ambigus est réalisée à l'aide de trajet. La valeur de chaque attribut dépend de la nature de son type :

- les types de bases sont directement affectés,
- les valeurs représentant des classes imbriquées sont affectées par une nouvelle liste d'affectations,
- les valeurs de classes identifiées sont affectées à l'aide d'une requête renvoyant un objet ou un ensemble d'objets de ces classes.

exemple 30 : création d'un objet dans la classe CLICHE

```
INSERT INTO région
  (
   nom = "Rhone Alpe",
   capitale = (     SELECT   *
                    FROM     commune
                    WHERE    nom = "Lyon"  ),
   contour = ("10 12 43 113"),
   cartographie = ((identificateur = "Rhône"), ( identificateur = "Isère"),
                   (identificateur = "Drome"), (identificateur = "Savoie"),
```

```
                    (identificateur = "Haute Savoie") )
  )
```

La syntaxe de cette commande tient compte des ensembles et des tableaux, leur affectation suit les mêmes règles que les attributs monovalués : liste de valeurs pour les types de base ou les classes imbriquées et requêtes pour les classes identifiées.

Le cas des classes descripteurs est traité de façon particulière : leurs objets sont créés par des fonctions. Il convient donc de donner comme valeur, une liste de paramètres utiles. Dans l'exemple 27, l'attribut "contour" est de type descripteur "rectangle", sa valeur est une liste de paramètres contenant uniquement une chaîne de caractères car une des fonctions de création des rectangles utilise une chaîne.

5.4.2 Spécialisation d'un objet

La spécialisation d'un objet consiste à prendre un objet d'une classe C et à l'insérer dans une de ses sous-classes. On utilise la même commande INSERT avec une liste d'affectations permettant d'affecter les nouveaux attributs de l'objet.

Pour spécifier la provenance de l'objet à spécialiser, on utilise un attribut du nom de la superclasse. L'objet à spécialiser est calculé par une requête sur sa classe d'origine.

exemple 31 : spécialisation d'un cliché

```
INSERT INTO objet_dans_cliché
  (
   cliché = ( SELECT * FROM cliché WHERE centre.x = 10 and centre.y = 20
                                        AND centre.z = 40 ),
   position = (x = 12, y = 11, z = 35),
   contour = (" 34 11 5 17"),
   numéro = 1001,
   objet_planimétrique.contour = (x = 12, y = 11, z = 35),
  )
```

Cette technique nous oblige à utiliser le nom des classes comme un attribut. On doit donc imposer une contrainte sur la définition des classes : les noms d'attributs doivent être différents des noms des super-classes.

Dans l'exemple 31, on spécialise le cliché identifié par le point (10,20,40) dans la classe "objet_dans_cliché". On crée ainsi un objet planimétrique de numéro 1001.

5.4.3 Mise à jour d'un objet

La commande de mise à jour d'objets correspond à la commande UPDATE en SQL. Elle intervient de la même façon sur les classes simples et sur les classes historiques, la mise à jour sur un objet historique signifiant la mise à jour de toutes ses versions.

Une commande de mise à jour peut modifier tous les attributs sauf les attributs clés de la classe. Cependant la date d'une version d'une classe historique peut être modifiée, on se trouve alors dans le cadre d'une commande UPDATE signifiant une création d'une nouvelle version.

exemple 32 : mise à jour d'une gare

```
UPDATE gare
SET  hauteur = 150,
     longueur = 200,
     contour = ( (x=12, y=23), (x=18,y=45) ),
     nb_voies = 25
WHERE nom = "gare de Lyon"
```

L'exemple 32 montre la mise à jour d'un objet de la classe "gare". Cette classe est une classe identifiée simple. Le système interprète donc facilement cette commande. Si la classe mise à jour est une classe historique, l'interprétation de la commande UPDATE peut être différente : la commande peut provoquer plusieurs opérations différentes sur les objets et les versions. Dans le paragraphe suivant, nous envisageons le cas d'une commande UPDATE provoquant uniquement la mise à jour d'une version.

5.4.4 Mise à jour d'une version

Il n'y a pas de différence syntaxique entre une mise à jour d'objets et une mise à jour de versions. Le système détecte la nature de la commande selon les attributs sur lesquels elle porte : une mise à jour de versions est activée lorsque la commande UPDATE porte sur des attributs non clés et historiques d'objets de classe historique.

exemple 33 : mise à jour d'une version d'un cliché

```
UPDATE cliché
SET  qualité = "moyen",
```

```
        type = "Landsat",
    WHERE numéro = 1098 AND date = 08/09/88
```

La commande de l'exemple 33 modifie la version du 08/09/88 du cliché numéro 1098. Les autres versions de ce cliché ne sont pas altérées. La même commande sans contrainte sur la date sera considérée comme une mise à jour d'objets et donc altérera toutes les versions du cliché.

Deux autres cas peuvent intervenir dans une commande UPDATE sur des classes historiques :

- la modification d'un attribut non historique : on se trouve alors face à une mise à jour d'objets. Ce cas de figure peut être couplé avec une mise à jour de versions :

  exemple 34 : commande double sur la classe "objet_dans_cliché"

```
UPDATE objet_dans_cliché
SET  position = (x=32, y=67, z=32),
     type = "Spot",
     qualité = "bon"
WHERE objet.numéro = 1100
AND   cliché.numéro = 3245
AND   date = 23/08/87
```

  La commande de l'exemple 34 modifie la version du cliché numéro 3245 et met à jour un objet dans la classe "objet_dans_cliché".

- la modification de la date : le système considère alors la commande comme la création d'une nouvelle version.

On constate donc qu'une seule commande UPDATE peut provoquer plusieurs opérations sur les objets et les versions.

5.4.5 Création d'une version

La commande UPDATE a été retenue pour exprimer la création d'une nouvelle version. La commande INSERT aurait aussi pu être choisie, cependant sa signification est trop liée à la définition d'un nouvel objet alors que la création d'une version est plus proche de la notion de trace des mises à jour d'un objet : la commande UPDATE correspond à cette approche.

Une création de version est donc détectée lorsqu'une commande UPDATE

altère la date d'une version dans une classe historique.

exemple 35 : création d'une nouvelle version d'un cliché

```
UPDATE cliché
SET     qualité = "bon",
        photo = (nom_fichier = "Landsat8009"),
        date = 01/06/89
WHERE numéro = 1001 AND  date = 08/04/89
```

L'exemple 35 montre la création d'une nouvelle version en utilisant une ancienne version d'un cliché : la nouvelle version reprend les valeurs de la version du 08/04/89 pour les attributs non affectés dans la commande.

5.4.6 Suppression d'un objet

La commande DELETE est utilisée pour effectuer des suppressions d'objets. Elle sert aussi à la suppression de versions. Pour le modèle ESTRELLA, supprimer un objet dans une classe historique correspond à en supprimer toutes les versions. Dans les classes simples, il n'y a pas d'ambiguité.

ESTRELLA interdit la suppression directe dans les classes imbriquées. Par contre, lors des suppressions d'objets identifiés, les objets imbriqués sont détruits.

Pour une classe historique, l'interprète SQLOMEGA distingue une suppression d'objets d'une suppression de versions selon la spécification de la date.

exemple 36 : suppression d'objets dans la classe CLICHE

```
DELETE FROM cliché
WHERE type = "Landsat"
```

Dans l'exemple 36, la commande DELETE supprime tous les clichés de type Landsat et donc détruit toutes les versions.

5.4.7 Suppression d'une version

Pour effectuer la suppression de versions, il faut donc contraindre la date des objets historiques dans la clause WHERE de la commande DELETE.

exemple 37 : suppression de certaines versions d'un objet

DELETE FROM cliché
WHERE type = "Spot" AND date < 01/08/87
Cette commande supprime toutes les versions de type Spot antérieures au 01/08/87 parmi les clichés.

5.4.8 Conclusion

SQLOMEGA permet d'effectuer toutes les opérations élémentaires sur les objets et les versions à partir des trois commandes SQL étendu (INSERT, UPDATE, DELETE).

La réduction d'un ensemble de sept opérations à trois commandes a été décidée d'une part pour rester dans le cadre de la norme SQL et d'autre par, parce que le choix de la bonne commande SQLOMEGA devant décrire une opération sur un objet, est naturel et évident.

De la même façon, l'interprète SQLOMEGA est en mesure de détecter facilement la nature des opérations demandées dans une commande : il se détermine suivant l'utilisation des attributs des classes, plus particulièrement suivant l'attribut de datation des versions.

5.5 L'INTERFACE PROGRAMMABLE

Dans le contexte MS2I, il est absolument indispensable de disposer d'une interface programmable. En effet, les applications comme la photo-interprétation ou la cartographie ne peuvent être développées uniquement à l'aide d'un langage interactif comme SQLOMEGA. Le contexte de ces applications (machines SUN, systèmes UNIX, programmeurs habitués au langage C, applications déjà développées en langage C) nous a contraint à choisir le langage C comme langage hôte pour notre SGBD.

A partir de ce choix, une étude de faisabilité est en cours, elle a mis en évidence trois modes d'interfaçe possibles entre C et ESTRELLA :

- l'**appel direct** : envoi direct d'une commande au SGBD par l'intermédiaire d'une fonction,

- la **précompilation avec couplage faible** : précompilation du programme, contenant directement des commandes SQLOMEGA, pour masquer l'utilisation d'une bibliothèque de fonctions spécifiques du dialogue entre C et OMEGA,

- la **précompilation avec couplage fort** : précompilation d'un programme, ne contenant plus de SQLOMEGA mais quelques constructions nouvelles proches du langage C de façon à perturber le

moins possible les techniques de programmation.

Ces modes s'inspirent des travaux déjà effectués dans le domaine du couplage entre les langages de programmation et les langages de base de données [Gamerman 87] [Bancilhon 88d].

5.5.1 L'appel direct

Avec l'appel direct, le programme envoie directement ses commandes au SGBD sous forme de chaînes de caractères. Il utilise une bibliothèque de fonctions comme par exemple, CONNECTION(), DECONNECTION(), ENVOYER_COMMANDE() et LIRE_OBJET() qui sont offertes par le SGBD SABRINA [Sabrina 88].

exemple 38 : programme C utilisant l'appel direct

```
CONNECTION(utilisateur, mot_de_passe);
curseur = ENVOYER_COMMANDE("select * from région");
nb = LIRE_OBJET(curseur,région);
while(nb<>0)
      {
        afficher(région);
       nb = LIRE_OBJET(curseur,région);
      }
DECONNECTION();
```

Ce type d'interface semble bien adapté pour des applications simples et des programmeurs maitrisant parfaitement le langage SQLOMEGA (SQL pour SABRINA).

Par contre, ce type d'interface provoque de nombreuses difficultés de programmation :

- les vérifications des commandes SQLOMEGA ne sont effectuées qu'à l'exécution,
- les résultats de requête ne sont accessibles que par des curseurs. On ne peut pas les manipuler globalement,
- les objets complexes doivent être recopiés intégralement en mémoire du programme ou sinon ils ne peuvent plus être manipulés.

5.5.2 La précompilation avec couplage faible

Contrairement aux appels directs, les programmes précompilés subissent un traitement avant d'être exécutés. Le pré-compilateur analyse et transforme ces programmes pour mettre en oeuvre des accès à la base de données. Le couplage faible signifie que dans un programme source, les accès à la base de données ne sont pas transparents. On trouve un exemple de couplage faible dans [Oracle 87] ou [Stonebraker 86].

Un couplage faible entre C et ESTRELLA serait mis en oeuvre en ajoutant la syntaxe SQLOMEGA à la syntaxe du langage C. La précompilation serait chargée de reconnaître les commandes SQLOMEGA et de les substituer par des fonctions d'une bibliothèque semblable à celle utilisée dans l'appel direct.

exemple 39 : programme C avant précompilation en couplage faible

```
région r;
CONNECTION();
SELECT * FROM région INTO r ;
printf("| région | capitale |\n");
while()
      {
       if (NOT FOUND) break;
        printf("| %s | %s |\n", r->nom, r->capitale->nom);
       GET_NEXT(r);
      }
DECONNECTION();
```

Dans l'exemple 39, on déclare une variable de type région, elle est affectée par le résultat d'une requête. La syntaxe "r->capitale->nom" permet l'accès au contenu d'un objet complexe géré par le SGBD.

Ce type d'accès est possible à condition d'effectuer une précompilation importante :

- substitution de "r->capitale->nom" par un appel de l'opérateur d'extraction,
- utilisation d'un gestionnaire d'objets en mémoire du programme. Dans [Benzaken 88], on montre l'utilisation d'une table d'indirections pour atteindre les objets recopiés en mémoire.

La figure 1 montre les différentes transformations du programme dans la cas d'une précompilation avec couplage faible.

Par rapport à l'appel direct, la précompilation avec couplage faible offre de nombreux avantages :

- elle permet d'intégrer SQLOMEGA dans le langage C,

- les vérifications de SQLOMEGA sont effectuées avant l'exécution par le précompilateur,
- les objets complexes sont accessibles.

Cependant elle comporte un inconvénient majeur : la complexité de la syntaxe et de la sémantique du nouveau langage C-SQLOMEGA.

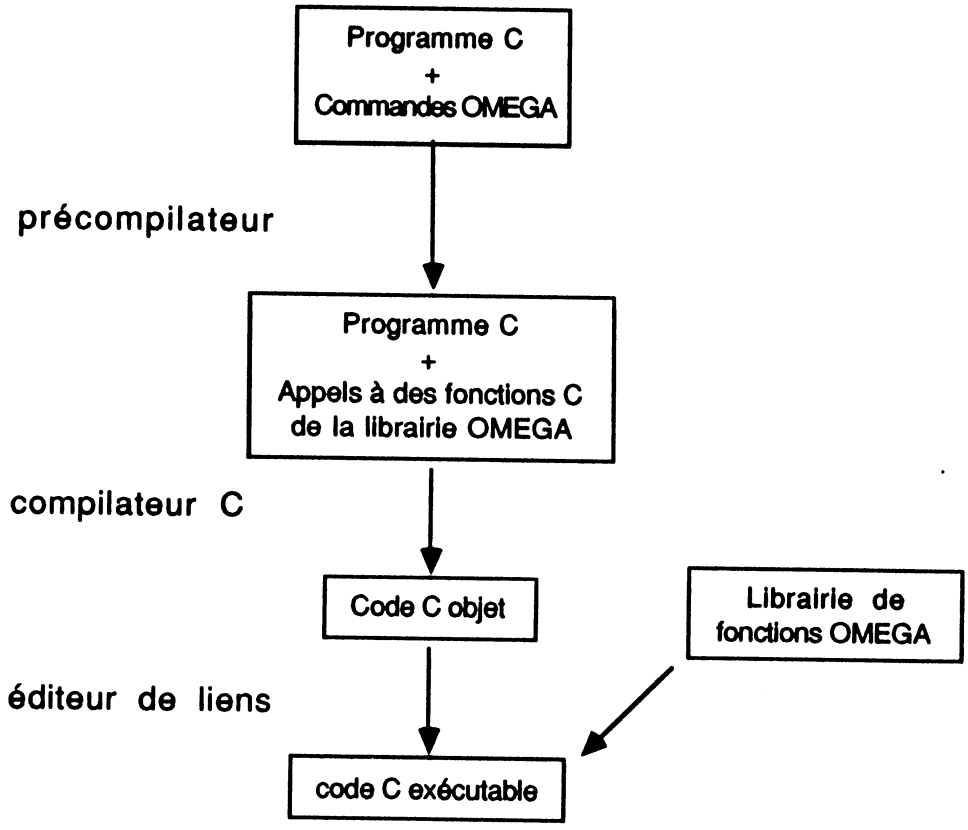


Figure 1 : Compilation du programme C

5.5.3 La précompilation avec couplage fort

Le but d'un couplage fort entre un langage de programmation et un langage de base de données est de résoudre le problème de la complexité. On trouve des exemples de couplage fort dans PASCAL/R, O2, ou GALILEO [Benzaken 85]. [Bancilhon 88d] distingue quatre types de couplage fort :

1- la définition d'un nouveau langage de programmation intégrant la gestion d'une base de données,
2- l'extension d'un langage de programmation pour la connection avec un modèle de données,
3- l'intégration de la gestion d'une base de données dans plusieurs langages existants,
4- la gestion des programmes comme des objets complexes de la base de

données.

Notre intention étant d'utiliser le langage C, seule la solution 2 nous intéresse. Pour ce faire, il faut exclure du couplage toutes les constructions particulières peu semblables au langage C, notamment les commandes SQLOMEGA.

Par contre, on peut introduire des notions voisines comme les ensembles ou la boucle "for each", qui apportent peu de modifications aux techniques de programmation.

exemple 40 : programme C avec couplage fort

```
région *r, R{};
CONNECTION();
R = db_read("région");
printf("| région | capitale |\n");
for each r in R
        {
         printf("| %s | %s |\n", r->nom, r->capitale->nom);
        }
DECONNECTION();
```

L'exemple 40 met en évidence trois nouvelles constructions :
- les structures de type ensemble R{},
- les fonctions d'accès à la base : db_read,
- la boucle for each.

On peut en imaginer d'autres comme les opérations entre ensembles, l'appel de fonctions déclarées dans ESTRELLA.

Pour implanter ces structures, la plupart des langages fortement couplés avec un SGBD utilisent un gestionnaire d'objets. Les objets sont recopiés dans la mémoire du programme lors de leur première utilisation, ensuite le programme n'accède plus à la base de données.

Par rapport aux deux interfaces précédentes, le couplage fort a l'avantage d'être facile à utiliser pour un programmeur C. Par contre, on perd l'utilisation de SQLOMEGA : les expressions de requêtes complexes doivent être programmées (en terme de db_read, for each ...) par les utilisateurs.

5.5.4 L'interface OMEGA-C

Le premier objectif de l'interface OMEGA-C est l'intégration de la base

de données OMEGA dans les applications déjà existantes et à venir. Ces applications sont principalement la photo-interprétation, la cartographie, l'aide au commandement et la télédétection. Nous avons vu qu'elles avaient toutes en commun un certain nombre de traitements et de requêtes : des traitements géographiques, des traitements d'images et des requêtes spatiales.

Ce type de traitements ne peut pas toujours être effectué interactivement, par contre les accès à la base de données sont relativement simples.

Les requêtes ne sont pas déterminées à priori. Elles doivent donc soit être traitées interactivement à l'aide de l'interface interactive, soit être dynamiquement construites dans les programmes.

Une application MS2I recouvre donc à la fois des programmes d'applications ayant des accès simples au SGBD et son interface interactive. Dans ces conditions, nous avons choisi un couplage fort avec précompilation pour interfacer OMEGA et le langage C. Cependant ce couplage fort doit permettre l'interrogation de la base de données avec des requêtes.

La syntaxe et la sémantique exacte du langage OMEGA-C sont en cours de définition. On peut donner les principes qui la guident :

- les nouvelles structures sont des extensions canoniques des constructeurs C : définition d'ensembles , boucle "for each" ...
- les variables de communication avec OMEGA ont un comportement identique aux variables C.
- les manipulations d'objets ESTRELLA ont une syntaxe identique et une sémantique voisine des manipulations de variables C,
- les accès au SGBD sont réduits aux commandes db_read, db_insert, db_delete et db_commit portant sur des objets ou des classes (db_commit valide les mises à jour).
  Afin de conserver certaines possibilités du langage SQLOMEGA, la fonction db_read acceptera en entrée des requêtes.

Nous avons prévu d'implanter cette interface en utilisant un gestionnaire d'objets en mémoire.

exemple 41 : programme OMEGA-C

```
int compter_région( nb_gd_villes )
int nb_gd_villes
{
  région *r, R{};
  int résultat = 0;
  CONNECTION();
```

```
    R = db_read("select * from région
                    where COUNT(grandes_villes) >= %d",nb_gd_villes);
    for each r in R
          {
           if (r->capitale != nil) résultat++;
          }
    DECONNECTION();
    return(résultat);
}
```

Dans l'exemple 41, nous montrons comment définir en OMEGA-C une fonction. Cette fonction pourra être compilée puis exécutée en relation avec la base de données OMEGA. Elle pourra aussi être incluse parmi les fonctions connues de la base de données pour pouvoir être appelée dans les requêtes SQLOMEGA.

5.6 CONCLUSION

Dans ce chapitre nous avons proposé deux interfaces pour le SGBD OMEGA:
- une interface interactive reposant sur le langage SQLOMEGA,
- un couplage entre le langage C et la base de données.

Le langage SQLOMEGA est une extension de SQL pour prendre en compte les notions du modèle ESTRELLA comme l'héritage, la composition d'objets, les ensembles, les tableaux, les versions ou les fonctions. Ce langage supporte les fonctionnalités traditionnelles des SGBD :
- opérations sur le schéma conceptuel,
- opérations sur les objets,
- accès associatif.

Le couplage entre C et OMEGA est en cours de définition. Il doit permettre l'utilisation des concepts du modèle ESTRELLA dans un programme d'application sans apporter de lourdes contraintes aux programmeurs. Son implantation doit mettre en oeuvre des techniques d'optimisation de transferts entre la mémoire de l'application et la base de données comme les gestionnaires d'objets.

Ces deux interfaces sont complémentaires pour les applications MS2I. L'interface programmable est destinée aux traitements de longue durée comme les traitements d'images, utilisant peu la puissance du modèle ESTRELLA. Le langage SQLOMEGA est offert aux utilisateurs interactifs qui veulent élaborer des requêtes complexes.

CHAPITRE 6

LE SERVEUR OMEGA

chapitre 6 : LE SERVEUR OMEGA

6.1 INTRODUCTION

Ce chapitre est consacré à la description de l'implantation du serveur de base de données OMEGA. Nous nous concentrerons plus particulièrement sur la gestion, le stockage et l'accès aux objets ainsi que sur l'interprétation des requêtes.

L'architecture générale d'OMEGA tient compte des nombreux travaux effectués dans ce domaine. Parmi toutes les architectures proposées pour les nouveaux SGBD, nous retiendrons quatre directions :

- la première repose sur l'extension d'un SGBD existant au moyen d'opérateurs supplémentaires (figure 1). Cette méthode a été choisie, par exemple dans les projets STARBUST [Lindsay 87] ou SABRINA. STARBUST conserve l'implantation traditionnelle des SGBD relationnels mais définit des extensions (index, contraintes d'intégrité...) associées aux relations. La gestion de ces extensions est diffusée dans toutes les couches du SGBD.
  [Kiernan 87] propose d'étendre le SGBD relationnel SABRINA pour la gestion de domaines complexes. Cette extension prévoit l'intégration d'un interprète LISP dans le corps du SBGD.

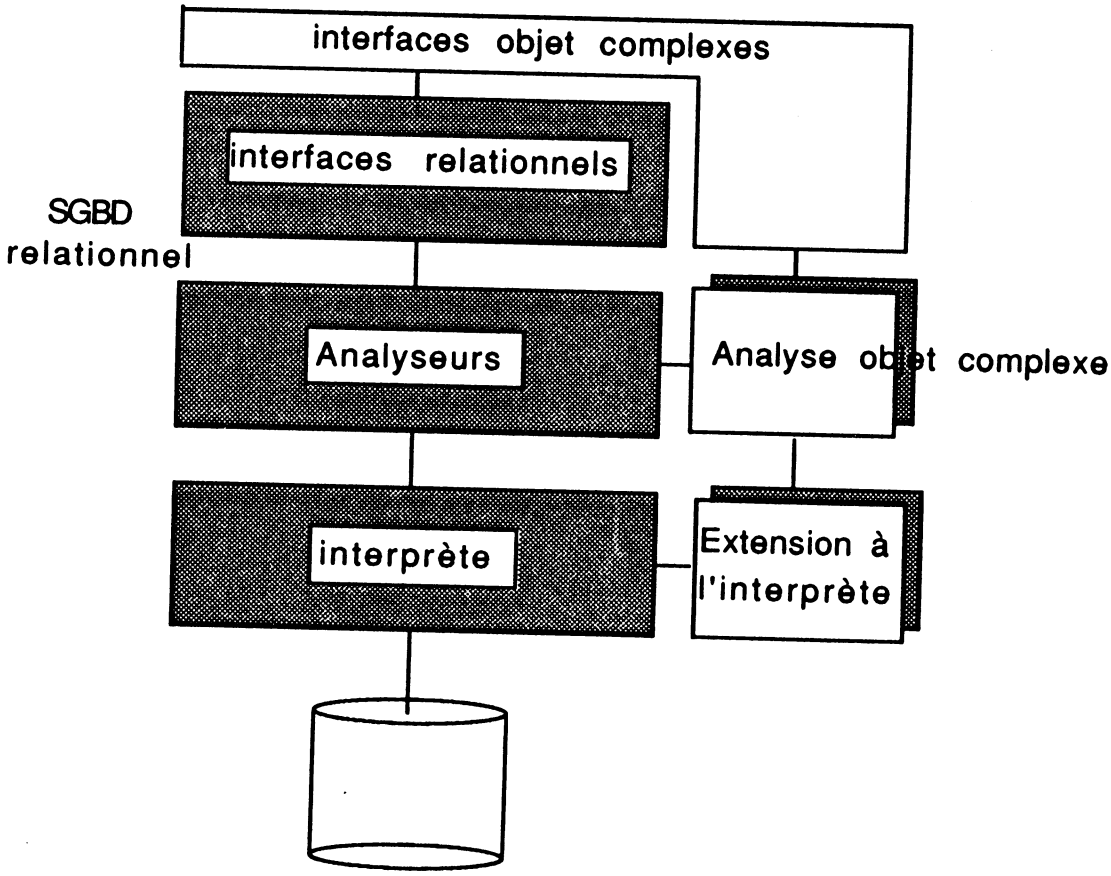


Figure 1 : extension d'un SGBD relationnel

- la seconde consiste en la réalisation d'une couche "gestionnaire d'objets complexes" au dessus d'un SGBD traditionnel [Valduriez 87] (figure 2). Cette technique a déjà été utilisée pour le projet TIGRE [Velez 84]. Elle a été proposée pour la gestion d'objets dans POSTGRES [Rowe 87].

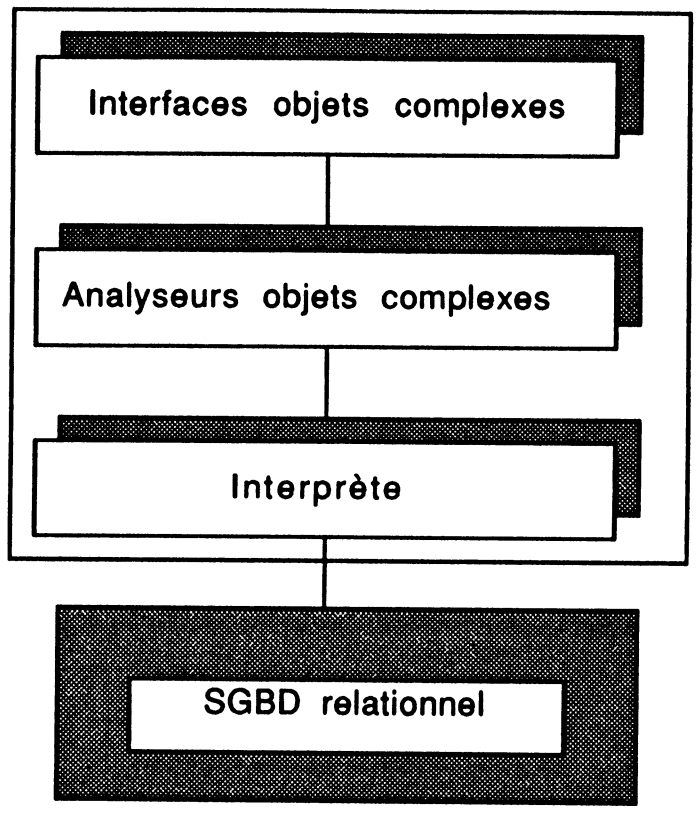


Figure 2 : réalisation d'une couche objet sur un SGBD relationnel

- la troisième méthode est la plus courante : développement complet d'un nouveau SGBD (figure 3). La plupart des nouveaux systèmes veulent mettre en oeuvre des techniques qui n'étaient pas nécessaires aux SGBDR : regroupement d'objets, indexation géographique... D'autre part, pour obtenir de bonnes performances, les concepteurs de ces systèmes pensent qu'il faut repartir de zéro.

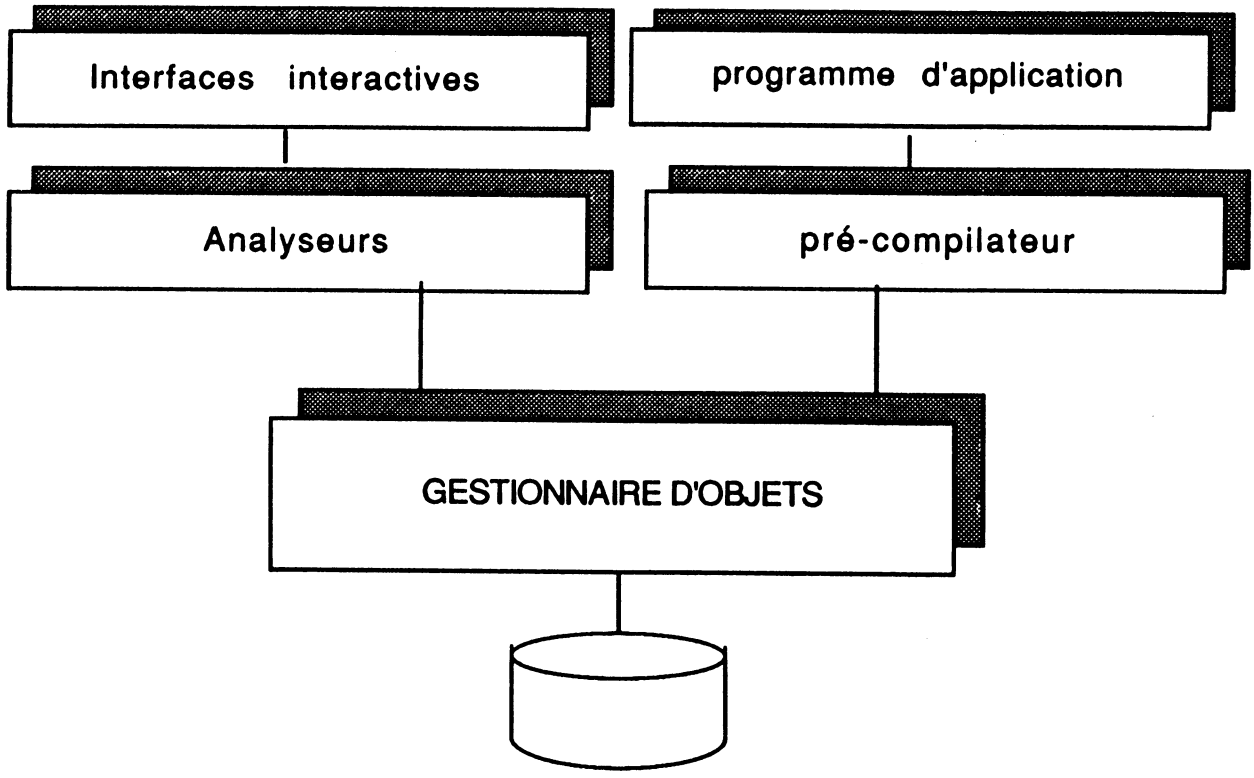


Figure 3 : Développement complet d'un SGBD orienté-objet

- la quatrième méthode consiste en la réalisation d'un noyau de bas niveau permettant de générer des couches spécifiques de chaque application (figure 4). Cette méthode est notamment employée par les générateurs de SGBD comme EXODUS, DASDBS, GENESIS ou GEODE.

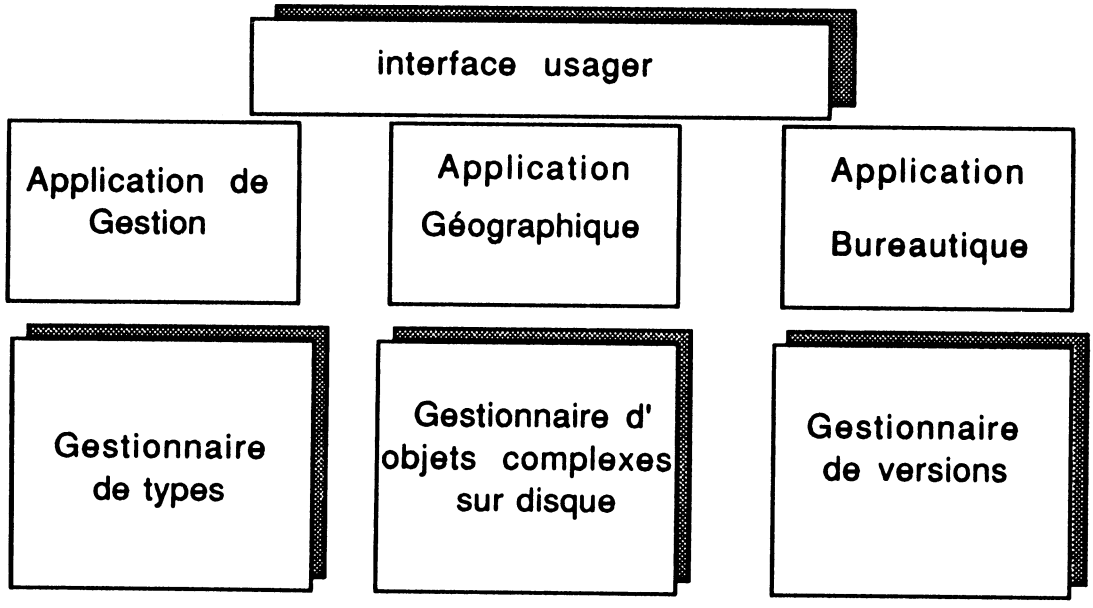


Figure 4 : réalisation de primitives pour un générateur de SGBD

Un prototype du SGBD OMEGA est en cours de développement. Il est implanté au dessus du SGBD ORACLE ce qui correspond à la seconde approche. Il utilise aussi d'autres gestionnaires de données pour gérer les objets particuliers comme les images, les textes ou les index géographiques.

Les raisons qui nous ont conduit à construire au dessus d'ORACLE sont multiples :

- en premier lieu, il convient de préciser que le groupe MATRA dont MS2I est une filiale, a un contrat avec ORACLE. Il n'était donc pas envisageable d'utiliser un autre SGBD relationnel commercialisé. D'autre part, nous ne disposons pas des sources d'ORACLE, ce qui écarte la première solution (adjoindre de nouvelles fonctionnalités à un SGBDR),
- la troisième possibilité repose sur le développement complet d'un nouveau SGBD. Cette possibilité a été écartée : le but de notre étude est de montrer la faisabilité d'un SGBD pour la géographie. Un prototypage rapide était possible à l'aide d'ORACLE alors que le développement total est un projet d'une toute autre envergure,
- la quatrième solution (développement de primitives puis génération de SGBD) a déjà été repoussée au niveau de la modélisation. Il s'agit d'un travail intéressant mais très complexe.

La seconde solution que nous avons choisie entraîne une certaine redondance dans les opérations (OMEGA puis ORACLE vont effectuer des vérifications sémantiques) et amoindrit l'intérêt d'un modèle orienté-objet en utilisant un stockage plat qui provoque de faibles performances. Par contre, elle nous apporte des avantages importants :

- ORACLE est un produit bien connu à MS2I, son cadre d'utilisation est donc assez favorable,
- le prototype que nous allons construire pourra tirer parti des fonctionnalités d'ORACLE dans de nombreux domaines : concurrence, sécurité...
- ORACLE dispose d'une interface SQL, SQLOMEGA se présente donc aux utilisateurs comme une extension de son interface interactive,
- les techniques de projection d'un modèle à objets complexes sur un modèle relationnel ont été fortement étudiées et peuvent être appliquées dans le cas d'OMEGA.

Ce choix conditionne toute l'implantation du système OMEGA. Dans la section suivante, nous décrivons l'architecture du système. Les sections

suivantes présentent les gestionnaires de classes et d'objets qui bien sûr, utilisent ORACLE. La section 4 démontre l'exécution des requêtes SQLOMEGA, notamment l'évaluation des opérateurs sur les classes et l'interprétation des fonctions. La section 5 montre l'utilisation des fonctions et des classes descripteurs pour manipuler des données multimédia. Ensuite nous présentons l'état d'avancement du prototype et une conclusion sur cette implantation.

6.2 ARCHITECTURE DU SGBD OMEGA

Le SGBD OMEGA est implanté sur un système UNIX, il utilise le SGBDR ORACLE ainsi que des SGF pour gérer des données volumineuses (images, cartes, ...). Il est connecté comme serveur d'objets à des stations de travail qui sont ses clientes via un réseau éthernet (voir figure 5).

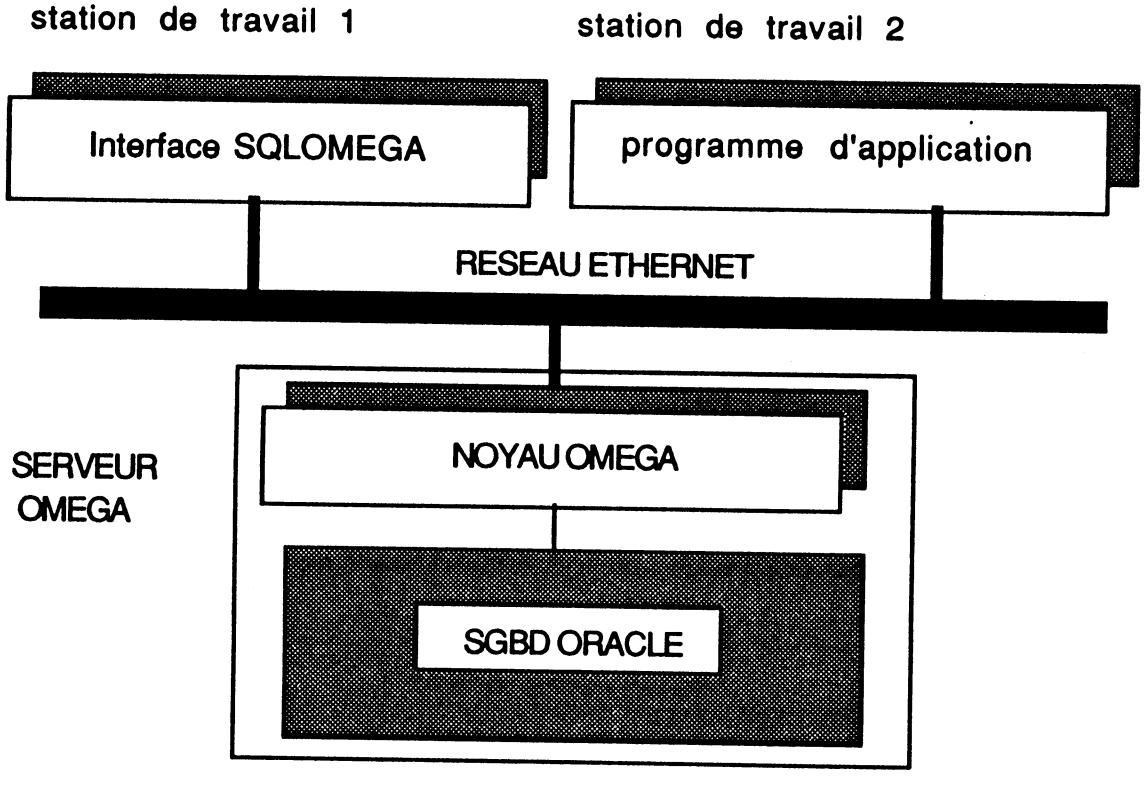


Figure 5 : CONFIGURATION RESEAU DU SGBD OMEGA

Le noyau d'OMEGA met en oeuvre en un seul processus UNIX, les possibilités du modèle ESTRELLA. Ce processus regroupe plusieurs fonctionnalités:

- gestion des classes : une classe est représentée par plusieurs relations dans ORACLE et sa définition est stockée dans des catalogues,

- gestion des objets : il s'agit de gérer le stockage et l'accès sur le disque en utilisant ORACLE,
- gestion des fonctions : la création, la destruction, le stockage et leur exécution avec la liaison dynamique,
- gestion des prédicats et des transactions: ce sont des opérations encore peu étudiées qui feront l'objet d'une étude ultérieure.

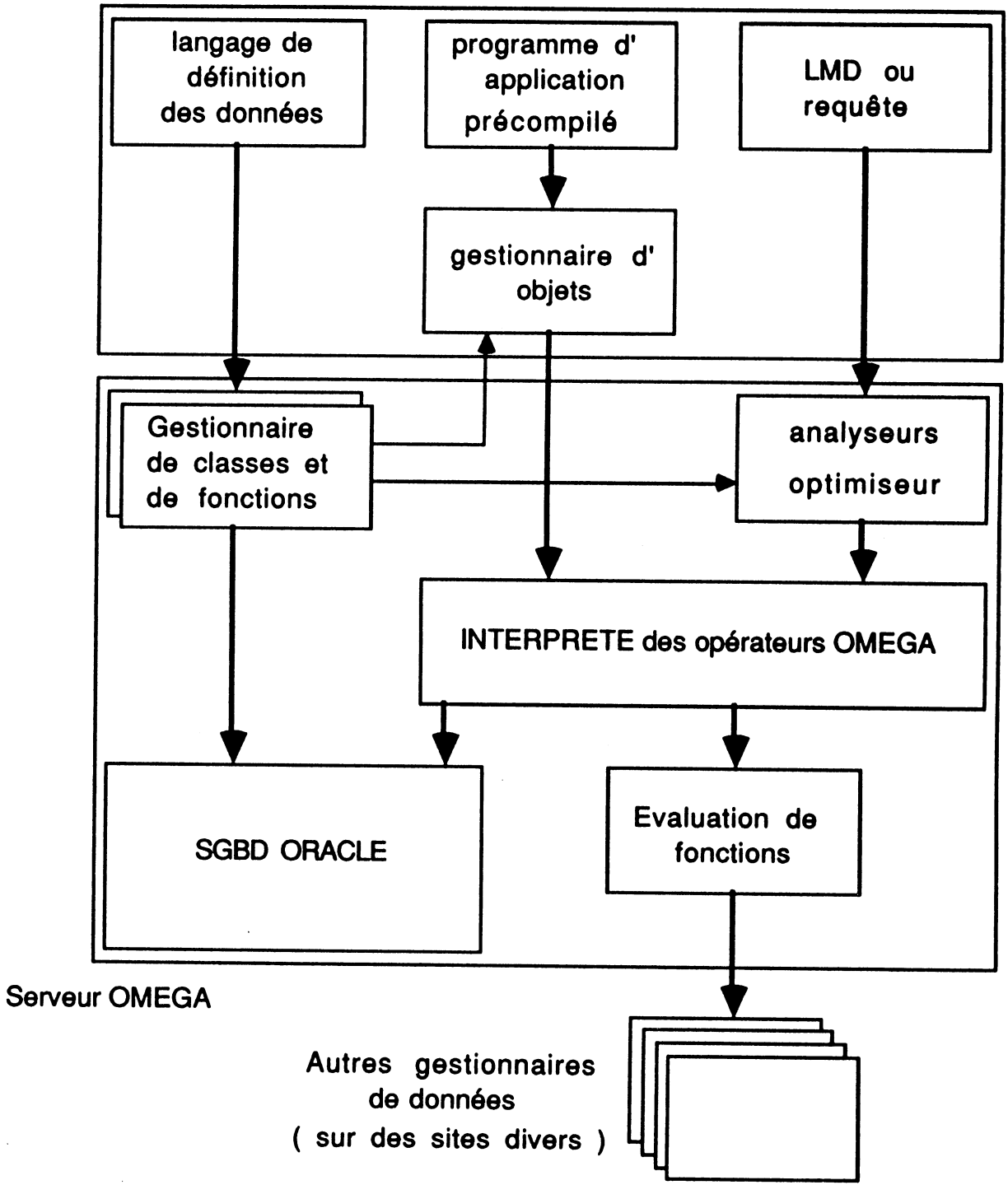


Figure 6 : ARCHITECTURE DU NOYAU OMEGA

L'interface interactive SQLOMEGA est un processus indépendant. Il met en oeuvre d'autres fonctionnalités classiques des langages de base de données dont certaines peuvent être intégrées au noyau d'OMEGA :

- analyse lexicale et syntaxique,
- analyse sémantique,
- transformation et optimisation de requêtes en opérateurs sur les classes,
- interprétation des opérateurs.

Les opérateurs sur les classes (définis au chapitre 5) sont programmés en utilisant les opérateurs disponibles dans ORACLE :

- la sélection sur une classe est obtenue en utilisant la sélection sur les relations,
- la projection est réalisée de la même façon,
- le produit de deux classes correspond aux produits de plusieurs relations,
- l'extraction d'objets est programmée à l'aide de jointures relationnelles.

L'interface avec le langage C met en oeuvre deux opérations différentes :

- précompilation des programmes,
- exécution en utilisant les gestionnaires de noyau OMEGA (cette exécution ne fait pas intervenir des notions de transformation ou d'optimisation de requêtes).

On introduit la notion d'identificateur interne (appelé "surrogate" dans la littérature) pour pouvoir gérer simplement les notions d'identité d'objets et de composition d'objets. Toutes les classes, tous les objets et toutes les fonctions ont un identificateur interne dans OMEGA. Cet identificateur est stocké dans un champ "surrogate" de toutes les relations utilisées par OMEGA dans ORACLE.

6.3 LA GESTION DES CLASSES

La description du schéma conceptuel en terme de classes est stockée dans un ensemble de catalogues. Ces catalogues sont en fait des relations dans ORACLE qui sont créées une fois pour toutes lors de la procédure d'installation de la base de données OMEGA.

L'interprétation des commandes du LDD modifie ces catalogues et crée les relations qui vont contenir les objets des classes. L'exécution des commandes du LMD est simplement traduite en opérations sur ces relations qui contiennent les objets [Valduriez 87] [Velez 84].

Les catalogues décrivent chaque particularité des définitions de classes:
- un catalogue pour l'identification des classes : C_CLASSE,
- un catalogue pour le treillis de classe : C_TREILLIS,
- un catalogue pour les attributs : C_ATTRIBUT.

D'autre part, un catalogue supplémentaire gère la définition des attributs multivalués (tableau borné ou ensemble) : C_SET_ARRAY.

Pour les autres éléments d'un schéma conceptuel OMEGA (fonctions et prédicats), le rôle de catalogue est échu à une classe particulière (FUNCTION ou PREDICATE).

6.3.1 Les catalogues

La première fonctionnalité du gestionnaire de classe est donc de stocker la description des classes dans les catalogues. Pour cela, le gestionnaire de classe utilise ORACLE : chaque catalogue correspond à une relation. Toutes les classes y sont identifiées par un surrogate. Les attributs sont eux aussi identifiés par un surrogate. Les constituants de ces relations "catalogue" permettent de stocker la définition des classes :

C_CLASSE : c_classe gère l'existence et la nature des classes
surrogate : entier -> identificateur de la classe
nom : chaîne -> nom de la classe
état : entier -> état de la classe
valeurs possibles : SIMPLE, HISTORIQUE, IMBRIQUEE, DESCRIPTEUR, VUE
path : chaîne -> chemin d'accès des classes descripteurs

C_TREILLIS : c_treillis gère les liens dans le treillis
ss_surrogate : entier -> identificateur de la sous-classe
su_surrogate : entier -> identificateur de la super-classe

C_ATTRIBUT : c_attribut identifie les attributs dans les classes
surrogate : entier -> identificateur de l'attribut
classe : entier -> identificateur de sa classe
type : entier -> identificateur du type de l'attribut
mode : entier -> modalité de définition de l'attribut
valeurs possibles : (MONOVALUE, TABLEAU, ENSEMBLE) x (CLASSE DE BASE, CLASSE IMBRIQUEE REFERENCE, DESCRIPTEUR)
père : entier -> identificateur de la superclasse d'où provient un attribut redéfini sur une sous-classe
défaut : entier -> identificateur de la valeur par défaut de l'attribut

clé : booléen -> appartenance de l'attribut à la clé de la classe
valeurs possibles : OUI, NON
statique : booléen -> accessibilité en mise à jour
valeurs possibles : OUI, NON
null : booléen -> possibilité de non affectation de l'attribut
valeurs possibles : OUI, NON

C_SET_ARRAY : c_array gère les dimensions des attributs multivalués
surrogate : entier -> identificateur de l'attribut
classe : entier -> identificateur de sa classe
nb_dim : entier -> nombre de dimensions
valeurs possibles : ZERO (ensemble)
N (tableau)
minmax: chaîne -> liste des bornes du tableau pour chaque dimension

Chaque commande du LDD entraîne des répercussions sur ces catalogues. La création de classe (commande **create class**) génère une série de n-uplets :

- un n-uplet dans le catalogue C_CLASSE,
- N n-uplets dans C_ATTRIBUT (autant que d'attributs cités),
- p n-uplets dans C_TREILLIS (autant que de super-classes),
- q n-uplets dans C_SET_ARRAY (autant que d'attributs multivalués).

exemple 1 : création d'une classe et répercussions dans les catalogues

```
CREATE CLASS image_radar ISA cliché
BEGIN
key   nom      : char(20),
      capteur  : radar NOT NULL,
      contenu  : image NOT NULL,
      angle    : ARRAY [1..10] OF real,
      échos    : SET OF points
END
```

On note S("xxxx") la fonction qui recherche dans C_CLASSE, le surrogate d'une classe de nom "xxxx".

La commande de l'exemple 1 provoque l'insertion des n-uplets suivants :

C_CLASSE :
(1000, "image_radar" , CLASSE SIMPLE , "")

C_TREILLIS :

(1000 , S(cliché))

C_ATTRIBUT :
(1001, "nom" ,1000, CHAINE , CLASSE DE BASE, 0, 0, OUI, NON , NON)
(1002, "capteur", 1000, S(radar) , REFERENCE , 0, 0, NON, NON , OUI)
(1003, "contenu", 1000, S(image), DESCRIPTEUR , 0, 0, NON, NON , OUI)
(1004, "angle" , 1000, REEL , TAB x BASE , 0, 0, NON, NON , NON)
(1005, "échos" , 1000, S(point) , SET x IMBRIQUEE, 0, 0, NON, NON , NON)

C_SET_ARRAY :
(1004, 1000 , 1 , "1..10")
(1005, 1000 , 0 , "")

Les autres commandes du LDD modifient de même les catalogues :

- la commande drop class "xxxx" supprime des catalogues les n-uplets concernant la classe "xxxx" et ses sous-classes,
- alter class modifie certains n-uplets des catalogues :
  - add attribut ajoute un n-uplet dans C_ATTRIBUT et s'il est multivalué dans C_SET_ARRAY,
  - drop attribut supprime un n-uplet dans C_ATTRIBUT et s'il est multivalué dans C_SET_ARRAY,
  - add history modifie l'état dans C_CLASSE et rajoute un attribut historique
  - drop history effectue l'opération inverse.

6.3.2 Représentation des classes dans un SGBDR

Outre la gestion de la définition des classes, le gestionnaire de classes doit préparer le stockage des objets : une classe est aussi un ensemble d'objets. Dans un SGBDR, un ensemble peut être stocké soit dans la valeur d'un attribut de grande taille, soit dans une relation. La première solution ne peut convenir car on ne peut pas manipuler facilement les éléments. C'est pourquoi, comme dans [Velez 84] [Valduriez 87], les ensembles (pour les attributs multivalués) et les classes sont représentées par des relations.

L'interprétation des commandes du LDD, après transformation des catalogues, agit sur la définition des relations dans ORACLE. Ces relations sont créées à l'interprétation de la commande **create class**. A chaque commande de création de classe, on procède de la façon suivante :

- mise à jour des catalogues,
- création des relations représentant la classe,

- création des relations représentant les attributs multivalués.

6.3.2.1 Relations représentant une classe non historique

Une classe non historique est représentée dans ORACLE, par une relation de même nom. Les constituants de la relation correspondent aux attributs propres de la classe. On ajoute le constituant "surrogate" qui permet de discriminer les objets stockés dans cette relation.

Les types de constituants de cette relation sont les mêmes que ceux définis dans la classe lorsqu'il s'agit de types de base. Les autres constituants sont de type entier : ils ont pour valeur le "surrogate" d'autres objets d'OMEGA.

Une instance d'une classe non historique est présente sous forme d'un seul n-uplet, dans la relation représentant la classe. Par contre, un objet est en général, stocké dans plusieurs n-uplets de différentes relations.

exemple 2 : transformation d'une classe simple en relation dans un SGBDR

soit C un classe définie telle que
superclasse(C) = {S1, S2, ... , Sn}
structure_propre(C) = [a1 : C1, ... , an : Cn]
clé(C) = [a1 : C1, ... , ap : Cp]

La liaison entre C et ses superclasses est gérée dans le catalogue C_TREILLIS. La relation créée dans ORACLE est définie comme suit :

relation(C) = (surrogate : entier,
a1 : T1
. . / . .
an : Tn)

si Ci est une classe de base alors Ti = Ci sinon Ti = entier

6.3.2.2 Relations représentant une classe historique

La représentation des classes historiques dans ORACLE met en oeuvre deux relations. La première R1 permet la gestion de l'identité des objets, la seconde R2 stocke les versions.

Les constituants de la relation R1 sont les clés de la classe (qui constituent l'identité de ses objets, immuables dans le temps) : un n-uplet de R1 représente une instance de la classe avec toutes ses versions.

Les constituants de R2 sont les autres attributs de la classe : un n-uplet

de R2 correspond à une version d'un objet de la classe.

Conformément à la règle d'identification dans ORACLE, on ajoute un constituant "surrogate" à chacune des relations. Dans R1, "surrogate" est l'identificateur de l'objet alors que dans R2, "surrogate" est l'identificateur de la version. Cependant, ce n'est pas suffisant pour gérer la dépendance entre les versions et les objets. Pour cela, on ajoute un nouveau constituant dans R2 : "S_objet" qui, pour chaque version, contient l'identificateur de l'objet.

exemple 3 : transformation d'une classe historique

soit H une classe définie telle que
superclasse(H) = {S1, S2, ... , Sn}
structure_propre(H) = [a1 : C1, ... , an : Cn, d : date]
clé(H) = [a1 : C1, ... , ap : Cp]

les relations R1 et R2 créées dans ORACLE sont définies comme suit :

R1 = (surrogate : entier,
a1 : T1,
../..
ap : Tn)

R2 = (surrogate : entier,
S_objet : entier,
ap+1 : Tp+1,
../..
an : Tn,
d : date)

si Ci est une classe de base alors Ti = Ci sinon Ti = entier

On peut remarquer que les commandes de transformation entre les classes historiques et les classes simples entraînent des bouleversements importants dans la structure des relations ORACLE.

6.3.2.3 Relations représentant un attribut multivalué

Les attributs multivalués comme les classes correspondent à des ensembles d'objets. Leur valeur est stockée dans le SGBDR de la même façon que les classes.

Lors de la création d'un attribut multivalué (commande **create class** ou **alter class add attribut**), l'interprète du LDD crée une relation.

exemple 4 : relation stockant les valeurs multiples

soit C une classe définie telle que

superclasse(C) = {S1, S2, ... , Sn}
structure_propre(C) = [a1 : C1, ... , E : { Ce }, ... ,
A : array (Ca), ... , an : Cn]
clé(H) = [a1 : C1, ... , ap : Cp]

relation créée pour gérer les attributs E et A:

```
relation(E) = ( surrogate      : entier,
                E              : Te     )

relation(A) = ( surrogate      : entier,
                n_ordre        : entier,
                A              : Ta     )
```

Les types Te et Ta sont définis de la même façon que dans les exemples 2 et 3.

Quelque soit le nombre de dimensions du tableau A, sa relation n'a qu'un constituant n_ordre pour gérer les indices : on applique une fonction classique de transformation d'indices en dimension N dans une indexation linéaire.

6.3.3 Autres fonctions du gestionnaire de classes

Nous avons vu deux fonctions essentielles du gestionnaire de classes : la gestion des définitions des classes et la gestion de leur représentation dans ORACLE. Ces deux fonctions sont activées lors de l'interprétation des commandes du LDD.

Une troisième fonction est importante : le gestionnaire de classes doit fournir aux autres composants du noyau OMEGA des informations sur la nature des classes nécessaires à leur fonctionnement. Il utilise pour cela le contenu des catalogues. Cette fonction est activée quasiment lors de l'exécution de toutes les commandes OMEGA.

Lors de l'utilisation du langage SQLOMEGA, l'analyseur sémantique et l'optimiseur émettent des demandes au gestionnaire de classes pour vérifier la validité des commandes et pour connaître les méthodes d'accès efficaces.

Ces mêmes demandes sont envoyées par le précompilateur C et le

gestionnaire d'objets lors de l'utilisation de programmes d'applications.

On peut supposer que le gestionnaire de classes aura à répondre à différentes demandes :

- existence de la classe : classe(C),
- structure d'une classe : structure(C),
- structure propre : structure_propre(C),
- clé et clé propre : clé(C),
- super-classes d'une classe : superclasse(C).

6.3.4 Initialisation de la base de données

Lors du démarrage initial de la base de données OMEGA, il faut préparer les relations ORACLE qui vont contenir les catalogues. Cette phase d'initialisation est effectuée une fois pour toutes par un processus spécifique indépendant du noyau OMEGA. Ce processus est appelé INIT_OMEGA.

6.4 LA GESTION DES OBJETS

Dans cette section, nous allons nous intéresser aux objets. Nous arborderons deux points : le stockage et les opérations élémentaires. Les opérations élémentaires sont les opérations du LMD : création, mise à jour et suppression d'objets ou de versions. On ne prend pas en compte les requêtes qui feront l'objet de la section suivante.

6.4.1 Représentation relationnelle des objets

Les objets sont des instances de classe. Ils sont stockés dans le support de stockage des classes : le SGBD relationnel ORACLE. Comme pour les classes, tous les objets ont un identificateur interne appelé "surrogate".

Soit O = (c,v) un objet d'une classe C. On distingue plusieurs modes de stockage selon la nature de C et de ses super-classes :

- C est une classe simple sans super-classe qui est représentée par une relation Ro dans ORACLE.

  O = (c,v) est stocké par un n-uplet dans Ro : Ro(O) = [S(c), c, V]
  S(c) est la fonction de calcul de l'identificateur interne de l'objet à partir de sa clé.
  V est le résultat de la substitution dans v de toutes les clés de objets par leurs identificateurs internes.

- C est une classe simple avec des super-classes simples Si, chacune

représentée par une relation Ri.

c et v peuvent être décomposés suivant la provenance des attributs :
c = (c0, c1, ... , cn) et v = (v0, v1, ... , vn)
O est stocké à l'aide d'un n-uplet par relation Ri :
Ri(O) = [S(c), ci, Vi]
S(c) est l'identificateur interne de O dans ORACLE
Vi sont les résultats des substitution de clés dans vi

- C est une classe historique sans super-classe. Elle est représentée par deux relations Ro et Rv.
  la définition O = (c,v) peut être étendue par O = (c, { dj, vj }) qui représente l'objet et ses versions.
  O est réparti sur les deux relations Ro et Rv :
  - un n-uplet dans Ro : Ro(O) = [S(c), c]
  - un n-uplet par version dans Rv : Rv(O,j) = [S(c,dj), S(c), dj, Vj]

  S(c) est l'identificateur de l'objet et S(c,dj) est l'identificateur de jème versions.

- C est une classe historique ou simple avec des super-classes historiques SHi et des super-classes simples Sli. Il s'agit du cas le plus général. on déduit sa solution à partir des trois cas particuliers précédents :
  O = (c , v) avec c = (c0, c1, ... , cn) et v = (v0, v1, ... , vn)
  si vi appartient à une classe historique SHi, elle peut être décomposée en : vi = { dij, vij }
  O est réparti sur l'ensemble des relations représentant C, les SHi et les Sli :
  - dans une Sli, on trouve : Ri(O) = [S(c), ci , Vi]
  - dans une SHi, on a un n-uplet dans Rio : Rio(O) = [S(c), ci]
    et un n-uplet par version : Riv(O,j) = [S(c,dj), S(c), dj , Vj]

Dans tous les cas, la valeur d'un attribut redéfini est stockée dans la relation de sa classe d'origine. Le lien entre la valeur et son emplacement est réalisé grâce à l'attribut "père" du catalogue C_ATTRIBUT.

L'identificateur interne de l'objet O est calculé une fois pour toutes : s = S(c). Il est propagé dans toutes les relations contenant des parties de O et permet ainsi de les lier entre elles.

Ce principe de représentation des objets crée quelques inconvénients :

- les objets sont dispersés sur plusieurs relations : l'accès à un objet

complet n'est pas simple, il nécessite des opérations de jointures,
- l'utilisation d'identificateur interne aurait pu être intégré au modèle de données comme dans [Lécluse 87].

Cependant, on en retire encore plus d'avantages :

- le stockage dans le SGBD relationnel n'est possible qu'en effectuant cette normalisation des objets : il permet un prototypage rapide sur un outil éprouvé. On récupére de nombreuses fonctionnalités base de données,
- les opérateurs de l'algèbre relationnelle peuvent être réutilisés : la reconstruction d'un objet avec des jointures mais aussi l'accès associatif à l'aide de l'opérateur de sélection,
- l'utilisation de l'identificateur interne permet une mise en oeuvre simple de la composition d'objets,
- la correspondance entre les clés et l'identificateur interne simplifie l'exécution des requêtes en ramenant les tests de comparaison sur l'égalité des objets à la comparaison de deux "surrogate".

exemple 5 : représentation d'une gare et d'une commune (voir annexe 1)

on considère deux objets g et c respectivement des classes gare et commune définis comme suit :

g = [numéro=1002, nom="saint lazare", nb_voies=25, type="gare"
ville = (SELECT * FROM commune WHERE nom = "Paris")]
c = [numéro=957, nom="Paris", nb_habitant=4000000, type="commune"]

g et c sont représentés dans les trois relations suivantes :

R(OBJET) = R1 (surrogate: int, numéro: int, type: char(20))
R(GARE) = R2 (surrogate: int, nom: char(20), nb_voies: int, ville: int)
R(COMMUNE) = R3 (surrogate: int, nom: char(20), nb_habitants: int)

Ils ont pour valeurs :

R1(g) = (S(g) , 1002 , "gare")
R1(c) = (S(c) , 957 , "commune")

R2(g) = (S(g) , "saint lazare", 25 , S(c))
R3(c) = (S(c) , "Paris" , 4000000)

Dans l'exemple 5, l'attribut "ville" d'une gare est monovalué. Il est

affecté par une requête sur la classe "commune". On peut faire deux remarques sur ce procédé :

- le résultat de la requête peut renvoyer 0, 1 ou N objets. OMEGA réagit de la manière suivante :
  - 0 objet : ville prend la valeur "non affectée",
  - 1 objet : ville reçoit l'identificateur interne de cet objet,
  - N objets : OMEGA signale cette anomalie et ne crée pas d'objet.
- la requête renvoie un objet qui est le résultat instantanné de son exécution. La gestion de la cohérence entre la requête et ses résultats est à la charge des utilisateurs. Pour ce faire, ils peuvent par exemple, faire porter les conditions sur les clés des objets sélectionnés.

En supposant que la requête est retournée un seul objet c, le surrogate de c est utilisé pour représenter la référence de g à c. Ce surrogate n'est pas accessible aux utilisateurs. Il n'est pas affecté lors des mises à jour de c : la relation de composition entre g et c n'est donc pas altérée.

L'identificateur interne sert donc pour la représentation de la composition d'objets, il est aussi utilisé pour les calculs d'égalité entre objets.

L'identificateur d'un objet est calculé lors de la création de l'objet dans la base de données. En fait, l'identificateur d'objets est obtenu en concaténant deux identificateurs : l'identificateur interne de la classe et le numéro d'instance dans la classe. Cet identificateur est unique puisque l'identificateur de classe est unique.

De même lors de la création d'un objet, le SGBD assure l'unicité de la clé. Les relations stockant une classe sont donc la représentation en extension d'une relation bi-univoque entre l'identificateur interne et la clé d'un objet. Ainsi tout test d'identité de deux objets (égalité de leurs clés communes) peut être exécuté par un test d'égalité de leur identificateur interne.

exemple 6 : recherche des gares de la même ville que la gare St Lazare

```
R: SELECT g.nom
   FROM   gare g , gare l
   WHERE  g.ville = l.ville
   AND    l.nom = "St Lazare"
```

La requête R est traduite en opérateur OMEGA :

R : (GARE X (GARE : [nom = "St Lazare"]) [ville = ville]) [nom]

au lieu de

R : E = (GARE : [nom = "St Lazare"]) -> ville
F = GARE -> ville
P = prédicat(E, F , clé(COMMUNE))
(E x F : P) [nom]

Le test sur l'égalité des surrogate permet d'optimiser le calcul des requêtes. Il supprime les extractions de sous-objets qui sont des opérations couteuses (voir exemple 6).

6.4.2 opérateurs sur les objets

Les opérateurs pris en compte pour la gestion des objets sont classiquement la création, la suppression, la mise à jour et l'accès aux objets. Ces opérateurs sont activés soit directement par un programme d'application précompilé, soit lors de l'évaluation d'une commande du LMD.

L'évaluation d'une commande interactive comprend plusieurs étapes : analyse lexicale et syntaxique, analyse sémantique, génération d'une arborescence représentant la commande et enfin exécution de l'arborescence par l'interprète d'OMEGA (voir figure 6).

La signification de chacune de ces phases sera détaillée dans la section suivante. Les opérateurs que nous allons décrire interviennent en phase d'interprétation de l'arborescence :

- l'opérateur de création (appelé CREER) peut générer un objet, une version ou les deux. Il prend en entrée la liste des valeurs des attributs (voir exemple 5). Il vérifie l'unicité des clés, calcule l'identificateur interne puis crée les n-uplets dans les relations représentant la classe. Cet opérateur est récursif : la création d'un objet ou d'une version entraîne la création des objets imbriqués.

- l'opérateur de mise à jour (appelé CHANGE) opère sans aucune différence sur les objets et sur les versions. Il adopte un mécanisme voisin de l'opérateur de création.

- l'opérateur de suppression (appelé DETRUIRE) sur une classe C reçoit en entrée l'identificateur interne de l'élément à détruire. Il peut s'agir d'un identificateur d'objets ou d'un identificateur de versions. Cet opérateur supprime les n-uplets contenant cet identificateur dans les relations représentant la classe C. Il a un fonctionnement récursif : les objets ou les versions imbriquées doivent être détruits.

- l'opérateur d'accès (appelé GET) reçoit en entrée un identificateur interne. Comme pour l'opérateur de suppression, il peut s'agir d'un objet ou d'une version. Cet opérateur reconstitue l'objet à partir des n-uplets qui le décrivent dans ORACLE.

6.5 EVALUATION DES COMMANDES SQLOMEGA

Dans cette section, nous donnons un aperçu des règles d'exécution des commandes SQLOMEGA. Comme le montre la figure 8, cette exécution traverse plusieurs couches :

- l'analyse lexicale et syntaxique,
- l'analyse sémantique,
- la génération et l'optimisation d'un arbre d'opérateurs sur les classes et sur les fonctions,
- l'interprétation de cette arbre.

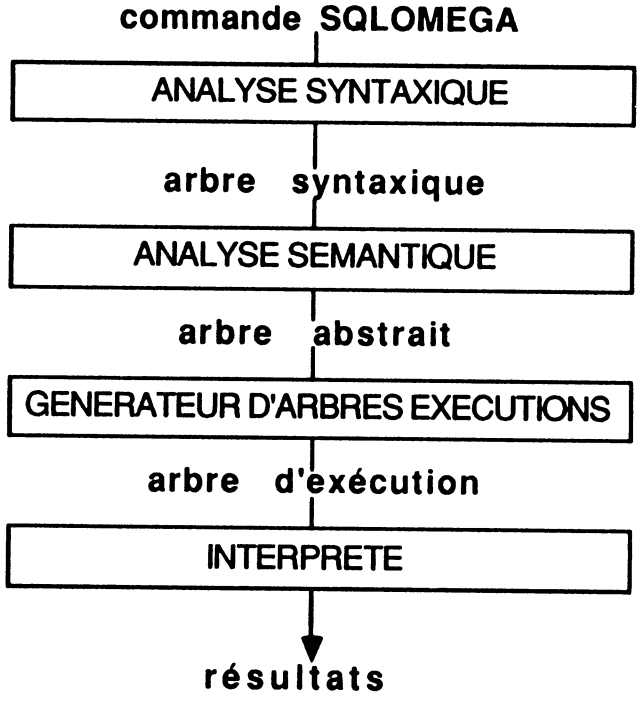


Figure 8 : exécution d'une commande SQLOMEGA

Nous nous intéresserons plus particulièrement à l'évaluation des requêtes notamment en ce qui concerne la génération d'arborescence et son interprétation.

6.5.1 L'analyse lexicale et syntaxique

Les analyseurs du langage SQLOMEGA ont été construits en utilisant les outils classiques d'UNIX : LEX et YACC. Ils vérifient la correction grammaticale des expressions en reprennant la syntaxe du langage défini dans l'annexe 2. L'analyseur syntaxique produit un arbre représentant la commande. Cet arbre prépare déjà les traitements à venir : l'arbre représentant une requête facilite la génération de l'arbre des opérateurs OMEGA en contenant une structure propre aux appels de fonctions. Cette structure isole les fonctions des autres manipulations et met en évidence ses paramètres.

6.5.2 L'analyse sémantique

L'analyse sémantique vérifie si la commande est sémantiquement correcte. Elle vérifie la cohérence au niveau des classes, des objets et des fonctions :

- existence des classes manipulées,
- existence et provenance des attributs cités dans la commande,
- validité des trajets dans le graphe d'objets : vérification que l'enchaînement des pas se traduit réellement par un enchaînement de classes,
- existence des fonctions et compatibilité des types entre les paramètres effectifs et les paramètres formels : cette compatibilité n'est que partiellement vérifiée puisque la liaison n'est réalisée qu'à l'exécution. Par contre, cette compatibilité partielle permet de réduire le domaine d'investigation de l'algorithme de liaison.
- vérification des types dans les expressions de calcul de la clause SELECT et les expressions de condition de la clause WHERE. Le type du résultat des fonctions prises en compte pour ces vérifications est le type du résultat formel.

Pour effectuer ces vérifications, l'analyseur interroge le gestionnaire de classes et de fonctions. Il dispose des fonctions présentées dans la section 6.3.3 :

- existence de la classe : classe(C),
- structure d'une classe : structure(C),
- structure propre : structure_propre(C),
- clé et clé propre : clé(C),
- super-classes d'une classe : superclasse(C).

Cette phase de vérification conduit à mettre à jour l'arbre syntaxique qui

provenait de l'analyse syntaxique. On appelle cet arbre un arbre abstrait. Ensuite, on procède à la décomposition de la commande pour faire apparaître les opérateurs OMEGA.

6.5.3 La génération de l'arbre d'exécution

Cette génération a lieu à partir de l'arbre abstrait. Dans le cas des commandes du LMD, la racine de l'arbre est un noeud dont la nature dépend du type de la commande :

- pour la commande INSERT, un noeud "CREER" préparant l'opérateur sur les objets CREER,
- pour DELETE, un noeud "DETRUIRE" préparant l'opérateur DETRUIRE,
- pour UPDATE, un noeud "CHANGER" préparant l'opérateur CHANGER.

La commande INSERT contient directement tous les paramètres nécessaires à son exécution : les attributs et leur valeur. Certaines valeurs peuvent être des requêtes. Si le cas se présente, le générateur produit un sous-arbre pour chacune puis les relie au noeud racine.

Les commandes DELETE et UPDATE interviennent généralement après une sélection. L'analyse sémantique isole cette sélection sous forme d'une requête. Le sous-arbre des racines des commandes DELETE et UPDATE est donc celui généré par la transformation de leur sous-requête. La commande UPDATE admet aussi des sous-requêtes, elles sont traitées de la même façon que pour la commande INSERT.

La génération d'e l'arbre d'une requête repose sur le principe suivant: l'interprétation de chaque noeud de l'arbre doit produire une classe temporaire (ensemble d'objets homogènes). La classe temporaire produite par le noeud racine fournit l'ensemble des résultats de la requête.

La génération de l'arbre d'exécution d'une requête a lieu en deux étapes :

- séparation entre les opérateurs sur les classes et les appels de fonctions : cette séparation provoque l'éclatement de l'arbre syntaxique. Elle produit des sous-arbres représentant des requêtes SQLOMEGA dépourvues d'appel de fonctions.

- génération d'arbres contenant uniquement des opérateurs sur les classes pour chaque sous-arbre.

6.5.3.1 Séparation des fonctions

La séparation entre les fonctions et les classes est prévue dès l'analyse syntaxique. L'arbre syntaxique est accompagné d'une structure décrivant les fonctions appelées et leurs paramètres. L'arbre syntaxique lui-même contient des entrées directes (sans passer par la racine) vers les noeuds représentant les fonctions.

exemple 7 : séparation entre classes et fonctions

```
SELECT o.numéro, o.type, surface(o.contour)
FROM   objet_planimétrique o , région r
WHERE  r.nom = "Picardie"
AND    surface(r.contour) > 100 * surface(o.contour)
AND    est_dans(o,r)
```

L'analyse sémantique produit la liste (appelée liste LF) suivante :

clause	Fonction	paramètres	type de résultat
select	surface	(o.contour)	réel
where	surface	(r.contour)	réel
where	surface	(o.contour)	réel
where	est_dans	(o, r)	booléen

La génération d'un arbre d'exécution séparant les fonctions des sous-requêtes se développe à partir de ces structures. Cette génération s'accomplit en 6 étapes :

1 - extraction d'une requête sans fonction

On extrait de la requête initiale, sa sous-requête sans fonction en supprimant les expressions contenant des fonctions de la clause SELECT et les conditions contenant des fonctions de la clause WHERE. Cette sous-requête appelée R0 s'exprime, pour l'exemple 7, de la façon suivante :

```
R0 :   SELECT   o.numéro, o.type
       FROM     objet_planimétrique o , région r
       WHERE    r.nom = "Picardie"
```

2 - préparation des paramètres effectifs

On rajoute dans la clause SELECT de la sous-requête, les paramètres

nécessaires aux fonctions (on utilise la liste LF). On obtient une requête R' sans fonction dont l'interprétation fournira les paramètres effectifs des fonctions.

exemple 8 : requête de base pour l'exemple 7

```
R1 :   SELECT   o.numéro, o.type, o.contour, r.contour, o.contour, o , r
       FROM     objet_planimétrique o , région r
       WHERE    r.nom = "Picardie"
```

On suppose que R' sera interprétée à l'aide des opérateurs sur les classes. Elle produira une classe temporaire T'. Dans le cas de l'exemple 7, on appelle T': T1 . Son type est :

[numéro: int, type: char(10), contour: rectangle, contour: rectangle, contour: rectangle, o: objet_planimétrique , r: région]

3- évaluation des fonctions de la clause WHERE

A partir de la classe temporaire T', il faut ajouter les conditions restantes. Pour cela, il faut évaluer les fonctions de la clause WHERE de la requête initiale.

Pour évaluer une fonction, on utilise l'opérateur "APPLY" que l'on définit comme suit :

Soit E un ensemble d'objets homogènes et f une fonction
p1,p2, ... ,pn des noms d'attributs des objets de E

APPLY : E, f, p1,p2, ... ,pn -> K
K = { k = [e, f: f (e.p1,e.p2, ... ,e.pn)] / e in E }

"APPLY" construit un ensemble d'objets K à partir d'un ensemble E en ajoutant un attribut f à chaque objet e de E qui a pour valeur le résultat de l'évaluation de la fonction. L'algorithme exécutant l'opérateur "APPLY" peut être simplement mis en oeuvre en utilisant la construction **for each** introduite au chapitre 5.

```
APPLY :
        K = {}
        for each e in E
          {
           res = eval_function( f , e.p1, e.p2, ... , e.pn)
           K = K u [e, f: res]
          }
```

"eval_function" a déjà été définie au chapitre 5 : c'est la procédure qui effectue la liaison dynamique avec le code de la fonction "f".

Pour calculer toutes les fonctions, il suffit d'enchaîner des opérateurs "APPLY". Cet enchaînement produit finalement une classe temporaire T" .

exemple 8 : arbre d'évaluation des fonctions de l'exemple 7

T2 = APPLY (T1, surface, r.contour)
T3 = APPLY (T2, surface, o.contour)
T4 = APPLY (T3, est_dans, o , r)

4 - évaluation des conditions non traitées dans la sous-requête

Les conditions restantes peuvent maintenant être évaluées en utilisant les opérateurs de classes sur T". Le générateur d'arborescence produit une nouvelle sous-requête R" portant sur T" en reprenant les conditions non interprétées dans la requête initiale :

exemple 9 : requête sur T4 évaluant les dernières conditions

```
R5 :    SELECT   *
        FROM     T4
        WHERE r.surface > 100 * o.surface
        AND    est_dans = true
```

L'interprétation de R" (R5 pour la requête de l'exemple 7) doit produire un ensemble d'objets E (T5 pour R5).

5 - évaluation des fonctions de la clause SELECT

Pour obtenir le résultat définitif de la requête, on doit évaluer les fonctions de la clause SELECT : on génère de nouveau une arborescence d'opérateur APPLY.

exemple 10 : fonction de la clause SELECT

T6 = APPLY(T5, surface, o.contour)

· 6 - restitution des attributs de la clause SELECT

La table temporaire, résultat de la cinquième étape, contient trop d'informations. Il faut la projeter sur les attributs sélectionnés : on génère une nouvelle requête sans fonction qui sera exploitée par les opérateurs d'OMEGA.

exemple 11 : projection de la table T6

```
R7 :    SELECT   numéro, type, surface
        FROM     T6
```

Le processus de séparation entre les classes et les fonctions conduit donc à la génération d'un arbre qui contient deux types de noeuds :
- des noeuds "APPLY" pour l'évaluation de fonction,
- des noeuds requêtes sans fonction pour la manipulation des classes.

Cet arbre peut être optimisé sous plusieurs aspects :

A- les calculs redondants de fonctions peuvent être éliminés. Par exemple, le second calcul de la fonction "surface" sur "o.contour" n'est pas nécessaire puisque la classe T5 contient déjà le premier résultat.

B- l'interprétation des conditions avec fonctions peut être modifiée. L'ordre d'évaluation des fonctions peut être permuté. On peut y insérer des requêtes afin de réduire progressivement la taille des classes temporaires. Pour cela, on doit tenir compte du facteur de sélectivité des fonctions.

C- jusqu à présent, notre arbre n'était en fait qu'une liste. Pour optimiser les calculs de fonctions et réduire la taille des classes temporaires, il est possible de diviser certaines requêtes sans fonction.

exemple 12 : optimisation en utilisant la sélectivité de la fonction "est_dans" et la redondance de "surface"

```
T1 =    SELECT   o.numéro, o.type, o.contour, r.contour, o.contour, o , r
        FROM     objet_planimétrique o , région r
        WHERE    r.nom = "Picardie"
T2 =    APPLY (T1, est_dans, o , r)

T3 =    SELECT   *
        FROM     T2
        WHERE    est_dans = true
```

```
T4 =    APPLY (T3, surface, r.contour)
T5 =    APPLY (T4, surface, o.contour)
T6 =    SELECT    *
        FROM      T5
        WHERE r.surface > 100 * o.surface

T7 =    SELECT    numéro, type, surface
        FROM      T6
```

Dans l'exemple 12, on insère un filtre sur la classe T2 en calculant au plus tôt la fonction "est_dans". La fonction "surface" n'est pas recalculée pour la clause SELECT.

remarque :

on pourrait fusionner les deux dernières requêtes. Ce travail est inutile puisque de toutes façons, elles vont être décomposées en arbres d'opérateurs sur des classes.

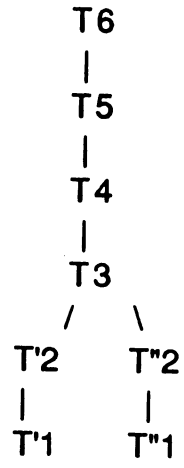
exemple 13 : séparation de la requête R1

La requête R1 peut être divisée en deux pour minimiser les calculs de surface. Dès lors, ce n'est plus la condition "est_dans(o,r)" qui devient la plus discriminante :

```
T'1 =   SELECT    o.numéro, o.type, o.contour, o.contour, o
        FROM      objet_planimétrique  o

T"1 =   SELECT     r.contour, r
        FROM      région r
        WHERE     r.nom = "Picardie"

T'2 =   APPLY (T'1, surface, r.contour)
T"2 =   APPLY (T"1, surface, o.contour)

T3 =    SELECT    *
        FROM      T'2 , T"2
        WHERE     r.surface > 100 * o.surface

T4 =    APPLY (T3, est_dans, o , r)
```

```
T5 =    SELECT    *
        FROM      T3
        WHERE     est_dans = true

T6 =    SELECT    numéro, type, surface
        FROM      T5
```

On obtient bien un arbre de classe temporaire :



Le choix entre les organisations des exemples 12 et 13 doit être fait lors de cette phase de génération d'arborescences. Aucune technique spécifique de décision n'a été étudiée. Il faudrait prendre en compte :

- la sélectivité d'une fonction,
- le temps de calcul de la fonction,
- le temps de calcul des requêtes sans fonction (ceci faisant intervenir la possibilité d'indexation des classes),
- le nombre d'éléments des classes.

6.5.3.2 Utilisation des opérateurs sur les classes

Dans cette section, nous nous intéressons aux requêtes SQLOMEGA sans appel de fonctions. Elles sont produites lors de la première partie de la génération d'arborescences d'une requête, dans les étapes 2,4 et 6 de la séparation classes/fonctions.

La seconde partie de la génération de l'arbre d'exécution consiste à transformer ces requêtes pour que l'interprète puisse les exécuter. Cette transformation repose sur le principe énoncé dans le chapitre 5 : toute requête SQLOMEGA sans fonction peut être exécutée par une suite d'opérateurs OMEGA : sélection, projection, produit, et extraction.

Une première méthode consiste à générer directement la suite

d'opérateurs à partir de la requête. Cette transformation n'est pas complexe. On peut s'inspirer des techniques déjà mises en oeuvre dans d'autres systèmes. Les systèmes relationnels (ORACLE ou SABRE) utilisent les opérateurs de produit, sélection et projection avec cependant l'absence d'opérateur d'extraction. Les systèmes à objets complexes (AIM ou VERSO) effectuent cette transformation avec les opérateurs relationnels plus un opérateur d'imbrication et un opérateur d'extraction (généralement appelés "nest" et "unnest").

Nous avons choisi une autre possibilité liée à notre support de stockage : le SGBD relationnel ORACLE. Notre but est de réaliser rapidement un prototype opérationnel. Pour ce faire, nous devons réutiliser au maximum les possibilités d'ORACLE. Nous utilisons donc les opérateurs du SGBD relationnel pour programmer les opérateurs OMEGA.

La seule entrée disponible sur ORACLE est le langage SQL. Pour réutiliser les opérateurs d'ORACLE, il faut donc entrer des commandes SQL. La transformation des requêtes en opérateurs OMEGA est donc ramenée à la génération d'une requête équivalente dans ORACLE.

Cette tache est simplifiée par les choix de représentations des classes et des objets dans des relations :

- une sélection OMEGA est traduite par une ou plusieurs sélections ORACLE,
- une projection OMEGA est traduite par une ou plusieurs projection sur certaines relations,
- un produit OMEGA est traduit par un produit de relations dans ORACLE,
- l'opérateur d'extraction est mis en oeuvre par des jointures sur les relations contenant les objets référencés. Ces jointures utilisent les identificateurs internes propres aux objets.

exemple 14 : transformation des sous-requêtes de l'exemple 12

```
R1 =    SELECT    o.numéro, o.type, o.contour, r.contour, o.contour, o , r
        FROM      objet_planimétrique o , région r
        WHERE     r.nom = "Picardie"

SELECT A.numéro, A.type, A.contour, R.contour, A.contour,
       A.surrogate, B.surrogate
FROM   objet_planimétrique A , région B
WHERE  B.nom = 'Picardie'

R3 =    SELECT    *
        FROM      T2
        WHERE     est_dans = true
```

```
SELECT A.numéro, A.type, A.o_contour, A.r_contour, A.o_contour,
       A.o_surrogate, A.r_surrogate, A.est_dans
FROM   T2 A
WHERE  A.est_dans = 'true'

R6 =   SELECT   *
       FROM     T5
       WHERE r.surface > 100 * o.surface

SELECT A.numéro, A.type, A.o_contour, A.r_contour, A.o_contour,
       A.o_surrogate, A.r_surrogate, A.est_dans, A.r_surface,
       A.o_surface
FROM   T5 A
WHERE  A.r_surface > 100 * A.o_surface

R7 =   SELECT   numéro, type, surface
       FROM     T6


SELECT A.numéro, A.type, A.o_surface
FROM   T6 A
```

Une fois la transformation des sous-requêtes terminée, l'arbre d'exécution contient deux types de noeuds :

- des noeuds "APPLY" pour exécuter des fonctions,
- des requêtes ORACLE qui exécutent des opérateurs sur des classes.

Le générateur transmet cet arbre à l'interprète.

6.5.4 L'interprète

L'interprète doit exécuter chaque noeud de l'arbre d'exécution. Les types de noeud connus sont :

- CREER : création d'un objet,
- CHANGER : mise à jour d'objets,
- DETRUIRE : suppression d'objets,
- APPLY : exécution de fonctions,
- requête ORACLE : exécution de requêtes ORACLE.

L'interprète exécute les trois premiers types de noeuds à l'aide des opérateurs de même nom définis pour les objets. Les deux derniers types de noeuds sont ceux de l'arbre d'exécution des requêtes.

L'interprète parcourt cet arbre à partir des feuilles. A chaque noeud, il génère une classe temporaire. En cas de branches parallèles dans l'arbre d'exécution (voir exemple 13), l'interprète peut se dupliquer pour optimiser les temps de calcul. L'interprète prend aussi en charge le changement des noms d'attributs en cas de collision. Il propage le nouveau nom dans l'arbre d'exécution.

Lors de l'exécution du noeud APPLY, l'interprète accomplit la liaison dynamique avec le code des fonctions. La procédure de liaison appelée "eval_function" met en oeuvre l'algorithme de recherche de la borne inférieure de l'ensemble des signatures applicables (voir chapitre 4). Cet algorithme est décidable et fini.

Une fois le code déterminé, la procédure "eval_function" lance son évaluation. Plusieurs mécanismes ont été envisagés :

- liaison entièrement dynamique : on peut utiliser les nouveaux outils de SUN UNIX 4.0 (on dépend alors fortement d'une machine). On peut aussi écrire un éditeur de liens dynamique,
- liaison statique à la définition de la fonction : on doit alors regénérer un exécutable d'OMEGA à chaque modification de la classe "FUNCTION",
- liaison statique avec un processus "serveur" de fonctions et appel dynamique du serveur par "eval_function". La modification de la classe "FUNCTION" ne modifie plus le noyau OMEGA mais rajoute ou supprime un service sur le serveur de fonctions.

Nous avons choisi la dernière méthode qui offre l'avantage de conserver la dynamicité de la liaison et qui permet un prototypage relativement rapide. Dans ces conditions, il faut assurer le passage des paramètres et le retour du résultat. Pour cela, nous comptons nous appuyer sur les possibilités du système UNIX à travers les services offerts par XDR et RPC.

Le résultat de l'interprétation d'une requête est une classe temporaire dont les attributs correspondent à la sélection de la requête initiale. Ce résultat est recopié dans un tampon puis envoyé à l'interface interactive pour l'affichage dans une fenêtre de visualisation (voir section 6.7).

6.6 STOCKAGE ET ACCES POUR LES TYPES ABSTRAITS

Nous rappelons que ce sont les descripteurs qui correspondent à la notion de type abstrait : un descripteur est la représentation d'un objet géré extérieurement. Toutes les manipulations sur ces objets sont invoquées via leur descripteur et réalisées par des fonctions.

Les descripteurs sont gérés dans le SGBD OMEGA. Ils se comportent comme des objets de base : ils appartiennent aux objets qui les créent et

ne peuvent pas être partagés. Ils sont créés ou détruits mais ne peuvent pas être mis à jour.

Tout comme une valeur de type abstrait, un descripteur ne peut pas être manipulé directement. Par exemple, ils sont créés par OMEGA lors de l'appel des fonctions de création des valeurs de type abstrait qu'ils représentent (voir exemple 17 et 18).

Les manipulations de types abstraits sont donc plus complexes que les manipulations des objets classiques du modèle ESTRELLA. Ces manipulations nécessitent des fonctionnalités supplémentaires dans certaines couches du noyau OMEGA :

- dans l'analyse sémantique : reconnaissance des types abstraits, recherche de la fonction appropriée, vérification des paramètres, mise à jour de l'arbre syntaxique,
- dans la génération de l'arbre d'exécution : nouvelles dispositions des noeuds,
- dans l'interprète : comportement particulier de certains opérateurs.

Les autres couches ne sont pas modifiées : les analyseurs lexicaux et syntaxiques ne peuvent pas identifier la présence de descripteurs.

Les transformations les plus complexes de l'arbre syntaxique interviennent lors des appels implicites de fonctions : OMEGA doit exécuter une fonction sur un descripteur sans qu'elle soit citée dans la commande. D'autres fonctions sur les descripteurs (les fonctions utilisateurs) sont appelées explicitement dans les commandes. Elles n'entraînent donc pas de grosses modifications de l'arbre syntaxique. Parmi les appels implicites, on distingue cinq cas :

1- l'affichage de descripteurs : attribut de type descripteur dans la clause SELECT d'une requête,

2- la comparaison entre deux descripteurs : utilisation du symbole "=" entre deux attributs de type descripteur, dans la clause WHERE d'une requête,

3- la création d'un descripteur : commande INSERT pour un objet contenant une valeur de classe descripteur,

4- la suppression de descripteurs : commande DELETE d'un objet dont un des attributs est de type descripteur,

5- la suppression puis la création de descripteurs : commande UPDATE portant sur un attribut de type descripteur.

Dans chaque cas, l'analyse sémantique doit transformer l'arbre syntaxique en remplaçant une opération sur un attribut par un appel de fonction. En plus elle doit mettre à jour la liste LF des fonctions contenues

dans la commande.

On rappelle que des fonctions élémentaires doivent être définies avec les classes descripteurs. Elles doivent permettre d'effectuer l'insertion, la suppression, la comparaison et la visualisation des types abstraits.

exemple 15 : définitions de la classe descripteur "rectangle"

```
create class rectangle isa descriptor
begin
  path : char(20) = "/BD/spatial/R+tree",
  identificateur : char(20),
  diagonale : int,
  surface : real
end
```

fonctions élémentaires sur les "rectangles" :

```
insert : char(80)                                  -> rectangle
insert : real x real x real x real                 -> rectangle
insert : real x rectangle                          -> rectangle

delete : rectangle                                 -> nil

egal : rectangle x rectangle                       -> boolean

print : rectangle                                  -> nil
```

autres fonctions :

```
surface :      rectangle                           -> real
diagonale :    rectangle                           -> int
chevauche :    rectangle x rectangle               -> boolean
```

L'exemple 15 montre la définition d'un descripteur de rectangle. Les rectangles sont effectivement stockés sur un support désigné par l'attribut "path" (/BD/spatial/R+tree). L'ensemble des fonctions qui permettent la manipulation des rectangles, comprend des fonctions élementaires et des opérateurs pour les utilisateurs.

Parmi les fonctions élémentaires, on trouve la création, la suppression, la comparaison et l'affichage. La fonction de création "insert" est plusieurs fois rédéfinie, elle corespond à plusieurs méthodes de création de rectangle :

- à partir d'une chaîne de caractères,
- en utilisant les coordonnées réélles de deux points,
- par homothétie sur un rectangle déjà créé.

Les fonctions "surface" et diagonale" sont importantes (bien que non indispensable). Elles permettent d'affecter les autres champs du descripteurs (voir exemple 17 et 18).

6.6.1 Transformation de l'arbre syntaxique des requêtes

La transformation de l'arbre syntaxique des requêtes recoupe les cas 1 et 2. Le cas 1 rejoint le problème de l'affichage. Il sera traité dans la section 7 de ce chapitre. Le second cas intervient exclusivement dans les clauses WHERE des requêtes.

exemple 16 : comparaison entre deux descripteurs

```
SELECT o1.numéro, o1.type, o2.numéro, o2.type
FROM   objet_planimétrique o1, objet_planimétrique o2
WHERE o1.contour = o2.contour
```

Il s'agit d'une modification simple de l'arbre syntaxique : l'analyseur doit isoler la condition portant sur deux descripteurs, remplacer l'opérateur de comparaison par la fonction "égal" (définie obligatoirement sur la signature "descripteur" x "descripteur"), et enfin insérer les deux attributs comme paramètres de cette fonction. L'analyse sémantique doit en plus rajouter une entrée dans la liste LF (liste des fonctions appelées).

exemple 17 : analyse sémantique pour la requête de l'exemple 15

```
SELECT o1.numéro, o1.type, o2.numéro, o2.type
FROM   objet_planimétrique o1, objet_planimétrique o2
WHERE égal(o1.contour, o2.contour)
```

La liste LF issue de l'analyse sémantique :

clause	Fonction	paramètres	type de résultat
where	égal	(o1.contour, o2.contour)	réel

6.6.2 Transformation de l'arbre syntaxique des commandes du LMD

Les cas 3,4 et 5 interviennent dans les commandes du LMD. Le troisième cas (la création des descripteurs) est le plus complexe : il faut d'une part, insérer un descripteur dans la relation ORACLE qui les contient et d'autre

part, exécuter une des fonctions "insert" (il y en a plusieurs à cause des possibilités de surcharge dynamique).

Afin de permettre cette double activité, nous autorisons deux modes de création de descripteurs :

- un mode direct où l'utilisateur entre les valeurs des attributs du descripteur,
- un mode indirect où l'utilisateur donne une liste de valeurs qui seront les paramètres de la fonction "insert".

L'insertion en mode direct est une possibilité contraire au principe des types abstraits qui a été définie pour les besoins du prototype. Nous voulions au plus vite manipuler des données multimédia : il fallait donc implanter rapidement des descripteurs sans pouvoir gérer les appels de fonction. Cette possibilité sera revue lorsque le prototype sera complet.

Actuellement, le choix du mode de création est effectuée lors de la définition de la classe descripteur par une directive supplémentaire en option :

CREATE CLASS <identificateur> ISA DESCRIPTOR [WITH DIRECT INSERT]

Par défaut, le mode de création est indirect. Si on suppose que la classe "rectangle" est en mode indirect alors que la classe "image" est en mode direct, on obtient, pour insérer un cliché, la commande suivante :

exemple 18 : création d'un cliché

```
INSERT INTO cliché
  (
   numéro = 2001,
   qualité = "bonne",
   contour = (200,1.23, 85.65, 0.18),
   photo = (identificateur = "paris_2001" )
  )
```

Dans cette commande, les attributs "contour" et "photo" sont respectivement de type "rectangle" et "image". L'attribut "contour" est affecté par une liste de valeurs correspondant à l'appel de la fonction :

INSERT : real x real x real x real -> rectangle

L'attribut "photo" est affecté, comme un objet imbriqué, par une liste

d'affectations pour les attributs de la structure de la classe "image".

Les deux modes sont implantés différemment :

- l'insertion en mode direct n'utilise pas d'appel de fonctions, il est donc inutile de transformer l'arbre syntaxique,
- l'insertion en mode indirect transforme l'arbre syntaxique pour utiliser la fonction "insert" sur le descripteur. Cette transformation tend à ramener la commande INSERT à une commande du mode direct : les attributs du descripteur sont affectés soit par des valeurs par défaut soit par des fonctions. Un attribut joue un rôle particulier : l'attribut "identificateur", il reçoit le résultat de l'exécution de la fonction "insert". Cet attribut réalise le lien entre la donnée et sa description, il est défini automatiquement à la création de la classe "descripteur", il prend pour valeur une chaîne de caractères.

exemple 19 : l'exemple 18 après l'analyse sémantique

```
INSERT INTO cliché
  (
   numéro = 2001,
   qualité = "bonne",
   contour = (      path = "BD/spatial/R+tree",
                    identificateur = insert(200,1.23, 85.65, 0.18),
                    diagonale = diagonale(200,1.23, 85.65, 0.18),
                    surface = surface (200,1.23, 85.65, 0.18)          ),
   photo = (identificateur = "paris_2001" )
  )
```

La liste LF issue de l'analyse sémantique :

clause	Fonction	paramètres	type de résultat
contour	insert	(200,1.23, 85.65, 0.18)	char(30)
contour	déplac-	(200,1.23, 85.65, 0.18)	int
contour	surface	(200,1.23, 85.65, 0.18)	real

Les deux derniers cas d'appel implicite de fonction interviennent dans les commandes DELETE et UPDATE. La transformation de l'arbre syntaxique de la commande DELETE demande aussi deux actions : il faut d'une part conserver la commande de suppression du descripteur dans ORACLE et d'autre part, rajouter l'appel de la fonction "delete". Cependant, contrairement à la commande INSERT, les attributs du descripteur ne sont pas nécessaires. En effet, suite à la phase de sélection, l'interprète disposera du "surrogate" du descripteur et pourra le supprimer sans

difficulté.

L'analyse sémantique est donc simple : il suffit de rajouter l'appel de la fonction "delete" dans la liste LF avec comme paramètre l'attribut "identificateur" du descripteur.

La liste LF issue de l'analyse sémantique :

clause	Fonction	paramètres	type de résultat
delete	delete	identificateur	char(30)

Nous ne détaillerons pas la transformation de l'arbre syntaxique de la commande UPDATE. Elle correspond à une synthèse entre les transformations opérées pour DELETE puis pour INSERT puisque les mises à jour de descripteurs correspondent rigoureusement à des suppressions suivies d'insertions.

6.6.3 Génération d'arbres et interprétation

Le travail du générateur d'arbres n'est que très peu modifié par la présence de types abstraits dans une commande. Dans la plupart des cas, la transformation de l'arbre syntaxique suffit à gérer cette présence.

L'arbre abstrait des requêtes contient des fonctions supplémentaires. Le générateur d'arbres d'exécution traite ces fonctions sur les descripteurs comme les fonctions classiques du modèle ESTRELLA :

- l'arbre syntaxique de la commande DELETE n'a pas été modifié. Par conséquent le générateur d'arbres d'exécution génère classiquement un noeud "DETRUIRE" et un sous-arbre de sélection.
- les arbres abstraits des commandes INSERT et UPDATE comportent des constructions nouvelles : un attribut peut être affecté directement par une fonction. Cette nouvelle structure est prise en compte par le générateur d'arbres : il génére un noeud racine de type "CREER" et substitue la valeur des attributs descripteurs par un noeud de type "APPLY" pour permettre à l'interprète d'exécuter ces fonctions.

L'interprète reçoit ces différents types d'arbres. Son comportement n'est pas modifié : il exécute chaque noeud et produit des classes temporaires. Par contre les opérateurs sur les objets sont étendus :

- l'opérateur CREER identifie les noeuds "APPLY" et rend la main à l'interprète pour son exécution,
- l'opérateur DETRUIRE lance après chaque suppression de descripteur dans ORACLE, l'évaluation de la fonction "delete" associée,
- l'opérateur CHANGER accomplit les fonctions des opérateurs DETRUIRE

et CREER.

La transformation de l'arbre syntaxique et la génération d'un arbre d'exécution particulier ne concernent pas les fonctions définies par les utilisateurs sur les descripteurs. Celles-ci sont traitées comme n'importe quelle fonction du modèle de données ESTRELLA.

La totalité de ces mécanismes assure le traitement des descripteurs conformément aux principes du modèle ESTRELLA.

6.7 LES INTERFACES AVEC LES UTILISATEURS

Deux types d'interfaces avec les utilisateurs ont été étudiés :

- l'interface programmable pour le langage C,
- l'interface conversationnelle interactive qui utilise le langage SQLOMEGA.

Les principes de ces deux interfaces ont été détaillés dans le chapitre 5. Nous allons nous intéresser à leur implantation.

6.7.1 L'interface avec le langage C

L'interface OMEGA-C a été spécifiée comme le couplage fort entre le langage C et le SGBD OMEGA. La syntaxe et la sémantique du nouveau langage appelé OMEGA-C ne sont pas complètement définies. Par contre les mécanismes que l'on devra utiliser pour les mettre en oeuvre sont déjà connus.

En effet, les applications utilisant l'interface OMEGA-C nous imposent certains critères :

- les applications utilisent OMEGA comme un serveur d'objets sur le réseau de stations de travail,
- les transferts d'objets entre le SGBD et le programme d'application doivent être optimisés,
- les erreurs de programmation doivent être détectées au plus tôt.

Les techniques nécessaires pour respecter ces contraintes ont déjà été étudiées. Le programme écrit en OMEGA-C est précompilé. Les accès aux objets d'OMEGA sont substitués à des appels à un gestionnaire d'objets. Le gestionnaire est un processus chargé de gérer les transferts entre OMEGA et la station de travail. La figure 7 montre le rôle du gestionnaire d'objets dans l'interface OMEGA-C.

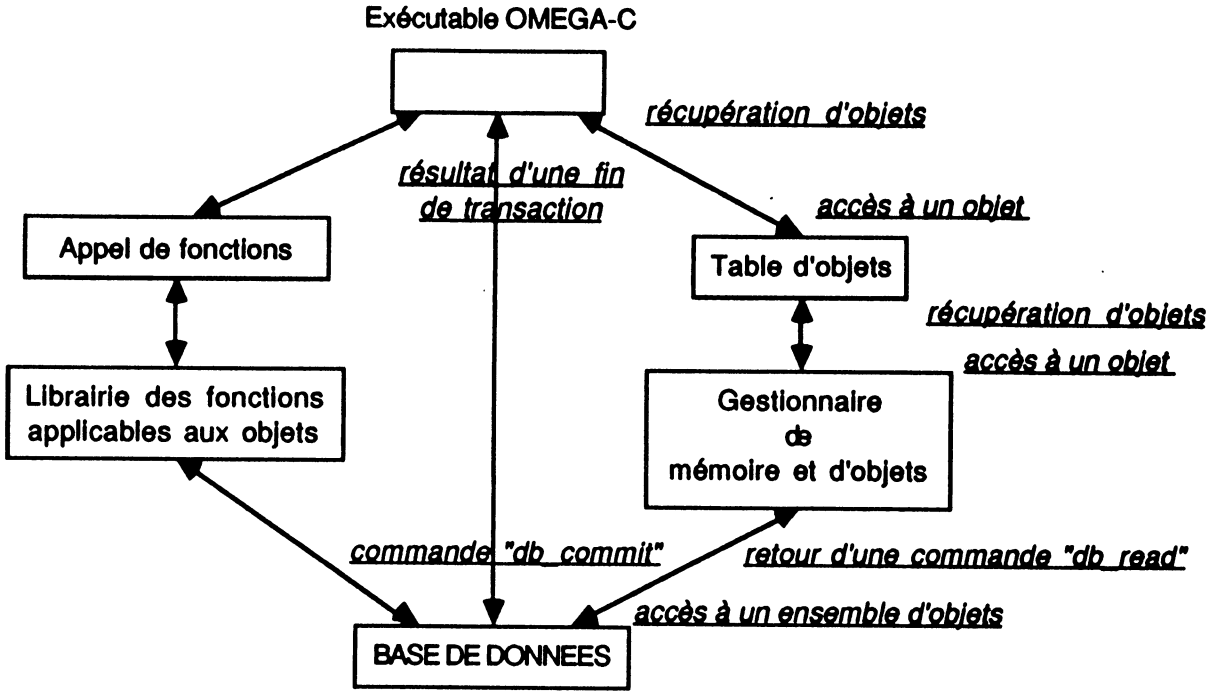


Figure 7 : Exécution du programme C

6.7.2 L'interface interactive

L'interface interactive prévue pour le SGBD OMEGA est une interface graphique (voir chapitre 5.6). Elle doit tenir compte de certaines contraintes systèmes :

- la configuration réseau : le SGBD OMEGA est un serveur SUN 3 dont les clients sont des stations de travail SUN ,
- les possibilités des stations de travail : il faut tenir compte du multi-fenêtrage et de la souris.

D'autre part, les nouvelles fonctionnalités du modèle ESTRELLA doivent être prises en compte par l'interface :

- l'affichage des objets complexes : il s'agit d'afficher le contenu d'un objet dont la structure peut être très profonde,
- l'affichage des valeurs des types abstraits : l'implantation de ces types n'est pas connue du SGBD. L'interface dispose d'une fonction "print" qui sera utilisée pour l'affichage.
- représentation graphique du schéma conceptuel d'une base de données OMEGA : cette représentation est plus complexe pour OMEGA que pour un SGBD relationnel.

Une première étape dans la construction de cette interface graphique a

été franchie avec la réalisation d'une interface interactive de dialogue avec la base de données à partir du langage SQLOMEGA. Cette interface ne permet pas la visualisation graphique du schéma. Elle offre des fonctionnalités équivalentes aux interfaces interactives des SGBD relationnels. Cependant elle prend en compte l'affichage des objets et la visualisation des descripteurs.

Cette interface repose sur trois processus :

- le noyau "OMEGA" qui exécute les commandes sur le serveur,
- le processus "OPUS" qui enregistre les commandes sur la station de travail,
- le processus "VISU" qui affiche des résultats sur une station de travail.

Un utilisateur ouvre une session de travail en activant "OPUS" sur sa machine. "OPUS" établit la communication avec OMEGA, puis transmet les commandes de l'utilisateur :

```
OPUS:
        ouvrir_communication_depuis(station_de_travail)
        lire(commande)
        tantque ( non_vide(commande) )
          compte_rendu = émettre(commande)
          afficher(compte_rendu)
          lire(commande)
        fin_tantque
```

"OMEGA" interprète les commandes et déclenche l'affichage des résultats sur la station de travail. L'affichage utilise une représentation tabulaire. Les résultats de type "objet" ou "descripteur" sont remplacés par des boutons respectivement de type "OBJ" et "DESC".

```
OMEGA(commande, station_de_travail) :
        résultat = exécuter(commande)
        émettre( compte_rendu(résultat))
        déclencher( "VISU" , résultat , station_de_travail )
```

Pour connaître le résultat caché derrière les boutons, l'utilisateur doit activer une fonction du processus "VISU" correspondant au type du bouton. Dans le cas d'un bouton de type "OBJ", la fonction "BOUTON" reçoit l'identificateur interne de l'objet. "BOUTON" active alors la fonction

d'accès aux objets GET puis déclenche un nouveau processus "VISU" avec le résultat.

BOUTON(OBJ, surrogate, station) :

résultat = GET(surrogate)
déclencher("VISU", résultat , station)

La fonction "BOUTON" sur un type descripteur a un autre comportement. Elle reçoit en entrée la valeur de l'attribut "identificateur" de la structure "descripteur". La fonction "BOUTON" effectue en premier le lien avec le code adéquat de la fonction "print" puis déclenche son exécution sur la station de travail.

BOUTON(DESC, ident , station) :

déclencher("eval_function" , (print, ident) , station)

Contrairement au SGBD relationnel dont la seule interface est le langage SQL (ORACLE, INGRES, ...), l'interface interactive ne se limite plus à la simple utilisation du langage SQLOMEGA : les deux fonctions de gestion des boutons dans le processus "VISU" sont des nouvelles entrées pour le SGBD OMEGA.

Ce type de nouvelles fonctionnalités sera développé plus profondément dans l'interface graphique, l'objectif principal d'un tel outil étant d'affranchir les utilisateurs du langage de la base de données.

6.8 CONCLUSION

Le prototype actuel d'OMEGA est disponible sur les SUN 3 disposant du SGBD relationnel ORACLE. Il est mono-utilisateur et met en oeuvre des parties du modèle ESTRELLA :

- gestion des classes : la gestion des valeurs par défaut et les transformations historiques/simples ne sont pas opérationnelles,
- gestion des objets : la commande UPDATE n'est pas implantée,
- gestion du temps : toutes les fonctions de manipulation des historiques n'ont pas été implantées,
- gestion des fonctions : actuellement seule une liaison statique est possible,
- gestion des descripteurs : seule l'utilisation en mode direct est disponible,
- gestion des prédicats : aucun prédicat n'est opérationnel.

La poursuite des développements devra permettre à court terme de rendre entièrement opérationnelle la gestion des fonctions et des descripteurs. D'autres développements sont prévus, ils dépendent des perspectives et des objectifs que l'on choisira pour la suite du projet.

Deux interfaces ont été étudiées. Seule l'interface interactive a été implantée dans la version réduite telle qu'elle a été exposée dans la section précédente. La réalisation de l'interface programmable est une des priorités.

Au cours du développement, les principes d'utilisation d'ORACLE ont été respectés :

- les classes et leur objets sont gérés dans le SGBDR,
- les opérations sur les classes sont exécutées par des requêtes SQL.

Le logiciel est entièrement écrit en langage C. Quatre personnes ont collaborées à sa réalisation :

- Philippe Gomanne (Ingénieur MATRA) a écrit le processus de visualisation (appelé "VISU") : 3 mois,
- Chou Lim (une stagiaire) a réalisé le processus d'interrogation (appelé OPUS) : 2 mois,
- Stephane Raszewski (Ingénieur sous-traitant) a développé le gestionnaire de classes : 2 mois,
- l'auteur de cette thèse a développé du reste du noyau du SGBD OMEGA : 6 mois.

Cela représente environ un an de travail pour un homme.

CHAPITRE 7

CONCLUSION ET PERSPECTIVES

Pour conclure cette thèse, on peut revenir sur les objectifs de départ et faire un bilan des travaux. Notre but initial était d'insérer une base de données dans les applications MS2I en ayant un minimum de désagréments (incompatibilité, incohérence, restructuration...). Cette première demande nous a conduit à l'étude des systèmes commercialisés : nous avons constaté leurs insuffisances. MS2I a alors choisi de spécifier son propre SGBD : c'est ce qui fait l'objet de notre étude.

Parmi les souhaits des applicataires MS2I, nous avons choisi de mettre en valeur les points suivants :

- le modèle du SGBD doit permettre une modélisation simple des applications géographiques. Ce modèle prend en compte les caractéristiques des données manipulées par MS2I :
    - les données sont multimédia : cartes numériques, images satellites, textes volumineux et structurés . . .
    - les données sont de grande taille : par exemple, une image du satellite SPOT panchromatique représente une zone de 60 sur 60 kilomètres et un volume de 36 mégaoctets,
    - les données sont liées à des référentiels du terrain (coordonnées terrestres),
    - les données évoluent dans le temps (les images satellites météo par exemple).
    - les liens sémantiques entre les données sont complexes : les données sont généralement organisées suivant des décompositions hiérachiques (régions, départements, communes). Elles sont liées par des relations de proximités. De plus elles sont multifacettes : par exemple, en reconnaissance aérienne un pont est vu sous son aspect géométrique, alors que dans un système d'aide à la circulation routière, il est vu sous son aspect "débit de véhicules",

- les opérateurs du SGBD doivent regrouper les opérations traditionnelles des SGBD (créations de base de données, mises à jour des objets, accès associatifs) et les opérateurs propres aux domaines d'applications de MS2I : opérateurs géométriques et traitements d'images,

- le SGBD doit fournir des interfaces appropriées à ses utilisateurs. On doit distinguer les utilisateurs interactifs des programmes d'applications.

Le modèle de données ESTRELLA, le langage SQLOMEGA sont les solutions que nous avons essayé d'apporter à MS2I. ESTRELLA est un modèle orienté

objet (notion de classes, objets, fonctions). Il permet une représentation correcte des données et des opérations des applications traditionnelles de MS2I (photo-interprétation, aide au commandement, cartographie, télédétection). Il est ouvert pour la gestion de types multimédia et pour l'intégration d'opérateurs géométriques ou de traitements d'images. ESTRELLA prévoit en plus la gestion de versions temporelles des objets.

Le langage SQLOMEGA permet l'utilisation simple et uniforme de toutes les possibilités du modèle ESTRELLA :

- accès associatif sur les classes,
- accès aux types abstraits (notamment les types multimédia),
- manipulation d'objets complexes,
- requêtes géométriques en utilisant des appels de fonctions.

L'interface OMEGA-C doit compléter cette étude. Elle est en cours de définition mais les principes conducteurs de son élaboration sont établis.

Ces travaux de conception ont été partiellement validés par un prototype opérationnel. Les principales commandes du langage SQLOMEGA ont été implantées. Elles permettent de créer une base de données ESTRELLA, d'ajouter ou de supprimer des objets et d'effectuer des requêtes.

Les autres fonctions de SQLOMEGA (mises à jour du schéma conceptuel et mises à jour des objets) n'ont pas été développées. Le logiciel actuel utilise le SGBD relationnel ORACLE. On peut donner quelques mesures pour évaluer son importance (voir figure 1).

OMEGA	LIGNES	CODE OBJET
Analyseur lexical et syntaxique	2000	83 Ko
Gestionnaire de classes	2600	100 Ko
Analyseur sémantique	4400	100 Ko
Générateur d'arborescence	2500	62 Ko
Interprète	2600	120 ko
Processus OMEGA	**14100**	**500 ko**

Figure 1 : taille du logiciel OMEGA

Outre OMEGA , deux autres processus ont été développés dans le cadre du projet "ESTRELLA". OPUS est une interface conversationnelle entre les utilisateurs interactifs et le SGBD OMEGA. VISU est un utilitaire de gestion de fenêtres pour l'affichage des résultats du SGBD.

Processus OMEGA	14100	500 ko
Processus OPUS	1350	65 ko
Processus VISU	1300	2000 ko
Total du projet ESTRELLA	**16750**	**2500 ko**

Figure 2 : taille du projet ESTRELLA

Un tel développement nous amène une remarque générale sur l'emploi des SGBD Relationnels (notamment d'ORACLE) dans le développement d'un système orienté objet.

Cette démarche offre des avantages théoriques :

- la correspondance entre les structures d'objets complexes et les structures relationnelles a été abondamment étudiée. Tous les écueils de cette démarche sont donc connus,
- les opérateurs du nouveau SGBDOO peuvent être mis en oeuvre simplement à l'aide de ceux du SGBD relationnel,
- les fonctionnalités de persistance, de partage et de sécurité sont déjà opérationnelles.

Ces avantages théoriques ont une contre-partie pratique non négligeable:

- le SGBD ORACLE ne dispose pas d'une interface algébrique (uniquement une interface SQL). La réutilisation de ses opérateurs n'est donc pas facile,
- l'interface ORACLE-C est d'une utilisation délicate. Elle n'est pas adaptée à l'environnement de programmation UNIX.

Dans l'état actuel du projet, on peut s'interroger sur ses perspectives et sur les extensions à apporter au système OMEGA. On peut distinguer deux directions :

- les extensions théoriques visant à compléter les concepts du modèle ESTRELLA et à les justifier plus formellement,

- les extensions pratiques pour améliorer l'utilisation d'OMEGA dans MS2I.

Du point de vue théorique, nous avons pu constater quelques anomalies au cours de la présentation du projet ESTRELLA :

- nous utilisons les constructeurs n-uplet, ensemble et tableau mais leur emploi n'est pas libre : nous pourrions étudier la création de classes d'ensembles de nature imbriquée, ne remettant pas en cause la sémantique du modèle ESTRELLA,
- les chemins multiples dans le treillis ne sont pas autorisés,
- les répercussions des modifications du schéma conceptuel sont peu étudiées,
- les relations entre les valeurs par défaut et le treillis ne sont pas abordées : les dépendances entre les valeurs par défauts calculées sur les classes peuvent conduire à des cycles sans fin (v1 est un valeur calculée à l'aide v2, elle-même calculée grâce à v1). Il faut étudier des méthodes d'évaluation telles que celles proposées dans POSTGRES pour les règles de production [Stonebraker 86b],
- nous définissons empiriquement des opérateurs sur les classes. Il serait intéressant de reprendre les résultats des études faites par d'autres systèmes orientés objets pour définir réellement une algèbre des objets.

Ces extensions sont importantes, cependant elles ne sont pas prioritaires. En effet plusieurs développements doivent d'abord être effectués afin de rendre OMEGA opérationnel :

- le prototype OMEGA doit être consolidé. Toutes les commandes SQLOMEGA doivent devenir opérationnelles. Nous devons mettre au point et développer l'interface OMEGA-C,
- le système OMEGA est actuellement mono-utilisateur. La gestion du partage dans un réseau de stations de travail doit être étudiée,
- aucune évaluation de performance n'a encore pu avoir lieu. Il est certain que l'utilisation d'une couche supérieure au SGBDR constitue un désavantage. Il faut donc étudier les possibilités d'indexation et d'optimisation offertes dans les SGBDR [Valduriez 87].

Ensuite, il est essentiel pour MS2I de diposer d'une version de base d'OMEGA fournissant les fonctionnalités nécessaires à ses activités :

- certains types multimédia doivent être offerts en standard : les types images, cartes, graphiques doivent être gérés par le SGBD et leur opérateurs élémentaires doivent être pré-programmés,
- les opérateurs spatiaux doivent aussi être disponibles bien que leur implantation ne soit pas prédéfinie : elle varie selon le format des

données (raster ou vecteur) et selon les index (quad-tree, R-tree ...).

Il reste donc beaucoup de travail avant qu'OMEGA puisse s'intégrer dans les applications MS2I. Nous sommes néanmoins convaincus que l'approche orienté objet est la meilleure solution pour les problèmes de MS2I.

ANNEXE 1

LES OBJETS PLANIMETRIQUES

Dans cette annexe 1, nous présentons le schéma conceptuel ESTRELLA d'une application : la gestion d'objets planimétriques.

Les utilisateurs cherchent à effectuer des relevés d'objets planimétriques (gare, ville, route, véhicule, ...) à partir de clichés aériens sur des régions (assez vastes) recouvertes par un ensemble de cartes numérisées :

- dans un premier temps, on cherche à situer le cliché par rapport aux régions et aux cartes disponibles. On utilise pour cela des points d'appui naturels visibles sur l'image.

- ensuite, on procède aux relevés des objets dont la description et la localisation seront gérées dans OMEGA.

1) Définition des classes descripteurs

```
create class image isa descriptor
begin
      path : char(20) = "/BD/spot",
      identificateur : char(20)
end;

create class carte isa descriptor
begin
      path : char(20) = "/BD/carte",
      identificateur : char(20)
end;

create class graphique isa descriptor
begin
      path : char(20) = "/BD/gks",
      identificateur : char(20)
end;

create class rectangle isa descriptor
begin
      path : char(20) = "/BD/spatial/R+tree",
      identificateur : char(20),
      diagonale : int,
      surface : real
end;
```

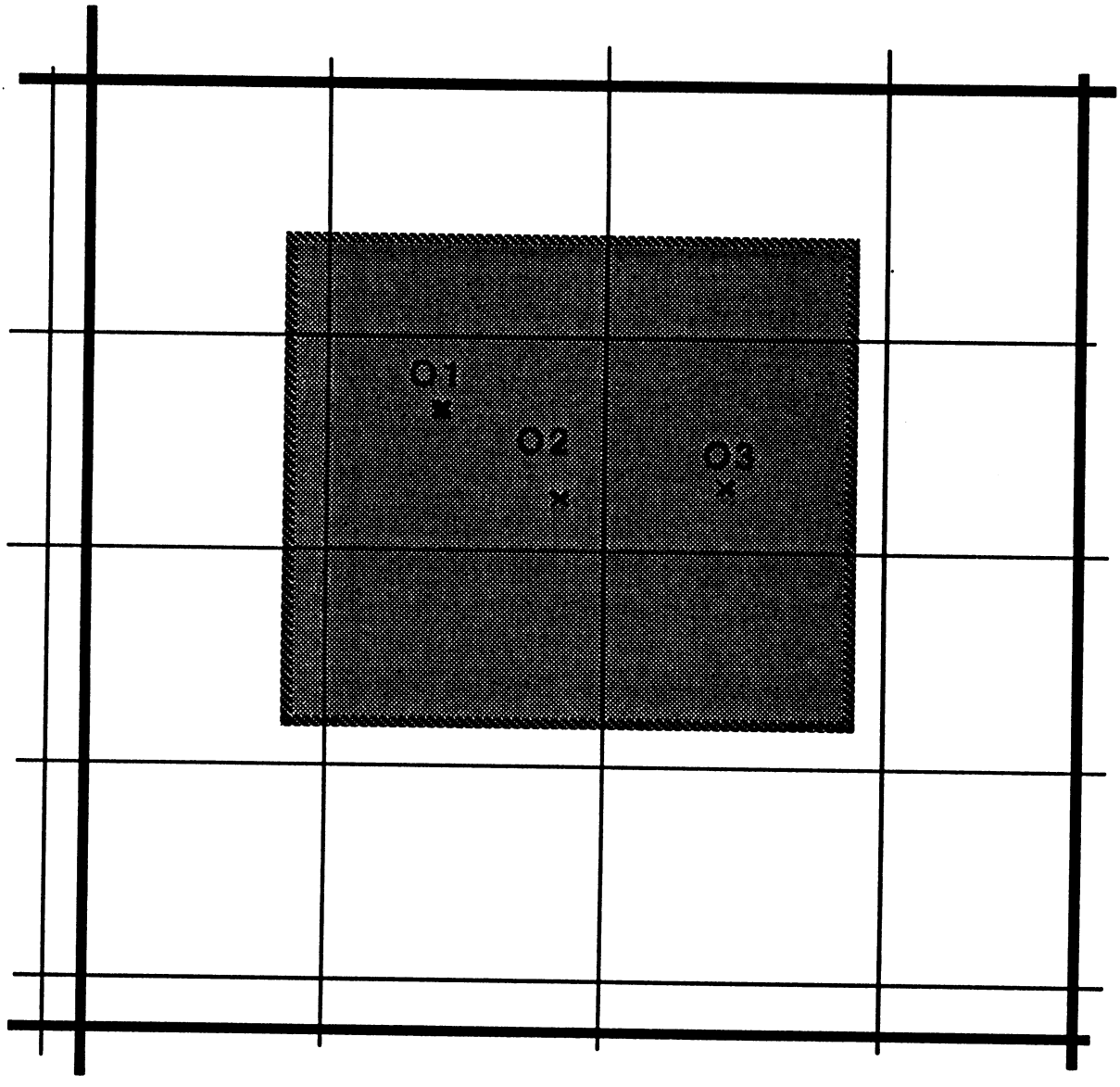
figure 1 : découpage en régions et cartes supperposées

carte au 1:50 000 ème

Région

cliché

O1 objet planimétrique



2) Définition des classes imbriquées

```
create class point isa universal
begin
      x : real,
      y : real,
      z : real
end;
```

3) Définition des classes simples

```
create class objet_planimétrique isa universal
begin
key  numéro          : int,
     hauteur         : real,
     densité         : real,
     matériau        : int,
     contour         : rectangle,
     type            : char(10)
end;

create class région isa universal
begin
key  nom             : char(20),
     capitale        : commune,
     grandes_villes  : array [1..100] of commune,
     contour         : rectangle,
     cartographie    : set of carte
end;
```

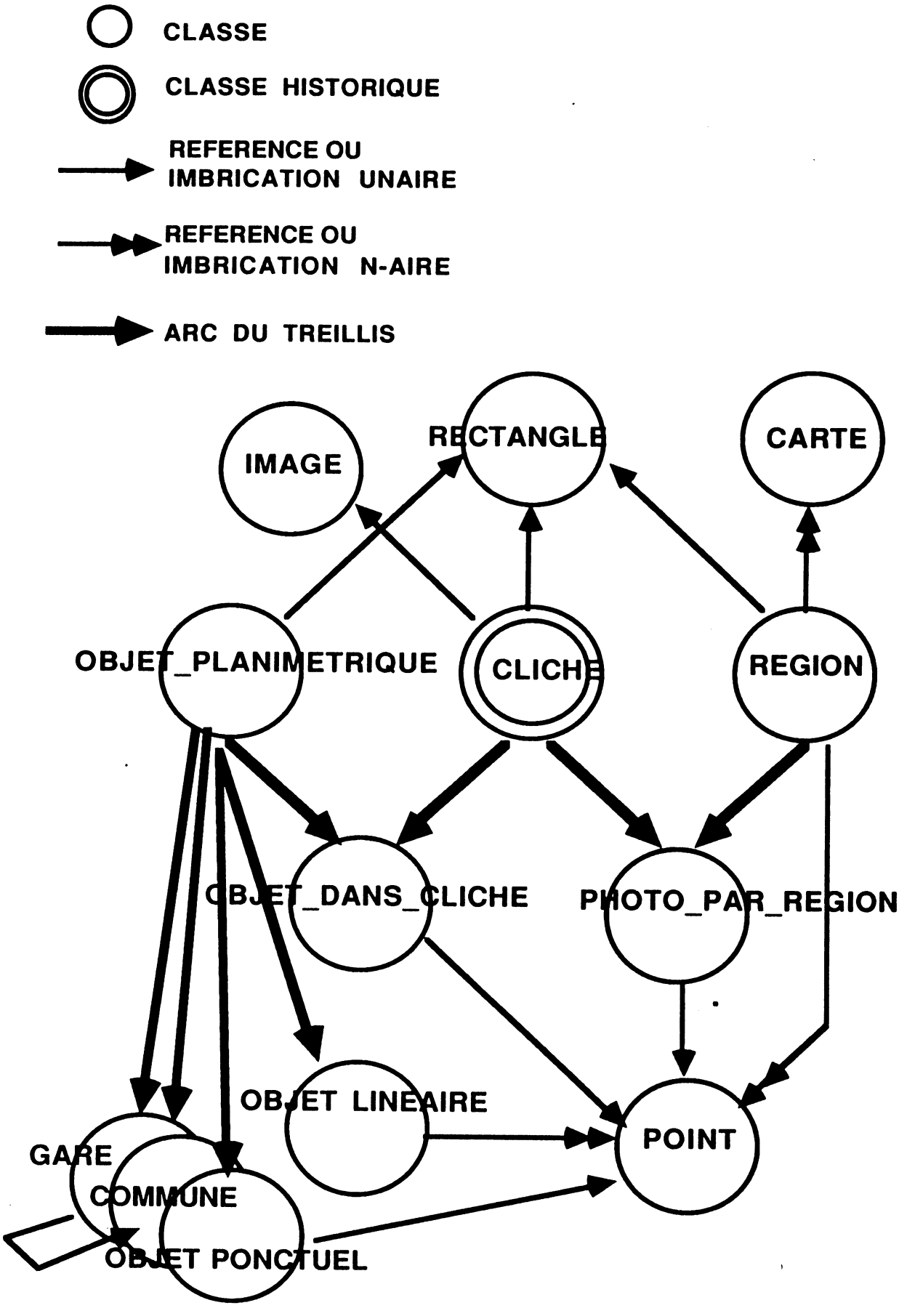
3) Création des classes historiques

```
create class historic cliché isa universal
begin
key  centre          : point,
     contour         : rectangle,
     qualité         : char(10),
     type            : char(15),
     photo           : image,
     date_instance   = date
end;
```

4) Création de sous-classes

```
create class cliché_par_région isa cliché, région
begin
      point_de_recouvrement  : point,
      zone_de_recouvrement   : rectangle,
      orientation            : real,
      cartes                 : set of carte
end;

create class objet_dans_cliché isa objet_planimétrique, cliché
begin
      position          : point,
      contour           : rectangle,
      carte             : carte
end;

create class objet_ponctuel isa objet_planimétrique
begin
      coordonnées                    : point
end;

create class objet_linéaire isa objet_planimétrique
begin
      points                         : set of point
end;

create class commune isa objet_planimétrique
begin
key   nom               : char(20),
      nb_habitant       : int,
      centre            : point,
      rayon             : real,
      plan              : graphique
end;

create class gare isa objet_planimétrique
begin
key   nom               : char(20),
      nb_voies          : int,
      ville             : commune,
      dessin            : graphique
end;
```

figure 2 : diagramme conceptuel



5) Définition de fonctions

A titre d'exemple, on va citer quelques fonctions utilisables. Elles sont insérées dans la classe FUNCTION.

EST_DANS : objet_planimétrique x cliché -> boolean

RECOUVREMENT : région x cliché -> objet_ponctuel

SURFACE : objet_planimétrique -> real

Certaines de ces fonctions sont définies sur les classes descripteurs :

INSERT : char() -> image
INSERT : char() -> carte
INSERT : char() -> graphique
INSERT : char() -> rectangle
INSERT : point x point -> rectangle

DELETE : image -> nil
DELETE : carte -> nil
DELETE : graphique -> nil
DELETE : rectangle -> nil

INTERSECT : graphique x graphique -> graphique
INTERSECT : rectangle x rectangle -> rectangle

CHEVAUCHE : rectangle x rectangle -> boolean

SURFACE : rectangle -> real

On donne des exemples d'insertions dans la classe FUNCTION pour quelques fonctions caractéristiques :

```
insert into function
  (
  nom = "EST_DANS",
  nb_paramètre = 2,
  paramètre = ( "objet_planimétrique", "cliché" ),
  résultat = "boolean"
  )
```

```
insert into function
  (
  nom = "RECOUVREMENT",
  nb_paramètre = 2,
  paramètre = ( "région", "cliché" ),
  résultat = "objet_ponctuel"
  )

insert into function
  (
  nom = "SURFACE",
  nb_paramètre = 1,
  paramètre = ( "objet_planimétrique" ),
  résultat = "real"
  )

insert into function
  (
  nom = "CHEVAUCHE",
  nb_paramètre = 2,
  paramètre = ( "rectangle" , "rectangle" ),
  résultat = "boolean"
  )

insert into function
  (
  nom = "SURFACE",
  nb_paramètre = 1,
  paramètre = ( "rectangle" ),
  résultat = "real"
  )
```

6) Définitions des prédicats

On définit cinq prédicats pour cette application :

P1 : une contrainte domaine sur le type des objets planimétriques

```
NULL [ { o.type / o in OBJET_PLANIMETRIQUE }
        MOINS
          { "gare" , "pont", "commune" } ]
```

P2 : une contrainte domaine sur la qualité des clichés

```
NULL [ { c.qualité / c in CLICHE }
       MOINS
        { "bon", "moyen", "mauvais" } ]
```

P3 : une contrainte sur la taille des objets planimétriques

```
NULL [ { o / o in OBJET_PLANIMETRIQUE
            & o.longueur x o.largeur > surface(o.contour) } ]
```

P4 : une contrainte forçant toutes les gares à être définies le plus précisément possible

```
NULL [ { o / o in OBJET_PLANIMETRIQUE & o.type = "gare" }
       MOINS
       { g / g in GARE }
     ]
```

P5 : une contrainte forçant toutes les communes à être définies le plus précisément possible

```
NULL [ { o / o in OBJET_PLANIMETRIQUE & o.type = "commune" }
       MOINS
       { c / c in COMMUNE }
     ]
```

On doit noter que les prédicats P4 et P5 définissent une partition de la classe objet_planimétrique entre ses sous-classes.

Les prédicats sont en fait des objets de la classe PREDICATE, on peut donner un exemple de création dans cette classe : les calculs ensemblistes sont décrits en terme de requêtes SQLOMEGA.

```
insert into PREDICATE
  (
  nom = "P1",
  creator = "Damier",
  type = "domaine",
  corps = "NULL [ SELECT  o FROM objet_planimétrique o
                  WHERE o.type not in { "gare" , "pont" , "commune" } ]"
  )
```

```
insert into PREDICATE
   (
   nom = "P2",
   creator = "Damier",
   type = "domaine",
   corps = "NULL [ SELECT  c FROM cliché c
                     WHERE c.qualité not in { "bon" , "moyen" , "mauvais" } ]"
   )

insert into PREDICATE
   (
   nom = "P3",
   creator = "Damier",
   type = "intra_classe",
   corps = "NULL [ SELECT  o FROM objet_planimetrique o
                     WHERE o.longueur x o.largeur > surface(o.contour) ]"
   )

insert into PREDICATE
   (
   nom = "P4",
   creator = "Damier",
   type = "inter_classe",
   corps = "NULL [ SELECT  o FROM objet_planimétrique o
                    WHERE o.type = "gare"
                     AND     o not in gare                              ]"
   )

insert into PREDICATE
   (
   nom = "P5",
   creator = "Damier",
   type = "inter_classe",
   corps = "NULL [ SELECT  o FROM objet_planimétrique o
                    WHERE o.type = "commune"
                     AND     o not in commune                           ]"
   )
```

ANNEXE 2

SYNTAXE DANS SQLOMEGA

Dans cette annexe, nous donnons la syntaxe du langage SQLOMEGA. Nous utilisont une grammaire BNF étendu avec les symboles suivants :

[x]	0 ou 1 instance de x
x *	0 ou plusieurs instances de x
x+	1 ou plusieurs instances de x
x | y	x ou y
(x | ... | y)	groupement d'alternatives mutuellement exclusives
MAJUSCULE	symbole terminal
<minuscule>	règle de grammaire
'x'	symbole terminal x
[x . . y]	alternative dans l'intervalle allant de x à y

```
< commande_omega > ::=      <création de classe>
                          | <suppression de classe>
                          | <création de vue>
                          | <suppression de vue>
                          | <création de prédicats>
                          | <suppression de prédicats>
                          | <création de fonction>
                          | <suppression de fonction>
                          | <ajout d'attribut>
                          | <suppresion d'attributs>
                          | <définition d'une valeur par défaut>
                          | <suppresion d'une valeur par défaut>
                          | <transformation simple-historique>
                          | <transfromation historique-simple>
                          | <création d'objet>
                          | <suppresion d'objet>
                          | <mise à jour d'objet>
                          | <requête>
```

---------- CREATION DE CLASSE ----------

```
<création de classe> ::=  CREATE CLASS [HISTORIC] <identificateur>
                          ISA (<identificateur> ',' )* <identificateur>
                          BEGIN
                          (<def_attribut> ',')* <def_attribut>
                          END

<def_attribut>       ::=  DATE_INSTANCE '=' <nom_attribut>
                        | [KEY] <nom_attribut> ':' <type> [NOT NULL]
                          [ '=' <valeur> [STATIC] ]

<nom_attribut>       ::=  (<identificateur> '.' )* <identificateur>

<type>               ::=  [ (SET OF | ARRAY <dimmension> OF ) ]
                          <identificateur> | <classe de base>

<dimmension>         ::=  '[' ( <bornes> ',' )* <bornes> ']'

<bornes>             ::=  <entier> '.' '.' <entier>
```

```
<classe de base>    ::=   BOOLEAN | INT | REAL
                        | CHAR [ '(' <entier sans signe> ')' ]
                        | DATE

<valeur>            ::=   <constante>
                        | '(' ( <affectation> ',' )* <affectation> ')'
                        | '(' ( <valeur> ',' )* <valeur> ')'
                        | <requête>

<affectation>       ::=   <nom_attribut> '=' ( <valeur> | <fonction> )

<fonction>          ::=   <identificateur>
                          '(' ( <parametre> ',' )* <parametre> ')'

<parametre>         ::=   <fonction>
                        | <trajet>
                        | <constante>

<trajet>            ::=   '*'
                        | ( <identificateur> [ <coordonnées> ] '.' )*
                          <identificateur> [ <coordonnées> ]

<coordonnées>       ::=   '[' ( <entier> ',' )* <entier> ']'
```

---------- SUPPRESSION DE CLASSE ----------

```
<suppression de classe> ::=  DROP CLASS <identificateur>
```

---------- CREATION DE VUE ----------

```
<création de vue>   ::=   CREATE VIEW < identificateur>
                          [ "(" (<identificateur> "," )*
                            <identificateur>          ")"      ]
                          AS '(' <requête> ')'
```

---------- SUPPRESSION DE VUE ----------

```
<suppression de vue> ::=   DROP VIEW <identificateur>
```

---------- CREATION DE PREDICAT ----------

<création de prédicat> ::= INSERT INTO PREDICATE
'(' 'nom' '=' <identificateur> ','
'créator' '=' <identificateur> ','
'type' '=' <type de prédicat> ','
'corps' '=' <prédicat>
')'

<type de prédicat> ::= 'domaine' | 'intra_classe' | 'inter_classe'

<prédicat> ::= (NULL | NOT NULL) '[' <requête> ']'

---------- SUPPRESSION DE PREDICAT ----------

<suppression de prédicat> ::= DELETE FROM PREDICATE WHERE
'nom' '=' <identificateur>

---------- CREATION DE FONCTION ----------

<création de fonction> ::= INSERT INTO FUNCTION
'(' 'nom' '=' <identificateur> ','
'nb_paramètre' '=' <entier> ','
'résultat' '=' <identificateur> ','
'paramètre' '=' '(' (<identificateur> ',')*
<identificateur> ')'
')'

---------- SUPPRESSION DE FONCTION ----------

<suppression de fonction> ::= DELETE FROM FUNCTION
WHERE 'nom' '=' <identificateur>
AND 'nb_paramètre' '=' <entier>
AND 'résultat' '=' <identificateur>
(AND 'paramètre' '[' <entier> ']'
'=' <identificateur>)*

---------- AJOUT D'UN ATTRIBUT ----------

<ajout d'un attribut> ::= ALTER CLASS <identificateur>
ADD <nom_attribut> ':' <type> [NOT NULL]
['=' <valeur> [STATIC]]

---------- SUPPRESSION D'UN ATTRIBUT ----------

<suppression d'un attribut>::=ALTER CLASS <identificateur>
DROP <identificateur>

---------- DEFINITION D'UNE VALEUR PAR DEFAUT----------

<définition d'une valeur par défaut> ::= ALTER CLASS <identificateur>
ADD DEFAULT VALUE <identificateur>
"=" <valeur>

---------- SUPPRESSION D'UNE VALEUR PAR DEFAUT----------

<suppression d'une valeur par défaut> ::= ALTER CLASS <identificateur>
DROP DEFAULT VALUE
<identificateur>

--------- TRANSFORMATION DE CLASSE SIMPLE EN HISTORIQUE ---------

<transformation simple-historique> ::= ALTER CLASS <identificateur>
ADD HISTORY
[WITH DATE_INSTANCE
'=' <nom_attribut>]
[FIRST VERSION IS <date>]

--------- TRANSFORMATION DE CLASSE HISTORIQUE EN SIMPLE ---------

<transformation historique-simple> ::= ALTER CLASS <identificateur>
DROP HISTORY
[WITH OBJECT AT <date>]

---------- CREATION D'OBJET ----------

<création d'un objet> ::= INSERT INTO <identificateur>
'(' (<affectation> ',')* <affectation> ')'

---------- SUPPRESSION D'OBJET ----------

<suppression d'un objet> ::= DELETE FROM <identificateur>
[WHERE <clause de condition>]

---------- MISE A JOUR D'OBJET ----------

<mise à jour d'un objet> ::= UPDATE <identificateur>
SET (<affectation> ',')* <affectation>
[WHERE <clause de condition>]

---------- REQUETE ----------

<requête> ::= SELECT (<résultat> ',')* <résultat>
FROM (<classe> ',')* <classe>
[WHERE <clause de condition>]

<résultat> ::= <fonction> | <trajet>

<classe> ::= <identificateur> [<lettre>]

<clause de condition> ::= ([NOT] <condition> <opérateur logique>)*
[NOT] <condition>

<condition> ::= <fonction>
| <terme> <opérateur de comparaison> <terme>
| <terme> <opérateur ensembliste> <ensemble>
| '(' <clause de condition> ')'

<terme> ::= <paramètre>

<ensemble> ::= <trajet> | '(' <requête> ')'

<opérateur logique>::= AND | OR

<opérateur de comparaison>::= '=' | '!=' | '<' | '>' | '<=' | '>='

<opérateur ensembliste> ::= IN | NOT IN

---------- DEFINITIONS GENERALES ----------

<identificateur> ::= <lettre> (<lettre> | <chiffre> | '_')*

<lettre> ::= ['A' . . 'Z"a' . . 'z']

<chiffre> ::= ['0' .. '9']

<constante> ::= <booléen> | <chaîne de caractères>
| <date> | <entier>
| <réel>

<booléen> ::= TRUE | FALSE

<chaîne de caractères> ::= '"' <caractère>* '"'

<caractère> ::= <lettre> | <chiffre> | <signe> | '&' | 'é' | '('
| '"' | '(' | 'ß' | 'è' | '!' | 'ç' | 'à' | ')' | '-' | '_' | '^'
| '$' | 'ù' | '=' | ':' | ';' | ',' | '?' | '.' | '/' | '+' | '%'
| '*' | '"' | 'º' | '¿' | '|' | '[' | ']' | '{' | '}' | '«' | '»'
| '`' | '£' | '@' | '#'

<date> ::= <jour> '/' <mois> '/' <année>

<jour> ::= ['1' . . '31']

<mois> ::= ['1' .. '12']
| JAN | FEV | MAR | AVR | MAI | JUI | JUL | AOU
| SEP | OCT | NOV | DEC

<année> ::= <entier>

<entier sans signe> ::= <chiffre>+

<entier> ::= [<signe>] <entier sans signe>

<réel> ::= <entier> ['.' <entier sans signe>]
['e' <entier>]

<signe> ::= '-' | '+'

ANNEXE 3

BIBLIOGRAPHIE

[Abiteboul 86] VERSO
S.ABITEBOUL, N.BIDOIT
"Non first Normal form relations: an algebra allowing data restructuring"
Journal of Computer and System Science, Décembre 1986

[Abiteboul 87a]
S.ABITEBOUL, S.GRUMBACH
"COL: a logic based language for complex objects"
Workshop on Database programming languages,
Roscoff (France) Septembre 1987

[Abiteboul 87b]
S.ABITEBOUL, S.GRUMBACH
"Bases de données et objets structurés"
TSI, Vol 6, N° 5, 1987

[Adiba 86a]
M.ADIBA, BUI QUANG NGOC
"Aspects historiques dans les bases de données généralisées"
Journées PRC, Giens, Avril 86

[Adiba 86b]
M.ADIBA, BUI QUANG NGOC
"Historical multimedia databases"
Conférence VLDB, Kyoto (Japon), Aout 1986

[Adiba 87a]
M.ADIBA, BUI QUANG NGOC
"Dynamic database snapshots, albums and movies"
Conférence TAIS: Temporal Aspects in Information Systems,
France, Mai 1987

[Adiba 87b]
M.ADIBA, BUI QUANG NGOC, C.COLLET
"Aspects temporels, historiques et dynamiques des bases de données"
TSI, N° Spécial Bases de Données, Nov 1987

[Adiba 88]
M.ADIBA, C.COLLET
"Management of complex objects as dynamic forms"
Conférence VLDB 1988, Los Angeles

[Andrews 87] V-Base
T.ANDREWS, C.HARRIS
"Combining language and database advances in an object-oriented development environment"
OOPSLA 87, Octobre 1987

[Bancilhon 87a]
F.BANCILHON, T.BRIGGS, S.KHOSHAFIAN, P.VALDURIEZ
"FAD: a powerful and simple database language"
VLDB 1987

[Bancilhon 87b] O2
F.BANCILHON, V.BENZAKEN, C.DELOBEL, F.VELEZ
"The O2, V0 Object Manager Interface"
Rapport Technique Altaïr 11-87, Septembre 1987

[Bancilhon 87c] O2
F.BANCILHON, V.BENZAKEN, C.DELOBEL, F.VELEZ
"The O2 Object Manager Architecture"
Rapport Technique Altaïr 14-87, Novembre 87

[Bancilhon 88a] F.BANCILHON
"Object-Oriented Database Systems"
Rapport Technique Altaïr 16-88, Janvier 88

[Bancilhon 88b] F.BANCILHON
"Object-Oriented databases"
Tutorial Conférence EDBT, Venise, Mars 1988

[Bancilhon 88c] F.BANCILHON, G.BARBEDETTE, V.BENZAKEN, C.DELOBEL,
O2 S.GAMERMAN, C.LECLUSE, P.PFEFFER, P.RICHARD, F.VELEZ
"The design and implementation of O2 an object oriented database sytem"
Conference OOPSLA 1988

[Bancilhon 88d] F.BANCILHON, D.MAIER
"Multilanguage Object-Oriented Systems: new answer to old database problems"
Rapport Technique ALTAIR, 21-88, Avril 88

[Banerjee 87a] J.BANERJEE, H.T.CHOU, J.GARZA, W.KIM et al.
ORION "Data model issues for Object Oriented applications"
ACM TOIS, Vol 5 Num 1, Janvier 1987

[Banerjee 87b] J.BANERJEE, W.KIM, H.J. KIM, H.F.KORTH
ORION "Semantics and implementation of schema evolution in Object-Oriented databases"
ACM SIGMOD, 1987

[Batory 88] D.S.BATORY, T.Y.LEUNG, T.E.WISE
GENESIS "Implementation concepts for an extensible data model and data language"
ACM TODS, Vol 13, N° 3, Septembre 1988 pp231-262

[Benzaken 87] V.BENZAKEN, C.DELOBEL, J.B.NDALA
O2 "Gestionnaires de mémoires et d'objets"
IV journées base de données avancées, Benodet 1988

[Bloom 87] T.BLOOM, S.B.ZDONIK
ENCORE "Issues in the design of Object-Oriented database programming languages"
OOPSLA 87, Octobre 1987

[Bui 86] BUI QUANG NGOC
"Aspects dynamiques et gestion du temps dans les SGBD généralisés"
Thèse de Doctorat de l'INPG, Grenoble, Novembre 1986

[Carey 86] M.CAREY, D.DE WITT, D.FRANK, et al.
EXODUS "The architecture of the EXODUS Extensible DBMS, a preliminary report"
Computer Sciences Technical Report 644, Mai 1986

[Carey 88] M.CAREY, D.DE WITT, S.VANDENBERG
EXODUS "A data model and a query language for EXODUS"
ACM SIGMOD conference, Chicago 1988

[Calkins 77] H.CALKINS, R.F. TOMLINSON
"Geographic information systems, methods and equipement for land use planning"
International commission on geographical data sensing and processing, Reston, Virginia 1977

[Calkins 83] H.CALKINS
"A pragmatic approach to GIS design"
International commission on geographical data sensing and processing, New York 1983

[Chang 77] S.K.CHANG, N.DONATO, B.H.Mck CORMICK, J.REUSS, R.ROCHETTI
GRAIN "A relationnal database system for pictures"
IEEE 1977

[Chang 81] N.CHANG, K.FU
IMAID "Picture query language for pictural data-base systems"
IEEE, Vol 0018-9162 Novembre 1981

[Chen 76] P.CHEN
"The entity relationship model - toward a unified view of data"
ACM TODS, vol 1, no 1, 1976

[Christodoulakis 86] S.CHRISTODOULAKIS, F.HO, M.THEODORIDOU
MINOS "The multimedia object presentation manager of MINOS : a symetric approach"
ACM SIGMOD conference, Mai 1986

[Codd 70] E.CODD
"A relationnal model of data for large shared data base"
Communication of the ACM, 1970

[Codd 79] E.CODD
"Extending the database relationnal model to capture more meaning"
ACM TODS, vol 4, no 4, 1979

[Collet 87] C.COLLET
"Formulaires complexes pour bases de données multimédia"
Thèse Nouveau Doctorat, USTMG, Novembre 1987

[Copernique] COPERNIQUE
"Serveur Multimédia"
Documentation technique

[Cowen 88] D.COWEN
"GIS versus CAD versus DBMS: what are the differences ?"
Photogrammetric engineering and Remote sensing, Vol 54, No 11, 1988

[Damier 88a] C.DAMIER, B.DEFUDE
"Un modèle de données pour les informations géographiques"
IV journées base de données avancées, Benodet 1988

[Damier 88b] C.DAMIER, B.DEFUDE
"The document management conponent of a multimedia data model"
ACM SIGIR conference, Grenoble Juin 1988

[Damier 88c] C.DAMIER, E.GOULPEAU
"Base de données multimédia militaire : conception du modèle de données ESTRELLA"
Rapport DRET, lot numéro 1, Décembre 1988

[David 88] B.DAVID
"Le model spatiarel: une extension du model relationnel pour gérer les données spatiales surfaciques"
IV journées base de données avancées, Benodet 1988

[Dayal 86] U.DAYAL, J.SMITH
PROBE
"PROBE: a knowledge base management system on knowledge base management system"
Springer Verlag, p227. 1986

[Delobel 82] C.DELOBEL, M.ADIBA
"Bases de données et systèmes relationnels"
DUNOD informatique, 1982

[Delobel 88] C.DELOBEL, C.LECLUSE, P.RICHARD
O2
"LOOQ : A query language for object-oriented database"
Colloque AFCET, Paris, Décembre 1988

[De Witt 88] D. De WITT
"Extensible Database Systems"
Tutorial, Conférence EDBT, Venise, Mars 1988

[Fauvet 88] M.C.FAUVET
"ETIC: un SGBD pour la CAO dans un environement partagé"
thése de l'université Grenoble 1, Septembre 1988

[Fishman 87] D.FISHMAN, D.BEECH, H.CATE, E.CHOW et al.
IRIS
"IRIS: an Object-Oriented database management system"
ACM TOIS, Vol 5, Num 1, Janvier 1987

[Frank 88] A.FRANK
"Requirement for a database management system for a GIS"
Photogrammetric engineering and Remote sensing, Vol 54, No 11, 1988

[Freston 87] M.FRESTON
"Structures de données multidimensionnelles"
3ème journées Base de données avancées, Port Camargue 1987

[Gallaire 88] H.GALLAIRE
"Knowledge databases"
Tutorial Conférence EDBT, Venise, Mars 1988

[Gamerman 87] S.GAMERMAN, C.LEPENANT, P.RICHARD
"Intégration des langages de programmations et des langages de manipulations de données : approche impérative"
Rapport Technique Altaïr, Avril 1987

[Gardarin 87] G.GARDARIN, E.SIMON
"Les systèmes de gestion de bases de données déductives"
TSI, Vol 6, N° 5, 1987

[Goldberg 83] A.GOLDBERG, D.ROBSON
"Smalltalk 80: the language and its implementation"
addison wesley 1983

[Goulpeau 86] E. GOUPLEAU, P.HOUEIX
"Base de données mulimédia : étude bibliographique préliminaire à l'étude théorique"
document MS2I Val de Reuil, Juin 1986

[Güting 88] R.H.GUTING
"Geo-relationnal algebra: a model and a query language for geometric database"
Conference EDBT, Venise, Mars 1988

[Hornick 87] M.F.HORNICK, S.B.ZDONIK
ENCORE "A shared, segmented memory system for an object-oriented database"
ACM TOOIS, Vol 5, N°1, Janvier 1987

[IEEE 87] IEEE
"Special issue on extensible database systems"
Database engineering, Vol 10, N°2 , Juin 1987

[Khoshafian 86] S.N.KHOSHAFIAN, G.P.COPELAND
"Object identity"
OOPSLA 86, Sepembre 1986

[Kiernan 87] G.KIERNAN, R.LE MAOULT, F.PASQUER
"Support de domaine complexe dans SABRINA : une approche par intégrat d'un interpréteur LISP"
3émé journée base de données avancées, Port Camargue, Juin 1987

[Kim 87] W.KIM, J.BANERJEE, HONG-TAI CHOU, J.GARZA, D.WOELK
ORION "Composite object support in an object-oriented database system"
OOPSLA 87, Octobre 1987

[Kim 88] W.KIM, HONG-TAI CHOU
ORION "Versions of schema for object-oriented databases"
Conférence VLDB, Los Angeles, Aout 1988

[[Krakowiak 87] S.KRAKOWIAK, M.MEYSEMBOURG, M.RIVEILL, C.ROISIN
GUIDE "Modèles d'objets et langage du système GUIDE"
Rapport Guide R2, Novembre 1987

[Laurini 88] R.LAURINI
"Modélisation et manipulation des objets et des connaissances spatiales en géomatique"
Laboratoire d'informatique appliquée, INSA LYON 1988

[Lécluse 87] C.LECLUSE, P.RICHARD, F.VELEZ
O2 "O2, an Object Oriented Data Model"
Rapport Technique Altaïr 10-87, Septembre 1987

[Lécluse 89] C.LECLUSE, P.RICHARD
O2 "The O2 database programming language"
Rapport Technique Altaïr 01-89, Janvier 1989

[Lindsay 87] STARBURST
B.LINDSAY, J.McPHERSON, H.PIRAHESH
"A data management extension architecture"
Conférence ACM SIGMOD 1987

[Linneman 87a] AIM
V.LINNEMAN
"Non Fist Normal Form Relations and recursive queries: an SQL-based approach"
Proc 3rd IEEE International Conference on Data Engineering, Los Angeles, Février 1987

[Linneman 87b] AIM
V.LINNEMAN
"Optimization of recursive queries over nested relations by differential technique"
IBM Heildelberg report TR 87-07005, Juillet 1987

[Lorie 84]
LORIE, MEIER
"Using relationnal DBMS for geographical databases"
Geoprocessing, No 2, 1984

[Mac Keown 86]
D. MAC KEOWN
"Digital cartography and photointerpretation from database viewpoint"
New application of database, ACADEMIC PRESS, 1986

[Maier 86] Gemstone
D.MAIER, J.STEIN, A.OTTIS, A.PURDY
"Development of an Object-oriented DBMS"
OOPSLA 86, Septembre 1986

[Manola 87] PROBE
F.MANOLA, J.ORENSTEIN, U.DAYAL
"Geographical information processing in Probe database system"
8éme symposium internationnal sur la cartographie assisté par ordinateur, Baltimore 1987

[Munoz-Baca 87]
G.MUNOZ-BACA
"Stockage et exploitation de dossiers médicaux multimédia au moyen d'une base de données généralisée"
thèse soutenu à l'USTMG, Grenoble 1987

[Nievergelt 84]
J.NIEVERGELT, H.HINTERBERGER
"The Grid File: an adaptable, symmetric multikey file structure"
ACM TODS, Vol 9, No 1, March 1984

[Oracle 87]
ORACLE FRANCE
"Interface de programmation"
Documentation ORACLE 1987

[Orenstein 86] PROBE
J.ORENSTEIN
"Spatial query processing in an object oriented database system"
ACM SIGMOD 1986

[Paul 87] DASDBS
H.B.PAUL, H.J.SCHEK, M.H.SCHOLL, G.WEIKUM, U.DEPPISCH
"Architecture and implementation of the Darmstadt database kernel system"
SIGMOD 1987

[Penney 87] Gemstone
D.J.PENNEY, J.STEIN
"Class modification in the Gemstone Object-Oriented DBMS"
OOPSLA 87, Octobre 1987

[Peuquet 84] D.J.PEUQUET
"A conceptual framework on comparaison of spatial data models"
Cartographica 1984

[Pistor 86] P.PISTOR, F.ANDERSEN
AIM "Designing a generalized NF2 model with an SQL-type language interface"
VLDB 1986

[Pistor 87] P.PISTOR
AIM "The Advanced Information Management prototype: architecture and language interface overview"
Journées Bases de Données Avancées, Port Camargue, Mai 1987

[Purdy 87] A.PURDY, B.SCHUCHARDT, D.MAIER
Gemstone "Integrating an object server with other worlds"
ACM TOIS, Vol 5, N°1, Janvier 1987

[Rechenmann 87] F.RECHENMANN
"SHIRKA: Système de gestion de bases de connaissances centrées-objets"
Manuel d'utilisation, ARTEMIS, Juin 1987

[Rieu 85] D.RIEU
"Modèle et fonctionnalités d'un SGBD pour les applications CAO"
Thèse de nouveau Doctorat, INPG, Grenoble 1985

[Rogala 85] J.P.ROGALA, J.ZIZAKA
EORID "Définition et réalisation d'une base de données d'images et de télédétection"
Centre scientifique IBM France, PARIS 1985

[Roth 87] M.A.ROTH, H.K.FORTH, D.S.BATORY
SQL/NF "SQL/NF: a query language for N1NF relational databases"
Information Systems, Vol 12, N°1, 1987

[Rowe 87] L.ROWE, M.STONEBRAKER
POSTGRES "The postgres data model"
VLDB brighton 1987

[Sabrina 88] INFOSYS
"Présentation du SGBD SABRINA"
Documentation Infosys, 1988

[Samet 88] H.SAMET
"Hierarchical representation of collections of small rectangles"
ACM computing survey, Vol 20, No 4, Decembre 1988

[Schaller 87] J.SCHALLER
ARC/INFO "The geographic information system ARC/INFO"
EURO CARTO VI, Brno 1987

[Schek 86a] H.J.SCHEK, M.H.SCHOLL
"The relational model with relation-valued attributes"
Information Systems Vol 11, N° 2, 1986

[Schek 86b] DSADBS
H.J.SCHEK, G.WEIKUM
"DASDBS : concepts and architecture of a database system for advanced applications"
rapport technique DVSI-1986-T1

[Scholl 88]
M.SCHOLL
"Manipulation of thematic maps"
Journées de travail, PRC BD3, groupe multimédia, Marseille, Avril 1988

[Scholl 89a]
M.SCHOLL
"VERSO : A database machine based on N1NF relations"
LNCS, Springer Verlag, no 361, 1989

[Scholl 89b]
M.SCHOLL, A.VOISARD
"Modeling of thematic maps: an application to geographic databases"
V journées base de données avancées, Genève, Septembre 1989

[Sellis 87]
T.SELLIS, N.ROUSSOPOULOS, C.FALOUTSOS
"The R+tree: a dynamic index for multi-dimensionnal objects"
13th VLDB, Brighton 1987

[Shipman 81]
D.SHIPMAN
"The functionnal data model and the data language DAPLEX"
ACM TODS vol 6, no 1. March 1981

[Skarra 86] ENCORE
A.H.SKARRA, S.B.ZDONIK
"The management of changing types in an object-oriented database"
OOPSLA 86, Septembre 1986

[Smith 87]
T.SMITH
"Requirements and principles for the implementation and construction of large-scale geographic information system"
Geographical information systems, vol 1, no 1, 1987

[Stonebraker 86a]
M.STONEBRAKER
"Inclusion of new types in database systems"
proc. second internationnal conference on database engineering, Los Angeles feb. 1986

[Stonebraker 86b] POSTGRES
M.STONEBRAKER, L.ROWE (Editors)
"The POSTGRES papers"
Memorandum No. UCB/ERL M86/85, Novembre 1986

[Takao 80] ADM
Y.TAKAO, S.ITOH, J.ISAKA
"An image-oriented database system"
Springer-Verlag 1980

[Tang 81] IDMS
G.TANG
"A management system for an integrated database of picture and alphanumerical data"
Computer graphics and image processing, Vol 16, 1981

[Tsichritzis 87]
D.TSICHRITZIS ed.
"Objects and Things"
Technical Report on Object Oriented System,
Centre Universitaire d'Informatique, Université de Genève, 1987

[Valduriez 85] P.VALDURIEZ, S.KHOSHAFIAN, G.COPELAND
"Implementation techniques for complex objects"
MCC Research Report, 1985

[Valduriez 87] P.VALDURIEZ
"Objets Complexes dans les systèmes de bases de données relationnels"
TSI, Vol 6 N° 5, 1987

[Velez 84] F.VELEZ
"Un modèle et un langage pour les bases de données généralisées. projet Tigre"
Thése de docteur ingénieur de l'INPG, Grenoble 1984

[Verso 86] J.VERSO (nom collectif)
VERSO "VERSO: a database machine based on non 1NF relations"
Rapport de Recherche INRIA N°523, Mai 1986

[Yamaguchi 80] K.YAMAGUCHI, N.OHBO, T.KUNII, H.KITAGAWA
ELF "ELF : extended relational model for large, flexible picture databases"
IEEE, vol CH1530-5 1980

[Zobrist 80] A.ZOBRIST, N.BRAYANT
IBIS "Design an image based information system"
Pictoral information systems edited by Springer-Verlag in 1980

AUTORISATION DE SOUTENANCE

DOCTORAT 3ème CYCLE, DOCTORAT INGENIEUR,
DOCTORAT DE L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

Vu les dispositions de l'Arrêté du **16 avril 1974,**

Vu les dispositions de l'Arrêté du **5 juillet 1984,**

Vu les rapports de M .DELOBEL...Claude.....

M .SCHOLL...Michel.....

M .DANIER...Christophe.....est autorisé(e)
à présenter une thèse en vue de l'obtention du .doctorat de l'Université
..Joseph Fourier - Grenoble I.....

Grenoble, le 23 JUIN 1989

Le Président de l'Université
Joseph Fourier - Grenoble 1

UNIVERSITE JOSEPH FOURIER GRENOBLE

A. NEMOZ