DRIVER : une couche objet virtuelle persistante pour le raisonnement sur les bases de données relationnelles

Franck LEBASTARD

N° d'ordre 93 ISAL 0030 Année 1993

THÈSE

présentée devant

L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

pour obtenir

LE GRADE DE DOCTEUR

Spécialité :

INGÉNIERIE INFORMATIQUE

par

Franck LEBASTARD

DRIVER : UNE COUCHE OBJET VIRTUELLE PERSISTANTE POUR LE RAISONNEMENT SUR LES BASES DE DONNÉES RELATIONNELLES

Soutenue le 26 mars 1993 devant la Commission d'Examen

Jury MM.	**P. PRÉVOT**	Président
	P. BAZEX	Rapporteur
	J. KOULOUMDJIAN	Examinateur
	S. MIRANDA	Rapporteur
	B. NEVEU	Examinateur
	F. RECHENMANN	Rapporteur

À mes parents,

À ma femme Nadine
et ma fille Laura.

Remerciements

Ce travail a été réalisé au sein du projet SECOIA, à l'INRIA Sophia-Antipolis. Le projet SECOIA, qui regroupe des chercheurs de l'INRIA et du Ministère de l'Équipement, fait partie du CERMICS, laboratoire de recherches commun à l'INRIA et à l'École Nationale des Ponts et Chaussées (ENPC).

Je remercie tout particulièrement :

- *Bernard LARROUTUROU*, Directeur du CERMICS, pour m'avoir accueilli dans son laboratoire, et avoir fait preuve à mon égard d'une constante bienveillance, d'une grande patience et d'une non moins grande compréhension;

- *Bertrand NEVEU*, Chef du projet SECOIA, pour la confiance qu'il m'a accordée pendant ces années de travail, pour sa disponibilité exemplaire et sa gentillesse. Je lui suis en particulier très reconnaissant d'avoir toujours accepté de lire à maintes et maintes reprises mes écrits que je lui soumettais;

- *l'INRIA*, pour les moyens extraordinaires qu'il met à la disposition de ses chercheurs. Je n'oublierai jamais la qualité de travail dont j'ai pu bénéficier ici.

Je tiens également à remercier :

- *Patrick PRÉVOT*, Professeur à l'INSA de Lyon, pour l'honneur qu'il me fait à présider le jury de ma thèse. Je le remercie également très chaleureusement pour avoir effectué en mon nom de nombreuses démarches administratives, m'évitant ainsi de multiples déplacements à Lyon;

- *Pierre BAZEX*, Professeur à l'Institut de Recherche en Informatique de Toulouse (IRIT), *Serge MIRANDA*, Professeur à l'université de Nice–Sophia-Antipolis et Directeur du MBDS, et *François RECHENMANN*, Directeur de recherche à l'INRIA et Chef du projet SHERPA, pour avoir accepté, malgré leurs emplois du temps extrêmement chargés, d'être les rapporteurs de mon travail;

- *Jacques KOULOUMDJIAN*, professeur à l'INSA de Lyon, pour avoir accepté de participer au jury;

- *Rose DIENG*, Chef du projet ACACIA à l'INRIA, pour avoir imaginé un joli nom pour mon système —à savoir, l'acronyme DRIVER—, pour la lecture complète qu'elle a bien voulu faire de mon mémoire et pour les remarques avisées qui en ont résulté. Je la remercie aussi (et surtout) pour son amitié et son éternelle bonne humeur;

- *Pierre HAREN*, Directeur Général de la société Ilog, ancien Chef du projet SMECI, pour m'avoir accueilli dans son projet alors que j'étais encore à l'école et pour avoir su me donner le goût de la recherche. Je n'oublierai jamais que je fais aujourd'hui ce métier passionnant grâce à lui;

- *Patrick ALBERT* et *Hakim ERHILI*, de la société Ilog, pour les nombreuses discussions que nous avons eues ensemble et qui se sont toujours révélées très utiles;

- *Jean-Patrick GIACOMETTI*, un de mes collègues préférés, pour sa grande aide à rédiger l'introduction et la conclusion de ma thèse;

- l'équipe du centre de documentation pour sa disponibilité en toutes circonstances, et en particulier *Catherine ALAUZUN*, pour l'énorme travail qu'elle a fait sur ma bibliographie;

- l'équipe du centre de calcul, qui est toujours là quand on a besoin d'elle;

- enfin, l'ensemble des membres des projets SECOIA et ACACIA avec qui j'ai apprécié de travailler et que je regrette de ne pouvoir tous citer.

"— Adieu, dit le renard. Voici mon secret. Il est très simple : on ne voit bien qu'avec le cœur. L'essentiel est invisible pour les yeux."

Antoine de Saint-Exupéry, "*Le Petit Prince*"

Table des matières

Table des figures

INTRODUCTION

L'Intelligence Artificielle (IA) aujourd'hui veut traiter des problèmes si importants qu'elle ne peut plus se passer des services des Bases de Données (BD).

En tant que chercheurs en IA, nous avons ressenti ce besoin d'exploiter, dans nos raisonnements, de grandes bases de connaissances, de taille bien supérieure à celle de nos bases habituelles. Nous nous sommes alors tournés vers les bases de données.

L'idée d'exploiter conjointement les techniques d'IA et BD est un des faits marquants de la recherche en informatique de ces dernières années. D'un côté, les chercheurs en BD se sont intéressés à la façon d'améliorer l'expressivité des données dans les BD (représentation de données implicites, de règles, ...). De l'autre, les chercheurs en IA ont eu de plus en plus besoin de manipuler et de raisonner sur de grandes quantités d'informations. Très naturellement, des travaux des deux domaines ont porté sur des thèmes communs pour finalement former un nouveau champ d'activité, connu sous le nom de *Systèmes à Grandes Bases de Connaissances*, ou *Bases de données expertes*, ou *SGBD déductifs*, ou encore *Systèmes experts persistants*.

Nous nous sommes tout de suite heurtés à une difficulté importante à laquelle se trouve confronté tout utilisateur de Système de Gestion de Bases de Données (SGBD) venant du Génie Logiciel ou des langages de programmation, et qui est connue sous le nom d'"*Impedance Mismatch*" [CM84]. Ce problème d'adaptation est double. D'une part, il tient au fait qu'on exploite élément par élément des données ensemblistes ("record at a time vs data set"); nous ne considérerons pas ici ce point d'achoppement important. D'autre part —et c'est à quoi nous nous limiterons—, il tient à la différence des modèles de données : le modèle de données d'un SGBD est généralement différent de celui de l'application qui l'exploite. Une conversion des données s'avère alors nécessaire à chaque échange d'information.

À l'heure actuelle, deux types de SGBD émergent[1] principalement : les SGBD relationnels et les SGBD orientés objet. Développeurs d'outils d'IA utilisant divers modèles objet, nous nous sommes naturellement d'abord intéressés aux seconds. Le problème de

[1] Sans être en opposition : approche "évolutionnaire" ↔ approche "révolutionnaire" [MVGP92].

l'Impedance Mismatch est apparu plus que jamais. En effet, les SGBD orientés objet actuels ne proposent pas à leurs utilisateurs de décrire le modèle objet qu'ils souhaitent voir géré. Nous avions donc, d'un côté, les modèles objet de nos outils, et de l'autre, celui proposé par le SGBD orienté objet. Dans tous les cas, une conversion était nécessaire pour transférer ou recopier un objet de la base dans l'environnement d'un outil, et inversement.

L'utilisation d'un SGBD relationnel nous a posé les mêmes problèmes. Le modèle relationnel propose en effet des structures de données qui sont également très différentes de celles de nos modèles objet, si bien que de semblables conversions sont également nécessaires.

Après étude de l'existant, nous avons finalement choisi de développer un système nouveau, cumulant de nombreux avantages des SGBD relationnels et des SGBD orientés objet.

Ce système, appelé DRIVER, *définit une couche objet au dessus des bases de données relationnelles.* Il propose des *correspondances* entre des concepts communs à de très nombreux modèles objet et ceux existant dans le modèle relationnel. Ces correspondances générales permettent d'associer des représentations objet à toute base de données relationnelle. En particulier, elles permettent, aux yeux de l'utilisateur, de transformer un SGBD relationnel en SGBD orienté objet.

Ce système se distingue des SGBD objet du marché par deux avantages.

D'abord, l'utilisateur **choisit le modèle objet** qui va être géré. Ainsi, les bases de connaissances auxquelles donne accès DRIVER sont directement visibles et accessibles dans le modèle objet de l'utilisateur. Pour la même raison, les objets de l'utilisateur peuvent très facilement devenir persistants, selon sa convenance.

Ensuite, la seconde originalité de DRIVER est qu'il permet la prise en compte **immédiate** de toute base relationnelle **sous forme objet** : tous les catalogues, toutes les bases de données disponibles sous forme relationnelle sont immédiatement utilisables sous forme de bases de connaissances objet. Et inversement, toute base de connaissances objet construite avec DRIVER peut être tout aussi rapidement proposée aux nombreux utilisateurs de SGBD relationnels. Ce second point fort de DRIVER est d'autant plus intéressant que le modèle relationnel est un standard complètement formalisé et unique, et que les SGBD basés sur ce modèle constituent aujourd'hui la quasi-totalité du marché.

Les difficultés qui se sont posées à nous ont été assez nombreuses.

La principale a été de préserver, dans la définition de nos correspondances, le double rôle de DRIVER, à savoir permettre aussi bien la prise en compte des données relationnelles dans un environnement objet que d'apporter la persistance aux objets de cet environnement. Pour cette raison, les correspondances relationnelles d'un certain nombre de concepts objet n'ont pas été identifiées de façon immédiate, en particulier celles des champs à objet ou encore l'absence de valeur dans les champs de type objet.

Un autre problème important a été posé par l'insertion d'objets dans la base. Ce problème cumulait en fait deux difficultés, la première qu'on peut résumer sous le titre d'"insertion de n-uplets dans une vue", et la seconde, liée au fait que les objets constituent

entre eux un graphe complexe.

Nous verrons plus avant que certaines difficultés n'ont pas encore été surmontées.

En particulier, si nous fournissons un outil efficace de filtrage d'objets *dans la base*, nous n'avons pas aujourd'hui de solution aussi efficace permettant le filtrage *sur l'ensemble de la couche objet virtuelle* (base + environnement) de DRIVER.

Plan de lecture

Cette thèse présente une étude détaillée du modèle DRIVER. Nous nous sommes efforcés de décrire le système de façon progressive de manière à permettre au lecteur de s'informer selon ses besoins. Le généraliste ou l'utilisateur se satisfera de la description générale qui est faite du système chapitre 2 tandis que le spécialiste poursuivra son investigation jusqu'aux études de l'implantation et des algorithmes de communication qui sont faites aux chapitres 3 et 4.

Le chapitre 1 présente l'état de l'art[2] en matière de Grandes Bases de Connaissances. Y sont répertoriés les travaux dans ce domaine, visant, pour certains, à développer le pouvoir expressif des modèles de données traditionnels, et, pour d'autres, à introduire des capacités de raisonnement au sein même des SGBD.

Le second présente les caractéristiques générales de DRIVER et de la couche objet virtuelle et persistante qu'il propose. Les principales fonctionnalités du système sont également décrites et les idées phares qui les sous-tendent, telles celles de *correspondance*, *schéma de correspondances*, *objet jumeau* ou encore *défaut d'objet*, sont introduites.

Le chapitre 3 présente le modèle relationnel, un modèle objet de référence[3], et propose des correspondances entre les deux. Les correspondances relationnelles des différents concepts objet sont considérées les unes après les autres.

Il se termine en présentant le méta-schéma, schéma de correspondances permettant de rendre persistantes les informations système contenues dans les schémas. La description du méta-schéma est une occasion de détailler un certain nombre de classes système de DRIVER.

Le chapitre 4 présente les principaux algorithmes utilisés par DRIVER. Sont notamment détaillés les algorithmes de chargement d'objets, d'insertion, de mise-à-jour et de suppression d'objets dans la base.

En annexe, le lecteur trouvera copie du manuel de référence de DRIVER, où sont présentées toutes les fonctionnalités du logiciel. Leur description est suivie d'un certain nombre d'exemples, dont un exemple de configuration du système pour un modèle objet donné —en l'occurrence celui de SMECI—, un exemple complet de schéma de correspondances et

[2] À paraître également dans [Leb93].

[3] En fait, résultant du regroupement de concepts objet communs à de nombreux modèles objet.

un exemple de session d'utilisation.

Chapitre 1

ÉTAT DE L'ART

Les Systèmes de Gestion de Base de Données (SGBD) sont devenus le cœur des systèmes d'information. Au fur et à mesure que leur puissance s'est développée, les utilisateurs se sont faits de plus en plus nombreux. Provenant de domaines variés, leurs besoins ont rapidement dépassé ceux des traditionnelles applications de gestion. Les nouveaux domaines d'application comme la Conception et Fabrication Assistée par Ordinateur (CFAO), l'Ingénierie Assistée par Ordinateur (IAO), la domotique, le traitement d'images, les systèmes à base d'hypertextes, les systèmes d'analyses géographiques et statistiques, l'informatique militaire temps réel, les systèmes experts en géneral, ont besoin de manipuler et de stocker de grandes quantités d'objets complexes.

Leurs besoins, mal satisfaits par les SGBD classiques, sont à l'origine de travaux de recherche qu'on peut regrouper en deux catégories :

- les études visant à développer le pouvoir expressif des modèles de données traditionnels;
- celles cherchant à introduire des capacités de raisonnement au sein même des SGBD.

1.1 Vers une meilleure expressivité des données

Un certain nombre de travaux cherchent à étendre les capacités d'expressivité du modèle relationnel (cf. §3.1.1) en lui adjoignant de nouveaux concepts. Il est en effet extrêmement restrictif d'imposer à un utilisateur de BD d'exprimer toute information sous la forme de simples n-uplets de valeurs atomiques. Non seulement cette modélisation des données est lourde et maladroite, mais elle entraîne en plus pour les applications des surcoûts en temps liés au traitement de requêtes inutilement complexes.

1.1.1 Extensions statiques du modèle relationnel

Dans le modèle relationnel, l'obligation pour une relation de ne comprendre que des attributs atomiques est imposée par la *première forme normale* (1FN) ou, en d'autres termes, par la logique du premier ordre. Les efforts de nombreux chercheurs visent donc à relâcher cette contrainte pour permettre l'utilisation de structures de données plus puissantes.

Un certain nombre d'extensions au modèle relationnel ont été développées ces dernières années pour permettre la gestion d'objets complexes. Les plus connues sont le modèle NF^2 de Jaeschke et Schek [JS82], le modèle des *relations emboîtées* de Fischer et Thomas [FT83], le modèle d'Abiteboul et Beeri [AB84, AG87, BA87] plus général mais aussi plus récent que les deux précédents, le modèle B-relationnel de Le Thanh [LeT86], le modèle de données Franco-Arménien *FAD* (Franco-Armenian Data model) de Bancilhon, Briggs, Khoshafian et Valduriez [BBKV87, BK89], le modèle *ALGRES* de Ceri, Crespi-Reghizzi,

Lavazza et Zicari [CCRG+88, CCRZ+90].

Généralement, on regroupe tous ces modèles sous la dénomination de *modèles NF^2* (Non-First Normal Form) car la première forme normale y est abandonnée. Dans ces modèles, le type des attributs n'est pas nécessairement atomique, il peut être lui-même une relation. Un attribut de type relation comprend lui-même des attributs qui peuvent être également de type relation, et ainsi de suite. Ces relations NF^2 sont également appelées *relations emboîtées.*

{ DEPARTMENTS }								
Deptno	**Mgrno**	{ **PROJECTS** }				**Budget**	{ **EQUIPMENT** }	
		Projno	**Pname**	< **MEMBERS** >			**Qu**	**Type**
				Empno	**Function**			
314	56194	17	CGA	39582	Leader	320,000	2	3278
				56019	Consultant		3	PC/AT
				69011	Secretary		1	PC
		23	HPAR	58912	Staff			
				90011	Leader			
218	71349	25	LEXI	72227	Staff	440,000	2	3278
				89211	Staff		2	PC/AT
				92100	Leader		1	3179
							1	PC/GA

Figure 1.1 : Un exemple de relation NF2

Un exemple de telle relation est présenté figure 1.1; il est tiré de [PD89]. La relation `DEPARTMENTS` comprend deux attributs de type relation `PROJECTS` et `EQUIPMENT`. `PROJECTS` en comprend lui-même un qui est `MEMBERS`. Les accolades "{"... "}" et les crochets "<"... ">" identifient les relations et les attributs de type relation. Les premières signalent les collections non ordonnées et les seconds, celles où l'ordre des éléments est préservé.

Les relations emboîtées minimisent la redondance de données et permettent un traitement efficace des requêtes puisque les jointures sont réalisées à l'intérieur même des relations.

Enfin, ces modèles sont particulièrement intéressants du fait qu'ils reposent sur des bases théoriques solides. Elles sont récapitulées dans [Lev92].

Trois systèmes majeurs concrétisent actuellement les efforts d'implémentation qui ont été faits dans ce domaine. Ce sont le système POSTGRES réalisé à Berkeley [RS87, SK91], le projet AIM (Advanced Information Management prototype) du centre IBM à Heidelberg [DKA+86, PA86, PT86, PD89], enfin le projet DASDBS (DArmStadt DataBase System) de l'Université de Darmstadt [DPS86, PSS+87, SPS87, SS89]. Le système VERSO de l'INRIA [BRS82, Bid86, SAB+89] a également été l'une des réalisations phares de ce domaine, mais son développement a été interrompu en 1989.

Pour finir, ajoutons que d'autres travaux récents cherchent à étendre la sémantique du modèle relationnel par des approches différentes.

Le modèle B-rel [BV92] reformule les concepts du modèle B-relationnel [LeT86] en préservant le caractère centré valeur du concept de relation tout en étendant le concept de domaine vers le concept de classe d'objets.

Le modèle relationnel V2 [Cod90] étend le modèle relationnel en introduisant un opérateur de jointure récursive et en offrant la possibilité d'utiliser des fonctions (prédéfinies ou définies par l'utilisateur) lors de la manipulation des données.

UniSQL [Kim92], le nouveau SGBD de Won Kim, offre pour sa part un modèle qui réunit les caractéristiques des modèles NF2 et des modèles orientés objet. Développé à la suite du SGBD orienté objet Orion (cf. §1.1.4), ce SGBD relationnel étendu bénéficie ainsi de l'expérience acquise par ses auteurs.

1.1.2 Les modèles de données sémantiques

Dans la plupart des modèles de données traditionnels comme les modèles relationnel, hiérarchique et réseau, l'information est représentée sous la forme de n-uplets de valeurs atomiques, comparables d'une certaine manière à de simples enregistrements dans un fichier. Les structures de données proposées à l'utilisateur y sont très proches de celles utilisées dans la représentation physique. Comme la sémantique des relations qui lient les données n'est pas explicitement précisée, toute description de connaissances se traduit invariablement par la perte de la signification des relations entre les données.

Les modèles de données sémantiques furent développés pour offrir un plus haut niveau d'abstraction dans la représentation des données et des relations qui les lient entre elles [HK87b]. Ils permettent aux concepteurs de BD de structurer les données de façon beaucoup plus naturelle.

Tout en ayant chacun leurs particularités, ces modèles présentent un ensemble de caractéristiques communes. Ils offrent une représentation explicite des objets, des attributs (fonctions) et des relations entre objets, des constructeurs de types, en particulier les constructeurs IS-A pour définir des graphes de types, enfin, des mécanismes permettant de dériver de nouvelles informations dans un schéma donné.

Un *type abstrait* est décrit comme un ensemble d'attributs. Ses instances sont des objets et possèdent de ce fait un identificateur interne. Ainsi, deux objets différents peuvent avoir même valeur. On peut définir des sous-types à un type abstrait au moyen des constructeurs IS-A (relation de *généralisation* du sous-type par le type père). Les objets instances de sous-types héritent des attributs définis pour les types ancêtres. Dans de nombreux modèles, un sous-type peut hériter de plusieurs types pères (héritage multiple).

Il existe deux autres constructeurs de types importants qui sont l'*agrégation* et le *groupement*. Le premier permet de définir un nouveau type qui est un agrégat[1] d'attributs, le second, un ensemble d'éléments de même type. L'agrégat est moins puissant que le type

[1] Produit scalaire

abstrait : on ne peut pas lui définir de sous-type et ses occurrences ne sont pas des objets, car ils n'ont pas d'identificateur interne. Une occurrence d'agrégat n'est identifiable que par sa valeur. Ainsi, deux objets différents peuvent avoir même valeur, pas deux occurrences d'agrégat.

La figure 1.2 présente un exemple de schéma tiré de [HK87b]. Il s'agit du schéma d'une base de voyageurs internationaux décrit dans un modèle sémantique générique spécifié par l'auteur. Ce schéma va nous permettre d'illustrer rapidement l'ensemble des concepts des modèles sémantiques qu'on vient d'évoquer.

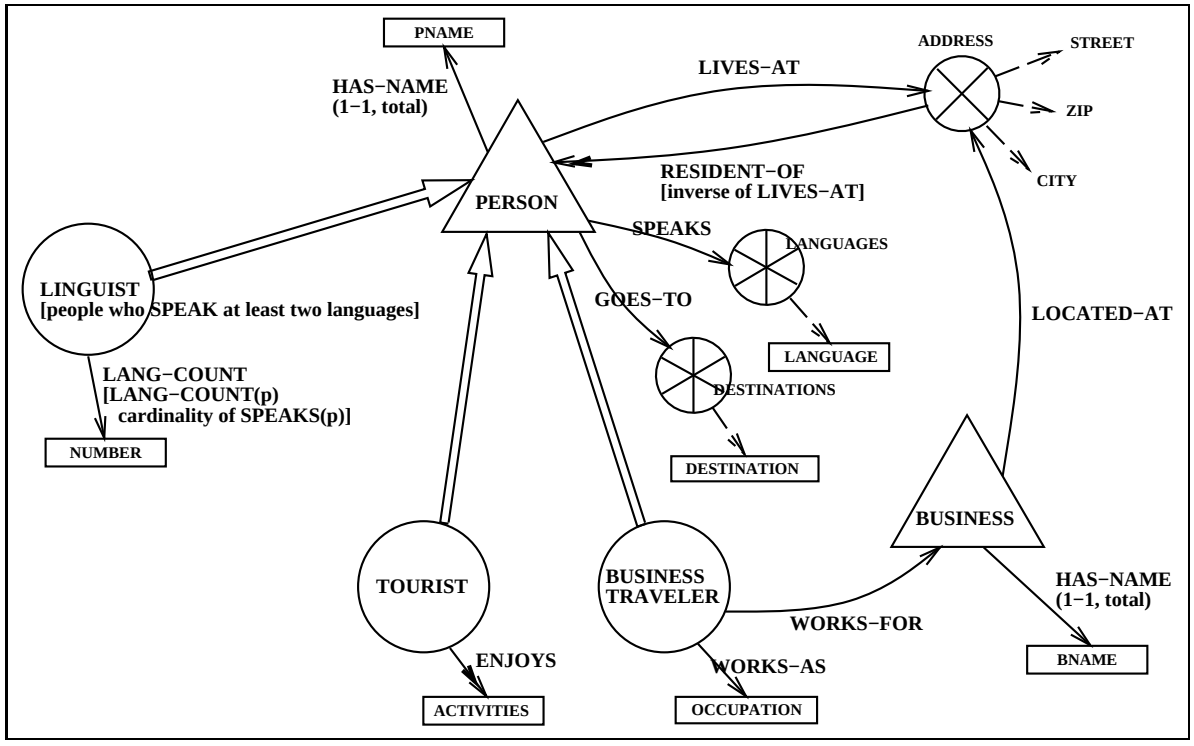

Figure 1.2 : Exemple de schéma décrit dans un modèle sémantique générique

Le schéma comprend deux types abstraits `PERSON` et `BUSINESS`, représentés par des triangles. `PERSON`, par exemple, compte quatre attributs `HAS-NAME`, `LIVES-AT`, `SPEAKS` et `GOES-TO`. `PERSON` a trois sous-types `LINGUIST`, `TOURIST` et `BUSINESS-TRAVELER`, représentés par des cercles. Les relations IS-A qui les lient à `PERSON` sont représentées par des flèches doubles.

Il existe deux catégories de sous-types dans les modèles sémantiques, les sous-types dont l'instanciation est spécifiée par l'utilisateur, et les sous-types dérivés. `TOURIST` et `BUSINESS-TRAVELER` sont de la première catégorie car une personne ne peut être l'un et/ou l'autre que si l'utilisateur de la base l'a explicitement précisé. Par contre, `LINGUIST` est un sous-type dérivé car toute personne parlant au moins deux langues en est automatiquement instance.

Le type ADDRESS de la base est un agrégat de trois attributs STREET, CITY et ZIP[2]. Les types LANGUAGES[3], DESTINATIONS et ACTIVITIES sont des groupements. Dans les schémas, les agrégats sont représentés par des nœuds $\times$ ($\otimes$) et les groupements par des nœuds $\star$ ($\circledast$). Le type ACTIVITIES n'est pas détaillé comme les deux autres. On précise à la place que l'attribut ENJOYS est multi-valué en doublant la tête de la flêche. Les deux représentations sont équivalentes.

Enfin, le schéma comprend un attribut dérivé LANG-COUNT. Cet attribut dérive de l'information (un nombre de langues parlées) à partir de l'attribut SPEAKS défini pour PERSON qui retourne l'ensemble des LANGUAGE parlés par la personne correspondante.

Les modèles de données sémantiques ont trois qualités principales :

- La première est d'accroître la séparation entre les niveaux conceptuel et physique. Illustrons ce point en comparant deux requêtes effectuant le même travail, la première consultant notre base de voyageurs, la seconde une BD relationnelle contenant les "mêmes" données. Nous souhaitons connaître la ville dans laquelle réside l'employeur de Mary. Nous l'obtenons dans une BD sémantique par :

  print LOCATED-AT(WORKS-FOR('Mary')).CITY

  Si notre base relationnelle comprend les deux tables BUSINESS(BNAME, STREET, CITY, ZIP) et EMPLOYEE(ENAME,EMPLOYER,OCCUPATION) qui contiennent les informations respectivement sur les entreprises et les employés de ces entreprises, la requête peut être :

  SELECT CITY
  FROM BUSINESS
  WHERE BNAME **IN**
  SELECT EMPLOYER
  FROM EMPLOYEE
  WHERE ENAME = 'Mary'

  Dans le premier cas, il n'est nullement fait allusion au niveau logique de la base; seuls les attributs (fonctions) sont utilisés pour retrouver l'information. Dans le second, il est nécessaire de "naviguer" dans la base en précisant les tables impliquées et les jointures à utiliser pour passer de l'une à l'autre.

- La seconde qualité de ces modèles est qu'ils permettent de diminuer fortement la surcharge sémantique des différents types de relation. Tout type de relation peut y avoir une représentation distincte, ce qui n'est bien sûr pas le cas dans le modèle relationnel.

[2] Notation : $ADDRESS = STREET \times CITY \times ZIP$
[3] Notation : $LANGUAGES = LANGUAGE*$

- La troisième est de fournir des outils d'abstraction performants pour manipuler les schémas. On peut par exemple, lors de l'édition d'un schéma, préciser le niveau de détail auquel on souhaite le visualiser. Grâce à la forte modularité que ces modèles apportent, on peut aussi très aisément isoler dans un schéma les informations concernant un type donné. Enfin, les mécanismes de dérivation peuvent aider à comprendre un schéma et à mieux l'exploiter en en extrayant encore de l'information qui y était latente, implicite. Ces mécanismes permettent de sélectionner des données, d'effectuer des calculs dessus pour mettre à jour la base ou retourner un résultat.

Trois modèles sémantiques sont particulièrement connus :

- le Modèle Entité-Relation (*ER*);
- le Modèle de Données Fonctionnel (*FDM*, Fonctional Data Model);
- le Modèle de Données Sémantique (*SDM*, Semantic Data Model).

Le modèle Entité-Relation (*ER*), proposé par Chen en 1976 [Che76], est considéré comme l'un des premiers modèles de données sémantiques, même si le terme "sémantique" n'était pas encore utilisé à ce moment-là. Il permet de modéliser le monde sous forme d'entités et de relations entre entités, les deux pouvant être représentés par un ensemble d'attributs mono-valués. Initialement, ce modèle n'avait pas de constructeur IS-A, mais des mécanismes ont été proposés depuis pour les lui ajouter [BLN86, TYF86].

Le Modèle de Données Fonctionnel (*FDM*) a été introduit en 1976 par Kerschberg et Pacheco [KP76]. Il a été le premier modèle sémantique à s'appuyer sur des relations fonctionnelles, c'est-à-dire les attributs. Il est simple et élégant. Il donne aux schémas une représentation graphique facilement compréhensible en faisant ressortir les types atomiques et les attributs au lieu des constructeurs de type, notamment l'agrégation et la généralisation. L'un des avantages principaux de ce modèle est de permettre l'utilisation directe des fonctions lors de la manipulation des données. Il fut le premier à être doté de mécanismes de dérivation [Shi81].

Le Modèle de Données Sémantique (*SDM*), présenté par Hammer et McLeod [HM81], a été l'un des premiers à tirer pleinement parti du constructeur "groupement" et des mécanismes de dérivation. Ces derniers ont par exemple été utilisés pour permettre d'avoir plusieurs perspectives, plusieurs points de vue sur de mêmes données sous-jacentes. SDM est particulièrement réputé pour ses multiples primitives de spécification d'attributs et sous-types dérivés. Il a par exemple quatre catégories de constructeurs IS-A :

- ceux définis par attributs,
- ceux définis par opérations sur d'autres types (union, intersection, ...),
- ceux définis comme domaine d'attribut,
- enfin, ceux spécifiés (ou contrôlés) par l'utilisateur.

Plus récent que ER et FDM, il est plus riche que ses deux prédécesseurs. Il est aussi plus complexe et donc plus difficile à utiliser.

De nombreux autres modèles de données sémantiques existent.

*SAM** (Semantic Association Model) [Su83, Su86] et *IRIS* [DKL85] sont deux autres modèles sémantiques hautement structurés, possédant l'ensemble des caractéristiques décrites dans notre exemple. SAM* est spécialisé dans les formes spéciales de l'agrégation. IRIS s'est particularisé en proposant à la fois constructeurs de types et attributs. Le système IRIS a par la suite évolué pour adopter un modèle orienté-objet [FBC+87].

Le modèle structurel (*SM*, Structural Model) [WEM79], *RM/T*, le développement conceptuel par Codd de son modèle relationnel [Cod79] et *GEM* [Zan83, TZ84] sont des modèles sémantiques définis au-dessus du modèle relationnel. Les systèmes correspondants ont été implémentés comme des couches au-dessus d'un SGBDR.

TAXIS [MBG+87], *GALILEO* [ACO85], *SHM+* (Extended Semantic Hierarchy Model) [BR84] et *INSYDE* [KM85] sont des modèles sémantiques axés sur la gestion dynamique des schémas. Dans ces modèles, les schémas peuvent évoluer en cours de transaction (évolution partagée).

Enfin, *Format* [HY84] et *LDM* (Logical Data Model) [KV84, Kup85] sont des modèles qui ont été développés à des fins de recherches théoriques.

1.1.3 Les modèles de données hyper-sémantiques

Les *modèles de données hyper-sémantiques* [PTE89] constituent un nouveau progrès par rapport aux modèles sémantiques pour représenter de façon pertinente les situations du monde réel. Cette nouvelle classe de modèles étend leurs capacités en leur permettant de gérer des connaissances à l'aide de concepts issus de la recherche en intelligence artificielle, comme l'inférence, le flou, ou les contraintes.

KDM (Knowledge/Data Model) de Potter et Kerschberg [PK86] est un exemple de cette nouvelle classe de modèles. KDM permet de capturer de façon homogène la sémantique des données et celles des connaissances. Données et connaissances y sont modélisées de manière unifiée et le formalisme permettant d'accéder aux unes et aux autres est également identique. Il n'y a pas de séparation données-connaissances.

Cette unité de représentation est rendue possible en dotant le modèle de trois nouveaux constructeurs de type qui s'ajoutent aux constructeurs classiques *généralisation* (relation "est un", IS-A), *classification* (relation "est instance de"), *agrégation* ("est une partie de"), *groupement* ("est membre de"). Ils sont :

- *contrainte.* Ce constructeur permet de placer une restriction sur un aspect d'un objet, une opération ou une relation avec d'autres objets/données. La relation qui lui est associée est "est une contrainte sur".
  Exemple : chaque niveau d'étude doit comprendre un certain nombre de matières, chacune devant être dispensée pendant un nombre d'heures donné;

- *heuristique.* Ce constructeur permet d'attacher un mécanisme de dérivation d'infor-

mations à des objets/données par la relation "est une heuristique sur".
Exemple : Les cours devant être suivis par un étudiant pour une période donnée sont déterminés en fonction du programme d'étude correspondant au niveau de l'étudiant, des cours qu'il a déjà suivis, et des cours qui seront donnés dans cette période.

- *temporel.* Ce constructeur permet de lier des types objet (types abstraits ou sous-types de type abstrait) au travers de relations synchrones ou asynchrones.
  Exemple : La modélisation d'un niveau d'étude doit prendre en compte le fait que les cours doivent être terminés pour que les examens puissent être passés, et que ces examens sont suivis d'un stage, puis d'un travail de rédaction de rapport sur le stage.

Pour finir, signalons que le paradigme représentationnel <Attribut, Objet, Valeur> est utilisé en KDM pour représenter les données, les méta-données, les connaissances et les méta-connaissances dans le modèle. Un avantage de cette représentation unifiée est que ces quatre types d'information sont directement accessibles et contrôlables par l'utilisateur. Sa finalité est de permettre la conception et le développement de *Systèmes de Bases de Données Experts* (EDBS).

1.1.4 Les modèles de données orientés-objet

Ces dernières années, les chercheurs en BD se sont particulièrement intéressés à intégrer les aspects comportemental et dynamique des données dans les formalismes de modélisation. Ce choix a été énormément influencé par le paradigme Orienté-Objet des langages de programmation. Il s'est traduit par l'apparition de *modèles de données orientés-objet.*

Il n'y a pas encore de définition universellement reconnue précisant ce qu'est un modèle orienté-objet. Cependant, un certain nombre de *règles d'or* ont été proposées par un groupe de chercheurs SGBDOO de renom [ABD+89] et doivent être satisfaites pour qu'un SGBD puisse prétendre au label "Orienté-Objet". Ces règles se scindent en deux groupes, l'un précisant ce qu'est un système orienté-objet (les huit premières), l'autre, ce qu'est un SGBD (les cinq dernières) :

- *Le système doit supporter les objets complexes.*
  Un objet complexe est un ensemble structuré de données, formé à partir d'objets plus simples à l'aide de *constructeurs.* Les objets les plus simples sont les entiers, les réels, les chaînes de caractères et autres données atomiques classiques. Les constructeurs sont nombreux, mais trois sont essentiels : l'*ensemble* (ou *Set*, le groupement des modèles sémantiques), la *liste* (ou *list*, l'ensemble ordonné) et le *n-uplet* (ou *tuple*, l'agrégation des modèles sémantiques). Les constructeurs doivent être orthogonaux entre eux, c'est-à-dire que tout constructeur doit pouvoir être appliqué à tout objet dans n'importe quel ordre.

- *Chaque objet doit avoir une identité propre.*
  Comme dans les modèles sémantiques, chaque objet doit être identifié par une référence interne unique. Cette règle permet l'égalité de valeur de deux objets différents.

Elle autorise aussi le partage d'objets et évite de ce fait toute duplication de données. La mise à jour des données s'en trouve simplifiée et plus efficace.

- *Les objets doivent être encapsulés.*
  La manipulation des objets et l'accès aux données qu'ils contiennent doivent être indépendants de leur représentation interne. De ce fait, ces opérations ne doivent être réalisées que par l'intermédiaire d'une interface constituée d'un ensemble de fonctions appelées *méthodes*. L'invocation d'une méthode sur un objet s'effectue par un *envoi de message*. Cet envoi de message précise l'objet concerné, le message c'est-à-dire le nom de la méthode, plus des paramètres éventuels.

- *Le système doit gérer la notion de classe ou de type.*
  La classe et le type sont différenciées par leurs rôles dans le système. La fonction du type est essentiellement de vérifier la correction d'un programme à la compilation alors que celle de la classe est plutôt d'être un outil de création et de manipulation d'objets, "une usine et un entrepôt à objets". Dans certains systèmes, la différence entre les deux est très subtile.

  Classe et type caractérisent un groupe d'objets ayant les mêmes propriétés; ils se décrivent par ces propriétés qui forment une structure et par des méthodes dont certaines sont les accesseurs aux propriétés des objets.

- *Ces classes (ou types) doivent hériter de leurs ancêtres.*
  Il doit être possible de définir une sous-classe à une autre classe précédemment définie. Cette sous-classe est une spécialisation de sa classe mère. Outre ses propres propriétés et méthodes, elle hérite en plus des propriétés et des méthodes de ses classes ancêtres[4].

- *L'appel de méthode doit pouvoir n'être résolu qu'à l'exécution.*
  Des méthodes de même nom peuvent être définies pour des classes différentes, y compris des classes d'une même hiérarchie. De ce fait, quand une fonction qui manipule des objets dont la classe n'est pas connue fait appel à une telle méthode, le système doit déterminer à l'exécution celle qui est concernée. Pour un objet donné, la méthode qui va être appliquée est celle définie pour sa classe si elle existe, sinon celle de sa[5] classe mère, et ainsi de suite.

- *Le système doit être calculatoirement complet*,
  c'est-à-dire permettre l'utilisation de n'importe quelle fonction calculable[6] lors de la manipulation des objets. La complétude est obtenue en offrant une connexion raisonnable avec les langages de programmation existants. De toute façon, l'interface de beaucoup de systèmes consiste en un jeu de primitives qui s'ajoutent à celles d'un tel langage.

[4] Classe mère, classe mère de classe mère et ainsi de suite.
[5] En héritage simple
[6] Au sens mathématique.

- *Le système doit être extensible.*
  Nous avons vu qu'il devait être possible de définir de nouveaux types qui s'ajoutent à ceux qui sont prédéfinis. Il ne doit y avoir aucune différence dans l'usage des types système et des types définis par l'utilisateur. Cette règle doit aussi être vraie pour les opérations sur les types.

- *Le système doit offrir la persistance aux données.*
  Par définition, on qualifie de *persistante* une information qui continue d'exister après l'exécution d'un programme et qui peut être réutilisée lors de nouvelles exécutions [ABC+83]. La persistance doit être orthogonale au modèle, c'est-à-dire qu'un objet doit pouvoir devenir persistant sans qu'il soit nécessaire de le copier explicitement dans la base.

- *Le système doit pouvoir gérer de larges bases de données.*
  La gestion du stockage secondaire est un trait classique des SGBD. Il est en général géré au moyen d'un certain nombre de mécanismes qui sont la gestion d'index, le regroupement de données, le transfert de données en mémoire, la sélection de chemins d'accès et l'optimisation de requête. Ces mécanismes doivent être totalement cachés pour l'utilisateur, de façon à assurer l'indépendance entre les niveaux logique et physique.

- *Le système doit permettre les accès concurrents.*

- *Il doit gérer les pannes.*
  En cas de panne matérielle ou logicielle, le système doit être capable de restaurer la base dans un état cohérent.

- Enfin, *le système doit fournir des fonctionnalités permettant d'interroger la base.*

Les huit premières règles définissent la base de tout modèle objet. La plupart des modèles objet se différencient les uns des autres parce qu'ils sont proposés au travers de langages de programmation différents et/ou parce qu'ils offrent des fonctionnalités supplémentaires par rapport au modèle minimal.

Par exemple, l'héritage peut être multiple, le modèle objet peut permettre la pose de contraintes sur les classes ou les propriétés, etc. Dans de nombreux modèles objet, les classes et les propriétés sont elles-mêmes des objets instances de leurs propres classes[7]. Ainsi, il est souvent possible d'enrichir de tels modèles par de nouvelles fonctionnalités en définissant de nouvelles propriétés ou de nouvelles sous-classes aux méta-classes et aux classes de propriétés. Dans les modèles objet modernes, tout est objet.

Un exemple de modèle objet –celui géré par DRIVER– est décrit en §3.1.2 et des exemples d'objets et de hiérarchies de classes sont présentés figures 3.3 et 3.4.

Les cinq dernières règles définissent un SGBD. Beaucoup de SGBD Orienté-Objet proposent un certain nombre de caractéristiques supplémentaires qui les rendent encore plus attrayants. Elles sont principalement la vérification et l'inférence de type, la distribution,

[7] Une classe de classe est une *méta-classe*.

des systèmes de transaction évolués et la gestion des versions d'objet.

On pourrait penser que les modèles objet ont une approche peu différente de celle des modèles sémantiques. C'est vrai en ce sens qu'ils fournissent tous deux des mécanismes pour construire des structures de données complexes avec relations entre objets. C'est inexact si on considère l'aspect comportemental des données. Dans les modèles objet, les classes comprennent les méthodes qui permettent d'effectuer des calculs complexes sur les données ou de définir le comportement des objets (comme graphiquement à l'écran, par exemple). Les méthodes ont un pouvoir d'expression incomparable par rapport aux attributs dérivés des modèles sémantiques.

Enfin, si les modèles de données sémantiques proposent des structures de données complexes particulièrement adaptées aux applications commerciales, ces structures sont insuffisantes pour les domaines d'application nouveaux où les types complexes tels les types multi-médias (texte, image, son) ou les types utilisateur très gourmands en occupation disque, sont requis [HK87a, SKL88]. Les nouvelles applications sont généralement interactives et requièrent des SGBD hautement dynamiques. Leurs utilisateurs ont besoin de contrôler le comportement des objets et de modifier dynamiquement les classes en leur rajoutant par exemple de nouvelles propriétés ou de nouvelles méthodes.

De très nombreux SGBD orienté-objet sont développés, même si peu ont pour l'instant atteint la phase de commercialisation. Ceux qui sont présentés ci-après ont été choisis parce qu'ils ont été l'objet ou à l'origine de très nombreuses publications.

GEMSTONE [CM84, MS86, MS87] a été le premier SGBD objet commercialisé et est de ce fait très populaire. Il est une extension du langage de programmation Smalltalk, duquel il adopte d'ailleurs le modèle objet. Il offre ainsi la persistance à ses objets et l'enrichit d'un langage de requêtes nommé OPAL qui permet un accès associatif des données.

ORION [KBB+87b, KBB+87a, BCG+87, BKKK87, KBC+88, Kim90] est un prototype de SGBD objet développé aux États-Unis. Il est implanté en Common Lisp et ses primitives s'ajoutent donc aux fonctions de ce langage. Les méthodes de classes doivent aussi être exprimées en LISP.

Il propose une hiérarchie de classes prédéfinies dont la racine est la classe `Object` (figure 1.3). `Object` a trois sous-classes qui sont `PrimitivType`, `Class` et `Collection`. `PrimitivType` est la classe mère de tous les types atomiques de base de ORION, à savoir entier, réel, chaîne de caractères, booléen, etc... `Class` est une méta-classe dont les méthodes permettent de créer et manipuler de nouvelles classes. Ses instances sont donc des objets classe. `Collection` est la classe des groupes[8] d'objets. Elle a pour sous-classe `Set`, classe des ensembles[9] d'objets. Les classes que définit l'utilisateur, `Employee`, `Salesman` et `Manager` dans la figure, ont pour ancêtre prédéfini `Object`. ORION génère automatiquement les classes ensemble sous-classes de `Set` qui leur correspondent et qui sont ainsi disponibles pour la définition des propriétés. L'héritage multiple est autorisé.

ORION est particulièrement efficace dans la gestion dynamique des schémas. Il per-

[8] Ensembles où un objet peut apparaître plusieurs fois.

[9] Au sens mathématique du terme.

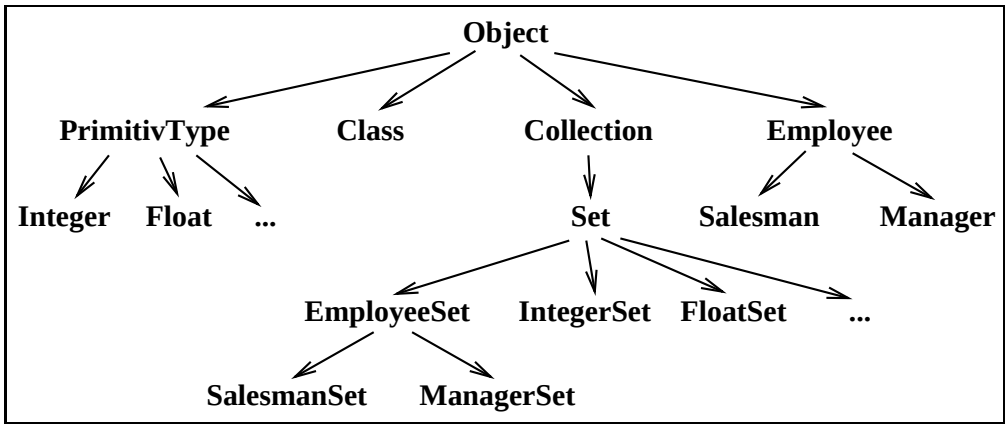

Figure 1.3 : Hiérarchie de classes sous ORION

met à tout moment les mises à jour de schéma qui sont aussitôt visibles des utilisateurs concurrents de la base. Ces mises-à-jour peuvent affecter les classes, les propriétés, les méthodes et le graphe d'héritage.

Ce SGBD est également réputé pour sa gestion efficace des objets, des versions d'objet, des transactions et des données multi-média.

O2 [BDK92] est un SGBD orienté-objet conçu et réalisé en France à l'INRIA et aujourd'hui commercialisé par la société O2 Technology. Il est implanté en C et C++. Ses caractéristiques générales, notamment son modèle objet, vérifient assez fidèlement les règles énoncées précédemment. O2 gère aussi l'héritage multiple, la vérification et l'inférence de type, la distribution, les transactions longues et les transactions emboîtées, les versions d'objet et les versions de BD. Par contre, il ne propose pas de gestion dynamique de schéma. Implanté en C et C++, il s'est aussi ouvert à d'autres langages comme Basic et Lisp (BasicO2 et LispO2).

ONTOS [Ont91] est un SGBD objet écrit en C++. Il est une réécriture du SGBD objet *VBASE* [Atw85, Ont86, AH87, Ont88] initialement écrit en C, dont la commercialisation avait été abandonnée suite à son manque de succès. Il s'intègre à C++ (v2.x) dont il rend persistant le modèle objet. Un des points forts d'ONTOS est son auto-descriptivité : le méta-schéma, le schéma et les données ont une représentation objet uniforme et l'interface d'accès à tous ces objets est également la même. Le fait de permettre à l'utilisateur du SGBD de manipuler les objets du schéma, par exemple les classes, rend possible une gestion dynamique de schéma très puissante. ONTOS gère également l'héritage multiple, la distribution, les transactions longues et les transactions emboîtées. Apparemment, il ne gère pas les versions d'objets.

IRIS [FBC+87] est un SGBD qui était initialement basé sur le modèle de données fonctionnel (cf. §1.1.2) et qui s'est ensuite doté des caractéristiques d'un modèle orienté-objet. Il vérifie toutes les règles énoncées définissant un SGBDOO tout en continuant

de bénéficier de son passé comme modèle sémantique. Une propriété est une fonction, une classe est un ensemble de fonctions, etc... En fait, le schéma apparaît comme un ensemble de types abstraits. La méta-base étant intégrée dans le graphe des types, la gestion dynamique des schémas est autorisée. IRIS gère aussi l'héritage multiple, l'intégrité référenciel et le contrôle de cardinalité, les versions d'objets. IRIS est écrit en C.

MATISSE [MAT92] est le SGBD objet successeur de *G-BASE* [Gra86a, Gra86b]. Comme lui, il ne vérifie que partiellement les règles précédemment énoncées. Ainsi, le seul constructeur d'objets existant est la *liste*. Le constructeur *ensemble* n'est pas supporté, ce qui est très contraignant. D'autre part, la persistance n'est pas orthogonale au modèle objet. Il n'est pas possible dans un programme de travailler sur des objets non persistants de même structure que les objets persistants. Ainsi, apporter la persistance à un objet qui ne l'est pas ne peut se faire que par recopie. Une qualité essentielle de MATISSE est sa gestion dynamique de schéma, rendue possible par le fait que la méta-base est décrite dans le modèle objet du SGBD. Une autre, originale, réside dans le fait qu'il est possible de poser toute sorte de contraintes, par exemple sur les attributs de classe, sur les liens entre objets (cardinalité,...). Leur expression peut être librement fixée par l'utilisateur.

De très nombreux autres SGBD orienté-objet ont été ou sont réalisés. Parmi eux, on peut citer *OM* [Mis84], *POMS* [CAC+84], *CPOMS* [BC85], *EXODUS* [CDRS86, CDF+86, CDV88], *Trellis/Owl* [SCB+86, OBS86], *PROBE* [MD86, Man87, DDGO87], *ObServer* [HZ87], *Polymnia* [But87], *VISION* [Car87], *MONADS* [RK87, Ros90], *MUSHROOM* [WWH87, WW90], *Mneme* [EMS88, ME89], *ODE* [AG89a, AG89b], *GEODE* [Puc89, PT89], *Cricket* [SZ90], *PAM* [KSD+90], *Objectivity* [LG91, MW91], *ObjectStore* [LLOW70], *NICE-C++* [MM91].

On peut préciser que comme Ontos, ObjectStore, Objectivity et NICE-C++ sont des C++ persistants.

1.2 Ajout de capacités de raisonnement aux SGBDR

De nombreux travaux ont pour objet de pourvoir le modèle relationnel de capacités de raisonnement. Leur base commune consiste à interpréter une base de données relationnelle comme une *théorie du premier ordre* [Nic79, GMN84, Rei84]. Dans une telle interprétation, les n-uplets d'une relation sont vus comme des *axiomes*, c'est-à-dire des formules logiques dont on pose la valeur de vérité à "vrai" (n-uplet = "proposition" vraie).

En fait, ces axiomes sont plus précisément des *formules atomiques instanciées*, car ils sont constitués d'un unique prédicat (ayant le nom d'une relation de la base !) n-aire, appliqué à un ensemble d'arguments ne comportant pas de variable libre.

On peut donc enrichir la théorie en lui ajoutant des axiomes plus complexes, par exemple des formules formées de phrases de prédicats pouvant comprendre en argument des variables libres. Ces axiomes rendent possible l'expression de *connaissances*, et par

voie de conséquence, la déduction de nouveaux faits appelés *théorèmes*. Par hypothèse, tout fait non démontrable est faux (négation par échec).

Ces axiomes peuvent être présentées sous forme de clauses en éliminant les quantificateurs des formules. En général (cf. plus loin), les clauses autorisées par les systèmes sont les *clauses de Horn*, de la forme $P1 \wedge P2 \wedge P3 \wedge \ldots \wedge Pn \rightarrow Q$. Ce sont des clauses formées d'une disjonction de prédicats avec un unique prédicat positif (Q) et 0 à N prédicats négatifs :

$$\{P1 \wedge P2 \wedge P3 \wedge \ldots \wedge Pn \rightarrow Q\} \Leftrightarrow \{Q \vee \neg P1 \vee \neg P2 \vee \ldots \neg Pn\}$$

Les clauses de Horn sont aussi appelées *règles logiques*. Par convention et pour en améliorer la lisibilité, les règles sont présentées "dans l'autre sens", la virgule symbolisant le connecteur "et" à la place de $\wedge$:

$$Q \leftarrow P1, P2, \ldots, Pn \quad \text{se lit} \quad \text{“}Q \text{ si } P1, P2, \ldots, Pn\text{”}.$$

Q est la tête de règle et $P1, P2, \ldots, Pn$ le corps.

Dans la base de données, Q est interprété comme une *relation dérivée*, encore appelée *relation déduite*, définie au moyen des Pi, relations de la base ou autres relations dérivées. Pour éviter les cycles lors du calcul des relations dérivées, on interdit dans toute règle que Q corresponde à une relation de la base. Par définition, l'ensemble des relations dérivées forment la *base en compréhension* (intentional database) tandis que l'ensemble des faits contenus dans les relations implantées constituent la *base en extension* (extensional database).

Prenons un exemple. Considérons les relations `parent`, `homme` et `femme` suivantes formant la base `lien_de_parenté` :

Table **parent**	
nom_parent	**nom_enfant**
pierre	paul
sophie	nicole
nadine	sophie
pierre	nicole

Table **homme**
nom
pierre
paul

Table **femme**
nom
nadine
sophie
nicole

Selon l'interprétation précédente, on peut voir dans la base `lien_de_parenté` la base en extension :

$$\begin{array}{ll} parent(pierre, paul) & homme(pierre) \\ parent(sophie, nicole) & homme(paul) \\ parent(nadine, sophie) & femme(nadine) \\ parent(pierre, nicole) & femme(sophie) \\ & femme(nicole) \end{array}$$

où *parent*, *homme* et $femme$ sont des prédicats et où chacune de leurs instanciations précédentes est considérée comme un axiome. Par convention, les termes constants commencent par une minuscule alors que les variables commencent par une majuscule.

Comme on l'a vu, on peut enrichir cet ensemble d'axiomes de règles logiques telles qu'on les a définies. Un exemple de base en compréhension peut être :

$$
\begin{array}{l}
pere(X,Y) \leftarrow parent(X,Y), homme(X) \\
mere(X,Y) \leftarrow parent(X,Y), femme(X) \\
fils(X,Y) \leftarrow parent(Y,X), homme(X) \\
fille(X,Y) \leftarrow parent(Y,X), femme(X) \\
grandpere(X,Y) \leftarrow pere(X,Z), parent(Z,Y) \\
grandmere(X,Y) \leftarrow mere(X,Z), parent(Z,Y) \\
ancetre(X,Y) \leftarrow ancetre(X,Z), parent(Z,Y) \\
famille(X,Y) \leftarrow ancetre(Z,X), ancetre(Z,Y)
\end{array}
$$

où *pere*, *mere*, $fils$, $fille$, *grandpere*, *grandmere*, *ancetre* et $famille$ sont des relations dérivéees.

On appelle *langage de règles* le langage permettant de définir les relations dérivées de la base en compréhension. Les langages de règles développés jusqu'à présent ont, dans leur grande majorité, adopté la syntaxe "à la Prolog" présentée ci-avant.

Un intérêt important de cette syntaxe est qu'elle permet d'exprimer aussi les requêtes : une requête est un axiome comportant des variables libres que l'on précède d'un point d'interrogation "?". Le système va alors déterminer, à l'aide de son démonstrateur de théorèmes, toutes les instanciations possibles des variables libres qui rendent vrai l'axiome correspondant à la question. Adressée à la base précédente, la question "De qui Nicole est-elle la fille ?" s'exprime ainsi :

```
?fille(nicole,X)
La réponse en est :
X = sophie
X = pierre
```

La référence en matière de langage de règles de SGBD est *Datalog* [Ulm86, CGT89]. Datalog peut être vu comme une variation de Prolog avec une sémantique ensembliste [GV90]. Cette sémantique rend le résultat d'un programme indépendant de l'ordre des clauses. Le langage est donc complètement déclaratif.

En dépit de ses propriétés formelles, *Datalog* est assez limité. En particulier, il n'a pas la puissance du calcul relationnel, même si, contrairement à lui, il gère la récursivité. La relation `ancetre` de la base en compréhension précédente en est un exemple. Datalog permet l'union, la projection et la jointure, mais pas la restriction et la différence.

Des extensions ont rendu ce langage plus attractif pour une utilisation réelle. Cependant, elles introduisent chacune leurs lots de restrictions d'usage, en particulier parce qu'elles rendent possible la déclaration de programmes dont la sémantique est ambiguë.

- La première a été de permettre l'utilisation de prédicats prédéfinis dans le corps des règles. Ces prédicats sont identifiés par des symboles spéciaux comme $<$, $\leq$, $>$, geq, $=$, $\neq$, et sont généralement utilisés en notation infixe. Ils permettent la restriction relationnelle. La règle suivante est un exemple d'utilisation d'un tel prédicat :
  $plus\_vieux(X,Y) \leftarrow age(X,A1), age(Y,A2), A1 > A2.$

- La seconde extension a été de permettre l'utilisation de la négation dans le corps des règles. En effet, en $Datalog$, le signe de négation "$\neg$" n'est pas autorisé à apparaître. La négation permet la différence relationnelle et en particulier la gestion des exceptions. Un exemple d'utilisation de la négation peut être :
  $non\_cadre(X)$
  $\leftarrow employe(X, Emploi), \neg employe(X, president), \neg employe(X, directeur).$

- La troisième termine de donner à $Datalog$ la puissance du calcul relationnel. Elle consiste à autoriser l'utilisation des fonctions arithmétiques et mathématiques classiques $+$, $-$, $/$, $*$, LOG, EXP, SIN, ...comme arguments de prédicats dans les règles. Un exemple d'utilisation de fonction arithmétique peut être :
  $metro(Depart, Dest, T1 + T2)$
  $\leftarrow metro(Depart, X, T1), metro(X, Dest, T2).$

De nombreuses autres extensions de $Datalog$ ont été proposées, permettant par exemple la mise-à-jour des faits par les règles ou l'usage de clauses plus générales que les clauses de Horn.

Deux d'entre elles sont particulièrement remarquables car elles permettent au SGBD relationnel sous-jacent de notamment supporter les modèles NF^2 présentés en §1.1.1. La première autorise la définition d'attributs de type relation, la seconde, d'attributs multi-valués :

- La définition d'attributs de type relation est rendue possible en étendant la notion de fonction et en en permettant la création par l'utilisateur.

  En première approche, le prédicat et la fonction se ressemblent car ils se caractérisent tous deux par un symbole (leur nom) et un ensemble d'arguments. Cependant, leurs rôles sont distincts : le prédicat est un terme de formule alors que la fonction est un argument de prédicat ou d'autres appels fonctionnels; la valeur d'un prédicat est booléenne alors que celle d'une fonction peut être quelconque.

  L'intérêt de la fonction définie par l'utilisateur est de rendre possible la structuration sous forme d'*objets complexes* des arguments de prédicat. En ce sens, on peut la comparer aux constructeurs `agrégation` des modèles sémantiques et `n-uplet` des modèles objet (cf. §1.1.2 et 1.1.4).

- La définition d'attributs multi-valués est, quant à elle, rendue possible en définissant un *constructeur d'ensemble* appelé *groupement* et noté $<>$. Ce constructeur permet

d'unifier une variable avec un ensemble de valeurs vérifiant une formule donnée. Par exemple, dans la règle suivante, la variable *Employes* est unifiée avec l'ensemble des noms des employés rattachés au département :

$$membre(Dept, Employes) \leftarrow employe(Nom, Dept), Employes =< Nom > .$$

Ce constructeur `groupement` est tout-à-fait comparable aux constructeurs `groupement` des modèles sémantiques et `ensemble` des modèles objet.

Il est proposé complété d'un certain nombre de prédicats ($member(Element, Set)$, $union(Set, Set1, Set2)$, ...) et fonctions ($COUNT$, MIN, MAX, AVG) prédéfinis utiles pour la gestion des ensembles. De plus, les prédicats prédéfinis usuels ($=$, $\neq$, $<$, $\leq$, ...) ont été élargis aux ensembles : par exemple, le comparateur $<$ appliqué à deux ensembles est un test d'inclusion stricte.

L'exemple suivant, tiré de [CGT89], a été réalisé en *LDL* (Logic Data Language) [CGK+90]. Il présente un ensemble de faits complexes comprenant des fonctions et des ensembles :

$person(name(joe, berger), birthday(1956, june, 30), children(\{max, sarah, jim\})).$
$person(name(joe, coker), birthday(1956, june, 30), children(\{bill, sarah\})).$
$person(name(bebe, suong), birthday(1958, may, 5), children(\{jim, max, sarah\})).$

Ici, *name*, *birthday* et *children* sont des symboles de fonctions. Les variables peuvent représenter des objets atomiques, comme des constantes, ou des objets composés. La règle suivante sélectionne les noms des personnes ayant même prénom et même date de naissance :

$$similar(X, Y) \leftarrow person(name(Z, X), B, C), person(name(Z, Y), B, D).$$

Par cette règle, on peut déduire le nouveau fait $similar(berger, coker)$.

La règle suivante définit un prédicat $eqchilds(X, Y)$ sélectionnant les noms (complets) des personnes dont les enfants ont exactement les mêmes prénoms :

$$eqchilds(X, Y) \leftarrow person(X, B, C), person(Y, D, C).$$

Par cette règle, on peut déduire le nouveau fait

$$eqchilds(name(joe, berger), name(bebe, suong)).$$

On constate ici que l'ordre des éléments d'ensemble n'intervient pas dans la comparaison.

Les ensembles peuvent être définis par énumération, comme dans la déclaration des faits précédents, ou à l'aide du constructeur `groupement` $<>$. La règle suivante construit l'ensemble des personnes ayant un enfant s'appelant Sarah :

$$sarahpar(< X >) \leftarrow person(name(A, X), B, C), member(sarah, C).$$

Cette règle génère le nouveau fait $sarahpar(\{berger, coker, suong\})$.

De très nombreux prototypes de recherche sont actuellement développés à partir de *Datalog* et de ses variations. La plupart sont réalisés en couplant un langage de programmation logique, PROLOG en général, avec un SGBD relationnel. Trois types de couplage peuvent être identifiés [GV90] :

- Le *couplage faible*. Cette première approche consiste à bâtir une interface d'appel au SGBD depuis le langage. L'utilisateur écrit ses programmes dans le langage et invoque explicitement le SGBD en utilisant des prédicats dédiés apportés par l'interface. En général, ces prédicats sont `select`, `insert`, `delete` et `update`, et provoquent l'envoi au SGBD de requêtes correspondantes.
- Le *couplage fort*. Cette fois, l'interprèteur de règles (PROLOG) est modifié de façon à rendre invisible le SGBD. L'utilisateur voit les n-uplets relationnels contenus dans la base comme des faits de son programme.
- L'*intégration*. Enfin, ici, les mécanismes de déduction sont intégrés au SGBD relationnel dont le noyau se trouve enrichi d'un gestionnaire de règles, d'un évaluateur de règles et d'opérateurs spécialisés réalisant efficacement certaines opérations coûteuses comme la récursion.

PROSQL [CW84] est une illustration de couplage faible. Ici, l'interface n'ajoute que le seul prédicat unaire `SQL` à Prolog. L'argument est une chaîne de caractères devant être une requête SQL de sélection, d'insertion ou de mise-à-jour. La requête peut comprendre des variables Prolog, instanciées ou non au moment de l'appel, selon qu'elles constituent des paramètres ou permettent de recueillir un résultat.

ALGRES [CCRG$^+$88, CCRZ$^+$90], *EPSILON* [Kou85, CFK$^+$86, Mol87], *NAIL!* (Not Another Implementation of Logic!) [MUG86, MNS$^+$87], un prototype réalisé par le CERT [CD88], un autre par Philips [DRV86] sont autant d'exemples de couplage fort.

POSTGRES [RS87, SK91], *SABRINA** [Kie89] sont des exemples d'intégration de capacités déductives à un SGBDR, le premier à Ingres, et le second à Sabrina. Les deux systèmes sont en fait développés par les équipes à l'origine des deux SGBDR qui disposent donc de leurs sources. Ils offrent tous deux le support d'objets complexes, spécifiés sous forme de types abstraits (ADT pour Abstract Data Types) définis par l'utilisateur. Ce sont donc des systèmes extensibles. Tous deux proposent également un gestionnaire de règles; celui de SABRINA* s'appelle *RDL1* . Les règles sont des règles de production de type *OPS-5* [BFKM85], de la forme "si `conditions` alors `actions`". La partie `conditions` est une expression du calcul relationnel et la partie `actions`, un série d'insertions et de suppressions de faits dans des relations déduites. Ces règles sont en effet orientées mise-à-jour; elles peuvent également être utilisées pour exprimer des déclencheurs (triggers en anglais) qui s'exécutent quand leurs conditions d'activation sont réalisées.

Certains systèmes correspondent à plusieurs catégories à la fois. C'est le cas d'un prototype développé à l'ECRC qui est en fait composé de trois sous-systèmes *EDUCE*, *DEDGIN* et *PROLOG/KB* [BDN$^+$86]. EDUCE réalise un couplage faible entre PROLOG et le SGBDR Ingres. DEDGIN est construit au-dessus d'EDUCE et réalise le couplage fort. PROLOG/KB est également construit au dessus d'EDUCE. Il assure le stockage des règles dans une base de données et offre un modèle sémantique à l'utilisateur, du type Entité-Relation plus constructeurs IS-A.

Le projet "Calculateur de cinquième génération" *FGCS* de l'ICOT (Institute for New Generation Computer Technology) [MKM+83, Fuc84, Ito86] a pour but de réaliser une "machine bases de données". Cette machine est en fait formée d'une machine matérielle et d'une machine logicielle (système d'exploitation). La première machine matérielle développée fut *DELTA*, maintenant remplacée par *KAPPA*. Toutes deux ont été exploitées avec *PROLOG PSI*, puis avec *PROLOG PIM* qui est une version parallèle de PSI. Dans ce projet, les couplages fort et intégration sont utilisés.

LDL (Logic Data Language) [TZ86, NT89, CGK+90], est un autre SGBD déductif proposant un langage qui étend *Datalog*, notamment par la gestion de la négation et des ensembles [Bee87], et des termes complexes [Zan85]. LDL est original notamment par le fait qu'il n'est pas construit au-dessus d'un SGBDR mais d'une machine bases de données parallèle *BUBBA* [Bor88].

1.3 Couplage d'un environnement IA avec un SGBD

Enfin, un certain nombre de travaux en IA visent à doter les systèmes raisonneurs de grandes bases de connaissances en les couplant à des SGBD. En général, un "traducteur" entre le modèle de données du système raisonneur et celui du SGBD est proposé de façon à permettre les transferts.

C'est la solution que nous avons retenu. La description du logiciel DRIVER [Leb90a, Leb90b, Leb92a, Leb92b] qui est la concrétisation de ce travail est l'objet de la présente thèse.

À notre connaissance, trois autres couplages ont été réalisés.

Le premier est celui qui a été réalisé entre Proteus [Rus89] et Orion [Kim90]. Orion est le SGBD orienté objet dont nous avons déjà parlé. Proteus est un système de maintien de la vérité (TMS) dans lequel les données ont une représentation à base de frames. Dans ce couplage, l'une des difficultés les plus significatives à surmonter a été la non-gestion par Orion du concept de méta-classe requis par Proteus. Mis à part ce point, les deux modèles de données offrent des similarités importantes et sont donc compatibles.

En fait, dans l'approche choisie, Proteus n'est pas différent d'une autre application construite sur Orion. Proteus exploite Orion par appels de fonctions tandis qu'Orion n'appelle jamais Proteus. Orion fournit simplement un support persistant aux instances de frames de Proteus qui seraient volatiles autrement. Le moteur de la base reste séparé du moteur d'inférence et le gestionnaire de connaissance ne maintient la connaissance que dans son domaine, c'est-à-dire en mémoire : la connaissance est donc bien séparée des données. Orion est simplement utilisé comme un stockage d'objets passif attendant les accès de Proteus.

Un couplage a également été réalisé entre KEE [FK85] et les SGBD relationnels. Le système qui réalise ce couplage est connu sous le nom de *KEEconnection* [KEE87]. Nous ne ferons pas de description détaillée de KEEconnection car le principe du couplage est

identique à celui qui a été retenu en DRIVER, à savoir basé sur la notion de schéma de correspondances.

Ajoutons seulement qu'à la différence de DRIVER,

- le modèle objet géré ne peut être choisi, c'est nécessairement celui de KEE. En fait, KEEconnection a été réalisé pour KEE et ne peut être utilisé sans lui.
- L'accès à un objet ne peut être réalisé qu'en le chargeant.
- L'absence de valeur dans un champ de type objet n'est pas autorisée.
- Quand un objet dans la base en référence d'autres par l'intermédiaire de champs objet, son chargement provoque nécessairement le chargement de ceux-là, et récursivement. Ainsi, si le réseau d'objets est suffisamment étoffé, toute la base peut être chargée en mémoire.
- Seuls les champs de type atomique et objet (simple) sont gérés. Il n'est pas possible de "mapper" des champs de type "ensemble", "ensemble ordonné", ou encore "calculé".

Enfin, un troisième type de couplage a été réalisé entre les systèmes à base de connaissance de type réseau sémantique et les SGBD orientés objet par le logiciel *PDKB* (Persistent Data/Knowledge Base) [HL90]. Afin de prendre en compte la représentation des connaissances des réseaux sémantiques, PDKB propose un ensemble d'axiomes formalisés sous forme d'expressions "à la Prolog" (logique du premier ordre) qui étendent les capacités de modélisation des SGBDOO. Ces abstractions formalisées incluent la généralisation/spécialisation (IS-A), l'agrégation, la composition, l'association et le groupement.

Ces travaux sont à rapprocher de ceux effectués sur Datalog , les modèles de données sémantiques (SDM) et les modèles entité-relation étendus. Mais la différence fondamentale entre PDKB et ces modèles est que PDKB propose les principes d'organisation sémantiques précédents et les axiomes de déduction formalisés dans le même environnement. PDKB pallie ainsi à l'incapacité de Datalog de représenter des situations du monde réel par abstractions sémantiques, et à celle des SDM d'offrir une représentation formelle du modèle de données.

1.4 Conclusion

Au cours de ce rapide tour d'horizon, nous avons pu constater qu'il existait de nombreuses façons de gérer de la connaissance dans une base de données et de raisonner dessus. Cependant, malgré cette diversité, les systèmes qui les implantent utilisent tous une architecture globale qui se range dans l'une des deux catégories suivantes [Sto90] :

- Dans la première, les données sont stockées dans le SGBD, et la connaissance, dans le système expert. Les données pertinentes selon la situation sont chargées à la demande dans l'espace de travail.

  Cette architecture est celle de l'exploitation de la plupart des systèmes que nous avons vus. C'est celle de beaucoup d'interfaces entre PROLOG et les SGBDR, c'est aussi celle de DRIVER et des applications utilisant les SGBD orientés objet.

  Son principal inconvénient provient du fait que les SGBD actuels sont incapables de prévenir l'interface que des données chargées en mémoire sont par ailleurs modifiées dans la base par un accès concurrent. Dans cette alternative, la gestion dynamique des données est donc un vrai problème.

- Dans la seconde, les données ET la connaissance sont stockées dans le SGBD.

  Il existe, comme nous l'avons vu, un certain nombre de prototypes ayant intégré un système de règles au sein d'un SGBD. Cependant, ces systèmes de règles sont primitifs par rapport à ceux existant en IA. Il n'y a pas de contrôle de l'inférence; il est impossible d'explorer des arbres de preuves et de revenir en arrière dans un raisonnement. En fait, les critiques de l'existant sont nombreuses et fondées. Mais, dans le futur, des bases de connaissances plus complexes seront certainement gérées de cette façon.

En dépit de ses inconvénients et de performances parfois médiocres, le premier type d'architecture semble donc être actuellement la meilleure solution pour le développement d'applications complexes ayant besoin de gérer de larges bases de connaissances.

Chapitre 2

DRIVER, COUCHE OBJET VIRTUELLE PERSISTANTE

2.1 Caractéristiques générales

Notre souci initial était de permettre au Générateur de Systèmes Experts SMECI [Sme92] d'accéder à des bases de données relationnelles et de les exploiter dans le cadre du raisonnement.

Nombreuses sont en effet les applications SMECI souhaitant pouvoir orienter leur raisonnement par la consultation de larges bases de données. La possibilité d'interroger une base permet par exemple de connaître les matériaux produits dans une région pour y minimiser le coût d'une construction, de consulter les normes techniques pour que cette construction soit conforme aux règles de l'art, de déterminer les fondations en fonction du site, de sélectionner sur catalogue les poutres d'acier utilisables selon le dimensionnement effectué par le système... On constate que les utilisations en conception sont innombrables. Elles le sont également dans d'autres domaines quand la base de données est utilisée comme support pour le partage d'informations. Un système expert spéculateur peut par exemple tirer profit de la consultation en temps réel des fluctuations boursières pour gérer des porte-feuilles de façon optimale. Un pilote automatique peut moduler l'allure et le cap d'un navire en fonction du trafic et de la météo tenus à jour dans une base de données commune, etc...

Une autre de nos préoccupations était d'apporter la persistance aux objets manipulés par SMECI qui présentent un intérêt pour une utilisation ultérieure, en particulier l'ensemble de ses bases de connaissances et le produit de ses analyses.

Il peut en effet être utile et très efficace de pouvoir retrouver les solutions proposées à des problèmes posés lors de sessions précédentes. Les applications raisonnant d'abord par analogie sur les cas déjà rencontrés avant d'éventuellement poursuivre en utilisant des codes de calcul plus complexes et plus coûteux en temps sont des exemples de systèmes demandeurs de persistance.

Enfin, bien que SMECI ait été à l'origine de nos travaux sur les bases de données, il nous a semblé important que l'outil qui allait nous permettre de gérer de grandes bases de connaissances persistantes en soit complètement indépendant de façon à pouvoir être ensuite utilisé de façon autonome.

2.1.1 Un pont entre deux mondes

Nos efforts se sont concrétisés au travers du système DRIVER [Leb90c, Leb90a, Leb90b] dont l'une des caractéristiques principales est de définir une couche objet pour les bases de données relationnelles.

DRIVER permet de présenter les données relationnelles sous forme d'objets complexes et les rend directement accessibles pour exploitation dans l'environnement objet du client[1]. C'est d'autre part ce même client qui précise le modèle objet dans lequel les données relationnelles sont exprimées. De cette façon, les objets sont particulièrement adaptés à l'usage qu'il souhaite en faire et peuvent être manipulés dans l'environnement de la même façon que ceux qui ne sont pas issus d'une base de données. Tous peuvent s'intégrer harmonieusement dans la même couche objet.

Par ailleurs, DRIVER permet également de rendre persistants dans une base de données relationnelle des objets de l'environnement qui ne représentent pas déjà des données d'une telle base.

Dans le premier cas, le système associe une représentation objet à des structures relationnelles. À l'inverse, dans le second, il associe une représentation relationnelle à des structures objet. À chaque fois sont utilisées de semblables correspondances mettant en rapport les deux représentations, correspondances en fait indépendantes du sens dans lequel s'effectue la transaction. Elles constituent le *schéma de correspondances* (figure 2.1) qui permet la traduction et le transfert des données d'un monde dans l'autre lors des échanges gérés par DRIVER entre la base et l'environnement objet.

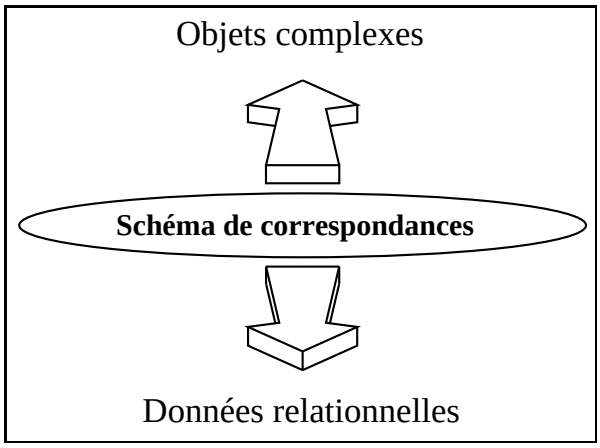

Figure 2.1 : Un schéma de correspondances entre deux représentations

Un schéma de correspondances contient trois catégories d'informations : celles relatives aux deux représentations de données, plus celles qui assurent leur rapprochement.

Sa construction peut être effectuée de différentes manières. En général, l'une des deux représentations existe déjà. Reste donc à définir l'autre et à indiquer les liens qui les unissent.

La solution la plus simple peut consister à générer automatiquement les éléments

[1] Environnement Le_Lisp pour l'instant puisque notre prototype a été développé dans ce langage.

manquants du schéma. DRIVER dispose de telles fonctionnalités qui permettent la prise en compte et l'exploitation immédiates d'une table relationnelle dans l'environnement et l'attribution tout aussi rapide de la persistance à une classe [Leb92b].

Cette possibilité peut n'être considérée que comme une solution à court terme. Elle est en effet assez brutale. Les éléments générés mécaniquement ne peuvent être que stéréotypés, ils manquent en plus de lisibilité. Il est en général préférable de décrire plus finement ses correspondances, notamment pour choisir une représentation objet pour des données relationnelles. Une des raisons principales est que seul l'utilisateur connaît la sémantique de certains attributs, en particulier ceux qui constituent des références vers d'autres tables. Le système n'a aucun moyen de la déterminer. Par souci de clarté, le terme "attribut" fera toujours référence dans notre propos à une colonne de table relationnelle tandis que le terme "champ" se rapportera à la propriété d'une classe d'objets.

Le schéma de correspondances peut également être construit à l'aide d'un éditeur graphique prévu à cet effet [BM92] (cf. §A.3). Le néophyte est alors pris par la main et guidé pas à pas. À chaque étape, les choix proposés tiennent compte de ce qui a déjà été fait, rendant impossible l'introduction d'incohérences ou d'incompatibilités.

L'utilisateur avancé préfèrera directement décrire son schéma à partir de l'interface fonctionnelle [Leb92b]. C'est la méthode la plus précise et la plus efficace. Un contrôle très strict de chaque déclaration est effectué, assurant ainsi la validité du schéma à chaque étape de sa construction. L'éditeur graphique dont on a précédemment parlé repose en réalité sur l'interface fonctionnelle. C'est une puissante aide qui permet de construire simplement un schéma sans connaître le nom des primitives et leurs arguments.

2.1.2 Une couche objet virtuelle persistante

Les données relationnelles dans la base sont interprétées comme des objets potentiels qui résultent des associations définies dans le schéma de correspondances. Ces objets potentiels sont appelés les *objets relationnels*. DRIVER permet l'accès immédiat aux objets relationnels depuis l'environnement. Cet accès peut par exemple être réalisé en construisant en mémoire les images des objets relationnels sélectionnés. Ces images sont appelées les *objets jumeaux* (cf. §2.3.4.2). Cependant, si le client le souhaite, l'accès peut être directement effectué dans la base de données sans construction dans l'environnement des jumeaux correspondants.

Du fait du couplage qu'il réalise avec les SGBDR, DRIVER élargit l'ensemble des objets directement accessibles et manipulables dans l'environnement en adjoignant aux objets spécifiquement de l'environnement les objets représentants de données relationnelles. L'ensemble de ces objets constituent la *couche objet virtuelle de* DRIVER. Comme le système fournit en outre toutes les fonctionnalités nécessaires pour administrer la persistance et l'attribuer à tel ou tel objet ou à telle ou telle classe, on qualifie également cette couche objet virtuelle de persistante (figure 2.2).

DRIVER donne accès aux objets relationnels par une fonction qui permet de les filtrer

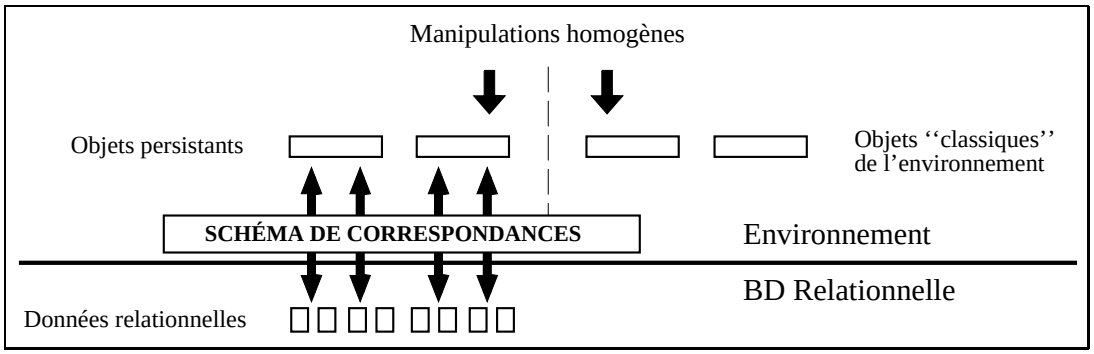

Figure 2.2 : La couche objet virtuelle

(cf. §4.4). Ainsi, il est possible de sélectionner dans la base des objets relationnels vérifiant certains critères et de les exploiter ensuite au sein de la couche objet, par exemple dans le cadre du raisonnement d'un système expert.

Le filtre est un prédicat à n variables où chacune d'entre elles représente une instance d'une classe précisée, décrite dans le schéma de correspondances. Le corps du prédicat est un ensemble de contraintes liant généralement ces n variables et pouvant être des comparaisons entre objets, entre champs, entre champs et valeurs. Les substitutions solutions sont donc des n-uplets d'objets relationnels. Les objets filtrés ne sont pas construits dans l'environnement. Sont retournées à la place les références des objets relationnels concernés qui permettent de construire ensuite des "objets-liens" vers ces objets relationnels.

Ces "objets-liens" vers la base de données sont appelés des *défauts d'objet*. Ils rendent dans l'environnement la manipulation d'un objet relationnel qui a été filtré strictement identique à celle d'un simple objet de même classe en mémoire. Toutes les méthodes associées à sa classe sont également valables pour les deux catégories d'objets si bien qu'un utilisateur final peut complètement ignorer, lorsqu'il manipule un objet, si ce dernier est une représentation de données relationnelles ou un simple objet de l'environnement sans rapport avec la base. Les défauts d'objet sont plus précisément décrits en §2.3.3 et en §4.1.1.2.

Au besoin, par exemple pour consulter ou mettre à jour la valeur d'un champ, il est possible, à partir du défaut d'objet, de construire dans l'environnement l'objet jumeau de l'objet relationnel correspondant.

On a vu que la couche objet pouvait comprendre des classes définies pour représenter des structures relationnelles. Elle comprend également celles à qui on a associé une représentation relationnelle pour pouvoir écrire leurs instances dans la base. Par souci de simplification, on qualifiera de *persistantes* l'ensemble de ces classes qui ont pour point commun d'être décrites dans un schéma de correspondances. Une classe persistante peut offrir la persistance à chacune de ses instances. Les informations propres à la classe sont elles-mêmes persistantes grâce au *méta-schéma* (cf. §3.3.2).

Les classes persistantes s'intègrent complètement aux hiérarchies de classes définies

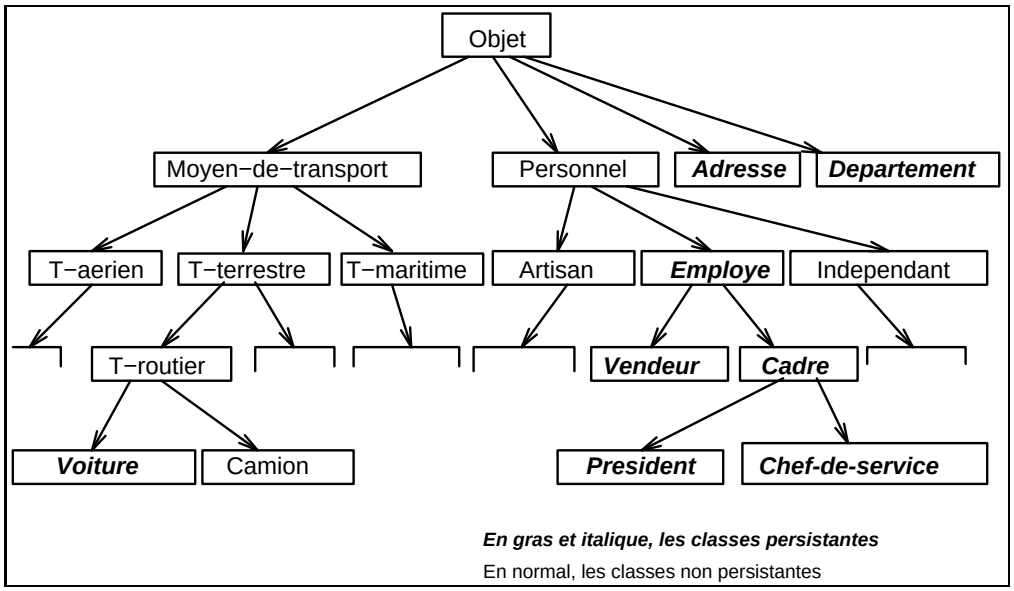

Figure 2.3 : Exemple de hiérarchie mêlant classes persistantes ou non

dans l'environnement (figure 2.3). Toute classe peut posséder une ou plusieurs sous-classes persistantes.

Une classe qui possède au moins une classe ancêtre persistante a des instances qui peuvent prétendre à la persistance. En effet, les instances d'une classe sont a fortiori instances des classes ancêtres. La classe hérite de toute qualité qui n'est pas redéfinie, notamment de la persistance. Malgré tout, seuls les champs hérités et décrits dans le schéma sont persistants. Les autres, en particulier les champs locaux, ne le sont pas.

Quand on définit une sous-classe persistante à une classe non persistante, les champs hérités ne sont généralement pas persistants, car sans correspondance relationnelle. Heureusement, il est possible de contourner cet obstacle en décrivant leurs correspondances dans le schéma de façon délocalisée, au niveau de la sous-classe (cf. §3.1.3.2).

2.1.3 Des représentations optimales

Il existe nombre de manières de représenter sous forme d'objets un ensemble de données relationnelles, comme il existe nombre de manières d'exprimer sous forme relationnelle un ensemble de classes.

La façon pertinente d'exprimer dans le monde objet une information contenue dans un attribut dépend de la sémantique qu'on lui prête et de l'utilisation qu'on va en faire.

Ainsi, pour telle application, tel programmeur donne comme correspondance à notre attribut un champ atomique (cf. §3.2.3) qui permet l'accès à des données brutes, par exemple des entiers. Dans une autre ou avec un autre programmeur, le même attribut peut servir à définir une jointure[2] qui est la correspondance d'un champ de type "objet".

[2] Au sens SGBD.

Un champ de type "objet" ou "ensemble d'objets" (lien entre objets) a en effet pour correspondance naturelle une jointure qui est un lien entre deux tables. Autre exemple : lorsque cette jointure est définie entre deux tables principales (correspondantes de deux classes) et est, par exemple, l'expression d'une relation 1:N, on a le choix entre lui associer un champ de type "objet" dans l'une ou définir un champ de type "ensemble d'objets" dans la seconde. On peut même lui associer les deux, moyennant certaines restrictions d'usage.

Le choix existe également pour affecter une représentation relationnelle à un ensemble de classes. Une classe et ses sous-classes sont associées à une table appelée *table principale* (cf. §3.2.2) et la plupart des correspondances des champs sont définies sur cette table. Cependant, il est possible de choisir la correspondance d'un champ sur une autre qui devient de ce fait *table secondaire* de la hiérarchie de classes précédente (cf. §3.2.3.1).

Par ailleurs, une jointure, définie pour accéder à une information, un ensemble d'informations, à un objet ou à un ensemble d'objets, peut être définie de multiples façons possibles.

Ces exemples donnent un aperçu de l'ensemble des choix qui s'offrent au concepteur d'un schéma de correspondances qui tente de coupler une représentation à l'autre. On peut en outre faire la remarque que le concepteur n'est nullement tenu, lorsqu'il définit un schéma, d'y décrire la totalité des structures auxquelles il a accès, que ce soit dans une représentation ou dans l'autre.

Quand son but est d'attribuer la persistance à un ensemble de classes, nul n'est besoin de décrire les autres. Il lui est également inutile de définir au sein d'une classe à rendre persistante, la correspondance de champs dont l'écriture dans la base de données ne présente pour lui aucun intérêt.

Quand il souhaite associer une représentation objet à des données relationnelles qu'il a besoin d'exploiter, il est également libre de ne décrire de la base que les structures qui l'intéressent et, bien sûr, de ne définir que leurs correspondances.

La structure des classes qui en résultent et la forme sous laquelle les données choisies s'expriment dans les objets sont ainsi adaptées aux besoins réels de l'utilisateur ou du système raisonneur.

Si un schéma semble optimal pour résoudre un certain problème, une autre formulation de ce même problème peut faire apparaître d'autres façons de le résoudre et laisser émerger un autre schéma possible. Une information à gérer dans la représentation objet peut y être exprimée de multiples façons et dans des champs de types différents. C'est la manière par laquelle cette information va être exploitée qui doit guider dans le choix de sa représentation. Cette flexibilité de la représentation objet associée aux données relationnelles permet de l'adapter et de la faire évoluer en fonction des besoins qui changent.

2.1.4 Exploitation de plusieurs bases

Il est possible de tirer profit de l'accès à plusieurs bases de données en définissant pour chacune d'entre elles un schéma de correspondances. Chaque schéma décrit alors comment chaque base est exploitée.

Lorsque plusieurs schémas ont été définis dans le même environnement, chacun d'entre eux y explicite un certain nombre de classes persistantes qui s'intègrent dans la hiérarchie des classes.

Une exploitation avancée de ces possibilités de connexions multiples est ébauchée en §2.4.

2.2 Gestion de la persistance

2.2.1 Granularité de persistance

Avec DRIVER, une information contenue dans un objet est dite persistante si elle a été déclarée persistante (!) et s'il existe un schéma de correspondances dans lequel est décrite sa correspondance relationnelle.

Le seul fait de définir une correspondance ne suffit pas à rendre persistantes les données concernées. En effet, dans un environnement comme celui d'un système expert, l'information évolue beaucoup et il n'est pas toujours souhaitable de conserver l'ensemble des données transposables dans la base grâce à une correspondance. Il vaut mieux avoir la possibilité d'attribuer cette persistance de façon plus précise en sélectionnant explicitement ce qui doit être retenu.

C'est le cas avec DRIVER. D'une manière générale, le schéma de correspondances permet de décrire la correspondance relationnelle de classes et de champs. Tout objet instance d'une classe décrite dans un schéma peut prétendre à la persistance. Deux objets appartenant à une même classe peuvent être différenciés parce que l'un possède cette qualité et l'autre non. Il appartient à l'utilisateur de l'accorder à ceux qui lui semblent présenter un intérêt pour une réutilisation ultérieure.

Au sein d'une même classe rendue persistante peuvent être combinés champs persistants ou non. À nouveau, il est possible de sélectionner, cette fois au cœur même des objets persistants, les informations pouvant prétendre à la pérennité. Les champs non persistants sont tout simplement ignorés pendant les écritures ou lectures d'objets dans la base.

Ces champs non persistants ne doivent pas forcément être déclarés dans le schéma. Ils n'y ont leurs places que si le schéma est également mis en œuvre pour définir dans le modèle objet client les classes auxquelles ils appartiennent. Dans ce cas, leur absence dans la description de la classe dans le schéma se traduirait par leur absence dans la définition de la classe dans le modèle objet utilisé. Quand le schéma n'est pas utilisé pour cela, quand, par exemple, les classes préexistent au schéma, nul n'est besoin de déclarer à DRIVER les champs qu'on ne désire pas rendre persistants.

Il peut être parfois utile de supprimer momentanément la persistance d'un champ. Cette opération peut facilement être réalisée avec DRIVER, par exemple pendant la phase

de mise au point de l'application ou encore pour augmenter localement les performances du système pendant certaines opérations de transfert de données entre l'environnement et la base. Le même type d'opération est également possible avec les classes et se traduit par une gestion encore plus précise de la persistance qui supporte ainsi la notion d'exception. Tel champ d'ordinaire persistant ne l'est plus dans certaines conditions d'exécution de l'application.

Enfin, il peut être parfois utile de décrire, au niveau d'une sous-classe persistante, la correspondance de champs définis dans une classe mère. Ceci est particulièrement vrai quand cette classe mère n'est pas persistante.

Quand, par exemple, le modèle objet utilisé présente une classe racine unique et que cette classe possède un champ `nom`, toutes les classes qui seront ensuite définies hériteront de ce champ `nom`. S'il est souhaitable de rendre l'une de ces classes persistante, il peut être tout aussi souhaitable d'également rendre le champ `nom` persistant sans pour autant devoir attribuer la persistance à la classe racine.

DRIVER permet ainsi de décrire au niveau d'une classe la correspondance de champs non locaux qui seront gérés comme les autres en ce qui concerne la persistance, mais qui seront ignorés lorsque la classe sera définie dans le modèle objet client en utilisant le schéma de correspondances.

2.2.2 Persistance des classes et des correspondances

Le schéma de correspondances contient des informations relatives aux classes et aux relations qu'il aide à rapprocher et relatives aux correspondances établies entre ces classes et relations. Ces trois types d'information sont aisément séparables pour effectuer des opérations spécifiques à chacune d'entre elles comme définir des classes dans le modèle objet utilisé ou créer des relations dans la base.

DRIVER assure la sauvegarde des informations structurelles qui lui sont nécessaires en gérant la persistance des schémas de correspondances.

Cette persistance des schémas et de l'ensemble des informations qu'ils contiennent est assurée grâce à un *méta-schéma* (cf. §3.3.1). Ce schéma associe des relations système à l'ensemble des classes système de DRIVER qui participent à la description des schémas de correspondances. Ces relations forment de par leur nature une sorte de méta-base, un *dictionnaire de données complexes.*

Il peut arriver que DRIVER soit instancié avec un modèle objet dont il ne puisse pas gérer l'ensemble des concepts, tel celui de méta-classe. Les informations sur les classes contenues dans un schéma de correspondances sont alors insuffisantes pour prendre en compte l'ensemble des caractéristiques du modèle objet. Apporter la persistance aux schémas de correspondances ne suffit donc pas pour rendre persistantes toutes les informations structurelles du modèle objet client.

Si des caractéristiques particulières de ce modèle objet ne sont pas implicitement gérées par DRIVER et que leur absence ne permet pas de reconstruire correctement l'ensemble

des informations structurelles, il est tout à fait possible de définir un schéma de correspondances en complément du méta-schéma, associant ces traits qui ne sont pas gérés à des tables "système" supplémentaires.

2.3 Transactions

Un objet persistant modifié ou créé dans l'environnement n'est pas immédiatement mis à jour dans la base. Cette phase d'écriture et de répercussion dans la base de modifications effectuées en mémoire doit être explicitement provoquée par un ordre de validation de *transaction*.

2.3.1 Transaction DRIVER et transaction SGBD

Une *transaction* DRIVER est une session propre à un ensemble d'objets persistants pendant laquelle toute modification de l'un d'entre eux n'est effective que dans la couche objet virtuelle à laquelle ils appartiennent. Tous les autres clients de la base, en particulier des données relationnelles correspondant à ces objets, continuent de les "voir" comme si elles n'avaient pas été modifiées. Les mises à jour ne sont visibles pour les autres qu'après validation de la transaction.

La notion de transaction existe aussi au niveau du SGBD et implique l'ensemble des données d'une base. C'est une session d'accès à une base, propre à un utilisateur. Elle se valide par exemple en SQL au moyen de la commande COMMIT.

Toute requête adressée au SGBD dans le cadre d'une telle transaction verrouille les données concernées et les rend provisoirement inaccessibles pour les autres transactions. Ce verrouillage existe quel que soit le type de la requête, qu'elle soit une sélection, une insertion, une mise à jour ou une suppression de n-uplet. Selon le SGBD sont verrouillées les seules données concernées, les pages[3] qui les contiennent ou les relations entières. Tout client voulant accéder dans sa propre transaction à des données verrouillées par un autre est mis en attente. Il reste bloqué jusqu'à ce que ces données soient à nouveau disponibles, jusqu'à ce que le "verrouilleur" valide sa transaction.

La transaction DRIVER ne se confond pas avec la transaction SGBD. Cet attachement aurait pu être préjudiciable pour toutes les applications nécessitant le partage de données. En fait, la validation de transaction SGBD est entièrement gérée par DRIVER qui l'effectue à bon escient, de façon totalement indépendante de celle de son propre système de transactions qui s'applique aux objets et uniquement sur demande.

Une transaction DRIVER implique soit l'ensemble des objets persistants de la couche virtuelle, soit un groupe d'objets qui est précisé à la validation. Sa granularité est d'un objet, c'est-à-dire que la plus petite transaction peut ne concerner qu'un objet (cf. §4.2.1.1). Si cet objet en référence d'autres directement ou indirectement par l'intermédiaire de

[3] La page est une entité de stockage propre aux BD

champs de type objet, il est possible d'inclure automatiquement ces objets référencés dans la transaction. Dans ce cas, ils sont également mis à jour dans la base lors de la validation.

Le fait que la transaction DRIVER n'implique pas forcément l'ensemble des objets persistants de la couche virtuelle est intéressant pour des raisons d'efficacité. De cette manière, à chaque phase de mise-à-jour de la base n'est concernée qu'une partie choisie de la couche objet, ceci se traduisant notamment par des gains en performances appréciables. Comme nous l'avons déjà souligné, la transaction SGBD, elle, implique à chaque fois l'ensemble des données contenues dans une base.

Enfin, les validations des deux types de transaction ne coïncident pas et sont effectuées selon des fréquences différentes. En DRIVER, une validation de transaction SGBD est faite[4] après chaque opération de transfert de données entre la couche objet et la base. Si un objet ou un groupe d'objets est chargé en mémoire, une validation SGBD libère aussitôt les relations verrouillées par la lecture. Si un champ d'objet est directement lu dans la base, cette lecture est immédiatement suivie d'une validation analogue.

L'insertion ou la mise à jour de données dans la base **se termine** également par une validation SGBD. Toute opération d'écriture dans la base d'un objet ou d'une information contenue dans un objet commence en DRIVER par une lecture dans la base des données correspondantes. Cette lecture, par contre n'est pas immédiatement suivie d'une validation SGBD et se traduit ainsi concrètement par le verrouillage des données consultées. Elles peuvent ensuite être mises à jour sans risque d'intervention intempestive d'un autre client. L'écriture est close par la validation de transaction SGBD qui rend visibles tous les éventuelles modifications de la base.

L'accès ou la modification d'informations contenues dans la base sont ainsi non bloquants pour les autres. Les validations de transaction SGBD qui accompagnent la plupart des accès de DRIVER à la base de données garantissent des temps de verrouillage du SGBD extrêmement courts, ce qui est très utile lors de l'exploitation d'une base partagée.

Si l'application DRIVER est seule à utiliser la base ou si l'on souhaite adopter une politique plus sûre de gestion des accès concurrents, il est possible d'éviter les fréquentes validations SGBD qui sont faites en changeant la valeur d'un l'indicateur système (driver-commit-after-reading). Ceci peut se traduire par une efficacité accrue du système car DRIVER n'effectue plus de validation de transaction SGBD qu'au moment où il valide la transaction DRIVER. Dans ce cas, les transactions SGBD et DRIVER sont confondues.

En conclusion, le mécanisme de transaction au niveau de la couche objet apporte un confort important dans la manipulation des objets persistants. Ceux-ci peuvent ainsi être modifiés, complétés, affinés sans subir les lourdeurs liées à l'écriture en temps réel sur disque.

De façon générale, le mécanisme des transactions permet de protéger l'intégrité et la sûreté des données qui restent cohérentes au sein de la base.

[4] Par défaut. Voir plus loin le rôle de l'indicateur système driver-commit-after-reading.

2.3.2 Maintien de la cohérence

Un certain nombre de modifications effectuées sur des objets peuvent être cohérentes dans leur ensemble et incohérentes si elles ne sont faites que partiellement. De telles modifications sont réalisées de façon sûre et validées conjointement par une même transaction alors qu'une mise-à-jour incrémentale aurait provoqué une incohérence momentanée au sein du SGBD.

Les transactions DRIVER comportent en plus des transactions SGBD un certain nombre de caractéristiques spécifiques qui vont dans le sens d'une plus grande sécurité pour les données contenues dans la base.

Une validation de transaction DRIVER permet de mettre à jour dans la base un groupe d'objets persistants. Les modifications effectuées dans l'environnement sur les objets persistants peuvent ainsi être validées de façon incrémentale, par exemple, après contrôle de leur validité.

Quand la base est partagée, un même environnement peut contenir simultanément des objets persistants dont la fréquence de validation dans la base doit varier de l'un à l'autre. On peut par exemple distinguer les objets qu'on sait uniquement manipulés par notre environnement et qui n'ont pas besoin d'être validés très fréquemment. À l'inverse, il y a ceux qui contiennent des informations partagées entre plusieurs systèmes et qui doivent être validés dès qu'ils sont modifiés. Grâce aux transactions DRIVER, ces différents types d'objets peuvent être gérés de façon optimale en minimisant les accès au SGBD.

En cas d'incohérence dans la couche virtuelle, au sein d'un groupe d'objets persistants précisément identifié, il est possible, en annulant la transaction en cours pour ces objets, de revenir en arrière et de leur redonner une valeur cohérente. Cette valeur est celle de leurs objets relationnels respectifs dans la base. Ils reflettent donc à nouveau très précisément les données relationnelles qu'ils représentent dans l'environnement. Le groupe est ainsi replacé dans un état qui a précédemment été reconnu valide.

Il est intéressant de remarquer que seul ce groupe est revenu à des valeurs antérieures. Le reste de la couche objet reste inchangé, en particulier les autres objets persistants modifiés qui n'ont pas encore été validés dans la base.

2.3.3 Défauts d'objet et objets relationnels

DRIVER permet une manipulation identique des objets de l'environnement et des *objets relationnels* au sein de la couche objet virtuelle. Rappelons que les objets relationnels sont les objets **potentiels** contenus dans la base de données qui résultent des correspondances effectuées.

Les objets relationnels sont atteints au moyen d'une primitive qui permet de les filtrer. Cependant, cette primitive ne renvoie pas directement les objets en question, mais leurs références. Ces dernières peuvent être confiées à une autre primitive qui délivre des objets système DRIVER qui "représentent" les objets relationnels correspondants dans l'environnement en l'absence d'objets jumeaux. Ces objets système sont les *défauts d'objet*.

Un défaut d'objet constitue un lien vers l'objet relationnel qu'il représente en mémoire (figure 2.4). Il contient toutes les informations nécessaires pour retrouver dans la base les données relationnelles associées et leur donner une représentation objet. En particulier, il connaît un schéma de correspondances, la classe principale de l'objet représenté et sa valeur de clé qui l'identifie de manière unique dans la base. Par le schéma de correspondances, il accède à la connaissance permettant de faire passer les données d'une représentation dans l'autre et à un canal de communication par lequel le dialogue avec la base est possible.

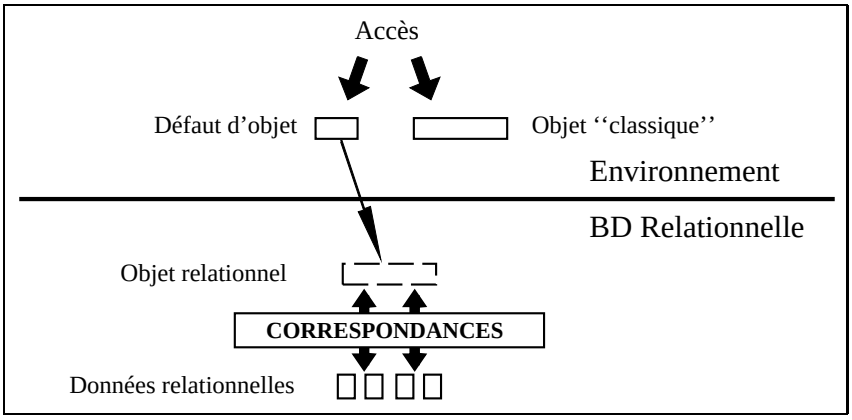

Figure 2.4 : Défaut d'objet et objet relationnel

À la demande et de façon transparente, le défaut d'objet est capable d'interroger la base et d'accéder aux informations souhaitées contenues dans l'objet relationnel qu'il représente.

En fait, il se manipule comme un objet classique de la couche objet. Pour l'utilisateur, l'objet qu'il met en œuvre "*est*" l'objet relationnel, car le défaut d'objet se comporte comme s'il était celui qu'il représente. Les champs définis pour la classe de l'objet relationnel sont normalement accessibles et peuvent être simplement consultés et mis à jour. Les méthodes sont également applicables. Le défaut d'objet répond correctement aux envois de message qui lui sont adressés et qui sont en réalité destinés à l'objet relationnel.

Le défaut d'objet est petit; il occupe très peu de place en mémoire. À l'opposé, il peut représenter des objets relationnels de taille quelconque qui peuvent donc être a fortiori très gros. Par cette technique du défaut d'objet, DRIVER permet ainsi l'exploitation, depuis l'environnement, de bases de données importantes. La mémoire n'est pas surchargée puisque les données n'ont pas besoin d'être chargées pour être prises en compte. Inversement, il est également possible, de ce fait, de décentraliser au niveau des bases de données certains modules de connaissance, parfois peu utilisés, qui, jusqu'à présent, étaient gérés en mémoire.

2.3.4 Accès aux champs et objets jumeaux

Le défaut d'objet permet d'accéder de deux façons différentes aux informations contenues dans l'objet relationnel.

2.3.4.1 L'accès direct

La consultation ou la mise-à-jour d'un champ de l'objet relationnel peut se traduire par un accès à la base où sont directement lues ou écrites les données relationnelles correspondant à la valeur du champ. Chaque accès se traduit par l'envoi d'une requête au SGBD. Après l'opération, le défaut d'objet est toujours un défaut d'objet.

Ce type d'accès est intéressant à plus d'un titre. Il permet de consulter certaines informations contenues dans de gros objets sans alourdir la mémoire plus que ne le réclamerait l'application. Dans le cas d'objets partagés, on accède ainsi à des valeurs qui sont toujours à jour. Des données issues d'autres utilisateurs de la base peuvent ainsi être immédiatement prises en compte.

Il présente cependant certains inconvénients. N accès à un objet se traduisent par N requêtes envoyées au SGBD, ce qui peut être lourd et manquer d'efficacité quand on travaille beaucoup avec les mêmes objets. C'est particulièrement vrai quand on sait que ces objets en question ne sont pas partagés avec d'autres clients du SGBD et que deux consultations successives d'un même champ ne risquent pas de renvoyer deux résultats différents. Enfin, un inconvénient de taille existe concernant la mise-à-jour. L'accès direct en écriture dans la base enfreint les règles du système des transactions DRIVER. Toutes les modifications effectuées ainsi ne peuvent pas être défaites par une annulation de transaction. L'usage de cette fonctionnalité en écriture doit donc être utilisée avec beaucoup de précautions.

2.3.4.2 Les objets jumeaux

L'autre alternative consiste à déclencher la construction d'un *objet jumeau* lors d'une tentative d'accès au champ d'un objet relationnel encore représenté par un défaut d'objet. L'objet jumeau est une image fidèle de l'objet relationnel. Seulement, contrairement à lui, il a l'avantage d'être un objet de l'environnement à part entière, héritant de toutes les propriétés du modèle objet client. À sa construction, il remplace purement et simplement le défaut d'objet si bien que toute connaissance de l'environnement qui référençait le défaut d'objet référence maintenant le jumeau. Une fois construit, le jumeau masque l'objet relationnel associé (figure 2.5). La primitive DRIVER d'accès aux objets relationnels renvoie d'ailleurs le jumeau s'il existe, au lieu du défaut d'objet.

Le jumeau est un objet de l'environnement ayant pour classe celle de l'objet relationnel. Quand il est construit, seules les valeurs atomiques et les ensembles sont chargés. Tous les objets relationnels auxquels il fait référence dans l'un de ces champs sont remplacés par des défauts d'objet. Là encore, si certains disposent déjà de leurs jumeaux, ce sont ces derniers qui sont retrouvés et utilisés.

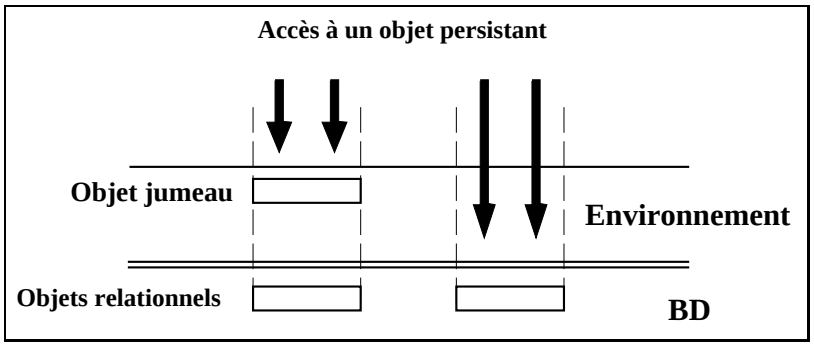

Figure 2.5 : Masquage de l'objet relationnel par l'objet jumeau

Il était bien sûr possible de placer et construire directement dans les champs objet les jumeaux des objets relationnels référencés. Mais, pour peu qu'ils référencent à leur tour de nouveaux objets et que le réseau entre objets soit suffisamment étoffé, on aurait vite fait de charger ainsi récursivement toute la base de données en mémoire. L'intérêt de la couche objet virtuelle s'en serait trouvé singulièrement amoindri.

Le principe des jumeaux s'intègre complètement à celui des transactions DRIVER. Quand un objet relationnel subit une tentative de modification, son jumeau est construit. C'est lui qui est mis à jour; les données relationnelles dans la base ne sont pas touchées. À ce moment, le contenu du jumeau ne correspond donc plus à celui de l'objet relationnel, mais cette différence n'est pas perceptible depuis l'environnement puisque le jumeau masque l'objet relationnel dès qu'il est créé. Tout nouvel accès sur l'objet relationnel est maintenant effectué sur le jumeau. Aucune requête n'est plus adressée au SGBD. À la validation de la transaction, toutes les modifications effectuées sur l'objet sont reportées sur l'objet relationnel, sur les données de la base. L'objet relationnel et le jumeau sont à nouveau strictement identiques. À l'inverse, si la transaction est annulée, tous les objets jumeaux sont remplacés par des défauts d'objet. Les modifications effectuées en mémoire sont ainsi perdues. L'objet jumeau disparaissant, le masquage de l'objet relationnel disparaît également.

Les avantages de cette gestion des accès aux objets relationnels sont assez importants. Tout d'abord, la préservation du système des transactions est une chose importante. Par ailleurs, en règle générale, le système gagne aussi en efficacité. Pour charger un objet (ou plusieurs) d'une classe comportant N champs de type "ensemble" ou "calcul multi–n-uplets", DRIVER effectue $1+N$ requêtes au SGBD (cf. §4.1.2.1). Les applications comprenant de nombreuses définitions de champs de ces types bénéficient alors de la diminution du nombre de communications avec le SGBD.

Un autre avantage des objets jumeaux concerne les systèmes raisonneurs. Cet avantage est décrit dans la section suivante.

Il est toujours possible d'accéder directement à un objet relationnel dans la base de

données, y compris quand son jumeau a été construit. Le fait que son représentant dans l'environnement soit un défaut d'objet ou un jumeau ne fait pas de différence à ce niveau; les primitives d'accès direct restent disponibles.

Cette possibilité est intéressante à plus d'un titre. On peut ainsi à tout moment surveiller l'évolution d'un paramètre dans une base de données partagée, quel que soit le représentant de l'objet relationnel dans l'environnement. Quand le jumeau est formé, il est également possible d'accéder directement à la valeur d'un champ dans la base pour la comparer avec celle contenue dans le jumeau. Enfin, l'accès direct à la base depuis l'objet jumeau permet aussi de gérer en mémoire la liste des valeurs successivement prises par un champ.

Quand l'objet jumeau a été construit, l'accès direct dans la base de données à la valeur d'un champ peut déclencher sur ce jumeau des *réflexes de mise-à-jour* qui sont inhibés par défaut.

En consultation, si la valeur du champ lue dans la base est différente de celle se trouvant dans l'objet jumeau, la différence indique que les données relationnelles correspondantes ont été modifiées par un autre utilisateur de la base depuis la construction du jumeau. Dans ce cas, DRIVER peut ne pas se contenter de rendre la nouvelle valeur du champ. Il peut également mettre à jour le champ dans l'objet jumeau.

En écriture, quand on accède directement à la base pour modifier le champ d'un objet relationnel, on peut également mettre à jour le champ correspondant de l'objet jumeau si sa valeur diffère de celle qu'on affecte dans la base.

Ces réflexes présentent un intérêt pour les applications où la base est partagée et où il est important d'avoir dans l'environnement des objets qui reflètent au maximum l'état de cette base de données.

Ils présentent cependant certains inconvénients importants qui empêchent de les laisser actifs par défaut. Leur principe est incompatible avec le système de transactions de DRIVER. Il est troublant par exemple qu'un accès en lecture dans la base se traduise automatiquement par l'écriture d'un champ dans l'environnement.

Parfois, des objets sont dépendants les uns des autres du fait de l'existence d'une relation liant certains de leurs champs. Si la consultation dans la base d'un champ prenant part à cette relation se traduisait par sa mise-à-jour brutale dans l'environnement, la cohérence du groupe d'objets pourrait être anéantie.

2.3.4.3 Objets jumeaux et raisonnement

Certains générateurs de systèmes experts comme SMECI disposent d'objets réversibles. Voyons en quoi les jumeaux présentent un réel intérêt pour ce type de système.

Retraçons d'abord rapidement le principe général de SMECI.

Le processus de conception, sur lequel est basé le raisonnement de SMECI, consiste à assembler des objets élémentaires qui peuvent être soit entièrement connus à l'avance, soit connus de manière partielle. Les paramètres inconnus sont alors déterminés par l'expert à

l'aide de règles. A l'aide de la connaissance dont il dispose, SMECI génère en parallèle les différentes solutions compatibles avec les spécifications et contraintes données au départ [Nev86].

Le moteur travaille sur des états qui représentent la solution en cours d'élaboration. Un état est donc caractérisé par les objets du processus de conception, plus ou moins précis selon l'état d'avancement de la solution.

Le moteur développe ainsi un arbre d'états (figure 2.6) où chaque branche correspond à une variante envisagée, où les nœuds le long de chaque branche sont les états de plus en plus précis par lesquels est passé le moteur. Une fois cet arbre complet, la feuille de chaque branche est soit une solution (feuille $n^o 402$ de notre exemple), soit un état qui a été reconnu incohérent par une règle de contradiction (feuilles $n^{os} 76, 79$, etc...), soit un état bloqué sur lequel aucune règle ne peut s'appliquer.

Au cours du raisonnement, lorsque, pendant la construction de l'arbre des états, le moteur reconnaît son état courant comme étant une feuille, il revient en arrière d'état en état jusqu'au premier où il lui reste au moins une variante à explorer. Il choisit la plus prometteuse par rapport à une fonction d'évaluation et la développe à son tour.

Quand on souhaite exploiter des objets relationnels dans le cadre du raisonnement apparaît le problème posé par la réversibilité des objets de l'environnement.

Si l'on supposait l'absence du mécanisme des objets jumeaux se poserait le problème de la non-réversibilité des données relationnelles dans la base. En effet, les SGBD relationnels ne gèrent pas en standard de liste, administrée par datation ou numéro de version, des valeurs successivement prises par un attribut au cours du temps. En cas de retour-arrière de SMECI, apparaîtrait la difficulté de remettre à jour, dans l'état à partir duquel repartirait le raisonnement, les objets relationnels qui auraient été directement modifiés dans la base au cours de l'exploration de la branche abandonnée. Une différence essentielle s'établirait au sein de la couche objet virtuelle entre les objets SMECI présents dans l'environnement qui sont réversibles et les objets relationnels qui sont incapables, en cas de retour-arrière du système, de retrouver leurs valeurs originelles.

Les objets jumeaux évitent cette distinction et conservent à la couche objet son homogénéité. Quand un objet relationnel est consulté par le système expert, DRIVER génère automatiquement le jumeau qui est un objet SMECI à part entière et le lui substitue pour la suite du raisonnement. Le jumeau comme tout objet SMECI est réversible. Il peut évoluer au fil du raisonnement et retrouver ses anciennes valeurs en cas de retour-arrière, sans risquer de perturber la cohérence de la base de données. Si le moteur revient en arrière vers un état antérieur à la création de ce jumeau, le masquage disparaît et c'est l'objet relationnel qui est à nouveau éventuellement unifié dans la poursuite du raisonnement.

2.4 Exploitation avancée des connexions multiples

Plusieurs schémas peuvent être définis, associant chacun une représentation objet à des données relationnelles.

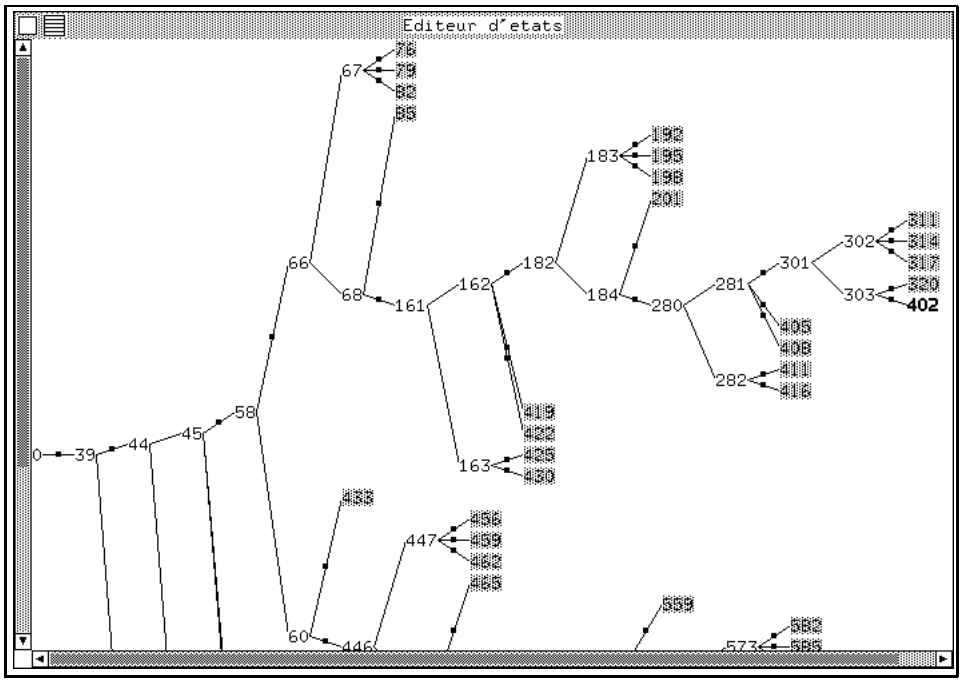

Figure 2.6 : Exemple d'arbre d'états de SMECI

2.4.1 Vers des objets de persistance distribuée

Aucune cohérence entre schémas n'est gérée par DRIVER. Lors de la construction d'un schéma, aucune vérification de compatibilité n'est effectuée entre les nouvelles déclarations et les autres schémas déjà existants.

Il est ainsi possible d'accorder la persistance à une classe dans un schéma en l'associant à une certaine table, puis de l'accorder dans un second à une sous-classe de la première en l'associant à une autre table. Cette redéfinition de la correspondance d'une classe au niveau d'une sous-classe est interdite en DRIVER au sein d'un même schéma (cf. §3.2.2).

Cette absence de contrôle peut être exploitée pour sauver par ce biais un objet en plusieurs parties, chacune dans une base différente. Dans le cas classe/sous-classe évoqué ci-dessus, les champs définis au niveau de la mère sont écrits dans une base, ceux qui sont locaux à la sous-classe sont écrits dans une autre. D'une façon plus générale, certains champs d'une classe peuvent être décrits dans un schéma, d'autres dans un autre, etc... Lors de chaque validation de transaction, selon le schéma courant sont écrits dans la base tels ou tels groupes de champs.

Ces objets relationnels *distribués* ont une identité propre dans chaque base. Chacune de leurs références est donc constituée d'un ensemble de couples schéma de correspondances - clé dans la base.

Le défaut d'objet, "objet-lien" vers un objet relationnel, n'est pas tout-à-fait adapté à référencer un tel objet relationnel distribué dans la version actuelle de DRIVER. En effet, il ne connaît qu'un seul schéma de correspondances et qu'une clé dans la base associée au

schéma. Cependant, étendre le système serait aisé. L'accès à un champ d'un objet se ferait simplement en recherchant la classe de l'objet et le champ dans les schémas référencés par le défaut d'objet. Une fois le champ retrouvé[5], le schéma qui le contient déterminerait le canal de communication, la base et la référence de l'objet dans cette base. La suite de l'opération, notamment la construction de la requête resterait la même.

Dans la version actuelle de DRIVER, un défaut d'objet pourrait référencer un objet relationnel distribué à condition, lors de l'accès aux champs, de préciser au défaut d'objet le schéma dans lequel est décrite la correspondance du champ concerné. Un mauvais trait est que, lors du déclenchement de la construction d'un objet jumeau, seuls seraient chargés les champs de l'objet définis dans le schéma référencé par le défaut d'objet.

En l'état actuel des choses, ce genre de manipulation d'objets à persistance distribuée est fortement déconseillé à un utilisateur non avancé.

2.4.2 Vers des représentations multiples contextuelles

Du fait de l'indépendance complète des schémas, aucune information issue d'un autre schéma ne peut être exploitée lors de la construction de l'un d'entre eux. Il est par exemple impossible de définir des champs de type objet destinés à recueillir des instances d'une classe déclarée dans un autre schéma. On peut cependant faire la remarque que ce genre d'opération n'aurait de sens que si les deux schémas étaient définis sur la même base. En effet, la correspondance d'un champ objet est une chaîne de jointures (cf. §3.2.3.2). Si les deux schémas étaient définis sur des bases différentes, des requêtes adressées au SGBD effectueraient des jointures entre tables de bases différentes, ce qui serait insensé.

Par contre, plusieurs schémas peuvent être définis pour une même base. Cette perspective peut être intéressante quand il s'agit de prendre en compte des données relationnelles dans l'environnement. On peut en effet définir plusieurs représentations objet pour de mêmes données, chacune étant exprimée dans un schéma différent.

Ces différentes représentations objet co-existent dans le même environnement. Leur intérêt est qu'elles sont toutes pertinentes pour une certaine tâche, dans un certain contexte. Elles permettent d'accéder à de mêmes données de façons multiples et optimales en fonction du contexte. De cette façon, les champs peuvent directement contenir l'information dans la forme la plus adaptée. Le filtrage des objets relationnels correspondants est lui-même plus efficace, les requêtes générées et adressées au SGBD étant optimisées.

Un autre intérêt de ces représentations objet multiples est la possibilité d'avoir simultanément dans l'environnement différents arbres d'héritage établis selon des critères différents et qui représentent de mêmes données. Les différentes classifications des êtres vivants sont des exemples de telles hiérarchies, indépendantes mais basées sur les mêmes données. La classification cladistique regroupe les êtres vivants selon leurs caractéristiques physiques communes tandis que la classification phénétique les regroupe en fonction de leurs similarités génétiques et hématologiques.

[5] Ces idées n'ayant pas été implantées, les problèmes de conflit de nom de champ n'ont pas été étudiés.

Là encore, les descriptions multiples de bases doivent être utilisées avec prudence. En effet, elles court-circuitent les vérifications effectuées lors de la construction d'un schéma, qui interdisent la représentation redondante d'informations dans l'environnement. Certaines données de la base peuvent par cette méthode se retrouver plusieurs fois en mémoire dans des objets de classes différentes. Si elles sont exploitées en simple consultation, elles ne sont la source d'aucun problème particulier. Si elles peuvent être modifiées et mises à jour dans la base, des incohérences peuvent être introduites.

Chapitre 3

LE SCHÉMA DE CORRESPONDANCES

3.1 Deux modèles de données différents

Les modèles relationnel et objet présentent, dans leur richesse d'expression, une différence essentielle qu'il est utile de souligner.

Nous allons examiner dans un premier temps le modèle relationnel puis le modèle objet que supporte DRIVER; nous verrons ensuite comment on peut les rapprocher afin d'être capable d'exprimer ou de traduire des données dans l'un comme dans l'autre.

3.1.1 Le modèle relationnel

Le modèle *relationnel* est dû à E. F. Codd [Cod70]. Les structures de données sont des *relations* (au sens mathématique) qu'on peut représenter par des tables à un instant donné. Les colonnes, caractérisées par un nom et un type simple, sont appelées des *attributs*, et les lignes, enregistrements successifs et occurrences de relations, sont appelées des *n-uplets*. Les types d'attributs sont toujours atomiques; les plus communément disponibles sont *entier*, *réel* et *chaîne de caractères*, mais il en existe d'autres, comme les types *date*, *argent* qui, cette fois, ne sont pas fournis par tous les SGBD.

Toute relation possède au moins une clé constituée d'un ou plusieurs attributs. La valeur d'une clé de n-uplet l'identifie de manière unique dans l'ensemble des n-uplets de la relation considérée.

Attribut	**type**	**longueur**	**Description de l'attribut**
empno	entier	2	Numéro de l'employé, *clé de la relation emp*
ename	caractères	20	Nom de l'employé
fname	caractères	20	Prénom
job	caractères	20	Emploi dans l'entreprise
mgr	entier	2	Numéro du chef de l'employé
sal	réel	8	Salaire
com	réel	8	Commission, pour certains employés
deptn	entier	2	Numéro du département de l'employé

Figure 3.1 : Structures relationnelles : `emp`, table des employés

La figure 3.1 présente un exemple de structure de table relationnelle, composée de différents attributs. La figure 3.2 montre et permet de comprendre aisément comment les données sont organisées sous forme de tableaux.

empno	ename	fname	job	mgr	sal	com	deptn
7839	king	paul	president		5000.		10
7566	jones	eric	manager	7839	2975.		20
7698	blake	harold	manager	7839	2850.		30
7782	clark	john	manager	7839	2450.		10
7902	ford	john	analyst	7566	3000.		20
7788	scott	pit	analyst	7566	3000.		20
7499	allen	jack	salesman	7698	1600.	300.	30
7521	ward	peter	salesman	7698	1250.	1400.	30
7654	martin	georges	salesman	7698	1250.	1400.	30
7655	james	peter	salesman	7698	1350.	1000.	30
7934	miller	paul	clerk	7782	1300.		10
7329	smith	john	clerk	7902	800.		20

Figure 3.2 : Données relationnelles : contenu de la table `emp`

L'intérêt du modèle relationnel provient de ses bases théoriques solides. Il est aussi de proposer des structures de données simples qui se construisent à partir de la théorie des ensembles.

Un certain nombre d'opérateurs s'appliquant aux relations permettent de définir des données, de les mettre à jour et de les rechercher. Ils constituent *l'algèbre relationnelle*. Chaque opérateur s'applique à des relations et produit comme résultat une relation. Ils comprennent les opérateurs classiques de la théorie des ensembles, comme l'*union*, l'*intersection* et la *différence*, plus d'autres permettant de composer des sous-ensembles de relations, comme la *projection*, la *restriction*, la *jointure* et la *division*.

Des langages construits sur l'algèbre relationnelle sont proposés pour définir et manipuler les données (DDL et DML[1]). Basés sur la logique des prédicats, ces *langages de requêtes* permettent d'exploiter les données relationnelles en se désintéressant complètement de leur implantation. Il existe un standard qui est *SQL*[2] [ANSI86, Mir90]. Ce langage a d'ailleurs fortement contribué au succès du modèle relationnel. Un des points forts de SQL est de fournir une interface uniforme aux administrateurs, programmeurs d'applications et utilisateurs finaux pour la définition, le contrôle et la manipulation des données.

[1] Data Definition Language, Data Manipulation Language
[2] Structured Query Langage.

3.1.2 Le modèle objet de DRIVER

3.1.2.1 Caractéristiques générales

Le modèle objet proposé par DRIVER est classique en IA.

Un objet, dans le sens objet complexe, désigne un ensemble de données structuré, identifié par une référence unique. L'objet est instance d'une classe, la classe étant la description purement statique d'un ensemble possible d'objets.

Les classes s'organisent en hiérarchies où les racines, c'est-à-dire les classes n'étant pas issues de classes plus générales, sont appelées les *classes principales*. Il n'y a pas nécessairement de classe principale unique. Une classe est définie par un ensemble de champs, qui sont les champs de la classe mère dont elle hérite, plus les champs locaux qui lui sont propres (Figures 3.3 et 3.4).

Dans l'implantation actuelle du système, l'héritage est simple, c'est-à-dire qu'une classe ne peut hériter directement que d'une seule autre classe.

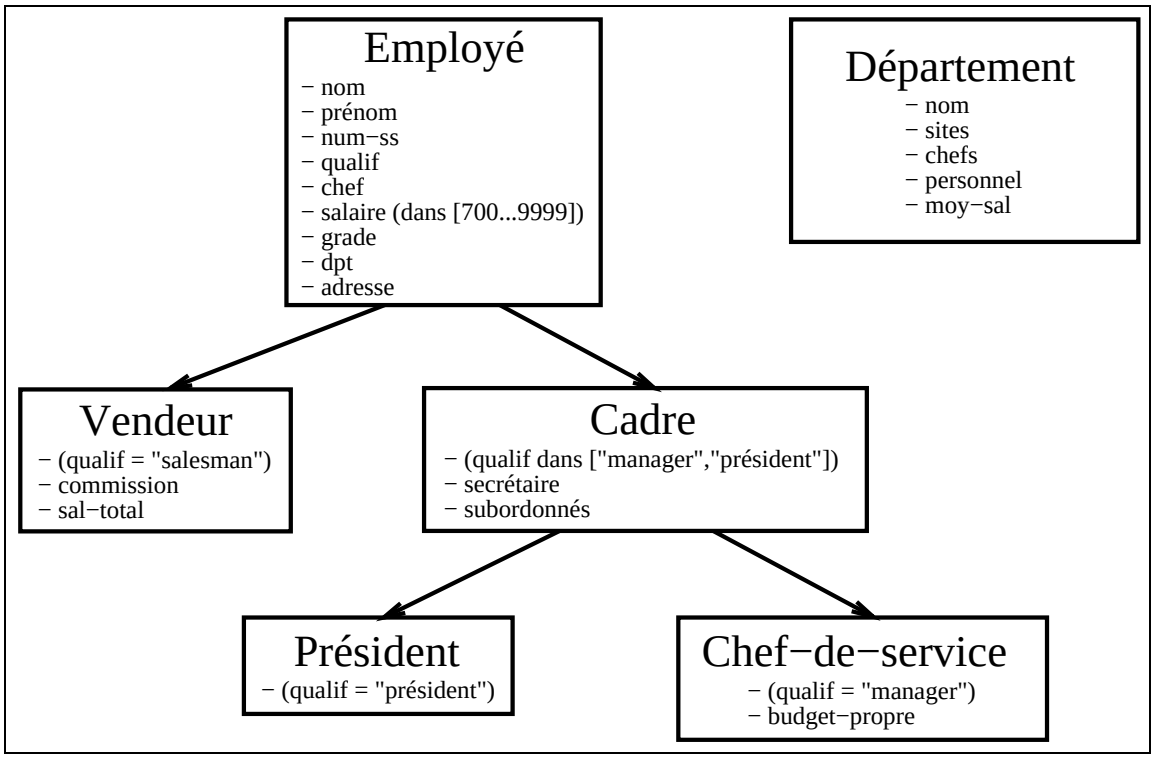

Figure 3.3 : Exemples de classes

Un champ, selon son type, est structuré en un certain nombre de facettes précisant la nature de la valeur qu'il contient; par exemple, le nom du champ, son type, la classe de l'objet dans le cas d'un champ de type objet, la définition des éventuelles contraintes,

Objet de classe Employé	
nom	miller
prénom	paul
num-ss	787309
qualif	clerk
chef	Object("clark")
salaire	(1300.
grade	2
dpt	Object("accounting")
adresse	Object("miller-addr")

Objet de classe Vendeur	
nom	ward
prénom	peter
num-ss	673595
qualif	salesman
chef	Object("blake")
salaire	1250.
grade	2
dpt	Object("sales")
adresse	Object("ward-addr")
commission	1400.
sal-total	2650.

Objet de classe Président	
nom	king
prénom	paul
num-ss	786259
qualif	president
chef	()
salaire	5000.
grade	5
dpt	Object("accounting")
adresse	Object("king-addr")
secrétaire	Object("bush")
subordonnés	(Object("jones") Object("blake") Object("clark"))

Figure 3.4 : Exemples d'objets

etc...

Des contraintes de toutes natures (cf. §3.1.3.2) peuvent être posées sur les champs atomiques et sont définies sous forme de prédicats dont la valeur de vérité indique si la contrainte est respectée ou non. L'application de ces prédicats aux valeurs candidates pour le champ permet de déterminer si elles vérifient ou non la contrainte. Ces contraintes peuvent être redéfinies au niveau des sous-classes et permettent de préciser de plus en plus strictement les valeurs autorisées pour chaque champ.

3.1.2.2 Persistance des objets

En fait, parler *DU* modèle objet est en réalité un peu inexact puisqu'en fait, DRIVER ne définit et n'offre pas réellement un nouveau et n-ième modèle objet. Il ajoute aux modèles existants, en particulier au modèle du client (cf. §3.1.2.3), une nouvelle caractéristique qui leur est complètement orthogonale, à savoir la persistance.

Cependant, cette persistance n'est accordée qu'à un certain nombre de propriétés du

modèle client qui correspondent aux concepts reconnus par DRIVER.

Bien entendu, il n'est pas possible de reconnaître l'ensemble des caractéristiques de tous les modèles objet. Un choix a été effectué. Afin de préserver la réflexivité du système et la symétrie dans l'utilisation de DRIVER (attribution de la persistance à des classes ou définition de classes destinées à la représentation de tables relationnelles), il n'est exploité des modèles objet que les propriétés qui peuvent avoir un sens vis-à-vis du modèle relationnel et de son rapprochement du monde objet.

Il pourra être possible de gérer d'autres caractéristiques, notamment parmi les données structurelles, mais cela devra être fait explicitement en définissant un schéma de correspondances entre elles et de nouvelles tables "système" à définir.

On parle malgré tout du modèle objet de DRIVER, car la description des informations concernant les classes à rendre persistantes ou à associer à des tables relationnelles s'effectue par le biais d'un langage à objets particulier à DRIVER, doté d'un certain nombre de caractéristiques, d'un formalisme et d'une syntaxe qui lui sont propres.

3.1.2.3 Une interface fonctionnelle objet

DRIVER s'utilise simplement avec différents modèles objet. Il effectue toutes les opérations sur les objets au travers d'une interface fonctionnelle qui invoque des méthodes sur des objets modèles. Pour avoir accès à un nouveau modèle, il suffit donc de définir un nouveau jeu de méthodes pour ce modèle.

Les envois de messages sont effectués sur des objets de classe Driver-Object-model. Prendre en compte un modèle objet peut consister à redéfinir les méthodes de cette classe. On peut aussi, et c'est préférable, créer des sous-classes à la classe Driver-Object-model et redéfinir les méthodes pour chacune d'entre elles (figure 3.5). De cette façon, différents modèles peuvent être utilisés dans un même environnement. Un mécanisme de *modèle courant* permet de déterminer à tout moment celui à utiliser.

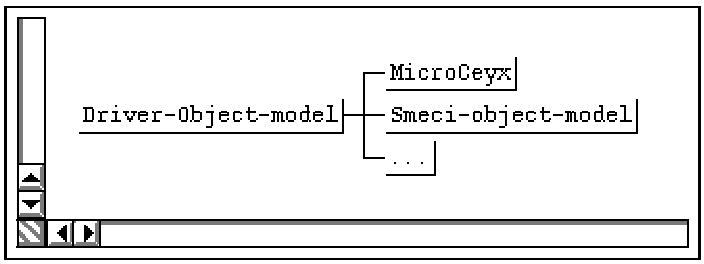

Figure 3.5 : Hiérarchies de classes de modèles objet

Pour le moment, DRIVER a été implanté en Le_Lisp v15 et, de ce fait, ne permet la prise en compte que des modèles objet disponibles dans cet environnement. Nous verrons cependant qu'aucun trait du système ne dépend spécifiquement de LISP et de son caractère interprété. DRIVER pourra par exemple être très rapidement porté en C++ pour

pouvoir être exploité par les utilisateurs de ce langage.

Le modèle objet *MicroCeyx* est disponible dans l'environnement DRIVER de base. La classe MicroCeyx, sous-classe de Driver-Object-model, et son jeu de méthodes associé y ont en effet été définis. MicroCeyx est le modèle objet utilisé par défaut. Les méthodes de la classe Driver-Object-model appellent en effet explicitement celle de la classe MicroCeyx.

Le modèle objet de SMECI est également disponible. La classe Smeci-object-model et ses méthodes ont été décrites. À titre d'exemple, les méthodes définies pour les modèles objet MicroCeyx et SMECI sont données en annexe B.4.2.

L'ensemble des méthodes permettent à DRIVER d'accéder à un certain nombre de fonctionnalités de base communes à tous les modèles objet.

Ces méthodes peuvent être facilement définies pour tout modèle, même si tel ou tel modèle objet n'est pas forcément capable d'exploiter la totalité des propriétés du modèle proposé par DRIVER.

Par exemple, le modèle objet de DRIVER gère la pose de contraintes sur les champs. MicroCeyx ne permet pas de le faire. Les contraintes ne peuvent donc pas être déclarées et exploitées en MicroCeyx, même si, dans le même temps, DRIVER en tire complètement profit.

Les méthodes composant l'interface fonctionnelle objet d'un modèle sont les suivantes :

- **Création d'une instance de classe**. Cette méthode a pour but de permettre à DRIVER de créer une instance de classe dans le modèle objet client et de récupérer cette instance.

  Comme pour chaque méthode de l'interface fonctionnelle objet, le premier argument est l'objet "modèle objet" auquel est adressé l'envoi de message de création d'instance de classe. Les suivants sont spécifiques à chaque méthode. Pour celle-ci, il n'y en a qu'un. Il s'agit de l'objet qui représente la classe en question en interne à DRIVER. Il contient toutes les informations la concernant. La classe de cet objet "classe", de nom Driver-class, est décrite en §3.3.2.3.

- **Lecture de la valeur d'un champ**. Cette méthode a pour but de permettre à DRIVER d'accéder à la valeur du champ d'un objet.

  Les arguments de cette méthode sont l'objet "modèle objet", le nom du champ et l'objet duquel on souhaite extraire l'information. Bien entendu, la méthode doit retourner la valeur du champ.

- **Affectation d'une valeur à un champ**. En complément de la fonction précédente, DRIVER a besoin de pouvoir affecter une (nouvelle) valeur au champ d'un objet. Il le fait par cette méthode.

  Ses arguments sont, en plus de ceux de la méthode précédente, la valeur à affecter au champ désigné.

- **Envoi de message**. Cette méthode permet à DRIVER d'envoyer un message à un objet.

  Ses arguments sont l'objet "modèle objet", le nom du message, l'objet auquel s'adresse le message, et la liste des éventuels arguments du message.

- **Test de l'existence d'une classe**. DRIVER a besoin de pouvoir tester si une classe a déjà été définie dans le modèle objet client.

  Les deux arguments de cette méthode sont l'objet "modèle objet" et l'objet système DRIVER représentant cette classe.

  Le résultat retourné doit être booléen.

- **Test de l'appartenance d'une instance à une classe**. DRIVER a également besoin de pouvoir tester si un objet est instance d'une classe au sens large, c'est-à-dire instance de cette classe ou de l'une de ses sous-classes. Cette méthode doit pouvoir le renseigner à ce sujet.

  Ses arguments sont l'objet "modèle objet", l'objet testé et l'objet DRIVER représentant la classe.

  Encore une fois, le résultat attendu est booléen.

Trois autres méthodes complètent l'interface fonctionnelle objet. Permettant à DRIVER d'utiliser des fonctionnalités de modèle objet beaucoup moins répandues que les précédentes, elles risquent de ne pas pouvoir être exprimées pour certains modèles objet.

- **Création de classe**. Cette méthode doit permettre à DRIVER de *créer une classe* dans le modèle objet client.

  Outre l'objet "modèle objet", elle nécessite en argument l'objet représentant la classe considérée. Le corps de la fonction doit consister à reformuler sous forme d'appels aux primitives du modèle client la création de classe, ses champs, les contraintes qui y sont associées, toutes données communes à ses futures instances.

  Cette méthode ne peut pas être définie pour la plupart des modèles objet des langages compilés. Ce n'est en fait pas très important car elle est invoquée uniquement si l'utilisateur souhaite utiliser les déclarations de classes en DRIVER pour créer, par la même opération, ces mêmes classes dans le modèle objet.

- **Mutation de classe**. Celle-ci doit permettre à DRIVER de *muter un objet d'une classe vers une autre.*

  Ses arguments sont l'objet "modèle objet", l'objet DRIVER représentant la future classe de l'objet à muter, et l'objet en question.

  Les modèles objet de Le_Lisp v15 offrant la mutation de classe, nous avons utilisé cette fonctionnalité pour optimiser le chargement et le traitement des objets. Certains algorithmes devront donc être revus lorsque nous implanterons DRIVER dans un environnement dont les modèles objet n'en disposent pas.

En fait, la mutation de classe est très utile quand on souhaite partager des informations de la base relationnelle sous-jacente entre plusieurs utilisateurs qui sont autorisés à la modifier, et donc à faire évoluer les classes des objets. Ceci permet par exemple d'utiliser la base pour pouvoir échanger ou partager certains objets entre systèmes indépendants.

Elle est également très pratique pour la gestion des défauts d'objet. Grâce à elle, le chargement d'un objet représenté par un défaut d'objet commence simplement par la mutation de cet objet de la classe des défauts d'objet vers sa classe principale dans la base. Cette opération est bien sûr suivie de la construction des valeurs des champs de l'objet.

- **Instances d'une classe**. La dernière méthode est utilisée par DRIVER pour récupérer l'*ensemble des instances d'une classe donnée*. Instances doit être pris au sens large puisque ce terme désigne également les instances des sous-classes.

  Elle a deux arguments qui sont l'objet "modèle objet" et l'objet DRIVER représentant la classe.

  Cette méthode n'est pas essentielle pour le fonctionnement du système.

  Elle n'est utilisée que dans le module DRIVER dédié à l'écriture des objets persistants dans la base de données, pour traiter un cas particulier de violation de contraintes d'intégrité sur les clés. Elle permet de déterminer l'ensemble des instances de la classe vérifiant correctement ces contraintes, et proposer à l'utilisateur de choisir parmi elles l'objet qui remplacera celui qui provoque l'incohérence.

  Ce cas de figure, extrêmement rare, n'apparaît que dans certaines formes exceptionnelles de schéma de correspondances et peut être souvent évité en modifiant le schéma à l'origine du problème. Il est examiné en détail en §4.2.2.2.

  Quand la fonction n'est pas exprimable avec le modèle objet client utilisé, il suffit de la définir de telle manière qu'elle provoque explicitement une erreur expliquant l'incohérence des données. DRIVER détecte le déclenchement de l'erreur et annule toute modification de la base depuis la dernière mise à jour validée, évitant ainsi de laisser la base dans un état indéterminé et inconsistant.

  L'utilisateur n'a plus alors qu'à corriger l'objet pris en défaut avant de re-provoquer une nouvelle mise à jour dans la base.

Un exemple de jeu de méthodes est donné en annexe §B.8; il permet d'utiliser DRIVER avec le modèle objet de SMECI.

3.1.3 Descriptions relationnelle et objet en DRIVER

Ces descriptions n'ont pas pour but de créer, d'un côté les tables correspondantes dans la base de données, de l'autre, les classes dans le modèle objet client, même si des primitives DRIVER permettent par ailleurs de le faire. Elles ont plutôt valeur de déclarations

d'existence pour permettre ensuite le rapprochement des structures relationnelles et objet décrites.

Ainsi, il convient de ne décrire que les relations et les classes qui présentent un intérêt pour la couche objet virtuelle persistante.

Il n'est pas non plus utile de décrire complètement telle ou telle relation dont on sait pertinemment que l'on n'en exploitera qu'une partie. De la même manière, il est sans intérêt de décrire en DRIVER les champs d'une classe auxquels on n'a pas l'intention d'accorder la persistance, toujours dans le cas, est-il besoin de le répéter, où ces déclarations ne sont pas utilisées pour créer les structures dans les mondes correspondants.

Il appartient au concepteur de sélectionner les informations dont il a besoin et d'établir ensuite les correspondances qui lui paraissent appropriées.

3.1.3.1 Description du schéma relationnel en DRIVER

La description d'un schéma relationnel se fait simplement en DRIVER en explicitant les tables en faisant partie, leurs attributs, types et contraintes d'intégrité sémantiques.

L'exemple suivant (listing 1) illustre les déclarations de trois tables d'un même schéma, `emp` décrite dans la figure 3.1, `person` contenant des renseignements personnels sur les individus, et `dept` la table des départements.

Listing 1. Description de relations en DRIVER

```
(driver-deftable emp
   (empno  integer 2 keypart)
   (ename  string 20 not-null)
   (fname  string 20 ())
   (job    string 20 ())
   (mgr    integer 2 ())
   (sal    float   8 ())
   (com    float   8 ())
   (deptn  integer 2 ()))

(driver-deftable person
   (pname    string 20 keypart)
   (fname    string 20 keypart)
   (ssnum    string 15 unique)
   (position string 10 ())
   (phone    string 12 ())
   (car      integer 2 ()))

(driver-deftable dept
   (deptno integer 2 keypart)
   (dname  string 20 ()))
```

`emp`, `person` et `dept` comprennent (au moins) les huit, six et deux attributs énoncés.

Chaque attribut est décrit par son nom, son type, sa taille dans le type, plus un mot réservé exprimant les éventuelles contraintes d'intégrité auquel il est éventuellement soumis (l'absence de valeur définie est représentée par le symbole ()).

Par défaut, la liste des types disponibles est integer, float et string. Cette liste peut être complétée en fonction des besoins et du SGBD utilisé.

Les contraintes d'intégrité imposables aux attributs sont classiques et standard :

- La première identifie l'attribut comme faisant partie de la clé de la relation. Elle se définit à l'aide du mot réservé keypart.
- La seconde interdit l'absence de valeur pour l'attribut en question dans un n-uplet. Elle se définit à l'aide du mot réservé not-null.
- La dernière interdit qu'un attribut ait même valeur dans deux n-uplets différents. Elle se définit à l'aide du mot réservé unique.

Dans l'exemple précédent, la clé de la relation `emp` est un attribut `empno`, celle de la relation `person` est constituée des deux attributs `pname` et `fname`. Une contrainte a été posée sur `emp.ename` précisant qu'un n-uplet de la relation doit forcément avoir un nom spécifié, une autre sur `person.ssnum` imposant cette fois que deux personnes ne peuvent avoir le même numéro de sécurité sociale.

3.1.3.2 Le langage objet de DRIVER

Décrire une classe consiste en DRIVER à :

- la nommer,
- la positionner dans les arbres de classes en précisant le nom de sa classe mère,
- expliciter les champs qui lui sont associés.

L'exemple suivant (listing 2) illustre la déclaration des classes `Employé`, `Vendeur` et `Département`, présentées dans la figure 3.3.

Listing 2. Description de classes en DRIVER

```
(driver-defclass Employe ()
   (nom            atom   symbol (localp ()))
   (prenom         atom   symbol)
   (num-ss         atom   symbol)
   (qualif         atom   symbol)
   (chef           object Employe)
```

```
   (responsable-de object2)
   (salaire        atom   float
         (constraint (lambda (s) (and (>= s 700.) (<= s 9999.)))))
   (grade          atom   fix (initval 0))
   (dpt            object Departement)
   (adresse        object Adresse)
   (tel-prive      atom   symbol)
   (vehicule       object Vehicule))

(driver-defclass Vendeur Employe
   (qualif (constraint (lambda (qual) (eq qual 'salesman))))
   (situ-famille atom      symbol)
   (commission   atom      float)
   (sal-total    monotexpr float))

(driver-defclass Departement ()
   (nom       atom       symbol (localp ()))
   (produits  object2set)
   (sites     atomset    symbol)
   (chefs     ordobjset  Cadre)
   (personnel objectset  Employe)
   (moy-sal   multitexpr float))
```

Le nom de la classe mère est précisé après le nom de la classe. S'il n'y a pas de classe mère, le symbole () exprime cette absence. `Employé` et `Département` n'héritent d'aucune autre classe connue de DRIVER. En tant que sommets d'arbres de classes, elles sont dites *classes principales*. `Vendeur` par contre est une sous-classe d'`Employé`.

La déclaration en DRIVER d'une classe n'héritant d'aucune classe mère n'implique pas que ce soit forcément le cas dans le modèle objet client. Cela exprime juste le fait que la classe en question n'hérite, dans le modèle objet client, d'aucune classe qui soit persistante du fait du même schéma de correspondances.

Les champs déclarés appartiennent en général à la classe correspondante[3]. Ils sont décrits par leurs noms, leurs types, certaines informations complémentaires qui varient avec le type du champ, plus un certain nombre de paramètres optionnels possédant, par défaut, une valeur déterminée.

Un certain nombre de types de champ sont proposés :

- **Champ à valeur atomique**. Désigné par le mot réservé `atom`, ce type regroupe les champs destinés à contenir une information élémentaire. Outre le type DRIVER du champ (atom) doit être précisé le type des futures valeurs contenues :

[3] Ce n'est pas toujours le cas, voir l'option `localp`.

- symbol. Le champ contiendra des symboles. Ce type est principalement intéressant pour une utilisation de DRIVER dans un environnement LISP. La première version de DRIVER a été écrite en Le_Lisp [Cha89];
- string. Contenu de type "chaîne de caractères";
- fix ou integer. Contenu de type entier;
- float. Contenu de type réel;
- (). Type du contenu inconnu ou "autre".

La gestion de l'absence de type telle qu'elle est faite ci-dessus peut être insuffisante lors de l'utilisation d'un modèle objet client fortement typé. De nouveaux types de valeur peuvent être rajoutés en fonction des besoins et du modèle objet client utilisé, à l'aide de la primitive DRIVER prévue à cet effet.

Les exemples de déclaration de champs atomiques (`nom`, `salaire`, `grade` d'`employé`, etc...) abondent dans la description des trois classes précédentes. Certains sont précisés d'informations optionnelles qui sont explicitées ci-après.

- **Champ ayant pour valeur un ensemble de valeurs atomiques**. Ce type de champ impose à chacun des éléments de l'ensemble d'être d'un même type à préciser. DRIVER ne gère en effet pas directement les ensembles hétérogènes. Nous verrons qu'il est malgré tout possible de le faire de façon artificielle en utilisant les méthodes de filtrage de champ (cf. §3.2.5.3). Ce type de champ est désigné par le mot réservé atomset.

  Le champ `sites` de la classe `Département` qui contient l'ensemble des noms des sites où le département a une antenne est un exemple de tel champ.

- **Champ ayant pour valeur un ensemble ordonné de valeurs atomiques**. Comme pour le type précédent, chaque élément de l'ensemble est d'un même type à préciser. Ici, en plus, l'ordre des éléments est important et doit être persistant lui aussi. Ce type de champ est désigné par le mot réservé ordatomset.

  Les classes proposées dans l'exemple ci-dessus ne comprennent pas de champ de ce type. On pourra néanmoins facilement déduire des exemples présentés une illustration de ce type de champ. Ainsi, le champ `sites` de la classe `Département` aurait pu être défini dans ce type, si l'ordre des éléments avait présenté un intérêt pour l'application. La déclaration de ce champ aurait alors été :

  ```
  (sites ordatomset symbol)
  ```

- **Champ à objet de classe connue (persistante)**. Ce type caractérise les champs destinés à référencer un objet dont la classe, ou au moins la classe principale, est connue. Bien entendu, le nom de cette classe, voire de la classe principale, doit être précisé. Le champ peut contenir toute instance de la classe en question ou de l'une de ses sous-classes. Le type est désigné par le mot réservé object.

  Les champs `chef` et `dpt` de la classe `Employé` sont des exemples de champ objet à classe connue.

- **Champ à objet de classe quelconque (mais persistante)**. Ce type caractérise les champs destinés à contenir un objet dont la classe ne peut être précisée. Il est désigné par le mot réservé object2.

  Le champ `responsable-de` d'`Employé` est un champ de ce type. Un employé peut en effet être dans notre application responsable d'un département (objet instance de la classe principale `Département`), d'un projet (objet instance de la classe principale `Projet`) ou d'un produit (objet pouvant, selon sa nature, être instance de telle ou telle classe principale).

- **Champ ayant pour valeur un ensemble d'objets de même classe (persistante)**. Ce type caractérise les champs destinés à contenir un ensemble d'objets dont la classe, ou au moins la classe principale, est commune et connue. Il doit également ici être précisé le nom de la classe des objets qui y seront référencés ou, à défaut, le nom de leur classe principale. Les objets pourront être instances de la classe précisée, ou de l'une de ses sous-classes. Ce type est désigné par le mot réservé objectset.

  Un champ de ce type est défini dans la classe `Département`. Il s'agit en l'occurrence du champ `personnel`, destiné à recueillir l'ensemble des employés de chaque département.

- **Champ ayant pour valeur un ensemble ordonné d'objets de même classe (persistante)**. Comme pour le type précédent, chaque objet élément de l'ensemble est (au moins) instance d'une même classe à préciser. Ici, en plus, l'ordre des objets est important et doit également être persistant. Ce type de champ est désigné par le mot réservé ordobjset.

  Le champ `chefs` de la classe `Département` est un exemple de champ de ce type.

- **Champ ayant pour valeur un ensemble d'objets de classes quelconques (mais persistantes)**. Ce type caractérise les champs destinés à contenir un ensemble d'objets de classes différentes et indépendantes, c'est-à-dire héritant de classes principales différentes. Ce type de champ est désigné par le mot réservé object2set.

  Le champ `produits` de la classe `Département` correspond à ce type. Il est destiné à recueillir l'ensemble des produits dont le département est responsable.

- **Champ ayant pour valeur un ensemble ordonné d'objets de classes quelconques (mais persistantes)**. Comme le type précédent, ce dernier caractérise les champs destinés à contenir un ensemble d'objets de classes différentes et indépendantes. Ici, en plus, l'ordre des objets est important et doit également être persistant. Ce type de champ est désigné par le mot réservé ordobj2set.

- **Champ à valeur atomique destiné à recueillir le fruit d'un calcul sur la base de données**. Cette catégorie de champ ne présente en fait un intérêt que dans la perspective où l'on souhaite rapprocher la classe qui le contient avec des structures relationnelles. Comme pour les champs atomiques doit être précisé le type des futures

valeurs contenues : `symbol`, `string`, `fix`, `float` ou `()` (voir description ci-dessus du champ atomique).

Bien évidemment, cette catégorie de champs ne peut être consultée qu'en lecture. Les calculs sur la base qu'on leur associe sont décrits lors de la phase de définition des correspondances.

On distingue en fait deux types de champs, selon la nature du calcul qu'on leur associe. Ces deux types découlent de deux types de calculs possibles :

- Le premier permet d'effectuer le calcul sur un seul n-uplet relationnel, généralement la correspondance principale de l'objet considéré, celui dont le champ calculé dont on parle sera affecté. Somme ou différence d'attributs du n-uplet sont des exemples d'expressions possibles. Ce type de champ est désigné par le mot réservé monotexpr.

  Un champ de ce type a été défini dans la classe `Vendeur`. Il s'agit de `sal-total`, destiné à recueillir le calcul de la somme `salaire+commission` (attributs `emp.sal` + `emp.com`) pour chaque vendeur (listing 3).

- Le second permet d'effectuer le calcul sur un ensemble de n-uplets, correspondant généralement à un ensemble d'objets. Ce genre de calcul permet de réaliser par exemple une moyenne ou la somme d'un ensemble de valeurs d'un attribut. Ce type de champ est désigné par le mot réservé multitexpr.

  Ce type de champ est également illustré, cette fois au sein de la classe `Département` par l'intermédiaire du champ `moy-sal`. Ce champ est destiné à recueillir le calcul de la moyenne des salaires des employés du département.

On a vu en §2.3.4 qu'on peut accéder à la valeur du champ d'un objet de deux façons différentes depuis la couche objet virtuelle. On peut faire la remarque que l'existence de ces deux possibilités présente un intérêt particulier pour l'exploitation des champs calculés :

- **Accès du premier type**. Le champ est calculé lors de la construction de l'objet jumeau et le résultat y est stocké. Par la suite, à chaque nouvel accès du premier type, la valeur est lue dans l'objet et n'est pas recalculée.
- **Accès du second type**. La valeur du champ est recalculée à partir des données contenues dans la base. Si le jumeau de l'objet relationnel a déjà été construit et si le réflexe de mise-à-jour est actif (cf. §2.3.4.2), le champ du jumeau est actualisé avec le résultat du dernier calcul.

  Le second type d'accès est évidemment plus coûteux puisqu'il se traduit par l'envoi d'une requête au SGBD à chaque consultation du champ. Il a en retour l'avantage de renvoyer des valeurs actualisées, ce qui est intéressant quand la base est partagée entre plusieurs utilisateurs.

Lors de l'"instanciation" de l'interface fonctionnelle objet, on pourra définir les accesseurs aux champs du modèle client de telle manière qu'ils génèrent par défaut un accès du premier type, et pour les champs calculés, un accès du second.

Les paramètres optionnels de déclaration de champ

Les déclarations des champs peuvent être complétées à l'aide de paramètres optionnels. Une option se présente dans la syntaxe de notre langage sous la forme d'une liste contenant au moins deux éléments dont le premier identifie l'option et le ou les suivants donnent son expression ou sa valeur.

- **Définition de la valeur initiale d'un champ**. La valeur initiale affectée à un champ lors de la création d'un objet peut être spécifiée.

  Dans notre implantation, l'expression permettant de calculer la valeur initiale du champ est une forme Lisp qui est évaluée lors de la création de chaque instance. Elle est introduite par le mot réservé initval.

  Cette option est illustrée, dans notre application, au sein de la classe `Employé`, par la définition d'une valeur initiale pour le champ `grade`. Toute nouvelle instance d'employé se verra affecter automatiquement un grade par défaut de valeur 0.

- **Contrainte d'un champ atomique**. DRIVER ne permet actuellement de contraindre que les champs atomiques non calculés.

  Dans notre implantation, l'expression de cette contrainte est une lambda-expression à un argument qui a valeur de prédicat. Elle est appliquée à toute valeur candidate du champ; si le résultat est vrai (au sens lisp), la contrainte est vérifiée; elle ne l'est pas dans le cas contraire. L'expression est introduite par le mot réservé constraint.

  Deux contraintes ont été définies dans l'application. La première, posée sur le champ `salaire` de la classe `Employé` impose à chaque employé d'avoir un salaire compris entre $700 et $9999. La seconde contraint les instances de la classe `Vendeur`. Elle leur impose d'avoir "salesman" comme valeur du champ `qualif` qui est hérité de la classe `Employé`. On remarquera que les objets qui sont précisément instances de la classe `Employé` ont un champ `qualif` sur lequel n'est posée aucune contrainte.

- **Déclaration d'un champ délocalisé**. Il est possible, par cette option, d'apporter la persistance à un champ défini au niveau d'une classe ancêtre d'une classe persistante. Sans cette facilité, il serait impossible de rendre persistantes des informations contenues dans des instances de classes persistantes, mais stockées dans des champs hérités de classes ancêtres non persistantes.

Dans les modèles objet où toute classe hérite d'une classe système unique, "Objet" par exemple, il arrive que soient accessibles dans cette classe racine des champs destinés à recueillir des informations communes à tous les objets, notamment leurs noms. Dans la perspective de vouloir associer par la suite une classe principale à une table relationnelle, il peut être utile de faire correspondre le champ `nom` de cette classe, hérité de la classe racine, à l'attribut `nom` de la relation.

On pourrait bien entendu décider de rendre directement persistante la classe racine. Il ne faut cependant pas oublier qu'une classe principale est associée à une relation principale (cf. §3.2.2) et que toute sous-classe de cette classe est associée à la même relation principale qu'elle. Toutes les classes persistantes devraient alors être associées à la même et unique relation ce qui ne manquerait pas d'être lourd et rapidement sans intérêt.

Cette option est identifiée par le mot réservé localp. La valeur par défaut est vrai (champ local).

Dans notre application, les champs `nom` des classes `Employé` et `Département` sont déclarés non locaux, ceci indiquant qu'ils sont hérités d'une classe mère dans le modèle objet client. Toute création d'une de ces deux classes dans ce modèle client déclenchée par la primitive DRIVER adéquate se traduira donc par l'absence de création de champ `nom` local.

3.2 Mise en correspondance des deux modèles

L'intérêt de définir des correspondances entre le modèle relationnel et le modèle objet canonique défini par DRIVER réside, rappelons-le, dans le fait, d'une part, de permettre la manipulation de données relationnelles sous forme d'objets dans un environnement désigné, et, d'autre part, de pouvoir décrire des objets de ce même environnement sous forme de données relationnelles pour les sauver dans une base de données relationnelle.

La définition de correspondances permet d'établir un pont entre les deux formalismes. Les différents concepts propres à chaque modèle ont donc été rapprochés le plus "naturellement" possible conformément à leurs sémantiques.

3.2.1 D'importantes différences conceptuelles

La mise en correspondance du modèle relationnel et d'un modèle objet n'est pas tout-à-fait immédiate car le modèle objet est beaucoup plus riche que son homologue relationnel. Il implique principalement en plus les notions de hiérarchie, d'héritage et d'envoi de message.

Le modèle relationnel ne permet d'établir explicitement aucune hiérarchie entre les relations d'un même schéma. Il ne sait gérer aucune sorte de lien d'héritage et il n'est pas

possible de définir une relation fille d'une autre et héritant de ses attributs. L'envoi de message est inexistant.

Concernant l'information contenue dans les structures, les possibilités de poses de contraintes sur les attributs sont relativement réduites.

Enfin, l'une des plus importantes restrictions imposées par le modèle relationnel est l'absence de types complexes pour les attributs, par exemple les types *ensemble* et *objet*.

3.2.2 Classe et relation, objet et n-uplet

La notion de relation s'apparente de façon assez naturelle à celle de classe. Nous les associons. En fait, nous rapprochons plus précisément une relation et une hiérarchie de classes, car nous imposons à toutes les sous-classes d'une classe dont on a défini la correspondance d'être associées à la même relation qu'elle (figure 3.6). La classe la plus générale associée à une relation est appelée *classe principale*. Sa relation associée est également qualifiée de *relation principale*.

Classe		Relation
Employé	⟷	emp
Vendeur	⟷	emp
Département	⟷	dept

Figure 3.6 : Associations de classes et de relations

De la même façon, nous rapprochons les notions de n-uplet relationnel et d'objet (figures 3.2, 3.4 et 3.7). L'un comme l'autre sont les données, instances, occurrences de leurs structures respectives, la relation pour l'un, la classe pour l'autre. Ainsi, en DRIVER, tout n-uplet existant dans une relation est un candidat objet, instance de la classe associée à cette relation.

Ce candidat objet est un objet à part entière, élément de la couche objet virtuelle, quand sa valeur est compatible avec les contraintes imposées aux instances de la classe. Ces contraintes, traduites en contraintes sur la relation grâce au schéma de correspondances, constituent une sorte de filtre qui détermine quels n-uplets sont des objets potentiels, *relationnels* (cf. §2.3.3) et lesquels ne le sont pas. Ceux qui ne sont pas retenus sont simplement ignorés par DRIVER. Tout se passe comme s'ils n'existaient pas.

Quand une relation est associée à une hiérarchie de classes, ses n-uplets qui vérifient au moins les contraintes imposées par la classe principale sont les correspondants d'objets relationnels instances de l'une d'entre elles. Un n-uplet de relation principale détermine précisément la classe de l'objet qui le représente en fonction des contraintes imposées par chaque classe qu'il respecte ou non. Cette classe devant être identifiée de manière unique,

Objet, classe		Relation(n-uplet)
Paul Miller, Employé	⟷	emp(7934,miller,paul,clerk,7782,1300.,,10)
Peter Ward, Vendeur	⟷	emp(7521,ward,peter,salesman,7698,1250.,1400.,30)
Paul King, Président	⟷	emp(7839,king,paul,président,,5000.,,10)

Figure 3.7 : Correspondances objet - n-uplet relationnel

toutes les sous-classes d'une même classe doivent pouvoir être différenciées entre elles, au moins par des contraintes différentes de façon qu'un n-uplet ne puisse vérifier au plus que celles de l'une d'entre elles. Une classe se différencie de sa mère par des contraintes plus astreignantes imposées aux objets, outre la définition de nouveaux champs qui s'ajoutent à ceux dont elle hérite.

Lors d'un filtrage sur la base de données, DRIVER classe automatiquement les objets relationnels choisis et leur attribue la classe la plus précise en fonction de leurs valeurs. Les contraintes s'organisant en arbre, l'appartenance à une classe est ainsi fonction de leur satisfaction ou non tout au long de la hiérarchie considérée.

Un objet relationnel détermine de manière unique le n-uplet qu'il représente dans la couche virtuelle grâce à la définition d'un *identificateur relationnel*, formé d'un nom de table principale et d'une valeur de clé dans cette table. Cet identificateur ne comporte pas d'information sur le SGBD et la base concernés, car chaque représentant d'objet relationnel dans l'environnement (défaut d'objet ou objet jumeau) connaît le schéma de correspondances dont il dépend, qui, lui-même, connaît le SGBD et la base pour l'exploitation desquels il a été défini.

3.2.3 Correspondances relationnelles des différents types de champs

Examinons maintenant comment établir des correspondances entre les différents types de champ du modèle objet et les structures relationnelles.

3.2.3.1 Correspondance du champ atomique

Il nous a semblé naturel d'associer attributs et champs atomiques.

Les attributs de la relation principale sont en général les correspondances de la plupart des champs atomiques de la classe principale associée (figure 3.8).

La plupart seulement, car il est en effet possible d'associer un champ atomique à un attribut d'une autre relation que la relation principale. Cette autre relation est alors appelée *relation secondaire*, associée à la classe en question. Une relation secondaire est liée à la relation principale au moyen d'une *jointure*.

De façon générale, une jointure se construit en liant par comparaison (égalité, inégalité, etc...) un ou plusieurs attributs de la relation principale à un ou plusieurs attributs de la relation secondaire, soit directement, soit par l'intermédaire d'autres tables qui devien-

Classe, champ		Relation.attribut
`Employé, nom`	⟷	`emp.ename`
`Employé, prénom`	⟷	`emp.fname`
`Employé, qualif`	⟷	`emp.job`
`Employé, salaire`	⟷	`emp.sal`
`Vendeur, commission`	⟷	`emp.com`
`Département, nom`	⟷	`dept.dname`

Figure 3.8 : Associations champ atomique - attribut de relation principale

nent également alors relations secondaires de la classe correspondante. Les comparaisons d'attributs peuvent être liées par `et` ou `ou`. En DRIVER, les jointures sont orientées; elles sont définies pour accéder, depuis une table origine, à une table destination.

Dans notre application, nous avons choisi de faire de la relation `person` (cf. §3.1.3.1) une relation secondaire de la classe `Employé`. Cela nous permettra de trouver au sein des instances d'`Employé` certaines informations issues dans cette table. Pour pouvoir y accé-

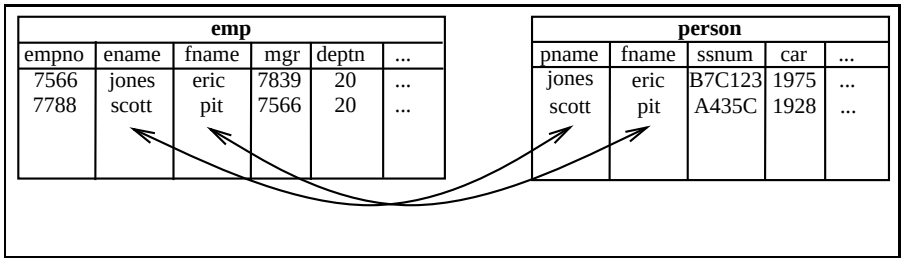

Figure 3.9 : Jointure de la table `emp` vers la table `person`

der depuis la relation `emp`, nous avons défini une jointure qui rapproche les deux tables (figure 3.9). Cette jointure exprime le fait qu'un employé (n-uplet de `emp`) et une personne (n-uplet de `person`) référencent le même individu s'ils ont même nom et même prénom. Son expression est :

```
emp.ename = person.pname and emp.fname = person.fname
```

Appelons-la J[`emp`→`person`]. Sa définition permet l'utilisation d'attributs de la table `person` pour la correspondance de champs d'`Employé`, notamment des champs atomiques (figure 3.10).

Cette jointure associe à chaque n-uplet de la relation principale un unique n-uplet de la relation secondaire. Elle est appelée *jointure élémentaire*. Une jointure élémentaire n'est pas autorisée à échouer : elle représente une liaison dite *forte*.

Classe, champ		Relation.attribut,	Jointures
`Employé, num-ss`	$\longleftrightarrow$	`person.ssnum,`	(J[`emp`→`person`])
`Employé, tél-privé`	$\longleftrightarrow$	`person.phone,`	(J[`emp`→`person`])

Figure 3.10 : Associations champ atomique - attribut de relation secondaire

Participant à la définition d'une correspondance de champ atomique, une jointure est nécessairement élémentaire. L'objet relationnel est constitué d'un n-uplet de la relation principale, plus un n-uplet dans chacune des relations secondaires associées à la relation principale. Si une jointure échouait, toute une partie de l'objet deviendrait inaccessible. En fait, l'algorithme d'accès aux données de la base rendrait l'objet complet totalement inaccessible (cf. §4.1.2).

La relation principale d'une classe et les relations secondaires liées par des jointures élémentaires à cette relation principale sont appelées les *relations élémentaires* associées à la classe considérée.

Toute relation élémentaire peut être utilisée à son tour comme base sur laquelle ou à partir de laquelle peuvent être définies de nouvelles correspondances de champs. Ses attributs peuvent être utilisés comme correspondances de champs ou pour définir de nouvelles jointures vers de nouvelles relations.

Chaque correspondance de champ atomique est donc accessible par une *chaîne de jointures* reliant la relation secondaire sur laquelle a été définie la correspondance à la relation principale. Le chaînage est simple. Chaque jointure étant orientée, l'ensemble doit en comporter une dont la table origine est la relation principale. La table destination de cette jointure doit être la table origine d'une autre, et ainsi de suite jusqu'à la table contenant l'attribut associé au champ. Une fois la chaîne reconstituée et les éléments ordonnés, toutes les jointures doivent avoir été utilisées. Si ce n'est pas le cas, la chaîne est *incohérente*. Si des éléments manquent, elle est *inconsistante*. Enfin, on peut faire la remarque que, si chaque jointure est élémentaire, la chaîne elle-même est élémentaire.

Une table ne peut pas être à la fois principale et secondaire pour des raisons de maintien de la cohérence entre la couche objet réelle et la base de données. Un objet relationnel est constitué d'un n-uplet dans chaque relation, principale ou secondaire, qui participe à la correspondance de la classe. Si un double rôle pouvait être accordé à une relation, la création d'un objet de la classe l'ayant pour table secondaire provoquerait par effet de bord la création d'un autre objet de classe celle l'ayant comme relation principale.

Une table secondaire pour une classe ne peut pas non plus être table secondaire pour une autre n'héritant pas de la même classe principale. Une table ne peut pas être table principale de deux classes principales différentes.

Pour des raisons similaires, l'usage multiple d'un attribut pour définir différentes cor-

respondances est interdit. Un attribut, qui a déjà été choisi comme correspondance de champ atomique ou qui intervient déjà dans la définition d'une jointure, ne peut plus être utilisé à d'autres fins. Il n'est plus disponible pour définir de nouvelles jointures ou de nouvelles correspondances de champs atomiques. Ceci n'interdit cependant nullement de réutiliser la jointure à laquelle il participe éventuellement dans la définition de nouvelles correspondances.

Les règles qui viennent d'être énoncées peuvent se révéler trop contraignantes pour certains cas particuliers d'application. Il peut par exemple être utile de pouvoir associer un champ atomique à un attribut qui participe également à la définition d'une jointure. Notre application comprend un tel exemple par le biais des attributs `emp.ename` et `emp.fname`. Ces deux attributs interviennent dans la jointure J[`emp`→`person`] définie plus haut (figure 3.9), mais nous souhaiterions également les associer à deux champs de la classe `Employé`, les champs `nom` et `prénom` (figure 3.8).

Afin de préserver la cohérence du système, les utilisations et correspondances multiples de relations sont autorisées dans la mesure où seule l'une d'entre elles est habilitée à écrire dans la base. Toutes les autres ne doivent permettre l'accès à l'information qu'en lecture.

Le même principe s'applique aux attributs intervenant à plusieurs reprises dans les définitions de correspondances. Pour des raisons de cohérence, ces attributs ne pourront être mis à jour dans la base que pour l'une de leurs participations.

Un attribut peut participer à une correspondance de deux façons différentes : dans une jointure ou comme correspondance d'un champ atomique. Une jointure est une contrainte qui lie des attributs et qui doit être impérativement vérifiée pour qu'un futur chargement des objets soit possible (cf. §4.2.1.3). Le report dans la base des valeurs des attributs qui la satisfont est nécessaire. Pour cette raison, si un attribut participe, entre autres correspondances, à une jointure, cette dernière étant nécessairement habilitée à écrire dans la base, toutes les autres correspondances ne devront permettre qu'un accès en seule lecture sur cette même base. Si un attribut intervient dans une jointure, toute autre utilisation de cet attribut comme correspondance de champ atomique ne sera possible que comme consultation de la base. Pour la raison évoquée ci-dessus et afin d'éviter les conflits ou les incompatibilités, il est strictement interdit de définir deux jointures comportant un même attribut.

Dans notre application, les attributs `emp.ename` et `emp.fname` pourront donc participer à la jointure J[`emp`→`person`] et être les correspondances des champs `nom` et `prénom` de la classe `Employé` si ces deux dernières correspondances ne permettent qu'un accès en lecture sur la base. Il est par ailleurs utile de préciser dès maintenant que si un nouvel objet persistant de la classe `Employé` est créé, son nom et son prénom seront bien sûr reportés dans la base. L'algorithme d'écriture d'un objet résout les contraintes posées sous forme de jointures (cf. §4.2.3.1) et "sait", quand c'est possible, exploiter le contenu des champs pour donner une première valeur aux attributs composant les jointures. Ces valeurs sont ensuite écrites dans la base. Ainsi, les champs `nom` et `prénom` vont être reportés dans la base à leur première validation. Seulement, ils ne pourront plus être modifiés ensuite pour éviter tout échec de la jointure à laquelle ils participent.

3.2.3.2 Correspondance du champ "objet"

Un champ de type objet représente un lien orienté d'un objet vers un autre. Quand la classe principale de l'objet référencé est connue, il est naturel d'associer ce lien entre classes à un lien entre les tables principales correspondantes.

Par définition, nous faisons correspondre ce type de champ à une chaîne de jointures qui relie ces tables principales et associe à chaque n-uplet de la première un n-uplet de la seconde (figure 3.12).

La dernière jointure de la chaîne doit avoir comme table initiale une relation élémentaire de la classe possèdant le champ et comme table finale la relation principale de l'autre classe. Elle est appelée *jointure objet*.

Présentons deux exemples de correspondance de champ objet. Le premier exploite l'existence, au sein de la relation `emp`, de l'attribut `deptn` qui contient le numéro du département auquel appartient l'employé. Une jointure entre les deux tables apparaît (figure 3.11) :

```
emp.deptn = dept.deptno
```

Cette jointure, que nous appelons J[`emp`→`dept`], est la correspondance naturelle du champ `dpt` de la classe `Employé` (figure 3.12). Il arrive que la chaîne de jointures se réduise comme ici à la seule jointure objet.

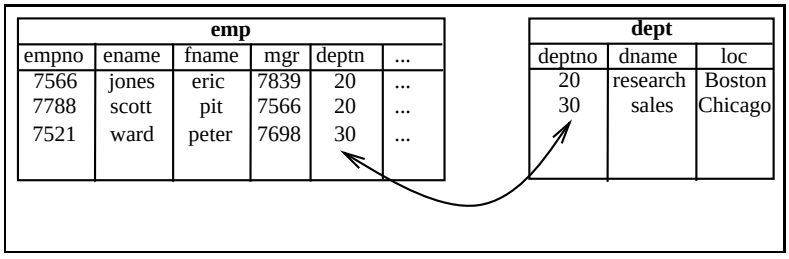

Figure 3.11 : Jointure de la table `emp` vers la table `dept`

Par ailleurs, nous constatons, dans la relation `person`, l'existence d'un attribut `car` qui contient, pour chaque personne, l'identificateur de son véhicule personnel. Cet identificateur correspond à une valeur de clé contenue dans l'attribut `vnum` de la table `vehicle`. On reconnaît une autre jointure objet possible, cette fois, entre les tables `person` et `vehicle` :

```
person.car = vehicle.vnum
```

Appelons cette jointure J[`person`→`vehicle`]. En admettant qu'on ait défini comme correspondance à la table `vehicle` la classe `Voiture`, on constate qu'on peut faire correspondre le champ `vehicule` de la classe `Employé` à la chaîne de jointures (**J**[`emp`→`person`]

J[person→vehicle]) (figure 3.12). Cette chaîne est cohérente et consistante. Elle relie bien les relations principales des deux classes concernées.

Classe, champ		Jointures
Employé, dpt	⟷	(J[emp→dept])
Employé, véhicule	⟷	(J[emp→person] J[person→vehicle])

Figure 3.12 : Associations champ objet - chaîne de jointures

Les modèles objet autorisent l'absence de valeur dans les champs de type objet des objets. On remarquera au passage que c'est un cas particulier où le champ de type objet a alors une valeur qui n'est pas un objet (!).

Cette caractéristique a été prise en compte. En première analyse, sa correspondance en termes relationnels est l'échec de la jointure objet participant à la correspondance du champ.

Les $N-1$ premières jointures de la chaîne sont élémentaires. Elles lient entre elles des relations élémentaires. Par contre, la dernière (la jointure objet) ne l'est pas. Elle relie une relation élémentaire de la classe considérée à une relation indépendante de cette classe, en l'occurrence, la relation principale de la classe de l'objet contenu. Si le champ objet ne contient pas de référence vers un objet, cette absence traduit ou se traduit par un échec de cette jointure.

En DRIVER, on normalise la gestion de l'absence de valeur dans les champs de type objet en imposant un certain nombre de restrictions aux jointures objet. Ces restrictions permettent de spécifier comment ces jointures sont autorisées à échouer. Elles permettent aussi de gérer efficacement ces échecs :

1. Les attributs de la relation destination (la relation principale de la classe de l'objet à référencer) impliqués dans la jointure sont obligatoirement et exclusivement les attributs constituant la clé de la relation.

2. Ces constituants de clé sont comparés à des attributs d'autres relations, appelés *attributs référentiels*, ou à des valeurs. À chaque fois, le comparateur est nécessairement l'égalité.

3. Toutes ces comparaisons doivent être reliées entre elles et avec le reste du corps de la jointure, s'il y en a un, par un `et`.

Une quatrième restriction est imposée aux jointures objet définies pour la correspondance de champ de type "objet". Cette restriction n'est donc pas caractéristique de toutes les jointures objet. Celles qui la vérifient sont dites *strictes* :

4. S'il existe, le reste du corps doit représenter un ensemble de liaisons fortes entre la relation origine de la jointure, relation élémentaire pour la classe possédant le champ, et les relations des attributs référentiels. Autrement dit, les relations des attributs référentiels doivent elles aussi être élémentaires pour la classe possédant le champ.

Par définition,
le champ objet d'une instance ne référence aucun autre objet (champ "vide", absence de valeur) si et seulement si l'un au moins des attributs référentiels vaut NULL[4].

Cette définition établit très précisément quelle correspondance relationnelle est donnée à l'absence de valeur dans un champ objet. Grâce à elle, le système est en mesure non seulement de détecter un champ objet vide au chargement, mais aussi d'écrire un tel champ vide dans la base. On connaît les attributs qui déterminent l'objet référencé (c'est-à-dire les attributs référentiels) et on sait quelle valeur leur affecter quand le champ est vide.

Prenons l'exemple des deux jointures objet J[`emp`→`dept`] et J[`person`→`vehicle`] que nous avons identifiées. Toutes deux respectent entièrement les restrictions imposées aux jointures objet strictes. Les seuls attributs des relations destinations qui participent sont respectivement `dept.deptno` et `vehicle.vnum`, chacun étant la clé de sa relation. Ils sont par ailleurs comparés par l'égalité à deux attributs `emp.deptn` et `person.car` issus de deux relations élémentaires de la classe `Employé`. L'intérêt des restrictions apparaît ici clairement : la consultation de chacun de ces deux attributs référentiels suffit pour déterminer le contenu des champs objet correspondants.

On constate que l'absence de valeur ne coïncide plus tout-à-fait avec l'échec de la jointure objet, comme nous l'avions dit en première analyse. Si tout champ objet vide se traduit effectivement en termes relationnels par un échec de jointure, l'inverse est faux; il est des cas d'échec qui ne correspondent pas à une absence de valeur dans un champ objet.

En effet, si tous les attributs référentiels possèdent une valeur, la référence de l'objet est considérée comme valide. Si, par ailleurs, la valeur de clé ne correspond à aucun n-uplet dans la relation principale jointe (raison d'échec de la jointure), l'existence de l'objet relationnel associé n'est pas remise en cause. Il est alors seulement annoncé *manquant* lors de toute opération visant à le charger ou à accéder à ses données (cf. §4.1.1.3).

Les restrictions permettent également une plus grande efficacité du système. L'algorithme d'accès aux données dans la base minimise le nombre de requêtes adressées au SGBD car elles sont extrêmement coûteuses en temps. Cet algorithme permet de charger en mémoire un nombre quelconque d'objets d'une même classe principale par l'envoi de

[4] NULL est la valeur "système" qui représente l'absence de valeur dans une base de données relationnelle.

$1 + N$ requêtes au SGBD, N étant le nombre de champs de type "ensemble" et "calcul multi–n-uplets" définis dans la classe (cf. §4.1.2). Sans les restrictions imposées aux jointures objet strictes, le coût de chargement serait de $1 + N + M$ requêtes, M étant le nombre de champs de type "objet" définis dans la classe (cf. §4.1.4.3).

Une requête supplémentaire serait nécessaire par champ objet. En effet, la jointure objet échoue quand le champ est vide. Comme, dans le résultat d'une requête, l'échec d'une jointure se solde par l'absence des n-uplets pour lequel cet échec s'est produit, la requête de chargement d'un groupe d'objets ne pourrait en aucun cas comprendre les objets relationnels ayant un champ objet vide.

Du fait des restrictions, les identités des objets contenus peuvent être obtenues dans le cadre de la requête principale de chargement. L'échec de la jointure objet au sein de cette requête est évité en supprimant la liaison faible qu'elle comprend. Pour connaître l'identité des objets référencés, il n'est plus nécessaire d'interroger le SGBD sur des valeurs de clé à chercher dans la relation principale associée à la classe de ces objets, vérifiant la jointure objet en liaison avec telle autre relation principale correspondant à la classe possédant le champ. On se limite à demander simplement la valeur des attributs référentiels puisqu'ils contiennent directement les références des objets qui nous intéressent. Ces attributs sont en plus toujours accessibles puisqu'ils appartiennent à des relations élémentaires de la classe possédant le champ. La sous-partie de la jointure objet qui les lie à la relation principale n'est pas autorisée à échouer.

Les restrictions imposées limitent objectivement les possibilités de jointure objet. Par exemple, une autre expression pour la jointure J[`person`→`vehicle`] comme :

```
person.car = vehicle.vnum and vehicle.year >= 1990
```

n'est pas autorisée. Ces restrictions ne gêneront cependant qu'un nombre très restreint d'utilisateurs : la plupart des jointures objet identifiables directement dans une base les respectent de façon naturelle. À l'inverse, elles permettent de définir une correspondance relationnelle claire pour l'absence de valeur dans un champ objet, valable aussi bien en termes de lecture que d'écriture de données dans la base. Elles rendent possible la gestion des objets manquants et offre une efficacité maximale au mécanisme de communication et de dialogue avec le SGBD.

Des liens entre objets plus ou moins forts

Dans les modèles objet, le champ de type objet permet de représenter un lien entre deux objets. Seulement, cette représentation ne donne aucune information sur la force de ce lien. Par exemple, il n'est pas possible d'imposer qu'un objet référencé dans un champ objet soit la seule valeur possible pour ce champ.

Il n'en est pas de même dans le modèle relationnel. En effet, la nature des attributs référentiels intervenant dans une jointure objet peut permettre de constituer une sorte de filtre et limiter ainsi l'ensemble des valeurs possibles pour le champ objet correspondant.

Certains attributs référentiels peuvent faire partie de la clé de leur relation. On peut alors considérer qu'ils ont alors un double rôle. D'une part, leurs valeurs participent aux clés des objets de la classe correspondante. D'autre part, du fait de leur fonction d'attributs référentiels, ils permettent également de référencer d'autres objets.

Comme une valeur de clé ne peut plus être modifiée une fois qu'elle a été attribuée, les valeurs de ces attributs référentiels sont par conséquent définitivement fixées quand un objet acquiert la persistance. De ce fait, elles transforment la jointure objet en filtre. Dans le champ objet associé ne sont plus ensuite autorisés que les objets dont la référence est compatible avec la contrainte exprimée par la jointure en question. Une jointure objet comprenant un ou plusieurs attributs référentiels éléments de clé est dite *restreinte*.

Quand un champ objet est associé à une jointure objet restreinte, l'absence de valeur dans ce champ est exprimée en donnant aux attributs référentiels non éléments de clé la valeur NULL. Si tous les attributs référentiels sont éléments de clé, l'absence de valeur dans le champ n'est pas autorisée. On peut d'ailleurs ajouter que, dans ce dernier cas de figure, pour un objet persistant donné, tous les attributs référentiels ont une valeur fixée. La référence de l'objet contenu est donc entièrement déterminée et le champ correspondant ne peut donc recevoir qu'un seul objet possible : celui dont la référence est donnée par les attributs référentiels. Une jointure objet dont tous les attributs référentiels sont éléments de clé est dite *fixée*.

Un exemple de champ objet associé à une jointure objet restreinte est présentée en annexe A.2 car notre base sur les employés n'en comporte pas. Par contre, elle présente un exemple de jointure objet fixée.

Le champ `adresse` de la classe `Employé` est associé à la jointure J[`emp`→`address`]. Cette jointure, dont la définition est `emp.empno = address.empno`, est fixée car tous les attributs référentiels qu'elle comprend, soit {`emp.empno`}, sont également éléments de clé.

Dans le modèle objet, cette association entre le champ `adresse` et une jointure fixée se traduit par le fait que tout objet `Employé` est inmanquablement et définitivement lié à un objet `Adresse` au moment où il devient persistant. L'absence de valeur ou le remplacement de l'objet `Adresse` autorisé par un autre ne sont pas admis. Le comportement du système lors d'une tentative de validation dans la base d'un objet ayant été modifié sans respecter ces règles est examiné en §4.2.2.2.

3.2.3.3 Correspondance du champ "ensemble d'objets"

Un champ de type "ensemble d'objets" représente lui aussi un lien entre objets. Il s'agit cette fois d'un lien multiple qui attache un ensemble d'objets à l'objet origine. Si les objets de l'ensemble sont d'une même classe connue, la correspondance de ce type de champ est également une chaîne de jointures entre tables principales. Cependant, cette chaîne associe cette fois à chaque n-uplet de la table origine un ensemble de n-uplets de la table

destination.

Encore une fois, les $N-1$ premières jointures de la chaîne sont élémentaires. Elles lient entre elles des relations élémentaires. La dernière jointure doit avoir comme table initiale une relation élémentaire de la classe possèdant le champ et comme table finale la relation principale de la classe des objets à référencer.

Si le champ associé permet seulement la consultation de données de la base, cette jointure n'est soumise à aucune restriction. Si le champ est classique, c'est-à-dire si les valeurs qui y sont contenues sont susceptibles d'être reportées ou mises à jour dans la base, la jointure doit être une jointure objet. Les restrictions qui lui sont imposées (cf. §3.2.3.2) sont nécessaires pour préciser encore une fois dans quelles conditions elle est autorisée à échouer.

La jointure objet n'est pas tenue d'être stricte, même s'il est préférable que ce soit le cas. En fait, seule une jointure représentant une liaison 1:N en a la possibilité; la restriction correspondante est impossible à respecter pour une jointure qui représente une liaison N:M.

Nous avons précédemment identifié une jointure J[`emp`→`dept`], de la table `emp` vers la table `dept` (figure 3.11). Son expression peut tout aussi bien permettre la définition d'une jointure J[`dept`→`emp`], de la table `dept` vers la table `emp`. Cette nouvelle jointure est la correspondance naturelle du champ `personnel` de la classe `Département`, destiné à recueillir les références des employés appartenant à ce département (figure 3.14). Comme nous l'avons vu, cette jointure, qui représente une liaison 1:N, est stricte; elle vérifie les quatre restrictions précédemment décrites.

Table **project**

projno	**pname**	**budget**
101	alpha	250000.
102	beta	175000.
103	gamma	95000.
...	...	...

Table **emproj**

projno	**empno**
101	7566
103	7566
101	7788
...	...

Figure 3.13 : Tables `project` et `emproj`

Prenons un autre exemple de jointure représentant cette fois une liaison N:M. Notre base de données comprend aussi deux tables `project` et `emproj`. `project` est la table des projets existant au sein de l'entreprise. Chaque employé peut participer à plusieurs projets, participations qui sont précisées dans la table `emproj` (figure 3.13). Considérons une classe `Projet` associée à la relation `project` et comprenant un champ `employés`. Ce champ est destiné à recueillir, pour chaque projet, les références des employés y participant. Sa correspondance relationnelle est une jointure J[`project`→`emp`] (figure 3.14) dont l'expression est :

```
project.projno = emproj.projno and emproj.empno = emp.empno
```

Cette jointure représente clairement une liaison N:M : un projet peut comprendre plusieurs employés et chaque employé peut participer à plusieurs projets. Dans la représentation relationnelle, la liaison N:M doit nécessairement s'exprimer sous forme d'une table indépendante dans laquelle chaque n-uplet représente un lien entre deux données. Ceci permet à chacune d'entre elles d'être en relation avec un nombre quelconque de données de l'autre ensemble. De ce fait, cette table indépendante ne peut pas être élémentaire pour l'une des deux classes. À chaque n-uplet de relation principale peut correspondre un nombre quelconque de n-uplet de cette table. Les quelques n-uplets de la relation `emproj` qui ont été présentés montrent qu'il y en a déjà deux qui correspondent au projet "alpha". La restriction imposée aux jointures objet strictes ne peut pas être vérifiée.

Classe, champ		Jointures
`Département, personnel`	$\longleftrightarrow$	(J[`dept`→`emp`])
`Projet, employés`	$\longleftrightarrow$	(J[`project`→`emp`])

Figure 3.14 : Associations champ "ensemble d'objets" - chaîne de jointures

On peut faire la remarque que la sémantique du lien qui lie un objet à un autre détermine le fait que le lien inverse mène soit à l'objet origine, soit à un ensemble d'objets dont fait partie l'objet origine (lien 1:1 ou 1:N). Une jointure exprimant un lien 1:N peut donc permettre de définir soit un champ de type "objet" dans l'une des deux classes, soit un champ de type "ensemble d'objets" dans l'autre. Si la jointure exprime un lien N:M, elle peut alors être la correspondance dans chaque classe d'un champ de type "ensemble d'objets".

Dans tous les cas, les deux définitions ne pourront être tolérées simultanément que si l'un des champs est déclaré accessible en lecture seulement. Cependant, il est utile de faire remarquer que, comme ces deux champs sont associés aux mêmes données relationnelles, la mise-à-jour dans la base de l'un des deux suffit pour que les deux aient une valeur correctement représentée par ces données relationnelles. À la charge du modèle objet client bien sûr de gérer la cohérence ou les éventuels liens inverses entre les deux champs.

Pour finir, signalons que les champs "ensemble d'objets" peuvent également être associés à des jointures restreintes ou fixées. Comme pour les champs de type "objet", ne pourront y être référencés que les objets qui vérifieront le filtre formé par les attributs référentiels contraints.

3.2.3.4 Correspondance du champ "ensemble de valeurs atomiques"

Un champ de type "ensemble de valeurs atomiques" est un champ multi-valué qui permet d'associer à chaque objet un ensemble de valeurs atomiques de même type. Nous

définissons comme correspondance à ce type de champ un attribut d'une table secondaire rattachée à la table principale par une chaîne de jointures (figure 3.15). Cette chaîne doit associer à chaque n-uplet de la relation principale un ensemble de n-uplets de la relation secondaire, desquels sont extraites les valeurs composant l'ensemble.

Classe, champ		**Relation.attribut,**	**Jointures**
`Département, sites`	$\longleftrightarrow$	`deptsite.sname,`	(J[`dept`$\rightarrow$`deptsite`])

Figure 3.15 : Association champ "ensemble d'atomes" - attribut

Comme pour le champ de type "ensemble d'objets", les $N-1$ premières jointures de la chaîne doivent être élémentaires et représenter des liaisons fortes. Encore une fois, la $N^{ième}$ ne peut pas être élémentaire puisqu'elle représente un lien 1:N ou N:M.

Un champ de ce type a été défini dans la classe `Département`. Il s'agit du champ `sites`, contenant, pour chaque instance, un ensemble de symboles représentant les noms des sites où se trouve implanté le département. Ce champ trouve naturellement sa correspondance au sein de la relation `deptsite` qui précisément gère dans la base les informations concernant les sites où sont représentés les départements (figure 3.16).

Table **deptsite**	
sname	**dept**
New-York	10
Boston	20
Chicago	30
Chicago	10
Los-Angeles	30
Austin	30
...	...

Figure 3.16 : Table `deptsite`

Cette correspondance relationnelle est l'attribut `deptsite.sname` (figure 3.15), rendu accessible de la table principale `dept` par la jointure J[`dept`$\rightarrow$`deptsite`] dont l'expression est :

```
dept.deptno = deptsite.dept
```

Cette jointure représente une liaison N:M.

3.2.3.5 Correspondances des champs "ensemble ordonné"

Tous les types de champs gérant des ensembles ordonnés d'éléments (valeurs atomiques ou objets) sont des champs de type "ensemble" pour lesquel l'ordre des éléments a de l'importance. La correspondance de chacun est donc presque identique à celle du type dont il se rapproche (mais qui n'est pas ordonné). On ordonne en plus les éléments de l'ensemble d'après les valeurs d'attributs de tables utilisées dans la jointure, en liaison forte (1:1) avec la table destination. Ces *attributs de tri* peuvent bien sûr faire tout simplement partie de la table destination.

Le premier définit l'ordre primaire, le second l'ordre secondaire, et ainsi de suite. Les éléments sont d'abord ordonnés en fonction de la valeur de l'attribut primaire de tri. En cas d'égalité, les éléments correspondants sont triés entre eux en fonction de la valeur de l'attribut secondaire, etc. Pour chaque attribut, l'ordre peut être ascendant ou descendant.

La classe `Département` présente un champ appartenant à cette catégorie. Il s'agit du champ `chefs` du type "ensemble ordonné d'objets", destiné, pour chaque département, à recueillir les références de ses cadres. La correspondance de ce champ est la jointure J[`dept`→`emp`], déjà décrite en §3.2.3.3, qui, contrairement à ce qui se passe pour le champ `personnel`, ne sélectionne pas l'ensemble des employés de chaque département. En effet, elle se trouve indirectement précisée par la classe des objets à filtrer (`Cadre`, ici), ce qui la complète des contraintes de cette classe (cf. §4.1.2).

Classe, champ		Jointures,	Attributs de tri
`Département, chefs`	⟷	(J[`dept`→`emp`])	(`emp.sal` (desc) `emp.ename` (asc))

Figure 3.17 : Correspondance d'un champ "ensemble ordonné d'objets"

Tous les attributs de `emp`, de `person` ou de n'importe quelle relation en liaison forte avec `emp` peuvent servir d'attribut de tri, à condition bien sûr d'éviter les correspondances multiples. Nous avons choisi `emp.sal` comme critère de tri primaire, et `emp.ename` comme critère de tri secondaire (figure 3.17). Dans ce champ, les cadres seront triés d'abord selon leur salaire, puis, en cas d'égalité, selon leur nom. Pour le premier, l'ordre descendant a été choisi; pour le second, l'ordre alphabétique a été préféré (ascendant).

Pour ne pas attribuer de rôle multiple à la jointure J[`dept`→`emp`] et aux attributs `emp.sal` et `emp.ename`, le champ `chefs` ne peut lui aussi être défini que comme une "fenêtre" de consultation de données de la base (accès en lecture seulement).

3.2.3.6 Correspondances des champs "objet de classe inconnue"

Définir la correspondance relationnelle d'un champ destiné à contenir un objet de classe inconnue (objet simple, ensemble d'objets, ensemble ordonné d'objets) n'est pas une chose immédiate. Cette correspondance est en plus vide de sens pour la démarche inverse qui consiste à vouloir associer une représentation objet à des structures relationnelles qu'on souhaite exploiter dans l'environnement. En effet, un attribut qui constitue une référence

vers une autre table permet naturellement de définir une jointure, correspondance d'un champ à objet de classe connue.

Ce type de champ devait pouvoir être rendu persistant car il existe dans les langages objet que nous utilisons et le fait de l'ignorer pouvait rendre l'utilisation de DRIVER sans intérêt pour certaines applications.

Nous associons le champ de type "objet de classe inconnue (mais persistante)" à un attribut de type "chaîne de caractères" (figure 3.18) dont les valeurs sont des références codées vers des objets relationnels de la même base. Chaque référence contient le nom de la relation principale de l'objet et la valeur de la clé du n-uplet correspondant.

Classe, champ		Relation.attribut,	Jointures
`Employé, responsable-de`	$\longleftrightarrow$	`emp.responsible_for`,	()
`Département, produits`	$\longleftrightarrow$	`deptproducts.pref`,	(J[`dept`$\rightarrow$`deptproducts`])

Figure 3.18 : Associations champ "objet de classe inconnue" - attribut

Les correspondances de ces types de champ à objet de classe indéterminée (objet simple, ensemble d'objets, ensemble ordonné d'objets) peuvent être rapprochées de celles des types gérant des valeurs atomiques (valeur atomique simple, ensemble de valeurs atomiques, ensemble ordonné de valeurs atomiques) : leur élément de base est associé à un attribut. On leur a défini en plus des filtres en lecture/écriture sur la base (cf. §3.2.5.3) qui permettent de transformer les valeurs. Le premier fonctionne dans le sens base vers couche objet. Il reçoit en entrée la valeur de l'attribut, c'est-à-dire une chaîne de caractères, et rend en sortie la valeur à placer dans le champ de l'objet, à savoir un pointeur sur l'objet contenu. Le second fonctionne dans le sens inverse. Il reçoit en entrée un objet et renvoie en sortie une chaîne de caractères codant la référence relationnelle de l'objet.

Deux champs de notre application sont des champs "objet de classe inconnue".

L'un est le champ `responsable-de` de la classe `Employé`, de type "objet de classe inconnue", référençant un objet dont l'employé est responsable. L'objet en question peut être une instance de `Département`, de `Projet` ou de classes correspondant à des produits. Sa correspondance doit être un attribut d'une relation élémentaire de la classe `Employé`. Définissons un nouvel attribut `responsible_for` de type `string` à la relation `emp` (figure 3.19). Nous l'associons au champ `responsable-de` (figure 3.18). Les valeurs qui y sont contenues et qui résultent de l'écriture d'objets persistants dans la base sont les références codées d'objets relationnels.

Le second est le champ `produits` de la classe `Département`, de type "ensemble d'objets de classe inconnue", qui référence les produits dépendant de chaque département. Encore une fois, ces produits peuvent être instances de classes indépendantes et différentes. Leurs références sont gérées dans l'attribut `pref` de la table `deptproducts` (figures 3.18 et 3.20). Chaque n-uplet de cette table fait état d'un lien entre un produit et un département. Une

Table **emp**				
empno	**ename**	**fname**	**...**	**responsible_for**
7839	king	paul	...	project/102
7566	jones	eric	...	dept/20
7698	blake	harold	...	dept/30
7782	clark	john	...	dept/10
7902	ford	john	...	project/101
7788	scott	pit	...	project/103
7499	allen	jack	...	software/743
7521	ward	peter	...	hardware/53
7654	martin	georges	...	
...	...	...	...	...

Figure 3.19 : Correspondance d'un champ objet2 : la table `emp` complétée

jointure J[`dept`→`deptproducts`], qui lie la table `deptproducts` à la table `dept`, permet d'identifier ce lien. Son expression est :

```
dept.deptno = deptproducts.deptno
```

Chaque produit pouvant être référencé par plusieurs départements, la jointure J[`dept`→`deptproducts`] est l'expression du lien N:M qui lie départements et produits.

Table **deptproducts**	
deptno	**pref**
20	software/743
30	software/874
30	software/743
30	hardware/53
...	...

Figure 3.20 : La table `deptproducts`

3.2.3.7 Correspondances des champs "calculés"

À l'inverse, les champs de type "calculés" ne présentent d'intérêt que pour associer une représentation objet à des données relationnelles. Ils permettent de consulter le résultat d'un calcul sur la base depuis la couche objet. Ils ont naturellement comme correspondance l'expression du calcul sur les attributs (figure 3.21).

Cette expression doit éventuellement être complétée par une chaîne de jointures qui, soit rapproche entre eux les attributs intervenant pour le calcul mono–n-uplet, soit sélectionne les n-uplets pris en compte dans chaque opération pour le calcul sur les ensembles de n-uplets.

Toutes les jointures de la chaîne doivent être élémentaires pour ce qui est du calcul mono–n-uplet, seulement les $N-1$ premières pour le calcul multi–n-uplets.

Classe, champ		Expression du calcul,	Jointures
`Vendeur, sal-total`	$\longleftrightarrow$	`(emp.sal + emp.com)`,	()
`Département, moy-sal`	$\longleftrightarrow$	`Moyenne(emp.sal)`,	(J[`dept`→`emp`])

Figure 3.21 : Correspondance de champs calculés

Notre application comprend deux exemples de champ calculé. L'un est associé à un calcul mono–n-uplet, l'autre à un calcul multi–n-uplets.

Le premier est le champ `sal-total` de la classe `Vendeur`. Ce champ va contenir, pour chaque vendeur, la somme (`salaire+commission`) des attributs `emp.sal` et `emp.com` du n-uplet correspondant (figure 3.21). Ces deux attributs se trouvant dans la relation principale, aucune jointure n'est nécessaire pour les rapprocher.

Le second est le champ `moy-sal` de la classe `Département`. Ce champ va recueillir, dans chaque département, la moyenne des salaires des employés en faisant partie. Il est associé à l'attribut `emp.sal` (figure 3.21). Les n-uplets correspondant aux employés de chaque département sont sélectionnés au moyen de la jointure J[`dept`→`emp`], déjà décrite en §3.2.3.3 et utilisée pour la correspondance des champs `personnel` et `chefs` de la classe `Département`.

3.2.4 Tables physiques et tables logiques

Le langage de description et la représentation objet sous-jacente des correspondances se devaient d'éviter de limiter les possibilités offertes par les définitions qui viennent d'être présentées.

Cela nous a amené à considérer en particulier un cas de champ objet où la jointure qui en est la correspondance est une *auto-jointure*, c'est-à-dire relie une table de départ et une table d'arrivée qui sont les mêmes. Ce cas correspond à un champ objet défini dans la classe C et destiné à recueillir des objets de cette même classe C. La jointure correspondante lie la table principale de la classe C à la table principale de la classe des objets contenus, soit la même table. La description d'une telle jointure s'avère impossible si l'on ne peut pas différencier dans l'expression les attributs de la table de départ et ceux de la table d'arrivée.

Ce type de jointure est descriptible en DRIVER grâce aux notions de *table physique* et

de *table logique*. Par définition, les tables physiques s'identifient aux tables définies dans la base de données. On peut donc déclarer dans l'environnement une table physique par table accessible dans la base. Associées à une même table physique peuvent être définies autant de tables logiques que nécessaire. Chacune d'entre elles est indépendante des autres, se désigne par un nom différent, mais référence en réalité la même table physique.

Dans l'auto-jointure, la table de départ et la table d'arrivée correspondent à la même table physique. On les différencie en utilisant deux tables logiques différentes. Le niveau logique permet dans la jointure de faire comme si les deux tables jointes étaient différentes et de référencer les n-uplets joints sous des noms différents, comme s'ils étaient contenus dans des tables différentes.

En DRIVER, on met généralement en œuvre des tables logiques.

Un certain nombre de primitives avancées permettent cependant de manipuler spécifiquement des tables physiques, mais leurs noms très explicites précisent sans ambiguïté la nature des objets manipulés.

Autre dérogation à la règle, les notions de relation principale et relation secondaire *caractérisent les tables physiques*. Quand une table logique participe à la correspondance d'une classe, sa table physique associée est table principale ou table secondaire de la même classe. Elle n'est plus disponible pour être table principale ou secondaire une nouvelle fois, que ce soit par le biais de la même table logique ou d'une autre.

Par ailleurs, la déclaration d'une table par driver-deftable, telles celles présentées en §3.1.3.1, se traduit automatiquement, outre la définition de la table physique, par la définition d'une table logique de même nom. C'est cette table logique qui sera ensuite référencée dans les déclarations à venir.

La correspondance d'une classe est une table bien évidemment logique. Seulement, pour des raisons de simplification, cette table principale ne pourra être qu'une table logique de même nom que la table physique associée.

Quand on définit la correspondance d'un champ, les attributs sélectionnés sont également logiques. Pour chaque jointure, la table de départ, la table d'arrivée, les tables intermédiaires sont des tables logiques. En particulier, une chaîne de jointures relie donc des tables logiques.

Classe, champ		Jointures
`Employé, chef`	⟷	(J[`emp→emp1(emp)`])

Figure 3.22 : Auto-jointure, correspondance de champ objet

Notre application sur les employés comporte un champ dont la correspondance est une

auto-jointure. Il s'agit du champ de type objet `chef` de la classe `Employé` (figure 3.23).

Le chef d'un employé est aussi une instance d'`Employé`. La jointure correspondante est donc une jointure de la relation `emp` vers la relation `emp`. Pour discerner dans la jointure la table de départ et la table d'arrivée, on rebaptise la seconde `emp1` (figure 3.23). Si on

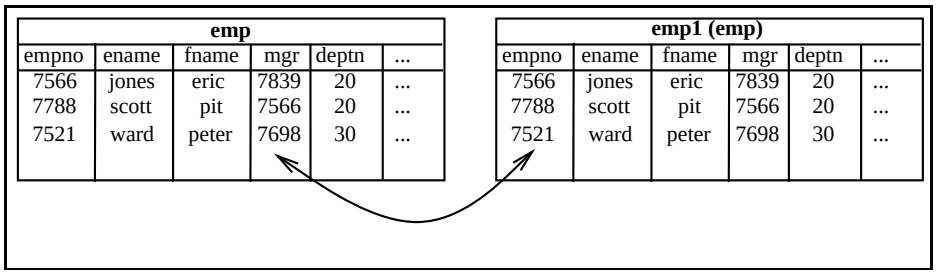

Figure 3.23 : Auto-jointure de la table `emp` vers elle-même

identifie donc la relation des "chefs" comme étant `emp1` (table logique définie sur la table physique `emp`), la jointure J[`emp`→`emp1(emp)`], qui représente dans la base le lien entre un employé et son chef, a pour expression :

```
emp.mgr = emp1(emp).empno
```

L'intérêt des tables logiques et physiques ne se limite pas à permettre l'expression des auto-jointures. Au travers d'un même objet relationnel, ces notions rendent possible le fait d'accéder à des informations contenues dans des n-uplets différents d'une même table secondaire. Il suffit pour cela d'associer aux champs en question, des correspondances sur des tables logiques différentes. Il importe malgré tout que les règles concernant la définition de correspondances multiples soient respectées pour maintenir la cohérence de la base (cf. §3.2.3.1).

3.2.5 Description des correspondances en DRIVER

3.2.5.1 Description de la correspondance d'une classe

Décrire une correspondance consiste en DRIVER à :

- rapprocher une classe principale et une relation principale;
- définir (éventuellement) les jointures dont on aura besoin;
- rapprocher chaque champ de sa correspondance relationnelle.

L'exemple suivant (listing 3) illustre la définition de correspondances entre les relations `emp`, `person` et `dept`, décrites en §3.1.3.1 (listing 1), et les classes `Employé`, `Vendeur` et `Département`, décrites pour leur part en 3.1.3.2 (listing 2).

Listing 3. Description de correspondances en DRIVER

```
(driver-defclass-map Employe emp
   (letjoins
       ((j1 (emp (p person)
            ((lambda (a1 a2 a3 a4)
                (and (eq a1 a3) (eq a2 a4)))
             (emp ename) (emp fname) (p pname) (p fname))))
        (j2 (emp (emp1 emp)
            ((lambda (a1 a2) (eq a1 a2))
               (emp mgr) (emp1 empno))))
        (j3 (emp (sg salgrade)
            ((lambda (a1 a2 a3)
               (and (>= a1 a2) (<= a1 a3)))
             (emp sal) (sg losal) (sg hisal))))
        (j4 (emp dept
            ((lambda (a1 a2) (eq a1 a2))
             (emp deptn) (dept deptno))))
        (j5 (emp address
            ((lambda (a1 a2) (eq a1 a2))
             (emp empno) (address empno))))
        (j6 (p vehicle
            ((lambda (a1 a2) (eq a1 a2))
             (p car) (vehicle vnum)))))
      (fields (nom       (emp ename) ())
              (prenom    (emp fname) ())
              (num-ss    (p ssnum)   (j1))
              (qualif    (emp job)   ())
              (chef      ()          (j2))
              (salaire   (emp sal)   ())
              (grade     (sg grade)  (j3) (readonly t))
              (dpt       ()          (j4))
              (adresse   ()          (j5))
              (tel-prive (p phone)   (j1))
              (vehicule  ()          (j1 j6)))))

(driver-defclass-map Vendeur
   (fields (situ-famille (p position)      ((Employe emp p)))
           (commission   (emp com)         ())
           (sal-total    ((lambda (a1 a2) (+ a1 a2))
                          (emp sal) (emp com))
                                          ())))

(driver-defclass-map Departement dept
   (letjoins
       ((j1 (dept (e emp)
            ((lambda (attr1 attr2) (eq attr1 attr2))
```

```
      (dept deptno) (e deptn))))
 (j2 (dept (ds deptsite)
     ((lambda (a b) (eq a b))
      (dept deptno) (ds dept))))
 (j3 (dept (dp deptproducts)
     ((lambda (a b) (eq a b))
      (dept deptno) (dp deptno)))))
(attrconstraint ((lambda (no) (> no 0)) (dept deptno))
                ())
(fields (nom       (dept dname)   ())
        (produits  (dp pref)      (j3))
        (sites     (ds sname)     (j2))
        (chefs     ()             (j1)
           (orders ((e sal) desc)
                   ((e ename) asc))
           (readonly t))
        (personnel ()             (j1) (readonly t))
        (moy-sal   ((lambda (a1) (avg a1)) (e sal))
                                  (j1)))))
```

Dans cet exemple, la classe `Employé` est associée à la relation `emp` tandis que la classe `Département` est associée à la relation `dept`.

Le nom d'une classe principale est suivie du nom de la table qu'on lui associe. Ce nom de table est inutile à rappeler pour les sous-classes (ou classes secondaires) qui, comme nous l'avons vu, sont associées à la même relation que leur classe principale. C'est la raison pour laquelle la correspondance `emp` de `Vendeur` n'est pas reprécisée.

L'association classe - relation est suivie des descriptions des correspondances des champs. Ces descriptions sont regroupées en une liste introduite par le mot réservé fields. Lorsqu'au moins une des correspondances comprend une jointure, cet environnement fields est englobé dans un environnement letjoins qui permet de décrire les jointures pour les référencer ensuite dans les correspondances de champs.

Des paramètres optionnels permettent de préciser la correspondance de la classe. Ici encore, une option se présente sous forme d'une liste composée d'au moins deux éléments dont le premier identifie l'option et le ou les suivants donnent son expression ou sa valeur.

- **Classe sans correspondance relationnelle** Il peut être utile de déclarer temporairement une classe sans correspondance, notamment dans la phase de mise-au-point. Cette option est faite pour cela.

  Elle est introduite par le mot réservé mappedp et sa valeur est booléenne.

- **Définition de contrainte d'attribut** Les contraintes sur les champs atomiques d'une classe se traduisent en contraintes sur les attributs correspondants lors du chargement des objets en mémoire. Il est aussi parfois utile d'imposer des contraintes sur des attributs de relations élémentaires de la classe qui ne sont les correspondances d'aucun champ atomique. Cette option permet de le faire. Ces contraintes s'ajoutent aux contraintes de classes et participent également aux opérations de filtrage d'objet lors des transferts de données en mémoire.

  Ceci offre la possibilité de déterminer la classe d'un objet relationnel au sein d'une hiérarchie en fonction de la valeur d'un attribut sans correspondance. De façon indirecte, la correspondance d'un tel attribut n'est pas un champ mais une classe : la valeur de l'attribut détermine la classe de l'objet. Une telle utilisation de cette option est faite au sein du méta-schéma (cf. §3.3.2.4).

  Cette option est identifiée par le mot réservé attrconstraint qui introduit un ou deux arguments. Le premier est l'expression de la contrainte, à nouveau présentée sous forme d'une lambda-expression à un argument qui a valeur de prédicat. Quand on l'applique à une valeur de l'attribut concerné, la contrainte est vérifiée si le résultat est vrai (au sens lisp); elle ne l'est pas dans le cas contraire. La lambda-expression doit être précisée de l'attribut associé à son argument. On la place dans une liste dont elle est le premier élément et où le second est l'attribut correspondant. Si l'attribut n'est pas dans la table élémentaire, une chaîne de jointures élémentaires permettant de l'atteindre doit être précisée. Cette chaîne constitue le deuxième argument facultatif de l'option.

  Notre application présente un exemple de contrainte d'attribut. L'attribut `deptno` de la relation `dept` n'a pas de champ atomique associé. Nous souhaitions le contraindre de façon à ne retenir comme objets relationnels que les n-uplets présentant une valeur positive pour cet attribut. Comme l'attribut appartient à la relation principale, aucune jointure n'est nécessaire; c'est pourquoi le deuxième argument du attrconstraint vaut ()).

Examinons maintenant comment on décrit une jointure en DRIVER. Nous nous intéresserons ensuite aux correspondances de champs.

3.2.5.2 Description d'une jointure

Les jointures sont définies au sein de l'environnement letjoins qui les rend disponibles localement, au sein de l'environnement fields englobé. Le letjoins prend deux arguments (figure 3.24) : le premier est une liste de déclarations de variables de jointures, le second est l'environnement fields où ces variables sont utilisables. Une déclaration de variable de jointure associe un nom de variable à une description de jointure.

(LETJOINS (<jvar1> ... <jvarN>) <fields-env>)
où
<jvarP> est de la forme **(<var-name> <join-declaration>)**
et
<join-declaration> est de la forme
(<begin-table-name> <end-table-name> <join-body>)
et
<join-body> est de la forme
(<lambda-expr> <attr1> ... <attrN>)

Figure 3.24 : L'environnement letjoins

Comme nous l'avons vu, une jointure représente pour nous un lien d'une table vers une autre. Elle se définit par sa table de départ, sa table d'arrivée et son expression.

Le corps de la jointure se décrit sous la forme d'un prédicat à n variables où chacune d'entre elles représente un des attributs intervenant dans la jointure. Ce prédicat est présenté sous la forme d'une lambda-expression[5]. Un n-uplet de valeurs des attributs concernés vérifie la jointure si l'application du prédicat à ces valeurs renvoie un résultat vrai (au sens lisp).

La lambda-expression doit être précisée des attributs associés à ses arguments. On la place dans une liste dont elle est le premier élément et où les suivants sont les attributs correspondants.

Prenons un exemple de déclaration de jointure. On a présenté dans la figure 3.9 une jointure liant la table `person` à la table `emp` dont l'expression est :

```
emp.ename = person.pname and emp.fname = person.fname
```

Présentée sous forme d'un prédicat Lisp où les attributs apparaissent par l'intermédiaire de variables, l'expression devient :

```
(lambda  (var1 var2 var3 var4)
         (and (eq var1 var3) (eq var2 var4)))
```

où `var1`, `var2`, `var3` et `var4` représentent respectivement `emp.ename`, `emp.fname`, `person.pname` et `person.fname`.

[5] Fonction anonyme et générique en lisp.

Si on complète la déclaration en précisant les tables origine et destination et les associations "variable de la lambda" - attribut, on obtient :

```
(emp person ((lambda (var1 var2 var3 var4)
                     (and (eq var1 var3) (eq var2 var4)))
             (emp ename) (emp fname) (person pname) (person fname))
```

On reconnaît ici celle qui a été décrite la première dans le letjoins et qui est identifiée par la variable `j1`. Seule subsiste une petite différence. Par concision, nous avons choisi, dans le listing 3, d'utiliser une table logique `p` plutôt que la table logique `person`.

Lors de la première utilisation d'une nouvelle table logique, il convient de préciser à quelle table physique elle est associée en remplaçant son nom par la liste "nom de table logique" - "nom de table physique" correspondante. Cette opération n'est pas nécessaire pour la première utilisation d'une table logique de même nom que la table physique associée.

Pour les tables intermédiaires qui n'apparaissent dans la déclaration qu'au moment de la description des attributs, le principe reste le même. La description d'un tel attribut devient `((new-log-table phys-table) attr)` au lieu de `(log-table attr)`.

À titre d'exemple, nous avons présenté en §3.2.3.3 la jointure J[`project`→`emp`] qui utilise une table intermédiaire `emproj`. En supposant qu'on souhaite utiliser la table logique `ep` associée à la table physique `emproj` pour exprimer cette jointure, la déclaration de la jointure serait :

```
(project emp
    ((lambda (a1 a2 a3 a4)
             (and (eq a1 a2) (eq a3 a4)))
     (project projno) ((ep emproj) projno) (ep empno) (emp empno))
```

La description des correspondances de la classe `Employé` comprend la définition de l'auto-jointure sur la table `emp` déjà présentée dans la figure 3.23. Cet exemple illustre la manière de décrire les auto-jointures.

Nous avons vu qu'une chaîne de jointures reliait des tables logiques. La correspondance du champ `véhicule` de la classe `Employé` est une chaîne de deux jointures (figure 3.12) qui relie la table `emp` à la table `vehicle` en passant par `person`. En termes logiques, la première jointure établit le lien entre la table `emp` et la table `p(person)`. La seconde doit

alors nécessairement lier la table `p(person)` à la table `vehicle`. Si une autre table logique que `p`, associée à la table physique `person`, y était utilisée, la chaîne serait inconsistante.

Ainsi, au sein d'un letjoins, un nom de table logique ne représente qu'une seule et même table logique. De ce fait, une table logique qui a été déclarée dans une jointure n'a pas à l'être à nouveau si elle est utilisée au sein d'une autre jointure. Dans l'application, `p` représente effectivement la même table logique dans les deux déclarations de jointures `j1` et `j6`.

Lors de la description des correspondances d'une sous-classe, il est parfois nécessaire de référencer une jointure définie lors de la description des correspondances d'une classe ancêtre. Cette jointure peut par exemple permettre d'accéder à un attribut d'une table secondaire de cette classe ancêtre.

Dans la description de la correspondance du champ, on remplace alors le nom de la variable —qui n'a plus aucune signification à cet endroit— par une référence explicite à la jointure en question. Cette référence est une liste de trois éléments comprenant successivement le nom de la classe pour la correspondance de laquelle la jointure a été définie, le nom de la table logique origine et enfin le nom de la table logique destination.

Le champ `situ-famille` de la classe `Vendeur` a pour correspondance l'attribut `position` de la relation `person`. La jointure dont on a besoin et qui lie les relations `emp` et `person` a déjà été décrite lors de la description des correspondances de la classe `Employé`. Il s'agit de la jointure référencée par la variable `j1` et qui joint la table logique `emp` à la table logique `p`. La référence explicite à la jointure en question est donc `(Employé emp p)`. Bien évidemment, l'attribut logique correspondance du champ est `p.position` (cf. §3.2.5.3).

3.2.5.3 Description des correspondances de champs

Quand nous avons établi la correspondance de chaque type de champ (cf. §3.2.3), nous avons associé chacun d'entre eux à un attribut (un calcul sur attributs pour les champs calculés), une ou plusieurs jointures ou aux deux. La syntaxe que nous avons choisie présente la correspondance de tout champ comme étant un couple attribut (ou calcul) - ensemble de jointures. Si l'un de ces deux éléments n'a pas de sens ou n'existe pas pour le champ considéré, on représente son absence par le symbole ().

Les correspondances de champ sont décrites au sein de l'environnement fields. Chacune est introduite par le nom du champ auquel on s'intéresse. Suivent les éventuels attribut (ou calcul d'attributs) et liste de jointures associés, complétés, selon les besoins, d'informations optionnelles.

Pour la plupart des types de champ, on déduit facilement la syntaxe à utiliser à partir des informations ci-dessus et des correspondances établies en §3.2.3.

Précisons seulement, pour les cas où la correspondance est un attribut, que cet attribut

doit appartenir à la table logique finalement atteinte par la chaîne de jointures associée.

Par exemple, le champ `num-ss` de la classe `Employé` est associé à l'attribut `ssnum` de la relation `person`. La jointure permettant d'atteindre cet attribut lie la table logique `emp` à la table logique `p`. De ce fait, la correspondance du champ est nécessairement `p.ssnum`.

Quand la référence de la jointure finale de la chaîne est une référence explicite, le principe reste le même. L'attribut correspondance du champ doit appartenir à la table logique destination de cette jointure. On l'a vu avec la description de la correspondance du champ `situ-famille` de la classe `Vendeur`.

Examinons plus en détail la description de la correspondance des champs "calculés" où l'expression d'un calcul sur la base se substitue à l'attribut.

Dans notre syntaxe, cette expression doit être présentée sous forme d'une lambda-expression dont les arguments représentent à nouveau les attributs intervenant dans le calcul. Comme pour la description des jointures, la lambda-expression doit être précisée des attributs associés aux arguments. On la place dans une liste dont elle est le premier élément et où les suivants sont les attributs correspondants.

Le calcul fait a priori intervenir des attributs de type numérique. Différents opérateurs sont disponibles, selon que le calcul est mono–n-uplet ou multi–n-uplets.

Pour un calcul mono–n-uplet, les quatre opérations de base (addition `add`, soustraction `sub`, multiplication `mul`, division `div`) peuvent être utilisées.

L'expression d'un calcul multi–n-uplets fait intervenir des opérateurs ensemblistes. Deux sont disponibles :

- Le premier `sum` permet d'effectuer la somme d'un certain nombre de valeurs d'un attribut, sélectionnées par jointure(s);
- Le second `average` (ou `avg`) permet d'effectuer la moyenne d'un certain nombre de valeurs d'un attribut, sélectionnées elles aussi au moyen d'une chaîne de jointures.

Deux exemples de notre application illustrent ces deux types de calcul. Les champs correspondant sont `sal-total` de la classe `Vendeur` et `moy-sal` de la classe `Département`. Leurs deux correspondances ont été décrites en §3.2.3.7.

Les paramètres optionnels de correspondance de champ

Les déclarations des correspondances de champs peuvent être complétées à l'aide des paramètres optionnels suivants :

- **Champ à accès en lecture seulement à la base de données**. On peut ou doit parfois préciser qu'un champ n'est qu'une "fenêtre" de consultation de données de la base. Comme nous l'avons déjà vu, ceci est nécessaire quand un attribut participe à plusieurs correspondances et évite d'avoir des problèmes de cohérence des données persistantes.

Cette option est identifiée par le mot réservé readonly. La valeur par défaut est faux (champ à accès en lecture et écriture dans la base), excepté pour les champs de type calculés qui sont de toute façon "readonly", quelle que soit la valeur éventuellement donnée à ce booléen.

Cette option est illustrée, dans notre application, au sein des correspondances des classes `Employé` et `Département`. Le champ `grade` de la classe `Employé` permet de consulter dans la table `salgrade` quel grade correspond à chaque salaire. Si ce champ n'était pas "readonly", la modification d'un grade d'un objet entraînerait lors du report dans la base une incohérence dans cette table : le nouveau grade remplacerait l'ancien pour la plage de salaire correspondante.

Le champ `personnel` de la classe `Département` est déclaré "accès en lecture seulement" car sa correspondance est une jointure objet déjà utilisée pour le champ `chefs`. Elle est également utilisée par le champ `moy-sal`, mais ce dernier est un champ calculé qui est "readonly" a fortiori.

- **Définition de filtres de champs** Des filtres peuvent être mis en place entre un champ et sa correspondance relationnelle. Toute valeur issue de la base et à affecter au champ ou encore contenue dans le champ et à reporter dans la base peut être modifiée par l'intermédiaire de tels filtres de façon à lui donner une forme adaptée à l'usage qu'on souhaite en faire.

  Deux filtres peuvent être créés par champ. L'un reçoit en entrée la valeur issue de la base de données et doit renvoyer la valeur à affecter dans le champ de l'objet. Le second reçoit en entrée la valeur issue du champ et doit renvoyer la valeur à transmettre au monde relationnel.

  Ces filtres sont mis en place sous forme de méthodes qui sont invoquées lors des transferts de données.

  Les méthodes driverloading sont invoquées pour filtrer une donnée issue de la base et destinée à être placée dans le champ d'un objet. Inversement, les méthodes driverwriting sont invoquées pour filtrer un contenu de champ à reporter dans la base de données.

  Une paire de filtres est déclarée pour le champ correspondant en utilisant cette option, introduite par le mot réservé filter. Ce mot réservé doit être suivi du nom donné à la paire de filtres, plus des mots clés précisant certains arguments à passer à ces filtres. La syntaxe de cette option est :

(filter <`filter-name`>)
ou
(filter (<`filter-name`> [class] [field] [object]))

Les crochets [ ] indiquent que le mot clé qu'ils encadrent est optionnel. L'utilisation de l'option filter nécessite la définition par l'utilisateur des deux méthodes de filtrages de noms

#:driverloading:<filter-name>
et
#:driverwriting:<filter-name>

qui assureront le filtrage des valeurs entre le champ correspondant et la base de données.

Si le mot clé filter introduit le seul symbole <filter-name>, les méthodes seront invoquées avec comme unique argument la valeur à filtrer. Si, au contraire, il introduit plusieurs paramètres, le premier doit être le nom du filtre <filter-name> suivi d'un ou plusieurs mots clés parmi class, field et object. Dans ce cas, les méthodes seront invoquées avec comme premier argument la valeur à filtrer, puis, selon les mots clés choisis, l'objet DRIVER qui représente la classe de l'instance à laquelle appartient le champ, l'objet DRIVER qui représente le champ concerné, ou encore l'instance concernée elle-même. Le passage de ces différents objets avec la valeur à filtrer permet de tenir compte du contenu de certains de leurs champs pour déterminer la valeur renvoyée par le filtre.

Il est possible en DRIVER de choisir dans quel ordre vont être chargés les champs d'un objet (voir fonction DRIVER-FIELD-LOADING-ORDER en annexe). Un intérêt de pouvoir choisir cet ordre peut être d'utiliser le contenu de certains champs de l'objet en cours de chargement – champs à charger en priorité donc – pour en construire d'autres.

Un certain nombre de types de champ atomiques ont des filtres qui leur sont associés par défaut. Ces types sont symbol, float, fix, integer et string. Il n'est donc pas nécessaire de définir des filtres par exemple pour un champ atomique de type symbol associé à un attribut relationnel de type string puisqu'il en existe par défaut. Bien entendu, il est tout-à-fait possible de le faire si c'est nécessaire.

À titre d'exemple, le jeu de méthodes défini pour filtrer un champ atomique de type symbol est présenté dans le listing 4.

Nous avons vu en §3.2.3.6 qu'on pouvait rapprocher les correspondances des types de champ à objet de classe indéterminée des types gérant des valeurs atomiques. Le jeu de méthodes de filtrage est lui aussi donné pour exemple dans le listing 4.

Listing 4. Méthodes de filtrage de champs

```
(de #:driverloading:symbol-filter (str)
    ;;; in: string  out: symbol
```

```
    (symbol () str))

(de #:driverwriting:symbol-filter (symb)
    ;;; in: symbol  out: string
    (string symb))

(de #:driverloading:compressed-object-reference (str)
    ;;; in: string  out: object
    (driver-get-compressed-key-object (symbol () str))

(de #:driverwriting:compressed-object-reference (obj)
    ;;; in: object  out: string
    (string (driver-object-compressed-key obj)))
```

- **Champ sans correspondance relationnelle** Il est possible de définir des champs comme étant sans correspondance relationnelle. Comme l'option de même nom pour les correspondances de classe, elle permet de déclarer un champ sans correspondance. Elle est notamment utile quand la correspondance du champ a été décrite et quand on souhaite temporairement ne pas en tenir compte.

  Elle est introduite par le mot réservé `mappedp` et sa valeur est booléenne.

- **Choix de critères de tri pour les champs "ensemble ordonné"** Nous avons vu en §3.2.3.5 que les éléments d'ensemble sont ordonnés d'après les valeurs d'attributs de tables utilisées dans la jointure et en liaison forte avec la table destination. Le premier attribut définit l'ordre primaire, le second l'ordre secondaire, et ainsi de suite.

  L'option est introduite par le mot réservé `orders`. Le ou les attributs doivent être précisés avec, pour chacun d'entre eux, l'ordre ascendant ou descendant (mots clés `asc` et `desc`) dans lequel doit se faire le tri.

  Notre application présente un exemple de champ de type "ensemble ordonné". Ce champ est le champ `chefs` de la classe `Département`. Il est destiné, pour chaque département, à recueillir ses dirigeants, ordonnés par ordre de salaire décroissant, puis, pour les gens ayant même salaire, selon leurs noms par ordre alphabétique.

3.3 Représentation interne des correspondances

3.3.1 Généralités

La description en DRIVER des structures de données relationnelles, des classes et des correspondances se traduit en interne au logiciel par la création d'objets système qui représentent ces informations. Ces objets sont instances de classes qui décrivent chacune un concept ou un type de correspondance.

Les schémas de correspondances sont eux-mêmes des objets, instances de la classe `Driver-mapping`, que les déclarations successives permettent de construire incrémentalement.

Il est possible de leur apporter la persistance car leurs classes sont décrites au sein d'un *méta-schéma de correspondances* `driver-metamapping` où leur sont associées des relations système.

Nous allons étudier la représentation des correspondances et nous intéresser aux différentes classes système définies en DRIVER en examinant le méta-schéma. D'une part, ce dernier comprend la description de la plupart d'entre elles. Enfin, c'est là l'occasion de découvrir un schéma de correspondances complet exploitant une bonne partie des fonctionnalités offertes par le logiciel.

3.3.2 Le méta-schéma de correspondances

3.3.2.1 Les schémas de correspondances

En DRIVER, les schémas de correspondances sont des objets de la classe `Driver-mapping`. Ils sont identifiés par le nom qu'on leur donne à la création. La notion de *schéma courant* permet de savoir à tout moment à quel schéma s'appliquent les déclarations de tables, de classes et de correspondances qui peuvent être faites. Un schéma courant est nécessairement défini à partir du moment où un au moins a été créé. Il ne peut y avoir qu'un schéma courant à un moment donné.

Au sein des objets schémas, la description des classes et de leurs correspondances est accessible par l'intermédiaire du champ `classes` (listing 5).

Les structures relationnelles sont accessibles via différents champs selon qu'on souhaite atteindre des tables ou des attributs, logiques ou physiques (champs `defattributes`, `deftables`, `attributes` et `tables`). Ces champs sont principalement utilisés pour retrouver des objets pendant la phase de construction du schéma. De la même façon, le champ `joins` permet d'accéder rapidement à une jointure définie dans le schéma sans devoir nécessairement explorer les correspondances des différents champs.

Un autre intérêt de ces champs est qu'ils permettent un chargement plus rapide du schéma stocké dans la base de données relationnelles (cf. §4.1.3.1) en augmentant sensiblement le nombre de liens directs existant entre le chargeur[6] (le schéma) et les objets

[6] Voir définition en 4.1.3.3.

qu'il référence. Le coût en requêtes du chargement est également considérablement réduit (cf. §4.1.3.3).

Listing 5. La classe `Driver-mapping` et sa correspondance

```
(driver-defclass Driver-mapping ()
   (name           atom      symbol)
   (classes        ordobjset Driver-class)
   (defattributes  objectset Driver-defattribute)
   (deftables      objectset Driver-deftable)
   (attributes     objectset Driver-attribute)
   (tables         objectset Driver-table)
   (joins          objectset Driver-join))

(driver-deftable db_mapping
   (mapnum  integer 2 keypart)
   (mapname string 20 ()))

(driver-defclass-map Driver-mapping db_mapping
   (letjoins
       ((j1 (db_mapping db_class
             ((lambda (a1 a2) (eq a1 a2))
              (db_mapping mapnum) (db_class mapping))))
        (j2 (db_mapping db_defattribute
             ((lambda (a1 a2) (eq a1 a2))
              (db_mapping mapnum) (db_defattribute mapping))))
        (j3 (db_mapping db_deftable
             ((lambda (a1 a2) (eq a1 a2))
              (db_mapping mapnum) (db_deftable mapping))))
        (j4 (db_mapping db_attribute
             ((lambda (a1 a2) (eq a1 a2))
              (db_mapping mapnum) (db_attribute mapping))))
        (j5 (db_mapping db_table
             ((lambda (a1 a2) (eq a1 a2))
              (db_mapping mapnum) (db_table mapping))))
        (j6 (db_mapping db_join0
             ((lambda (a1 a2) (eq a1 a2))
              (db_mapping mapnum) (db_join0 mapping)))))
       (fields (name          (db_mapping mapname) ())
               (classes       ()                   (j1)
                                      (orders ((db_class c_ord) asc)))
               (defattributes ()                   (j2))
               (deftables     ()                   (j3))
               (attributes    ()                   (j4))
               (tables        ()                   (j5))
               (joins         ()                   (j6)))))
```

Un autre champ, absent dans la déclaration présentée car non persistant, est un dictionnaire d'accès aux objets persistants, instances des classes dont la correspondance est décrite dans ce schéma. Ce dictionnaire permet de retrouver un objet en connaissant sa référence relationnelle.

Enfin, un autre encore contient des objets système DRIVER permettant de gérer la suppression d'objet (cf. §4.3). Ces objets sont utiles pendant les validations de transaction pour déclencher la suppression physique des objets dans la base et des références système aux objets détruits en mémoire, en particulier leur référençage ou le référençage de leurs clés dans les dictionnaires.

La correspondance de la classe `Driver-mapping` est une relation de nom `db_mapping` qui ne comprend que deux attributs. Le premier contient le numéro des schémas et le second, leurs noms. La plupart des champs de la classe `Driver-mapping` étant de type "ensemble d'objets", leurs correspondances sont des jointures. Ces jointures —des égalités entre deux attributs— ne font qu'affirmer à chaque fois la sémantique des attributs `mapping` des relations principales jointes, à savoir, être des références à la clé `mapnum` de la relation `db_mapping`.

Enfin, on notera que l'ordre des objets est important dans le champ `classes`. Cet ordre est celui dans lequel interviennent les différentes classes lors du chargement des objets jumeaux en mémoire (cf. §4.1.2.2). L'ordre est déterminé d'après les valeurs de l'attribut `c_ord` de la relation `db_class`, classées par ordre croissant.

3.3.2.2 Les tables et attributs physiques et logiques

Les tables physiques et les tables logiques sont respectivement représentées par des instances des classes `Driver-deftable` et `Driver-table`, les attributs physiques et les attributs logiques, par des instances des classes `Driver-defattribute` et `Driver-attribute`. Ces quatre classes sont présentées dans le listing 6.

Une table physique comprend un nom et une clé formée d'attributs (champs `name` et `key`). Le nom est unique si bien qu'une table physique peut être retrouvée par son nom. Dans la version actuelle de DRIVER, aucun champ ne renseigne sur les attributs qui la composent car le système n'a à aucun moment besoin de cette information pendant son exécution. Par ailleurs, la primitive `driver-table-attributes` renvoie ce résultat pour les besoins de l'utilisateur.

On peut s'étonner que le champ `key` référence des attributs logiques plutôt que des attributs physiques. Cet état de fait a pour seule raison l'amélioration des performances qu'il apporte dans certaines routines de construction de requêtes. Cette efficacité accrue est même en réalité la seule chose qui justifie l'existence de ce champ.

On peut ajouter que toute création de table physique s'accompagne automatiquement de la création de la table logique de même nom et que toute création d'attribut physique s'accompagne aussi de la création de l'attribut logique de même nom. Ainsi, lors de la

définition d'un nouvel attribut faisant partie de la clé, l'ajout d'un attribut logique dans ce champ `key` ne pose aucun problème. Cet attribut est de toute façon créé en même temps que son homologue physique et est donc disponible.

Listing 6. Les classes de représentation relationnelle

```
(driver-defclass Driver-deftable ()
   (name atom symbol)
   (key  objectset Driver-attribute))

(driver-defclass Driver-defattribute ()
   (deftable object Driver-deftable)
   (attrname atom   symbol)
   (status   atom   symbol)
   (type     atom   symbol)
   (typelen  atom   float))

(driver-defclass Driver-table ()
   (deftable object Driver-deftable)
   (alias    atom   symbol))

(driver-defclass Driver-attribute ()
   (table        object Driver-table)
   (defattribute object Driver-defattribute))
```

Contrairement à la table physique qui ne connaît pas ses attributs, l'attribut physique, lui, possède, par l'intermédiaire de son champ `deftable`, un lien sur la table dont il fait partie. Il a également un nom (champ `attrname`), un type et sa longueur (champs `type` et `typelen`), un statut (champ `status`) qui indique s'il est soumis à contrainte (`unique`, `not-null`) ou s'il fait partie de la clé (`keypart`). On le retrouve par le nom de sa table et son nom propre.

Les tables logiques sont des références logiques de tables physiques. On a vu qu'elles permettaient entre autres la manipulation de plusieurs n-uplets d'une même table au sein de la même expression (filtre, jointure, requête). De façon naturelle, elles ont un nom (champ `alias`) et un lien sur leur table physique (champ `deftable`). On retrouve aussi une table logique par son nom qui est unique.

Le système crée automatiquement les tables logiques de même nom que les tables physiques. Dans ces objets, le champ `alias` vaut simplement `()`. DRIVER retrouve leurs noms au besoin en allant consulter le nom de la table physique référencée dans le champ `deftable`.

Un attribut logique est un attribut de table logique. Il est composé d'un lien sur une table logique (champ `table`) et d'un lien sur un attribut physique (champ `defattribute`).

Il est identifié par les noms de sa table logique et de son attribut physique associés.

Listing 7. Correspondances des classes de représentation relationnelle

```
(driver-deftable db_deftable
   (dtnum   integer  2 keypart)
   (mapping integer  2 ())
   (dtname  string  20 ()))

(driver-deftable db_defattribute
   (danum    integer  2 keypart)
   (mapping  integer  2 ())
   (daname   string  20 ())
   (deftable integer  2 ())
   (status   string  20 ())
   (type     string  20 ())
   (typelen  float    4 ()))

(driver-deftable db_table
   (tnum     integer  2 keypart)
   (mapping  integer  2 ())
   (deftable integer  2 ())
   (talias   string  20 ()))

(driver-deftable db_attribute
   (anum    integer 2 keypart)
   (mapping integer  2 ())
   (table_  integer 2 ())
   (dattr   integer 2 ()))

(driver-defclass-map Driver-deftable db_deftable
   (letjoins
       ((j1 (db_deftable (attr db_attribute)
             ((lambda (a1 a2 a3 a4 a5 a6 a7 a8 a9)
                 (and (eq a1 a2) (eq a3 'keypart)
                      (eq a4 a2) (null a5)
                      (eq a6 a7) (eq a8 a9)))
              ((da db_defattribute) deftable) (db_deftable dtnum)
              (da status) (db_table deftable) (db_table talias)
              (attr dattr) (da danum) (attr table_) (db_table tnum)))))
       (fields (name (db_deftable dtname) ())
               (key  ()                   (j1) (readonly t)))))

(driver-defclass-map Driver-defattribute db_defattribute
   (letjoins ((j1 (db_defattribute (dt db_deftable)
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_defattribute deftable) (dt dtnum)))))
             (fields (deftable ()                         (j1))
                     (attrname (db_defattribute daname)  ())
```

```
                    (status   (db_defattribute status)  ())
                    (type     (db_defattribute type)    ())
                    (typelen  (db_defattribute typelen) ())))))

(driver-defclass-map Driver-table db_table
   (letjoins ((j1 (db_table (dt db_deftable)
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_table deftable) (dt dtnum)))))
             (fields (deftable ()                (j1))
                     (alias    (db_table talias) ()))))

(driver-defclass-map Driver-attribute db_attribute
   (letjoins ((j1 (db_attribute db_table
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_attribute table_) (db_table tnum))))
              (j2 (db_attribute (da db_defattribute)
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_attribute dattr) (da danum)))))
             (fields (table        () (j1))
                     (defattribute () (j2)))))
```

Les correspondances définies pour les champs de ces classes sont particulièrement simples pour la plupart, des attributs de tables principales et des jointures entre deux attributs.

Seule la jointure correspondance du champ `key` de la classe `Driver-deftable` est un peu complexe. Elle fait intervenir neuf attributs appartenant à quatre tables différentes. Elle lie la table des attributs logiques à celle des tables physiques. Son but est de sélectionner, pour chaque table physique, les attributs logiques élément de clé de même nom que leurs attributs physiques.

À partir de la table physique, on sélectionne les attributs physiques éléments de clé par la jointure A :

```
db_defattribute.deftable = db_deftable.dtnum
   and db_defattribute.status = 'keypart'
```

Toujours à partir de la table physique, on sélectionne la table logique de même nom par la jointure B :

```
db_table.deftable = db_deftable.dtnum
     and db_table.talias is null
```

Enfin, on identifie les attributs logiques en indiquant qu'ils sont associés aux attributs physiques filtrés par la jointure A et qu'ils ont pour table logique celle sélectionnée par la jointure B. Ceci définit la jointure C :

```
db_attribute.dattr = db_defattribute.danum
  and db_attribute.table_ = db_table.tnum
```

L'association de ces trois jointures A, B et C permet d'identifier les attributs logiques souhaités pour chaque table physique.

On notera que beaucoup d'attributs intervenant dans cette jointure ont déjà une correspondance dans le schéma. Pour que cette définition de jointure soit acceptée par DRIVER, on est tenu de déclarer le champ comme "read only". Le fait que ce champ ne puisse pas être écrit dans la base n'a pas d'importance puisque la seule chose qui conditionne l'appartenance d'un attribut à la clé d'une table est la valeur de son champ `statut` qui doit valoir keypart pour que ce soit le cas.

3.3.2.3 Classes et correspondances de classes

Les classes et leurs correspondances sont décrites dans le schéma par les objets de la classe `Driver-class` (listing 8). Ces objets servent en principe à décrire la correspondance des classes. Cependant, leur rôle les amène à également contenir la description des classes elles-mêmes.

Un objet (correspondance de) classe possède un nom (champ `name` qui l'identifie de façon unique.

L'éventuelle classe mère peut elle-même être décrite en DRIVER, auquel cas cette description (également instance de `Driver-class`) est référencée dans le champ `super`. On voit ici que l'implémentation actuelle du système ne prévoit que l'héritage simple. Prendre en compte l'héritage multiple ne se traduirait, au niveau de la représentation des connaissances de DRIVER, que par la transformation du type du champ `super` de object en ordobjset.

Listing 8. La classe `Driver-class` et sa correspondance

```
(driver-defclass Driver-class ()
   (name         atom      symbol)
   (table        object    Driver-table)
   (fields       ordobjset Driver-field)
   (restrictionp atom      symbol)
   (restrictions objectset Driver-restriction)
   (super        object    Driver-class)
```

```
   (mappedp      atom      symbol))

(driver-deftable db_class
   (cnum    integer  2 keypart)
   (cname   string  20 ())
   (mapping integer  2 ())
   (c_ord   integer  2 ())
   (table_  integer  2 ())
   (restrip string   1 ())
   (super   integer  2 ())
   (mappedp string   1 ()))

(driver-defclass-map Driver-class db_class
   (letjoins ((j1 (db_class db_table
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_class table_) (db_table tnum))))
              (j2 (db_class db_field
                   ((lambda (a1 a2) (eq a1 a2))
                    (db_class cnum) (db_field class))))
              (j3 (db_class db_join0
                   ((lambda (a1 a2) (eq a1 a2))
                    (db_class cnum) (db_join0 class))))
              (j4 (db_class (mother db_class)
                   ((lambda (a1 a2) (eq a1 a2))
                    (db_class super) (mother cnum)))))
             (fields (name         (db_class cname)   ())
                     (table        ()                 (j1))
                     (fields       ()                 (j2)
                                        (orders ((db_field f_ord) asc)))
                     (restrictionp (db_class restrip) ())
                     (restrictions ()                 (j3)
                                        (filter cl-restrictions))
                     (super        ()                 (j4))
                     (mappedp      (db_class mappedp) ()))))
```

Les contraintes imposées à la classe sont contenues dans le champ `restrictions`. Ce champ est en fait géré comme un index où sont placés des couples clé-valeur. Les valeurs jouent le rôle de contraintes et les clés qui permettent d'y accéder sont les objets contraints. Ce sont soit des champs, soit des attributs généralement sans correspondance. Une contrainte posée sur un champ dans une classe ancêtre peut être redéfinie. Dans ce cas, la clé est bien sûr l'objet champ de la classe ancêtre.

Quand la classe est contrainte, le champ `restrictionp`, qui est un booléen, a une valeur "vraie". On peut faire la remarque que le champ `restrictions` peut être vide, et `restrictionp` à "vrai". Cela arrive quand aucune contrainte n'a été définie spécifiquement pour la classe, mais qu'au moins une autre existe au niveau d'une classe ancêtre. La classe en hérite alors naturellement.

On a vu que la correspondance d'une classe était une table logique (cf. §3.2.2 et §3.2.4). Cette table est conservée dans le champ `table`. Elle est nécessairement la même que celle de la table principale. Dans l'implémentation actuelle du système, dans un but de simplification, cette correspondance est forcément une table logique dont le nom est identique à celui de sa table physique associée.

Les correspondances des champs sont contenues dans le champ `fields`. L'ordre dans lequel ces objets sont placés détermine celui dans lequel les champs vont être chargés de leurs valeurs à la construction des jumeaux.

Le champ `mappedp` permet de déclarer provisoirement une classe comme étant sans correspondance, sans devoir mettre à vide les champs `table` et `fields`. Il suffit en effet pour cela de simplement mettre à "faux" ce champ dont la valeur est généralement "vraie". Une nouvelle inversion de la valeur de ce booléen rétablit la persistance de la classe.

Enfin, `Driver-class` comprend quatre autres champs, absents dans la déclaration présentée car non persistants.

Le premier n'est utilisé que dans les classes principales. Il contient pour ces classes une table d'accès aux identificateurs relationnels des objets persistants (cf. §3.2.2). La clé d'un identificateur est l'objet persistant lui-même.

Le second contient une version "compilée" des contraintes de la classe et des jointures qui permettent d'atteindre les attributs contraints.

Les deux autres référencent respectivement un objet Driver-select (cf. §4.1.4) et un objet Driver-class-saving (cf. §4.2.3.2) dès qu'ils ont été construits. Ces deux objets contiennent des informations relatives au chargement et à l'insertion des instances de la classe.

La correspondance ne présente aucune particularité si ce n'est le filtre défini pour le champ `restrictions`. On a vu que le contenu de ce champ n'était pas directement l'ensemble des contraintes appliquées à la classe, mais un index permettant de les manipuler facilement. Un filtre est donc nécessaire pour transformer les ensembles de contraintes lus dans la base en index avant affectation dans le champ et inversement, les index en simples ensembles de contraintes avant écriture dans la base.

3.3.2.4 Champs et correspondances de champs

Les champs et leurs correspondances sont décrits par les objets de classe principale `Driver-field` (listing 9). De façon similaire aux classes, ces objets servent à décrire la correspondance des champs. Cependant, leur rôle les amène à également contenir la description des champs eux-mêmes.

Toute une hiérarchie de classes de champ a été définie sous `Driver-field` (figure 3.25) et permet de représenter les correspondances spécifiques à chaque type de champ.

`Driver-field` n'est pas la classe d'un type particulier de correspondance de champ

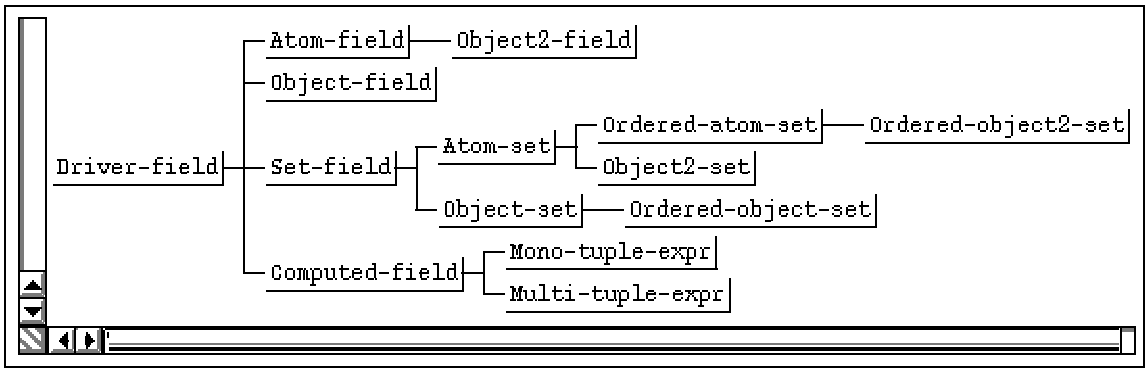

Figure 3.25 : Hiérarchie des classes de champ

et aucun objet ne peut en être précisément instance. Ses champs décrivent simplement les informations communes à tous les types.

Les objets (correspondance de) champ ont un nom (champ `name`) —qui est le nom du champ— et qui les identifie de façon unique au sein de leur classe. Cette dernière est référencée dans `ownerclass`.

La plupart d'entre eux, en fait tous sauf les champs à objet de classe connue (`Object-field`, `Object-set` et `Ordered-object-set`), contiennent des valeurs atomiques ou des ensembles de valeurs atomiques, telles des entiers, des réels, des symboles ou des chaînes de caractères. Ce sous-type, quand il existe, est alors contenu dans `ftype`.

Il est toujours possible de définir un filtre pour un champ, quel que soit son type. Un filtre en comprend en réalité deux. La ou les valeurs lues dans la base transitent par l'un avant leur affectation dans le champ tandis que le contenu du champ transite par le second avant d'être reporté dans la base. Ce doublet est identifié par un nom (cf. §3.2.5.3). Le système a connaissance qu'un filtre a été défini sur un champ quand le champ `filter` de l'objet correspondant contient précisément ce nom.

Nous avons vu qu'il était possible de préciser une valeur initiale pour un champ. Le champ `init-val` contient alors l'expression (au sens Lisp) qui est évaluée à chaque création d'objet. C'est le résultat de cette évaluation qui est aussitôt affecté au champ correspondant.

Listing 9. La classe Driver-field et sa correspondance

```
(driver-defclass Driver-field ()
   (name       atom      symbol)
   (ownerclass object    Driver-class)
   (ftype      atom      symbol)
   (filter     atom      symbol)
   (init-val   atom      ())
   (mappedp    atom      symbol)
```

```
   (read-only  atom      symbol)
   (localp     atom      symbol)
   (joins      objectset Driver-join))

(driver-deftable db_field
   (fnum       integer  2 keypart)
   (fname      string  20 ())
   (classfield string  20 ())
   (type       string  20 ())
   (class      integer  2 ())
   (f_ord      integer  2 ())
   (filter     string  20 ())
   (init_val   string  50 ())
   (mappedp    string   1 ())
   (r_only     string   1 ())
   (localp     string   1 ())
   (attr       integer  2 ())
   (objclass   integer  2 ())
   (a_lambda   string 200 ())
   (groupedp   string   1 ()))

(driver-deftable db_fieldjoins
   (field integer 2 keypart)
   (join  integer 2 keypart))

(driver-defclass-map Driver-field db_field
   (letjoins ((j1 (db_field db_class
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_field class) (db_class cnum))))
              (j2 (db_field db_join0
                  ((lambda (a1 a2 a3 a4) (and (eq a1 a2) (eq a3 a4)))
                   (db_field fnum) (db_fieldjoins field)
                   (db_fieldjoins join) (db_join0 jnum)))))
             (fields (name       (db_field fname)    ())
                     (ownerclass ()                  (j1)
                     (ftype      (db_field type)     ())
                     (filter     (db_field filter)   ())
                     (init-val   (db_field init_val) ())
                     (mappedp    (db_field mappedp)  ())
                     (read-only  (db_field r_only)   ())
                     (localp     (db_field localp)   ())
                     (joins      ()                  (j2))))))
```

Comme pour les classes, `mappedp` précise si le champ a une correspondance, ou non. La valeur de `mappedp` est généralement "vrai". Elle peut aussi être provisoirement mise à "faux" pour des champs dont on a pourtant décrit la correspondance mais dont on souhaite momentanément "débrancher" la persistance. Ce peut être intéressant, par exemple,

pendant la phase de mise-au-point du schéma.

On a vu qu'on pouvait décrire en DRIVER au niveau d'une classe donnée la correspondance d'un champ hérité d'une classe ancêtre (cf. §3.1.3.2). En général, cette classe ancêtre n'est pas décrite dans le schéma de correspondances et n'existe que dans le modèle objet client. Lors de la description de la classe, on indique que le champ déclaré n'est pas un de ses champs propres en positionnant la valeur du champ `localp` à "faux". Ce champ, qui a valeur de booléen comme le précédent, vaut donc "vrai" dans la plupart des cas.

Enfin, `joins` contient la chaîne de jointures participant à la correspondance du champ. Il n'aura pas échappé au lecteur que tous les types de champ peuvent en effet avoir recours aux jointures pour atteindre un ou plusieurs attributs ou pour sélectionner un ensemble de valeurs pertinentes.

La hiérarchie des types de champ est un exemple de hiérarchie de classes associée à une relation où le critère qui détermine la classe de chaque objet relationnel est la valeur d'un attribut sans correspondance. Nous avons même, dans le cas présent, isomorphisme entre les valeurs possibles de l'attribut et les classes de la hiérarchie des champs.

Un n-uplet de la relation `db_field` détermine la classe précise de son objet relationnel associé en fonction de la valeur de l'attribut `classfield`. Chaque valeur possible de l'attribut correspond à un type de champ. La correspondance de la classe `Driver-field` elle-même n'est pas contrainte parce qu'aucun n-uplet de la relation `db_field` n'est à rejeter.

Une jointure peut servir à définir la correspondance de plusieurs champs et une correspondance peut elle-même faire intervenir plusieurs jointures. Le lien qui unit les objets champ et les objets jointure est donc de type N:M. Comme nous l'avons vu, une table indépendante des tables élémentaires de chaque classe est alors nécessaire pour gérer ce lien. Cette table est la table `db_fieldjoins`. La jointure correspondance du champ `joins` utilise le fait que les deux attributs de cette table sont des références aux clés des relations `db_field` et `db_join0`.

Le champ atomique

Les champs atomiques "classiques" sont représentés par des instances de la classe `Atom-field`. Ce sont ceux que nous désignons sous le terme `atom` lors de leurs déclarations. Ils ont pour correspondance un attribut logique, qui est recueilli dans `attr`. Nous avons vu que l'éventuelle chaîne de jointures permettant d'atteindre l'attribut était contenue dans `joins` qui est local à la classe `Db_field`.

La correspondance de cette classe de champ n'a rien de particulier. On remarquera simplement l'expression de la contrainte imposée à l'attribut `classfield` pour la classe `Atom-field`. On constate aussi ici que `Object2-field` est une sous-classe de `Atom-field`.

Listing 10. La classe `Atom-field` et sa correspondance

```
(driver-defclass Atom-field Driver-field
   (attr  object Driver-attribute))

(driver-defclass-map Atom-field db_field
   (attrconstraint ((lambda (attr) (memq attr '(atom object2)))
                    (db_field classfield)))
   (letjoins ((j1 (db_field db_attribute
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_field attr) (db_attribute anum)))))
             (fields (attr () (j1)))))
```

Correspondance des champs calculés

La classe `Computed-field` correspond à une catégorie de champs atomiques un peu particulière puisqu'il s'agit de celle des champs calculés. Ces champs contiennent également des valeurs atomiques, mais qui sont les résultats de calculs sur la base. La correspondance d'un champ de ce type est cette fois l'expression d'un calcul.

Nous avons vu que les termes de ce calcul sont exprimés par l'utilisateur sous forme d'une lambda-expression, englobée dans un environnement où on précise quels attributs sont associés aux variables (cf. §3.2.5.3). Cette lambda-expression "englobée" est conservée dans le champ `a-lambda` (figure 11).

Listing 11. Les classes des champs calculés

```
(driver-defclass Computed-field Driver-field
   (a-lambda atom ()))

(driver-defclass Mono-tuple-expr Computed-field)

(driver-defclass Multi-tuple-expr Computed-field
   (groupedp   atom symbol))

(driver-defclass-map Computed-field db_field
   (attrconstraint ((lambda (attr) (memq attr '(monotexpr multitexpr)))
                    (db_field classfield)))
   (fields (a-lambda (db_field a_lambda) ()
                           (filter cf-alambda))))

(driver-defclass-map Mono-tuple-expr db_field
   (attrconstraint ((lambda (attr) (eq attr 'monotexpr))
                    (db_field classfield))))
```

```
(driver-defclass-map Multi-tuple-expr db_field
   (attrconstraint ((lambda (attr) (eq attr 'multitexpr))
                    (db_field classfield)))
   (fields (groupedp   (db_field groupedp) ())))
```

Bien que seul champ déclaré de la classe `Computed-field`, `a-lambda` est en fait accompagné de deux autres. Ceux-là n'ont pas d'intérêt à être persistant car leurs contenus peuvent être déduits de celui de `a-lambda`. Le premier contient l'expression du calcul compilée sous forme d'objets système. Ces objets, qui sont décrits en §4.1.4, permettent notamment de générer facilement l'expression du calcul sous une forme SQL. Le second champ contient l'ensemble des attributs logiques associés aux variables de la lambda-expression.

La classe `Computed-field` a deux sous-classes, `Mono-tuple-expr` et `Multi-tuple-expr`. Elles correspondent aux deux types de calcul possibles sur la base, calcul mono–n-uplet et calcul multi–n-uplets (cf. §3.1.3.2 et §3.2.3.7), identifiés par les mots réservés monotexpr et multitexpr.

La classe `Mono-tuple-expr` n'a pas de champ propre, persistant ou non.

La classe `Multi-tuple-expr` en possède trois, dont un seul est persistant. Le champ `groupedp` contient un booléen qui vaut "vrai" quand la clause SQL group by est nécessaire dans la requête. Sa valeur est "vrai" par défaut.

Les deux autres permettent de stocker l'objet requête correspondant au champ ainsi qu'un dictionnaire donnant accès aux valeurs de ce champ pour chaque objet. La clé d'une valeur est l'identificateur relationnel de l'objet "propriétaire". Ce dictionnaire est construit après interrogation de la base (cf. §4.1.4.3). Pendant le processus de construction des objets en mémoire, il est utilisé pour restituer à chaque objet sa valeur pour ce champ.

Un filtre a été défini entre le champ `a-lambda` et son attribut correspondant. Lors de l'écriture dans la base, il assure la conversion de l'expression Lisp en chaîne de caractères.

Par ailleurs, un appel à la méthode driver-object-loaded termine le chargement d'un objet (cf. §4.1.2.2). Les deux méthodes définies pour `Mono-tuple-expr` et `Multi-tuple-expr` assurent la reconstruction des champs non persistants.

Correspondance du champ objet

La classe `Object-field` est la classe des champs de type "objet". Elle n'a qu'un champ persistant `class` (listing 12) qui contient la classe de l'objet contenu. Comme nous l'avons vu, la chaîne de jointures correspondance de ce type de champ est stockée dans un champ propre à la classe `Driver-field`.

Listing 12. Le champ objet et sa correspondance

```
(driver-defclass Object-field Driver-field
   (class object Driver-class))

(driver-defclass-map Object-field db_field
   (attrconstraint ((lambda (attr) (eq attr 'object))
                    (db_field classfield)))
   (letjoins ((j1 (db_field db_class
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_field objclass) (db_class cnum)))))
             (fields (class () (j1)))))
```

`Object-field` possède aussi quatre autres champs, non persistants car non décrits dans les déclarations de la classe.

L'un d'entre eux contient toutes les jointures élémentaires de la chaîne correspondante, c'est-à-dire toutes sauf la dernière qui est la jointure objet. Un autre contient l'ensemble des attributs composant la clé de la relation associée à la classe de l'objet contenu. Un autre encore contient l'ensemble des attributs comparés par égalité aux attributs prédédents dans la jointure objet. Ce sont les valeurs de ces derniers qui sont demandées au SGBD dans la requête de la classe et qui permettent de déterminer la référence des objets contenus. Enfin, le dernier contient des informations relatives à la composition de la jointure objet.

Une méthode driver-object-loaded a également été définie pour la classe `Object-field`. Elle est appelée dès qu'un objet champ de type "objet" a été chargé. Elle assure ici aussi la reconstruction des champs non persistants à partir des informations contenues dans ceux qui le sont.

Correspondance des champs ensemble

La classe `Set-field` est la classe des champs de type "ensemble". Elle ne possède aucun champ persistant propre mais deux champs non persistants qui jouent les mêmes rôles que les champs non persistants de la classe `Multi-tuple-expr`. Nous avons vu que, à l'instar de ces champs calculés "multi–n-uplets", les champs "ensemble" nécessitent une requête individuelle pour déterminer leurs valeurs. L'un de ces deux champs non persistants contient la requête de lecture du champ dès qu'elle est construite, l'autre, un dictionnaire qui, après interrogation de la base, contient les valeurs pour ce champ des objets en cours de chargement. Ces valeurs sont accessibles par les identificateurs relationnels des objets.

Listing 13. Les champs "ensemble"

```
(driver-defclass Set-field Driver-field)

(driver-defclass Atom-set Set-field
   (attr object Driver-attribute))

(driver-defclass Ordered-atom-set Atom-set
   (orders ordobjset Driver-order))

(driver-defclass Object-set Set-field
   (setobjclass object Driver-class))

(driver-defclass Ordered-object-set Object-set
   (orders ordobjset Driver-order))
```

`Atom-set` est la classe des champs de type "ensemble d'atomes". Elle comprend un champ `attr` qui référence l'attribut logique correspondance du champ.

`Ordered-atom-set` est une sous-classe d'`Atom-set`, spécialisée sur les ensembles ordonnés d'atomes. L'ordre des éléments est précisé par des objets `Driver-order` contenu dans le champ `orders`. Le premier précise l'ordre primaire, le second l'ordre secondaire, et ainsi de suite.

`Object-set` est la classe des champs de type "ensemble d'objets" de classe connue. Elle possède un champ persistant `setobjclass` qui référence la classe de l'objet contenu. Un autre, non persistant, existe. Il contient l'ensemble des attributs composant la clé de la relation associée à la classe des objets contenus.

`Ordered-object-set` est le pendant de la classe `Ordered-atom-set` pour `Object-set`. Elle possède le même champ `orders` pour gérer l'ordre des objets dans chaque ensemble.

Listing 14. Correspondances des champs "ensemble"

```
(driver-defclass-map Set-field db_field
   (attrconstraint ((lambda (attr)
                      (memq attr '(atomset ordatomset objectset
                                   ordobjset object2set ordobj2set)))
                    (db_field classfield))))

(driver-defclass-map Atom-set db_field
   (attrconstraint ((lambda (attr) (memq attr '(atomset ordatomset
                                                object2set ordobj2set)))
                    (db_field classfield)))
```

```
   (letjoins ((j1 (db_field db_attribute
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_field attr) (db_attribute anum)))))
             (fields (attr () (j1)))))

(driver-defclass-map Ordered-atom-set db_field
   (attrconstraint ((lambda (attr) (memq attr '(ordatomset ordobj2set)))
                    (db_field classfield)))
   (letjoins ((j1 (db_field db_order
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_field fnum) (db_order field)))))
             (fields (orders () (j1) (orders ((db_order rank) asc))))))

(driver-defclass-map Object-set db_field
   (attrconstraint ((lambda (attr) (memq attr '(objectset ordobjset)))
                    (db_field classfield)))
   (letjoins ((j1 (db_field db_class
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_field objclass) (db_class cnum)))))
             (fields (setobjclass () (j1)))))

(driver-defclass-map Ordered-object-set db_field
   (attrconstraint ((lambda (attr) (eq attr 'ordobjset))
                    (db_field classfield)))
   (letjoins ((j1 (db_field db_order
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_field fnum) (db_order field)))))
             (fields (orders () (j1) (orders ((db_order rank) asc))))))
```

Les classes `Object2-set` et `Ordered-object2-set` sont les sous-classes respectives de `Atom-set` et `Ordered-atom-set`. Ceci explique les contraintes imposées à l'attribut `classfield` pour ces deux classes.

Les champs objet de classe inconnue

Un champ de type "objet de classe inconnue" est associé, nous l'avons vu, à un attribut dans lequel est stockée la référence relationnelle de l'objet sous forme d'une chaîne de caractères. De ce fait, toutes les classes de cette famille sont les sous-classes des classes "atomiques" correspondantes, à savoir `Atom-field` pour `Object2-field`, `Atom-set` pour `Object2-set` et `Ordered-atom-set` pour `Ordered-object2-set`.

Listing 15. Les champs objet2 et leurs correspondances

```
(driver-defclass Object2-field Atom-field)
```

```
(driver-defclass Object2-set Atom-set)

(driver-defclass Ordered-object2-set Ordered-atom-set)

(driver-defclass-map Object2-field db_field
   (attrconstraint ((lambda (attr) (eq attr 'object2))
                    (db_field classfield))))

(driver-defclass-map Object2-set db_field
   (attrconstraint ((lambda (attr) (eq attr 'object2set))
                    (db_field classfield))))

(driver-defclass-map Ordered-object2-set db_field
   (attrconstraint ((lambda (attr) (eq attr 'ordobj2set))
                    (db_field classfield))))
```

De ce fait, les correspondances à ces classes ne comprennent que les restrictions des contraintes imposées aux classes mères.

3.3.2.5 La classe Ordre et sa correspondance

Un objet de la classe `Driver-order` permet de décrire dans quel ordre des n-uplets solutions à une requête doivent être retournés à DRIVER pour traitement. Cet ordre conditionne celui dans lequel les valeurs issues des n-uplets vont être placées dans le champ.

L'ordre est déterminé par la valeur d'un attribut stocké dans le champ `attr`. Le tri peut se faire par ordre croissant ou décroissant. Selon le cas, le champ `ordtype` vaut `asc` (ascendant) ou `desc` (descendant).

Listing 16. La classe Ordre et sa correspondance

```
(driver-defclass Driver-order ()
   (attr    object Driver-attribute)
   (ordtype atom   symbol))

(driver-deftable db_order
   (ordnum  integer 2 keypart)
   (field   integer 2 ())
   (attr    integer 2 ())
   (ordtype string  4 ())
   (rank    integer 2 ()))

(driver-defclass-map Driver-order db_order
   (letjoins ((j1 (db_order db_attribute
```

```
        ((lambda (a1 a2) (eq a1 a2))
         (db_order attr) (db_attribute anum)))))
(fields (attr    ()                  (j1))
        (ordtype (db_order ordtype) ()))))
```

Dans notre implantation, un objet `Driver-order` est attaché à un champ. C'est pourquoi la relation `db_order` contient le numéro du champ correspondant.

Ce champ, qui est bien sûr de type "ensemble ordonné", peut être en relation avec plusieurs objets `Driver-order`. L'un définit l'ordre primaire, un autre l'ordre secondaire, et ainsi de suite. Le rang de chacun est stocké dans l'attribut `rank` de la relation `db_order`.

De façon réflexive, cet attribut est utilisé dans la correspondance des champs `orders` des classes `Ordered-atom-set` et `Ordered-object-set`. Il détermine lors du chargement l'ordre des objets `Db_order` dans ces champs.

3.3.2.6 Jointures, contraintes et leurs correspondances

Les jointures et les contraintes sont, en DRIVER, des objets instances de classes ayant même racine `Driver-join0` (cf. figure 3.26). Leurs natures sont similaires. Leur rôle commun est de préciser quelles règles doivent respecter les n-uplets filtrés pour être des objets relationnels de telle ou telle classe.

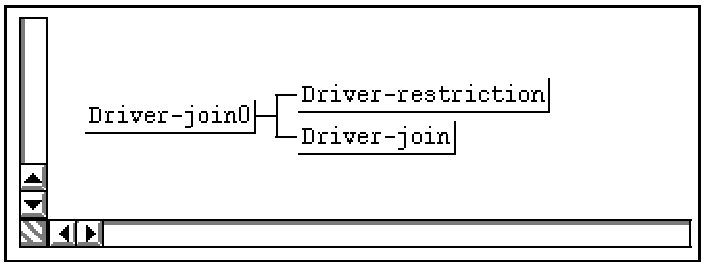

Figure 3.26 : Hiérarchie des classes de contraintes et jointures

Dans chaque cas, leur expression est demandée au concepteur du schéma sous forme d'une lambda-expression, nue pour les contraintes, englobée pour les jointures dans un environnement où on précise quels attributs logiques sont associés aux variables (cf. §3.1.3.2 et §3.2.5.2). Cette lambda-expression "englobée" (qui est reconstituée en ce qui concerne les contraintes) est conservée dans le champ `a-lambda` (listing 17).

Listing 17. Jointures et contraintes

```
(driver-defclass Driver-join0 ()
   (a-lambda atom ()))
```

```
(driver-defclass Driver-restriction Driver-join0
   (class object    Driver-class)
   (field object    Driver-field)
   (attr  object    Driver-attribute)
   (joins objectset Driver-join))

(driver-defclass Driver-join Driver-join0
   (table1 object Driver-table)
   (table2 object Driver-table))
```

`Driver-join0` possède un autre champ, non persistant cette fois, qui contient la représentation objet de la "a-lambda". Cette représentation est construite par le compilateur *CLERIC*[7] de DRIVER (cf. §4.4.2). Elle est formée d'objets système dont les classes sont décrites en §4.1.4. Ces objets répondent à un certain nombre de messages, notamment à `generate-sql` qui retourne une expression SQL directement utilisable dans une requête.

Une contrainte aide à préciser une classe : cette classe est référencée dans `class`. Elle est posée sur un champ ou un attribut; selon le cas, les champs `field` et `attr` permettent d'accéder à l'objet correspondant.

Enfin, l'éventuel attribut sur lequel est posé la contrainte peut se trouver dans une table secondaire associée à la classe. Dans ce cas, la chaîne de jointures qui permet de l'atteindre est contenue dans `joins`.

Une jointure permet d'atteindre une table logique cible à partir d'une table logique de départ. La première est accessible par le champ `table1`, la seconde par le champ `table2`.

Listing 18. Correspondances des jointures et contraintes

```
(driver-deftable db_join0
   (jnum     integer  2 keypart)
   (mapping  integer  2 ())
   (jtype    string  12 ())
   (a_lambda string 200 ())
   (class    integer  2 ())
   (field    integer  2 ())
   (attr     integer  2 ())
   (table1   integer  2 ())
   (table2   integer  2 ()))

(driver-deftable db_restrijoins
   (restri integer 2 keypart)
   (join   integer 2 keypart))
```

[7] **C**ompilateur de **L**ambda **E**xpression en **R**eprésentation **I**ntermédiaire de **C**ontrainte.

```
(driver-defclass-map Driver-join0 db_join0
   (fields (a-lambda (db_join0 a_lambda) ()
                          (filter j0-alambda))))

(driver-defclass-map Driver-restriction db_join0
   (attrconstraint ((lambda (attr) (eq attr 'restriction))
                    (db_join0 jtype)))
   (letjoins ((j1 (db_join0 db_class
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_join0 class) (db_class cnum))))
              (j2 (db_join0 db_field
                   ((lambda (a1 a2) (eq a1 a2))
                    (db_join0 field) (db_field fnum))))
              (j3 (db_join0 db_attribute
                   ((lambda (a1 a2) (eq a1 a2))
                    (db_join0 attr) (db_attribute anum))))
              (j4 (db_join0 (join db_join0)
                   ((lambda (a1 a2 a3 a4)
                       (and (eq a1 a2) (eq a3 a4)))
                    (db_join0 jnum) (db_restrijoins restri)
                    (db_restrijoins join) (join jnum)))))
             (fields (class () (j1))
                     (field () (j2))
                     (attr  () (j3))
                     (joins () (j4)))))

(driver-defclass-map Driver-join db_join0
   (attrconstraint ((lambda (attr) (eq attr 'join))
                    (db_join0 jtype)))
   (letjoins ((j1 (db_join0 (t1 db_table)
                  ((lambda (a1 a2) (eq a1 a2))
                   (db_join0 table1) (t1 tnum))))
              (j2 (db_join0 (t2 db_table)
                   ((lambda (a1 a2) (eq a1 a2))
                    (db_join0 table2) (t2 tnum)))))
             (fields (table1 () (j1))
                     (table2 () (j2)))))
```

L'appartenance d'un objet relationnel à telle ou telle classe est fonction de la valeur de l'attribut `db_join0.jtype` qui vaut restriction ou join selon les cas.

Un filtre a été défini pour le champ `a-lambda` de `Driver-join0`. Il permet de transformer l'expression Lisp en chaîne de caractères pour l'écriture dans la base, et vice-versa.

Des méthodes driver-object-loaded ont été définies pour chacune des classes

`Driver-restriction` et `Driver-join`. Elles assurent la reconstruction de la représentation objet de la "a-lambda" à partir du contenu des différents champs persistants.

Chapitre 4

COMMUNICATIONS ET TRANSFERTS DE DONNÉES

4.1 Chargement des objets relationnels

4.1.1 Généralités

4.1.1.1 Déclenchement d'un chargement

Dans la couche objet virtuelle, la construction d'un objet jumeau peut survenir de deux façons différentes :

- Elle peut d'une part être explicitement demandée. Pour des raisons d'efficacité, il est en effet parfois préférable de charger au préalable dans l'environnement des objets dont on sait qu'on va ensuite en avoir besoin. Quand les accès aux objets relationnels sont nombreux, il peut être lourd de les charger au coup par coup, au hasard d'accès à des objets encore représentés par des défauts d'objet.
- Elle se produit également par effet de bord, à la suite d'un envoi de message ou d'une tentative d'accès sur un objet relationnel encore représenté par un défaut d'objet. L'objet jumeau est construit et substitué au défaut avant que ne soit réellement effectué l'envoi de message ou l'écriture - lecture du champ.

Dans le premier cas, l'utilisateur fait directement appel à la primitive driver-load-class-objects. Cette fonction permet de charger dans l'environnement un ensemble d'objets relationnels de même classe principale. Sont requises en arguments la classe concernée et les valeurs des clés des objets relationnels à charger.

Dans le second, le système invoque la primitive driver-load-object-default dont l'algorithme est présenté figure 4.2. Cette fonction fait elle-même appel à driver-load-class-objects.

4.1.1.2 Chargement à partir d'un défaut d'objet

Un défaut d'objet peut au besoin procéder au chargement de l'objet relationnel qu'il représente car il contient toutes les informations nécessaires pour l'identifier et le charger.

La classe driver-object-default, classe des défauts d'objets, comprend un champ key, défini pour recueillir l'identificateur relationnel de l'objet représenté (listing 19). Le champ mapping contient le schéma dans lequel sont décrites ses correspondances.

Deux autres champs contiennent des informations qui peuvent être déduites des deux premières. Elles sont malgré tout déterminées et stockées une fois pour toute à la création du défaut car elles sont coûteuses à calculer mais nécessaires pour toutes opérations de

lecture de données dans la base qui implique le défaut.

Listing 19. Classe des défauts d'objet

```
(driver-defclass driver-object-default ()
   (mapping      object Driver-mapping)
   (main-class   object Driver-class)
   (key          atom   symbol)
   (expanded-key atom   ()))
```

La première, contenue dans le champ expanded-key, est la valeur de la clé du n-uplet de l'objet relationnel représenté. Elle est déduite de l'identificateur de l'objet, composé, rappelons-le, du nom de la relation principale et d'une valeur de clé dans cette relation.

La seconde est la classe principale de l'objet relationnel représenté. Elle est déduite du nom de la relation principale et du schéma dans lequel sont précisées les correspondances de chaque relations. Une fois déterminée, elle est conservée dans le champ main-class.

Un défaut d'objet est présenté figure 4.1. Il représente l'objet relationnel de classe `Vendeur` et de nom `allen`.

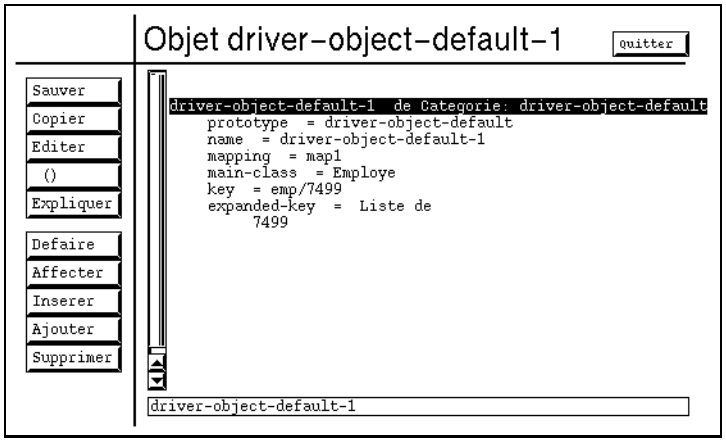

Figure 4.1 : Un défaut d'objet dans l'environnement de SMECI

On remarquera (listing 19) que driver-object-default est décrite dans le modèle objet de DRIVER. Ce n'est bien sûr pas pour lui apporter la persistance, ce qui n'a aucun sens. L'intérêt réside dans le fait que cette déclaration se traduit par la création de la classe correspondante dans le modèle client.

driver-object-default ne peut être qu'une classe du modèle client et en aucun cas une classe système. Lors de la construction d'un objet jumeau précédemment représenté dans l'environnement par un défaut d'objet, il est nécessaire de faire muter l'objet correspon-

dant de la classe driver-object-default vers la classe de l'objet relationnel. Pour que cette opération soit possible, les deux classes doivent être définies dans le même modèle objet, donc dans le modèle client.

4.1.1.3 Gestion des objets manquants

algorithme driver-load-object-default($Default$)
avec driver-current-mapping(mapping.$Default$)
  faire driver-load-class-objects(main-class.$Default$, {expanded-key.$Default$});
si driver-object-default-p($Default$) = $\top$
alors faire driver-missing-object-processing($Default$);
retourner $Default$;
fin

algorithme driver-missing-object-processing($Default$)
faire error("Missing relational object",$Default$);
fin

Figure 4.2 : Algorithme de chargement d'un défaut d'objet

La figure 4.2 présente l'algorithme de chargement d'un défaut d'objet. L'opération consiste principalement à charger l'objet relationnel au moyen de la fonction driver-load-class-objects après avoir précisé le schéma de travail.

Il peut arriver que le défaut d'objet ait été défini pour un identificateur relationnel ne correspondant à aucun n-uplet. Ceci peut se produire par exemple quand l'intégrité référentielle n'a pas été rétablie après une suppression de n-uplet effectuée dans la base par une transaction antérieure ou concurrente. Le défaut représente alors un objet relationnel dit *manquant*.

La tentative de chargement d'un tel objet échoue naturellement car le SGBD ne trouve aucune solution à la requête définie pour lire dans la base le n-uplet correspondant. Faute d'informations, la fonction driver-load-class-objects laisse le représentant de l'objet manquant sous forme de défaut d'objet.

Un défaut d'objet représente nécessairement un objet relationnel manquant quand une tentative de chargement ne produit aucun effet sur lui et qu'il reste un défaut d'objet. La primitive driver-missing-object-processing est alors appelée. Cette fonction provoque par défaut une erreur "Missing relational object", mais elle peut être redéfinie par l'utilisateur pour effectuer un traitement propre à son application.

4.1.2 L'algorithme de chargement

4.1.2.1 Coût en requêtes

Comme tous les algorithmes de DRIVER amenés à dialoguer avec le SGBD sous-jacent, l'algorithme de chargement des objets load-class-objects (figure 4.3) est conçu pour minimiser le nombre de requêtes adressées à la base. En effet, les coûts en temps de l'envoi d'une requête, de son traitement par le SGBD et de la récupération du résultat rendu sont tels que ce choix était nécessaire pour obtenir les meilleures performances. On peut ajouter que la complexité d'une requête n'influe que très peu sur le temps de traitement dont la plus grande partie est due aux transferts de données entre l'environnement et la base.

Dans cette perspective, nous nous sommes attachés à minimiser le nombre de requêtes nécessaires pour charger un groupe d'objets d'une classe donnée. DRIVER permet le chargement de la plupart des champs propres à une classe au moyen d'une *requête principale*, la réponse à cette requête comprenant un n-uplet par objet relationnel. Seuls les champs de type "ensemble" et "calcul multi–n-uplets" requièrent d'être chargés à part, au moyen d'une requête supplémentaire pour chacun d'entre eux. En effet, les réponses à ces requêtes comportent plusieurs n-uplets pour une seule valeur de champ (cf. §4.1.4.3).

Si une requête permet de charger en une fois tous les champs propres à une classe (abstraction faite des champs de type "ensemble" et "calcul multi–n-uplets"), elle ne permet cependant pas de charger en même temps les champs hérités des classes ancêtres. Une requête est nécessaire par classe élément de la hiérarchie. Ainsi, par exemple, si un n-uplet de la relation `emp` (figure 3.1) est un objet relationnel de la classe `Chef-de-service` (figure 3.3), trois requêtes sont nécessaires pour le charger : une pour charger les champs de la classe `Employé`, une pour charger ceux de la classe `Cadre`, et enfin, une dernière pour charger ceux de la classe `Chef-de-service`.

Par ailleurs, on ignore de quelle classe sera précisément un objet avant de le charger; on n'est sûr que de sa classe principale. Pour cette raison, le système est tenu d'envoyer au SGBD une requête pour chacune des classes de la hiérarchie. Quand l'objet est au moins instance de la classe pour laquelle une requête a été envoyée, la réponse comprend un n-uplet contenant les valeurs de ses champs propres à cette classe. Dans le cas contraire, elle ne comprend aucun n-uplet pour l'objet en question.

Certains se demanderont pourquoi tous les champs de l'objet ne peuvent être chargés en une seule fois. Cela n'est en fait possible que si on connaît la classe précise des objets concernés avant l'envoi de la requête de chargement.

Un autre algorithme de chargement aurait en effet pu être d'interroger d'abord la base sur la classe des objets à charger, puis d'envoyer une requête pour chaque classe dont on a au moins un objet à charger. Un avantage de cet algorithme aurait été d'éviter que DRIVER ait besoin d'une primitive de mutation de classe (cf. §3.1.2.3). En effet, on connaît ici la classe des objets au moment de les créer.

Soient N le nombre de classes de la hiérarchie, M le nombre de champs de type "ensemble" et "calcul multi–n-uplets" qu'elle comprend,
le coût C de cet algorithme (en nombre de requêtes) est :

$$C = (N + \alpha + \beta)$$

$$\text{avec } 1 \leq \alpha \leq N \text{ et } 0 \leq \beta \leq M.$$

Une requête est en effet nécessaire par classe pour déterminer quels objets en sont instances. De une à N autres sont également nécessaires pour charger les objets, selon qu'au mieux, ils sont tous précisément de la même classe, et, qu'au pire, l'ensemble comprend au moins un objet de chaque classe. β représente pour sa part le nombre de champs de type "ensemble" et "calcul multi–n-uplets" que comprennent les classes dont au moins un objet est à charger.

Pour la même opération, le coût C' de l'algorithme utilisé par DRIVER est :

$$C' = (N + \beta)$$

Il nécessite une requête par classe, plus une requête par champ de type "ensemble" et "calcul multi–n-uplets" à charger. Comme ne sont traités parmi ces champs que ceux qui appartiennent aux classes dont au moins un objet est à charger, β a la même valeur dans les deux coûts. On constate que l'algorithme utilisé par DRIVER est plus économique. On verra d'autre part que, du fait d'une optimisation, son coût est plus exactement *majoré* par $(N + \beta)$ (cf. §4.1.2.3).

Afin de préserver sa clarté, l'algorithme de chargement présenté dans la figure 4.3 ne comprend pas l'optimisation dont on vient de faire état. La version complète sera explicitée dans un deuxième temps (figure 4.4).

4.1.2.2 Détails de l'algorithme

La fonction driver-load-class-objects, accessible à l'utilisateur final, permet de charger dans l'environnement en une opération un nombre quelconque d'objets relationnels de même classe principale. Les objets choisis doivent être désignés au moyen de la valeur de clé de leurs n-uplets respectifs. Si la liste de clés est remplacée par le symbol t, tous les objets de la classe sont chargés.

L'ensemble des valeurs de clé indiquées est compilé sous forme d'une expression restrictive permettant, lors des requêtes au SGBD, de ne filtrer que les n-uplets des objets concernés. Cette expression est en réalité structurée sous forme d'objets système dont les classes sont décrites en §4.1.4. Ces objets répondent à un certain nombre d'envois de message comme generate-sql qui retourne une expression SQL directement utilisable dans une requête. L'objet qui représente cette expression restrictive est désigné sous le

```
algorithme driver-load-class-objects(Main_class, Object_keys)
soit Object_filter_operator ← build-object-filter(Main_class, Object_keys);
faire load-class-objects(Main_class, Object_filter_operator);
fin
```

```
algorithme load-class-objects(Main_class, Object_filter_operator)
soit Classes ← {Main_class} ∪ all-subclasses(Main_class);
pour tout C ∈ Classes
    faire load-class-part-objects(C, Object_filter_operator);
fin
```

```
algorithme load-class-part-objects(Class, Object_filter_operator)
soit Select ← class-select(Class)
et   Tuples ← ask-database(Select, Object_filter_operator);
pour tout T ∈ Tuples
    faire build-class-part-object(Class, Select, T);
fin
```

```
algorithme build-class-part-object(Class, Select, Tuple)
soit Key ← object-key-from-tuple(Select, Tuple)
et   Object ← get-object(Key);
faire change-object-class(Class, Object);
pour tout F ∈ fields.Class
    faire si mappedp.F = ⊤
        alors soit Fvalue ← field-value-from-tuple(F, Select, Tuple);
            faire set-field-value(F, Object, Fvalue);
faire driver-object-loaded(Class).Object;
fin
```

Figure 4.3 : Algorithme de chargement d'objets de même classe

nom d'*opérateur de filtrage d'objets* (`Object_filter_operator` dans les descriptions d'algorithme).

Une fois l'opérateur de filtrage construit, driver-load-class-objects fait appel à load-class-objects dont l'algorithme est présenté dans la figure 4.3.

load-class-objects utilise la primitive all-subclasses pour déterminer l'ensemble des classes héritant de la classe principale. Ceci permet d'effectuer ensuite, pour chacune d'entre elles, le chargement de leurs instances sélectionnées par l'opérateur de filtrage.

L'ordre dans lequel se trouvent les classes dans la variable *Classes* est important. Une classe fille doit toujours être placée après sa mère. En effet, quand, suite à l'interrogation de la base, un objet est reconnu comme étant au moins instance de la classe en cours de traitement, il est muté vers cette nouvelle classe. De cette façon, au fil du processus s'affine la classe de chaque objet.

Reprenons l'exemple du n-uplet de la relation `emp`, support d'un objet relationnel de la classe `Chef-de-service`. Au début du chargement, la variable *Classes* vaut {`Employé`, `Vendeur`, `Cadre`, `Président`, `Chef-de-service`}. Le processus commence donc par le chargement des instances de la classe `Employé`. Chef de service, notre objet relationnel est a fortiori reconnu comme étant un employé. Si l'objet était auparavant représenté dans l'environnement par un défaut d'objet, d'instance de driver-object-default il devient instance d'`Employé`. S'il était déjà représenté par un objet jumeau, de la classe de cet objet jumeau (`Chef-de-service` ou autre) il redevient employé.

Après chargement des champs propres de la classe `Employé`, le processus continue par le chargement des instances de la classe `Vendeur`, car cette classe est la suivante dans la variable *Classes*. Notre objet n'étant pas un vendeur, cette opération n'a aucune répercution sur lui. Suit le traitement de la classe `Cadre`. Il est cette fois concerné. Reconnu comme étant instance de cette classe, d'employé il devient cadre. Les champs propres à la classe `Cadre` sont affectés. La classe `Président` est la suivante. Notre objet n'étant pas reconnu comme président, cette opération n'a pas non plus d'incidence sur lui. Le processus se termine par le chargement des objets `Chef-de-service`. Étant reconnu comme tel, notre cadre est promu chef de service et ses champs propres à cette classe sont construits.

On comprend pourquoi une classe fille ne peut jamais se trouver avant sa mère dans la liste contenue dans la variable *Classes*. Si cette règle n'était pas respectée, l'algorithme de chargement serait mis en défaut. On ne pourrait jamais avoir dans l'environnement des objets jumeaux instances de classes placées avant leurs mères dans la variable *Classes*.

Dans l'exemple précédent, admettons que les classes `Cadre` et `Chef-de-service` soient inversées dans *Classes*. Cette variable aurait pour valeur {`Employé`, `Vendeur`, `Chef-de-service`, `Président`, `Cadre`}. Après traitement de la classe `Chef-de-service`, notre précédent objet serait bien un chef de service, mais avec les champs propres à la classe `Cadre` non affectés. Lors du traitement de la classe `Cadre`, reconnu comme étant l'une de ses instances, il serait muté de `Chef-de-service` à `Cadre`. Outre le fait que sa classe

ne serait pas correcte, on perdrait en plus les valeurs de ses champs propres à la classe `Chef-de-service`.

La solution à ce problème pourrait consister à n'autoriser, en fait de mutation de classe, que la spécialisation des objets. Ce serait oublier que, si la base de données est partagée et si des modifications faites par d'autres utilisateurs se traduisent sur nos objets relationnels par des changements de classes, ceux-ci peuvent être aussi bien des généralisations que des spécialisations. Lors d'un second chargement d'objet relationnel dont la classe a évolué entre-temps, DRIVER doit être capable de ré-ajuster la classe de objet jumeau correspondant, que ce soit dans un sens ou dans l'autre. Généraliser un objet doit donc rester possible.

L'algorithme load-class-part-objects permet de charger les champs propres à une classe d'un groupe d'objets sélectionnés par un opérateur de filtrage d'objets. Il fait appel à la fonction class-select qui lui rend un objet requête de la classe Driver-select (requête de consultation de données, cf. §4.1.4). Les objets requêtes répondent à un certain nombre de messages comme generate-sql qui, rappelons-le, retourne la requête SQL qui leur correspond.

Pour chaque classe, la requête est bien sûr spécifique. On peut malgré tout la traduire de façon générique par :

Parmi les n-uplets de la table principale,
lesquels sont des objets relationnels au moins instances de cette classe ?
Renvoyer la valeur des champs propres à la classe pour ces objets.

Complétée de l'opérateur de filtrage, la requête est restreinte aux seuls objets choisis. La fonction ask-database adresse au SGBD cette requête complétée de l'opérateur de filtrage et retourne l'ensemble des n-uplets correspondant aux objets solution.

L'appel à la fonction build-class-part-object permet ensuite de construire la partie propre à la classe de chacun de ces objets. Le représentant dans l'environnement de l'objet relationnel (jumeau ou défaut) est tout d'abord retrouvé à l'aide de sa référence contenue dans le n-uplet. Si la fonction get-object n'en trouve pas, elle crée alors un défaut d'objet et le retourne. L'objet recueilli est ensuite muté vers la classe considérée puis sont chargés ses champs locaux. Quand un champ a été déclaré "avec correspondance", sa valeur est extraite du n-uplet et calculée par la fonction field-value-from-tuple. Elle est ensuite affectée dans l'objet par la fonction set-field-value.

Une fois tous les champs chargés, le message driver-object-loaded est envoyé à l'objet. La méthode driver-object-loaded, qui par défaut ne fait rien, peut être redéfinie pour chaque classe. En fonction de sa classe, l'objet peut ainsi se voir appliquer des traitements appropriés, comme la reconstruction de champs non persistants à partir des informations contenues dans ceux qui viennent d'être chargés. À titre d'exemple, la recompilation de nombreux objets du schéma de correspondances est effectuée au travers de cet envoi de

message lors du chargement d'un schéma contenu dans la base de données relationnelles (cf. §3.3.2.4 et §3.3.2.6).

4.1.2.3 Optimisation de l'algorithme de chargement

Si on utilise l'algorithme présenté dans la figure 4.3, le coût de chargement d'un ensemble d'objets de même classe principale est de $(N + \beta)$ requêtes, comme nous l'avons vu.

En réalité, celui qui est effectivement utilisé par DRIVER (figure 4.4) a un coût C' qui est plus précisément *majoré* par ce nombre :

$$C' \leq N + \beta$$

Il bénéficie en effet d'une optimisation qui réduit le nombre de requêtes à adresser au SGBD au strict nécessaire.

Au fur et à mesure qu'avance le processus de chargement se précisent les classes des objets en cours de construction. Quand un objet est reconnu comme étant (au moins) instance d'une certaine classe de la hiérarchie, on peut aussitôt en conclure qu'il n'est pas instance d'une des sœurs de cette classe. Du fait de ce principe, on est parfois en mesure de détecter des classes non encore traitées et dont les objets en cours de chargement ne peuvent être les instances. Ces classes et toutes leurs descendances sont alors enlevées de la file d'attente.

L'élagage de l'arbre des classes est très économique car il évite ainsi la construction et l'envoi au SGBD de requêtes sans intérêt qui sont malgré tout aussi coûteuses en temps que les autres.

L'algorithme optimisé est présenté figure 4.4. On peut remarquer que l'ensemble des classes à traiter est géré sous forme d'une file d'attente. L'intérêt de ceci est double. D'une part, il est possible d'accéder par cette file aux classes en attente d'être traitées et de connaître ainsi l'état d'avancement du chargement. Enfin, le contenu de la file peut facilement être modifié, par exemple, pour retirer une classe dont le traitement est inutile.

Dans notre syntaxe, une file se "lit" de la gauche vers la droite, le prochain élément à en sortir étant le plus à gauche.

Illustrons le nouveau processus de chargement par un exemple. Soit la hiérarchie de classes décrite dans la figure 4.5, considérons le chargement de deux objets relationnels désignés par $o5$ et $o8$, instances respectives des classes $C5$ et $C8$. Au début du processus, la variable $Class\_queue$ vaut, par exemple[1], $\{Main\_class, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13\}$.

Consécutivement au traitement de $Main\_class$, aucune réduction de la file d'attente n'est possible car on n'a pas encore suffisamment d'information sur la classe des deux

[1] L'ordre n'est pas strict. La seule règle est qu'une classe fille soit nécessairement derrière sa mère dans la file d'attente.

```
algorithme load-class-objects(Main_class, Object_filter_operator)
soit Class_queue ← all-subclasses(Main_class)
et   Objects ← load-class-part-objects(Main_class, Object_filter_operator);
tant que Class_queue ≠ {}
      soit C ← pop-element(Class_queue);
      faire load-class-part-objects(C, Object_filter_operator)
      et    reduce-class-queue(Class_queue, Objects);
fin
```

```
algorithme load-class-part-objects(Class, Object_filter_operator)
soit Select ← class-select(Class)
et   Tuples ← ask-database(Select, Object_filter_operator)
et   Objects ← {};
pour tout T ∈ Tuples
    soit Object ← build-class-part-object(Class, Select, T);
    faire Objects ← Objects ∪ {Object};
retourner Objects;
fin
```

```
algorithme build-class-part-object(Class, Select, Tuple)
soit Key ← object-key-from-tuple(Select, Tuple)
et   Object ← get-object(Key);
faire change-object-class(Class, Object);
pour tout F ∈ fields.Class
    faire si mappedp.F = ⊤
        alors soit Fvalue ← field-value-from-tuple(F, Select, Tuple);
            faire set-field-value(F, Object, Fvalue);
faire driver-object-loaded(Class).Object;
retourner Object;
fin
```

Figure 4.4 : Algorithme optimisé de chargement d'objets de même classe

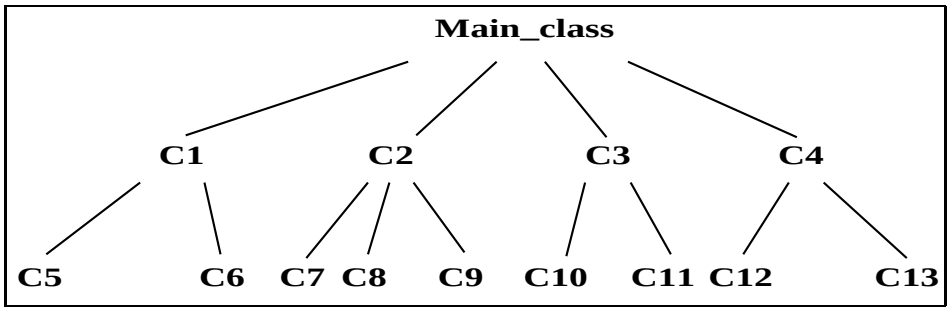

Figure 4.5 : Une hiérarchie de classes

objets. Il en est de même suite au traitement de $C1$, car si $o5$ a été reconnu comme une instance de $C1$, cela n'a pas été le cas de $o8$. On ignore donc encore à quelle sous-hiérarchie appartient cet objet. La fonction reduce-class-queue laisse $Class\_queue$ à $\{C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13\}$.

$C2$ reconnaît ensuite $o8$ comme étant l'une de ses instances. Suite au traitement de cette classe, l'ensemble des classes encore à interroger peut être réduit, car on sait à quelles sous-hiérarchies appartiennent les objets à charger, lesquelles sont pertinentes à explorer. Suite au traitement de $C2$, la fonction reduce-class-queue réduit $Class\_queue$ de $\{C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13\}$ à $\{C5, C6, C7, C8, C9\}$.

De la même façon, suite au traitement de $C5$, reduce-class-queue réduit la file de $\{C6, C7, C8, C9\}$ à $\{C7, C8, C9\}$. On évite ainsi de traiter $C6$. Le traitement de $C7$ qui vient après ne permet pas de réduire la file d'attente. Enfin, celui de $C8$, reduce-class-queue réduit $Class\_queue$ de $\{C9\}$ à $\{\}$. Le chargement est terminé.

On constate, par cet exemple, que la réduction de la file d'attente peut être extrêmement importante et faire gagner un temps considérable. La hiérarchie est formée ici de 14 classes. Si on ignore les champs de type "ensemble" et "calcul multi–n-uplets", 6 requêtes seulement ont été nécessaires pour construire nos deux objets dans leurs classes respectives.

Algorithme de réduction de la file d'attente des classes

L'algorithme de la fonction reduce-class-queue qui réduit la file d'attente des classes est donné dans la figure 4.6.

Son principe est simple. Soit une classe C dans la file d'attente, s'il n'existe pas au moins un objet parmi ceux en cours de chargement dont la classe est ancêtre de C, alors C et toutes ses sous-classes peuvent être enlevées de la file sans préjudice pour le chargement en cours.

algorithme reduce-class-queue($Class\_queue, Objects$)
soit $New\_class\_queue \leftarrow \{\}$;
pour tout $C \in Class\_queue$
    si $\exists\ O \in Objects$, subclass-p(C, object-class(O)) $= \top$
    alors $New\_class\_queue \leftarrow New\_class\_queue\ \cup \{C\}$;
faire $Class\_queue \leftarrow New\_class\_queue$;
fin

Figure 4.6 : Algorithme de réduction de la file d'attente des classes

4.1.3 Chargement récursif d'objets

4.1.3.1 Intérêt et danger

Pour des raisons d'efficacité, il est parfois intéressant de charger dans l'environnement en même temps qu'un objet tous ceux qui lui sont liés, tous ceux qu'il référence directement ou indirectement. Même si ce choix peut être coûteux en occupation mémoire, le gain en performances qui en résulte peut le justifier.

DRIVER a recours à cette fonctionnalité pour charger dans l'environnement un schéma de correspondances contenu dans la base de données.

Nous avons vu que la persistance pouvait être attribuée aux schémas grâce au méta-schéma (cf. §3.3.1). On peut donc a priori choisir comme schéma courant un schéma persistant dont tous les objets contenus sont encore représentés sous forme de défauts d'objet. Si chacun d'entre eux était chargé au besoin, quand il est consulté, les premières utilisations du schéma se traduiraient par un nombre important de requêtes adressées au SGBD, certaines pour les besoins de l'application, la plupart pour la construction en mémoire du schéma.

On comprendra aisément qu'il est beaucoup plus efficace de procéder à un chargement complet du schéma de correspondances avant son exploitation que de construire au coup par coup les objets qui le composent selon les besoins. La différence de coût en requêtes entre les deux méthodes en témoigne (cf. §4.1.3.3).

La plupart des applications n'ont bien entendu pas intérêt à charger récursivement leurs objets persistants. Il n'est pas rare que les réseaux d'objets relationnels contenus dans la base soient tellement étoffés que le chargement récursif d'un objet provoque la construction en mémoire d'une image de la base entière.

Cette fonctionnalité doit donc être utilisée avec circonspection si on souhaite éviter de saturer inutilement l'environnement avec des données sans grand intérêt.

4.1.3.2 Principe et détails de l'algorithme

L'algorithme load-deep-objects (figure 4.7) permet de charger récursivement un groupe d'objets donné.

Son principe général est le suivant. À chaque appel de load-deep-objects, on détermine l'ensemble des objets liés directement ou indirectement à l'un des objets donnés en argument. On extrait ensuite de cet ensemble les défauts d'objets qui s'y trouvent et on procède à leur chargement. Après leur construction, les objets nouvellement chargés sont à leur tour confiés à load-deep-objects.

De cette façon, tous les objets référencés sont chargés de proche en proche. L'appel récursif à load-deep-objects s'arrête quand le groupe d'objets passé en argument ne contient plus directement ou indirectement de défaut d'objet.

```
algorithme driver-load-deep-objects(Objects)
faire driver-rollback-objects(Objects)
et    load-deep-objects(Objects);
retourner Objects;
fin
```

```
algorithme load-deep-objects(Objects)
soit All_objects ← driver-all-linked-objects(Objects)
et   Defaults ← get-defaults(All_objects);
si Defaults ≠ {}
alors soit Main_classes ← involved-main-classes(Defaults);
      pour tout MClass ∈ Main_classes
          soit MClass_objects ← main-class-objects(MClass, Defaults)
          et   Object_keys ← object-default-keys(MClass_objects);
          faire driver-load-class-objects(MClass, Object_keys);
      faire load-deep-objects(Defaults);
fin
```

Figure 4.7 : Algorithme de chargement récursif d'objets

Détails de l'algorithme

Au début du processus, tous les objets à charger sont représentés par des défauts d'objets.

Si certains d'entre eux ont déjà été chargés, l'appel à driver-rollback-objects provoqué par driver-load-class-objects permet de les re-transformer en défauts d'objets. Il est à no-

ter que la fonction driver-rollback-objects n'a d'effet que sur les objets qui lui sont passés en argument. Elle ne change pas en défauts les objets liés aux premiers. Ainsi, quand le chargement récursif d'un objet a déjà été effectué, une seconde tentative ne se traduit que par le rechargement du premier niveau. Pour un réel rechargement récursif, il convient de préciser que l'on souhaite également le rechargement des objets contenus :

```
algorithme load-deep-again(Objects)
soit All_objects ← driver-all-linked-objects(Objects)
faire driver-load-deep-objects(All_objects);
fin
```

la primitive driver-all-linked-objects calculant l'ensemble complet des objets liés directement ou indirectement à l'ensemble passé en argument.

À l'aide de la primitive involved-main-classes sont déterminées les classes principales dont un objet relationnel au moins est à charger. Il va ensuite être procédé au chargement des instances de chacune d'entre elles par un appel à driver-load-class-objects.

Il est intéressant de remarquer que la variable $MClass\_objects$ ne référence pas forcément que des défauts d'objets, comme on serait tout d'abord tenté de le penser. Parfois, lors du chargement d'un champ pour lequel un filtre a été défini, se produisent par effet de bord des chargements d'objets du réseau. L'expression d'un filtre peut en effet provoquer de tels chargements. Pour éviter à ces objets d'être chargés une seconde fois, la primitive objects-default-keys ne retourne que les clés des objets de $Object\_keys$ encore représentés dans l'environnement par un défaut d'objet.

4.1.3.3 Coût de l'algorithme

Encore une fois, l'algorithme a été optimisé de façon à minimiser le nombre de requêtes adressées au SGBD. À chaque appel de load-deep-objects, une seule exécution de driver-load-class-objects charge tous les objets de la classe principale correspondante.

Considérons un réseau de q objets relationnels, chacun étant instance d'une classe parmi N.

Par définition, on appelle *chargeur* de ce réseau un objet ou un ensemble d'objets donnant accès, au travers des différents types de champ objet (object, object2, objectset, ...), à tous les éléments du réseau. Le chargement récursif de ce chargeur se traduit donc par un chargement complet du réseau dans l'environnement.

Soit un objet o_1, chargeur de ce réseau,

Soit $P(\{o_1\})$ le nombre d'appels à load-deep-objects nécessaire pour charger le réseau à partir d'o_1,
le coût C'' de chargement de ce réseau à partir de o_1 est :

$$C''(\{o_1\}) = \sum_{i=1}^{N} \sum_{j=1}^{P(\{o_1\})} C_{ij}$$

où C_{ij} est le coût de chargement d'un groupe d'objets de classe principale i lors du $j^{ème}$ appel à load-deep-objects.

Cette matrice des coûts C est elle-même définie à partir de la matrice de chargement Q.
On a :

$$C_{ij} = \begin{cases} 0 & \text{si } Q_{ij} = 0 \\ C'_i & \text{si } Q_{ij} \neq 0 \end{cases}$$

où Q_{ij} est le nombre d'objets de la classe i chargés lors du $j^{ème}$ appel à load-deep-objects, avec

$$\sum_{i=1}^{N} \sum_{j=1}^{P(\{o_1\})} Q_{ij} = q$$

et où C'_i est le coût de chargement d'un groupe d'objets de classe principale i. Pour simplifier les expressions, on assimilera C'_i à son majorant $N + \beta$ (cf. §4.1.2.3).

C'' dépend du chargeur choisi. Ce coût est en général d'autant plus réduit que le chargeur choisi est en liaison directe avec un maximum d'éléments du réseau.

En comparaison, le coût C''' de chargement du même réseau objet par objet est :

$$C''' = \sum_{i=1}^{q} C'_{classe(q_i)}$$

où q_i est le $i^{ème}$ objet parmi les q, et $C'_{classe(q_i)}$ le coût de chargement d'un groupe d'objets de sa classe principale. C''' est, pour sa part, indépendant du chargeur choisi.

Soit Q_i le nombre d'objets de classe i à charger,

$$Q_i = \sum_{j=1}^{P(\{o_1\})} Q_{ij}$$

C''' devient :

$$C^{'''} = \sum_{i=1}^{N} \sum_{j=1}^{Q_i} \underbrace{C'_{classe(q_j)}}_{C'_i} = \sum_{i=1}^{N} C'_i \sum_{j=1}^{Q_i} 1 = \sum_{i=1}^{N} Q_i.C'_i$$

Comparer $C^{''}(\{o_1\})$ et $C^{'''}$ revient à comparer, en se restreignant à une classe i donnée,

$$\sum_{j=1}^{P(\{o_1\})} C_{ij} \quad \text{et} \quad Q_i.C'_i$$

La valeur de $P(\{o_1\})$ dépend de la structure du réseau[2]. Par conséquent, la présence de cette variable dans une expression de coût restreinte à une classe perd de sa signification et de son intérêt.

Si on fait la remarque que les appels à load-deep-objects pendant lesquels aucun groupe d'objets de la classe i n'est chargé n'ont aucune influence sur la valeur de $C^{''}(\{o_1\})$ restreinte à i, on constate qu'on peut remplacer dans cette expression P par $\min(P(\{o_1\}), Q_i)$.

On a

$$\sum_{j=1}^{\min(P(\{o_1\}),Q_i)} C_{ij} \quad \leq \quad Q_i.C'_i$$

Il y a égalité dans le cas d'une chaîne d'objets :

$$\begin{cases} P(\{o_1\}) = Q_i \\ \forall j \in [1, P(\{o_1\})], \quad Q_{ij} \neq 0 \quad \Rightarrow \quad C_{ij} = C'_i \end{cases}$$

Prenons un exemple. Considérons notre base de données "fil rouge" sur les employés d'une entreprise.

Avec les correspondances définies dans le schéma `map1`, la base comprend 44 objets ($q = 44$) instances de 11 classes différentes. Sept d'entre elles sont principales ($N = 7$).

La hiérarchie de la classe `Employé` en comprend cinq; aucune ne possède de champ de type "ensemble" ou "calcul multi–n-uplets". Son coût de chargement est cinq ($C'_{Employé} = 5$). La classe `Département` n'a pas de sous-classe mais quatre champs "ensemble" et un champ "calcul multi–n-uplets". Son coût de chargement est six ($C'_{Département} = 6$). Suit un tableau récapitulatif de l'ensemble des coûts C'_i de chargement des différentes classes principales :

[2] et du choix du chargeur, bien sûr.

$$\begin{pmatrix} C'_{Employé} \\ C'_{Département} \\ C'_{Projet} \\ C'_{Adresse} \\ C'_{Véhicule} \\ C'_{Software} \\ C'_{Hardware} \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ 3 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Prenons comme chargeur l'objet employé nommé `scott`.

La fonction load-deep-object est appelée la première fois avec un défaut d'objet représentant `scott`.

Les matrices $C(\{\text{scott}\})$ et $Q(\{\text{scott}\})$ valent :

$$C(\{\text{scott}\}) = \begin{pmatrix} 5 & 5 & 5 & 0 & 5 & 0 & 5 & 0 \\ 0 & 6 & 0 & 6 & 0 & 6 & 0 & 0 \\ 0 & 3 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \qquad Q(\{\text{scott}\}) = \begin{pmatrix} 1 & 1 & 5 & 0 & 4 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 5 & 0 & 4 & 0 & 1 \\ 0 & 1 & 1 & 5 & 0 & 4 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Finalement,

$$P(\{\text{scott}\}) = 8, \quad C''(\{\text{scott}\}) = 61 \quad \text{et} \quad C''' = 113$$

Le choix du chargeur influe énormément sur le coût du chargement récursif. Le coût est en général d'autant plus faible que le chargeur est pertinent (!) et qu'il est composé d'un choix judicieux d'objets. À titre de comparaison, voici un exemple de coût de chargement où le chargeur est composé des trois objets `Département` {`accounting`, `research`, `sales`} :

$$C(\{\text{acc., res., sales}\}) = \begin{pmatrix} 0 & 5 & 0 \\ 6 & 0 & 0 \\ 0 & 0 & 3 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \qquad Q(\{\text{acc., res., sales}\}) = \begin{pmatrix} 0 & 12 & 0 \\ 3 & 0 & 0 \\ 0 & 0 & 3 \\ 0 & 0 & 12 \\ 0 & 0 & 11 \\ 0 & 2 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$P(\{\text{acc., res., sales}\}) = 3, \quad C''(\{\text{acc., res., sales}\}) = 18$$

Le coût 18 obtenu est optimum (à comparer à $C''' = 113$) puisque chaque classe n'est interrogée qu'une seule fois.

Il faut noter que ce nombre majore en réalité le coût réel de l'opération puisque les coûts de chargement des différentes hiérarchies, qui varient en fonction des ensembles d'objets à charger, sont eux-mêmes majorés par les valeurs qui leur ont été données.

4.1.4 Les requêtes de consultation

4.1.4.1 Représentation des requêtes de consultation

DRIVER demande au SGBD les valeurs des objets relationnels instances d'une classe au moyen d'une requête de consultation de données. La requête propre à chaque classe est construite après les déclarations de correspondances ou au premier chargement d'objets de la classe, et est ensuite réutilisée pour les suivants. Comme nous l'avons vu, elle est restreinte à un groupe d'objets particulier en la complétant d'un opérateur de filtrage.

Une requête de consultation est représentée en DRIVER par un objet de la classe Driver-select. Sa structure est particulièrement bien adaptée pour construire incrémentalement la requête et générer ensuite son expression en SQL [ANSI86].

Nous n'allons pas ici détailler la syntaxe du langage SQL. Le lecteur peu familier de ce langage pourra en trouver une description complète dans de nombreux ouvrages, par exemple dans [Mir90].

```
SELECT [ALL | DISTINCT] target_list
    FROM table(s)
    [WHERE search_cond]
    [GROUP BY col(s)]
    [HAVING search_cond]
    [UNION subselect]
    [ORDER BY col(s)]
```

Figure 4.8 : Syntaxe d'une requête SQL de consultation de données

La syntaxe générale de la requête SQL de consultation est donnée figure 4.8. Disons, en quelques mots, que la requête est introduite par le mot-clé SELECT, qu'elle s'adresse à un ensemble de tables et reçoit en réponse une table extraite de cet ensemble. Les colonnes de cette table extraite sont les attributs et expressions déclarées dans la `target_list` et ses n-uplets sont formés des valeurs des colonnes qui vérifient les contraintes exprimées dans la `search_cond`.

Des exemples de requêtes SQL de consultation des tables `emp` et `person` sont présentés figure 4.9 et les réponses peuvent être comparées aux contenus des tables correspondantes donnés en §B.5.

La classe Driver-select (listing 20) présente une structure dans laquelle on retrouve les principales articulations d'une requête SQL de consultation de données. Comme Driver-select est en fait une sous-classe de Driver-subselect, les différentes informations sont réparties entre les champs des deux classes.

```
$ sql smecidemo
INGRES TERMINAL MONITOR – Copyright (c) 1981, 1990 Ingres Corporation
INGRES SunOS Version 6.3/02 (su4.u42/05)
continue
* select ename,fname,job from emp where sal>2000;
*\g
```

ename	fname	job
king	paul	president
jones	eric	manager
blake	harold	manager
clark	john	manager
ford	john	analyst
scott	pit	analyst

```
continue
* select emp.ename,emp.deptn,person.phone from emp,person
*     where emp.ename=person.pname and emp.fname=person.fname
*             and person.position='married';
*\g
```

ename	deptn	phone
jones	20	40-783539
clark	10	40-893723
king	20	40-374845
miller	10	40-223233
ward	30	40-378467
scott	20	44-003231

```
continue
* select pname,fname,position from person
*     where not exists(select * from emp
*                           where emp.ename=person.pname and emp.fname=person.fname);
*\g
```

pname	fname	position
clark	laura	married
ford	laura	married
king	anita	married
miller	rita	married
scott	helen	married
ward	suzie	married

Figure 4.9 : Requêtes SQL de consultation de données sous Ingres

Listing 20. Les classes de représentation des requêtes d'interrogation

```
(driver-defclass Driver-subselect ()
   (tables ordobjset Driver-table)
   (where  objectset Driver-join0))

(driver-defclass Driver-select Driver-subselect
   (distinctp   atom      symbol)
   (columns     ordobjset (Driver-attribute Driver-sql-op))
   (grouped-by  atom      symbol)
   (order-by    ordobjset Driver-order))
```

Driver-subselect est la classe des sous-requêtes. Une sous-requête est caractérisée en DRIVER par la forme "*exists(select * from tables where conditions)*". C'est un élément de restriction utilisé dans la partie where d'une autre requête.

Le symbole * qui se substitue à la `target_list` indique que les colonnes de la table résultat doivent être la totalité des attributs des tables interrogées. De ce fait, une sous-requête n'est déterminée que par deux ensembles d'informations, les tables logiques interrogées et les conditions restrictives.

Driver-subselect comprend les deux champs qui permettent de les recueillir. Le premier, tables, contient les tables interrogées et le second, where, les conditions, à savoir, les jointures qui lient les tables entre elles et les restrictions qui sélectionnent les n-uplets solution.

La classe Driver-select comprend un certain nombre de champs propres qui s'ajoutent à ceux qu'elle hérite de Driver-subselect. L'ensemble de ces champs rend possible l'expression de toute requête SQL que le système pourrait avoir besoin d'envoyer au SGBD.

distinctp, qui a valeur de booléen, précise si tous les n-uplets reçus en réponse à la requête doivent être différents.

columns contient la liste des attributs logiques à consulter et des expressions à calculer. Les expressions y sont représentées sous forme d'objets de classe racine Driver-sql-op (cf. §4.4.2). La syntaxe utilisée pour décrire ce champ indique simplement que ce dernier peut recueillir des instances des deux classes Driver-attribute et Driver-sql-op[3].

groupedp, autre booléen, indique si la commande SQL group by doit être utilisée dans la requête. Nous verrons qu'elle est particulièrement utile dans les requêtes de lecture des champs de type "calcul multi–n-uplets" où le regroupement des n-uplets est nécessaire.

Enfin, order-by précise si l'ordre dans lequel sont rendus les n-uplets solutions est important. Dans l'affirmative, le champ contient un ensemble d'objets de la classe Driver-order. Le premier fixe l'ordre primaire, le second, l'ordre secondaire, et ainsi de suite.

[3] Cette syntaxe est en fait interdite. Les deux classes Driver-subselect et Driver-select sont décrites ici dans le langage objet de DRIVER pour des raisons de clarté. Dans l'implémentation, ces deux classes système sont décrites en MicroCeyx.

Dans la version actuelle du système, les commandes union et having d'une requête de sélection n'ont pas de représentation objet car DRIVER ne les utilise pas. Leur éventuelle prise en compte future ne poserait aucun problème.

Comme de nombreuses classes de DRIVER, Driver-subselect et Driver-select répondent au message generate-sql. Leurs méthodes construisent et renvoient sous forme d'une chaîne de caractères la requête SQL correspondant à un objet.

4.1.4.2 Construction d'une requête de classe en lecture

Une requête de classe en lecture permet d'interroger la base sur la valeur des champs propres à cette classe d'un groupe de ses instances. Les champs hérités de classes ancêtres doivent être lus par les requêtes de ces classes.

Un objet relationnel est instance d'une classe s'il vérifie ses contraintes. Afin d'éliminer les n-uplets non concernés, on initialise, à la création de l'objet requête, le champ where en y plaçant un objet de classe Driver-sql-op qui regroupe l'ensemble des contraintes de la classe correspondante (listing 4.10). On le complète des jointures nécessaires pour atteindre les attributs contraints depuis la table principale.

L'examen des n-uplets solution doit permettre de déterminer lequel correspond à quel objet. Dans ce but, le champ columns est affecté de l'ensemble des attributs composant la clé de la relation principale. Chaque valeur de clé identifie un objet.

Pour leurs parts, le champ tables est initialisé avec cette relation principale et distinctp est mis à "vrai".

À titre d'exemple, la figure 4.11 présente un objet requête initialisé pour la classe `Employé`, tel qu'il est retourné par la fonction initialise-class-select (listing 4.10). L'objet CCC[`Employé`][4], qui regroupe les contraintes de la classe `Employé`, est seul dans where car tous les attributs participant à son expression (`emp.sal` et `emp.qualif`) sont dans la table principale.

Chaque champ propre à la classe est ensuite consulté. À chaque fois, l'objet requête est complété d'attributs, d'expressions et de jointures, en fonction du type du champ (cf §3.2.3). La complétion de la requête pour un champ est faite au moyen d'un envoi de message info-for-select sur cet objet champ. Ainsi, selon le type de correspondance, telle ou telle méthode est ainsi atteinte et exécutée :

- La correspondance relationnelle d'un champ atomique est un attribut, éventuellement complété d'une chaîne de jointures permettant de l'atteindre. L'attribut est ajouté au contenu du champ columns et les jointures au contenu du champ where. Quand un élément, attribut (logique) ou jointure, figure déjà dans un champ, il n'y est pas ajouté à nouveau. Quand une nouvelle table logique est introduite dans

[4]CCC pour Complete Class Constraint.

```
algorithme class-select(Class)
soit Select ← initialise-class-select(Class);
pour tout F ∈ fields.Class
    faire si mappedp(F) = ⊤
        alors faire info-for-select(Select).F;
retourner Select;
fin
```

```
algorithme initialise-class-select(Class)
soit Select ← create-select-object()
et   Class_restriction ← complete-class-constraint(Class)
et   Joins ← constraint-joins(Class, Class_restriction);
faire distinctp.Select ← ⊤
et    columns.Select ← table-key(table.Class)
et    tables.Select ← { table.Class}
et    where.Select ← {Class_restriction} ∪ Joins;
retourner Select;
fin
```

Figure 4.10 : Algorithme de construction d'une requête de classe en lecture

Objet de classe Driver-select	
tables	(`emp`)
where	(CCC[`Employé`])
distinctp	t
columns	(`emp.empno`)
grouped-by	()
order-by	()

Figure 4.11 : Objet Driver-select initialisé pour la classe `Employé`

l'objet requête, table de l'attribut ou table d'un attribut utilisé dans une jointure, elle est ajoutée au contenu du champ tables.

Dans notre exemple, le champ nom de la classe Employé est associé à l'attribut emp.ename (figure 3.8). L'appel à la méthode info-for-select pour ce champ se traduit simplement par l'ajout de cet attribut au contenu de columns.

Le champ num-ss de la même classe est associé à l'attribut person.ssnum (figure 3.10). L'envoi du même message sur l'objet num-ss provoque l'ajout de person.ssnum à columns, de person à tables et de J[emp→person] à where.

- La méthode info-for-select des champs calculés mono–n-uplet est peu différente. Elle ajoute, au lieu d'un attribut, l'expression du calcul associée au champ au contenu de columns. Le traitement des tables et des jointures est identique.

- La correspondance d'un champ objet est une chaîne de jointures dont la dernière est une jointure objet stricte (cf. §3.2.3.2). Du fait des restrictions qui lui sont imposées, la jointure objet peut être analysée et disséquée. On en extrait les attributs qui sont comparés aux éléments de clé de la table principale jointe. Ce sont eux qui sont consultés dans la base. La méthode info-for-select des champs objet les ajoute au contenu de columns. Le reste du corps de la jointure objet, quand il existe, est ajouté au contenu du champ where avec les $N-1$ premières jointures de la chaîne.

  Le champ véhicule de la classe Employé est associé à la chaîne de jointures (J[emp→person] J[person→vehicle]) (figure 3.12). L'examen de la jointure objet J[person→
  vehicle] laisse apparaître que l'ensemble des attributs à consulter au titre de ce champ est (person.car). person.car est donc ajouté au contenu de columns. Aucun reliquat de la jointure objet ne subsistant et J[emp→person] étant déjà présente dans where, aucune nouvelle jointure n'est ajoutée au contenu de ce champ.

  Le champ chef de la classe Employé est associé à l'auto-jointure J[emp→emp1(emp)] (figure 3.22). Du fait de la décomposition de la jointure objet, l'appel à la méthode info-for-select pour ce champ se traduit par le simple ajout de l'attribut emp.mgr au contenu de columns.

- Le traitement des champs de type "ensemble" ou "calcul multi–n-uplets" est différent. Ces champs ont besoin d'une requête propre pour déterminer leurs valeurs.

  Un champ de type "ensemble" est associé à une chaîne de jointures qui sélectionne un ensemble de n-uplets par objet solution, c'est-à-dire par n-uplet solution à la requête de classe. Une requête spécifique à chaque champ de type "ensemble" est nécessaire.

  La correspondance d'un champ de type "calcul multi–n-uplets" détermine une valeur par objet. Cependant, les n-uplets sur lesquels portent les calculs sont différents des n-uplets solution à la requête principale. Une requête par champ de type "calcul multi–n-uplets" est également nécessaire.

Objet de classe Driver-select	
tables	(`project emproj emp`)
where	(J[`project→emp`] CCC[`Employé`])
distinctp	t
columns	(`project.projno emp.empno`)
grouped-by	()
order-by	()

Figure 4.12 : L'objet Driver-select du champ `employés` de la classe `Projet`

Les méthodes info-for-select de ces deux classes de champ ne complètent donc pas l'objet requête de la classe mais en construisent de nouveaux. Après leurs constitution, ces objets requêtes sont stockés dans les objets champs correspondants (cf. §3.3.2.4).

Prenons l'exemple du champ `employés` de la classe `Projet`. Ce champ de type "ensemble d'objets" est associé à la jointure J[`project→emp`] (figure 3.14). L'objet requête créé pour ce champ est initialisé pour la classe `Projet`. columns est complété des attributs constituant la clé de la relation principale de la classe des objets référencés, soit `emp`. J[`project→emp`] est insérée dans where. Le champ `employés` devant recueillir des instances de la classe `Employé`, l'objet CCC[`Employé`] est également ajouté au contenu de where. L'objet requête finalement établi par la méthode est présenté figure 4.12.

Objet de classe Driver-select	
tables	(`dept emp`)
where	(CCC[`Département`] J[`dept→emp`])
distinctp	t
columns	(`dept.deptno [avg(emp.sal)]`)
grouped-by	t
order-by	()

Figure 4.13 : L'objet Driver-select du champ `moy-sal` de la classe `Département`

Les champs de type "ensemble d'atomes" sont gérés de façon similaire. La méthode diffère seulement par le fait que le champ columns n'est pas complété de la clé de la relation jointe mais de l'attribut associé au champ et qui contient les valeurs des éléments des ensembles.

Les champs de type "calcul multi–n-uplets" sont également gérés de la même façon. Ici, c'est l'expression du calcul qui est ajouté au contenu du champ where. À titre d'exemple, l'objet requête du champ `moy-sal` de la classe `Département` est présenté

figure 4.13. La correspondance de ce champ est l'expression d'un calcul qui détermine la moyenne des salaires des employés de chaque département (figure 3.21).

Objet de classe Driver-select	
tables	(emp person salgrade)
where	(CCC[Employé] J[emp→person] J[emp→salgrade])
distinctp	t
columns	(emp.empno emp.ename emp.fname person.ssnum emp.job emp.mgr emp.responsible_for emp.sal salgrade.grade emp.deptn person.phone person.car)
grouped-by	()
order-by	()

Figure 4.14 : L'objet Driver-select de la classe Employé

- La méthode info-for-select des champs de type "ensemble ordonné" affecte en plus le champ order-by des ordonnateurs[5] adéquats. Le premier détermine l'ordre primaire, le second l'ordre secondaire, et ainsi de suite. Les éventuelles jointures qui permettent de les atteindre sont ajoutées au contenu du champ where.
- Les méthodes des champs "Objet de classe inconnue" (object2, object2set, ordobj2set) sont pour leurs parts identiques à celles des champs atomiques correspondants (atom, atomset, ordatomset).

La figure 4.14 présente l'objet requête complet correspondant à la classe Employé, tel qu'il est retourné par la fonction class-select.

Ces objets requêtes sont ensuite exploités par la fonction ask-database (figure 4.4) pour interroger le SGBD et charger les données relationnelles pertinentes dans l'environnement. La requête SQL est obtenue en envoyant le message generate-sql à l'objet requête.

Ã titre d'exemples, les requêtes SQL générées pour les objets des figures 4.12, 4.13 et 4.14 sont présentées §4.1.4.4.

4.1.4.3 Exploitation des résultats

La fonction field-value-from-tuple détermine la valeur du champ d'un objet à l'aide de la requête et du n-uplet correspondant à l'objet :

algorithme field-value-from-tuple($Field, Select, Tuple$)
soit $Primary\_value \leftarrow$ primary-field-value-from-tuple($Select, Tuple$).$Field$;
retourner driver-loading-type-filter($Field, Primary\_value$);
fin

[5] Objets de la classe Driver-order.

Une nouvelle fois, un envoi de message est effectué sur l'objet champ pour orienter le traitement des données en fonction du type du champ et de sa correspondance. La valeur "brute" retournée par la méthode primary-field-value-from-tuple appelée est ensuite confiée à la fonction driver-loading-type-filter qui la fait passer dans un filtre driverloading avant de la retourner :

- La méthode primary-field-value-from-tuple des champs atomiques est particulièrement simple. Elle consiste à retourner la valeur de l'attribut associé au champ contenue dans le n-uplet.
- Celle des champs de type "calcul mono–n-uplet" n'est pas différente : la valeur du champ est également directement lue dans le n-uplet.

emp empno	emp ename	emp fname	person ssnum	emp job	emp mgr	emp respons.	emp sal	salgrade grade	emp deptn	person phone	person car
7329	smith	john	J3847Y2	clerk	7902		800.	1	20	44-332047	1550
7499	allen	jack	76373Y	salesman	7698	software/743	1600.	3	30	78-383726	2004
7521	ward	peter	128347	salesman	7698	hardware/53	1250.	2	30	40-378467	1253
7566	jones	eric	B7C123	manager	7839	dept/20	2975.	4	20	40-783539	1975
7654	martin	georges	234FS1	salesman	7698		1250.	2	30	40-276951	3005
7655	james	peter	K4G262	salesman	7698		1350.	2	30	40-449323	
7698	blake	harold	A473SE	manager	7839	dept/30	2850.	4	30	44-183211	1429
7782	clark	john	462SQ1	manager	7839	dept/10	2450.	4	10	40-893723	1230
7788	scott	pit	XA435C	analyst	7566	project/103	3000.	4	20	44-003231	1928
7839	king	paul	A34F4	president		project/102	5000.	5	10	40-223233	1232
7902	ford	john	6D3210	analyst	7566	project/101	3000.	4	20	40-374845	4328
7934	miller	paul	95Z0H5	clerk	7782		1300.	2	10	43-127772	1276

Figure 4.15 : N-uplets solutions à la requête de la classe `Employé`

- La méthode des champ objet va consister à reconstruire l'identificateur relationnel de l'objet contenu (cf. §3.2.2) et à le proposer à la fonction get-object pour retrouver l'objet correspondant dans l'environnement. S'il n'y en a pas encore, un défaut d'objet est créé et retourné.

  La figure 4.15 présente l'ensemble des n-uplets solutions à la requête de la classe `Employé`. Considérons par exemple la construction du champ `dpt` de l'employé `smith`, dont le n-uplet est en première position. L'attribut `emp.deptn` valant 20, l'identificateur de l'objet référencé est `dept/20`. get-object(`dept/20`) retourne l'objet correspondant.

  L'identificateur construit peut ne correspondre à aucun objet de la base. Il s'agit alors d'un identificateur d'objet relationnel *manquant* (cf. §4.1.1.3). Un défaut d'objet est cependant toujours retourné, car get-object n'effectue aucune vérification d'existence sur l'objet dont on lui fournit l'identificateur.

- Les méthodes primary-field-value-from-tuple des champs de type "ensemble" ou "calcul multi–n-uplets" exploitent la requête propre du champ construite par la méthode info-for-select correspondante.

project.projno	emp.empno
101	7788
102	7499
102	7521
101	7566
103	7566
101	7329
102	7654
102	7698
103	7782
101	7839
102	7566
101	7902
103	7934

dept.deptno	avg(emp.sal)
10	2916.667
20	2443.75
30	1660.

Figure 4.16 : N-uplets solutions aux requêtes des champs `employés` de la classe `Projet` et `moy-sal` de la classe `Département`

Lors du chargement d'un groupe de P objets d'une classe comportant un tel champ, la méthode primary-field-value-from-tuple correspondant à ce type de champ est appelée P fois.

Au premier appel, la requête est adressée au SGBD et la réponse est traduite sous forme d'un dictionnaire : les clés sont les identificateurs relationnels et les valeurs sont les valeurs du champ pour les objets correspondant aux identificateurs. Une fois construite, le dictionnaire est stocké dans l'objet champ et la valeur du champ du premier objet traité est retournée.

Aux appels suivants, le système ne fait plus que rechercher dans le dictionnaire la valeur du champ correspondant à l'objet en cours de traitement.

Reprenons comme exemples les champs `employés` de la classe `Projet` et `moy-sal` de la classe `Département`.

Les n-uplets solutions à leurs requêtes respectives (requêtes et objets requêtes présentés en §4.1.4.2) sont donnés figure 4.16. À partir de ces résultats sont construits les dictionnaires d'accès aux valeurs des champs présentées figure 4.17.

Une fois ces dictionnaires construits, l'accès à la valeur d'un champ à partir de l'identificateur relationnel de l'objet est immédiat.

- Les différents types de champs "Objet de classe inconnue" sont gérés de la même façon que les types atomiques correspondants. Seul diffère leur filtre de chargement. Ce dernier fait passer les identificateurs relationnels dans la fonction get-object pour retrouver dans l'environnement l'objet correspondant.

clés	valeurs
project/101	{Object(emp/7788), Object(emp/7566), Object(emp/7329), Object(emp/7839), Object(emp/7902)}
project/102	{Object(emp/7499), Object(emp/7521), Object(emp/7654), Object(emp/7698), Object(emp/7566)}
project/103	{Object(emp/7566), Object(emp/7566), Object(emp/7782), Object(emp/7934)}

clés	valeurs
dept/10	2916.667
dept/20	2443.75
dept/30	1660.

Figure 4.17 : Dictionnaires d'accès aux valeurs des champs `employés` de la classe `Projet` et `moy-sal` de la classe `Département`

4.1.4.4 Optimisation des requêtes de sélection

L'objet CCC[`Employé`] regroupe les contraintes de la classe `Employé`. L'expression SQL qui lui correspond est :

```
((emp.sal >= 700. and emp.sal <= 9999.) or
 (emp.job = 'salesman' and (emp.sal >= 700. and emp.sal <= 9999.)) or
 (emp.job in ('manager','president') and (emp.sal >= 700. and emp.sal <= 9999.)) or
 (emp.job = 'president' and (emp.sal >= 700. and emp.sal <= 9999.)) or
 (emp.job = 'manager' and (emp.sal >= 700. and emp.sal <= 9999.)));
```

Ce genre de restriction SQL est extrêmement coûteuse en temps et en mémoire pour le SGBD qui doit la traiter. Pour sa part, Ingres refuse même de la résoudre et provoque une erreur "requête trop complexe".

DRIVER possède un optimisateur de restriction qui permet de réduire singulièrement ce genre d'expression. Les réductions sont obtenues au moyen des règles d'équivalence présentées dans la figure 4.18.

Considérons par exemple la contrainte CCC[`Employé`] ci-dessus. Son expression peut être exprimée sous la forme :

$$A \vee (B \wedge A) \vee (C \wedge A) \vee (D \wedge A) \vee (E \wedge A)$$

Par applications de la seule règle (7), on constate qu'elle est équivalente à : A.

Les simplifications ne sont pas toujours aussi directes, notamment quand les contraintes sont redéfinies au niveau des sous-classes. Cependant, du fait du mécanisme d'héritage, on arrive souvent à réduire les expressions de façon très importante.

Après éventuelles simplifications, les requêtes SQL générées pour les objets des figures 4.12, 4.13 et 4.14 sont respectivement :

$$C_1 \wedge C_2 = C_2 \wedge C_1 \text{ et } C_1 \vee C_2 = C_2 \vee C_1 \quad (1) \text{ et } (2)$$

$$(\bigwedge_i^N C_i) \vee [Expression] = \bigwedge_i^N (C_i \vee [Expression]) \quad (3)$$

$$Not(\bigwedge_i^N C_i) = \bigvee_i^N Not(C_i) \quad (4)$$

$$Not(\bigvee_i^N C_i) = \bigwedge_i^N Not(C_i) \quad (5)$$

$$C_1 \wedge ... \wedge C_{N-1} \wedge (C_1^{'} \vee ... \vee C_k \vee ... \vee C_N^{'}) = C_1 \wedge ... \wedge C_{N-1} \quad \text{avec } k \in [1, ..., N-1] \quad (6)$$

$$C_1 \vee ... \vee C_{N-1} \vee (C_1^{'} \wedge ... \wedge C_k \wedge ... \wedge C_N^{'}) = C_1 \vee ... \vee C_{N-1} \quad \text{avec } k \in [1, ..., N-1] \quad (7)$$

$$\bigvee_i^N (C = value_i) = C \in \{value_i, i = 1, ..., N\} \quad (8)$$

Figure 4.18 : Règles d'équivalence utilisées pour la réduction des restrictions SQL

- ```
  select distinct project.projno,emp.empno
  from   project,emproj,emp
  where  project.projno = emproj.projno and
         emproj.empno = emp.empno and
         (emp.sal >= 700. and emp.sal <= 9999.);
  ```

- ```
  select   distinct dept.deptno,avg(emp.sal)
  from     dept,emp
  where    dept.deptno > 0 and dept.deptno = emp.deptn
  group by dept.deptno;
  ```

- ```
  select distinct emp.empno,emp.ename,emp.fname,person.ssnum,
         emp.job,emp.mgr,emp.responsible_for,emp.sal,salgrade.grade,
         emp.deptn,person.phone,person.car
  from   emp,person,salgrade
  where  (emp.sal >= 700. and emp.sal <= 9999.) and
         (emp.ename = person.pname and emp.fname = person.fname) and
         (emp.sal >= salgrade.losal and emp.sal <= salgrade.hisal);
  ```
  ```

4.2 Écriture des objets dans la base

4.2.1 Algorithme général d'écriture

4.2.1.1 Généralités

L'écriture[6] d'un objet dans la base ne peut se produire que lors d'une validation de transaction explicitement provoquée par un appel à la primitive driver-commit-objects.

Cette fonction cumule en fait deux rôles distincts :

- Le premier est de permettre la validation des modifications effectuées en mémoire sur les objets persistants. Les données relationnelles correspondantes sont mises à jour et tout nouvel accès sur la base, notamment concurrent, prend en compte les modifications.
- Le second est d'apporter la persistance à des objets de l'environnement, à condition toutefois que leurs classes soient connues du schéma de correspondances courant. Cette opération se traduit cette fois par l'insertion dans la base de nouvelles données relationnelles.

driver-commit-objects concilie ces deux rôles car il est possible de lui préciser en argument l'ensemble des objets à valider dans la base, qu'il s'agisse d'objets déjà persistants ou non.

Cette possibilité permet de réduire la granularité de la transaction DRIVER à un objet. Son intérêt majeur est l'efficacité accrue qu'elle procure lors des validations de transaction.

Quand les objets à valider sont précisés, deux nouvelles possibilités s'offrent à l'utilisateur :

- Sont non seulement sauvés dans la base les objets indiqués, mais également l'ensemble des objets qu'ils référencent directement ou indirectement.
- Ne sont sauvés que les objets précisés.

  Cependant, dans ce second cas, sont également écrits dans la base les objets non encore persistants directement référencés par l'un au moins des objets à valider. Si l'un d'entre eux référence lui-même un nouvel objet non persistant, ce dernier est également écrit dans la base, et ainsi de suite.

  Ces écritures induites sont nécessaires pour reporter fidèlement dans la base de données les objets précisés. Les clés des objets référencés doivent être déterminées pour que leurs valeurs puissent être correctement reportées dans les n-uplets des objets à écrire dans la base. Si cela n'était pas fait, les champs de type objet correspondants devraient être considérés comme vides, seraient écrits comme tels, et seraient réellement vides lors d'une nouvelle consultation.

[6] Par définition, nous différencions *écrire* d'*insérer* et *mettre à jour*. *Écrire* peut être l'un et/ou l'autre. *Insérer* implique l'idée d'ajout, de nouveauté, et *mettre à jour*, de modification de l'existant.

```
algorithme driver-commit-objects(Objects, Deep)
soit Objects_to_save ← objects-to-save(Objects, Deep);
faire commit-objects(Objects_to_save);
retourner Objects;
fin
```

```
algorithme commit-objects(Objects)
faire initialise-saving-buffer()
et    initialise-tuple-buffer()
et    build-object-buffer(Objects)
et    check-class-constraints-on-objects(Objects);
tant que object-buffer() ≠ {}
      soit Class_object_buffer ← next-class-object-buffer();
      faire commit-class-objects(Class_object_buffer);
faire valid-new-object-keys();
fin
```

```
algorithme commit-class-objects(Class_object_buffer)
soit Class ← class-object-buffer-class(Class_object_buffer)
et   Object_keys ← {};
si build-p(class-tuple-buffer(Class)) = ⊥
alors faire build-class-tuple-buffer(Class_object_buffer);
tant que Class_object_buffer ≠ {}
      soit Object ← next-object(Class_object_buffer)
      et   Tuple ← get-object-tuple(Class, Object)
      et   Object_key ← commit-object(Class, Object, Tuple);
      faire Object_keys ← Object_keys ∪ {Object_key};
retourner Object_keys;
fin
```

```
algorithme commit-object(Class, Object, Tuple)
si ∃ Tuple
alors retourner update-object(Class, Object, Tuple)
sinon retourner write-object(Class, Object);
fin
```

Figure 4.19 : Algorithme général d'écriture d'un ensemble d'objets

L'algorithme de driver-commit-objects est présenté figure 4.19. Le premier argument est l'éventuelle liste des objets à valider. Le second est un booléen qui précise le mode de validation choisi quand des objets sont précisés. Ces deux arguments sont transmis à objects-to-save qui détermine l'ensemble complet des objets à écrire dans la base :

```
algorithme objects-to-save(Objects, Deep)
si Objects = ⊤
alors retourner all-persistent-objects-in-memory()
sinon si Deep = ⊤
        alors retourner driver-all-linked-objects(Objects)
        sinon retourner Objects;
fin
```

Si *Objects* vaut $\top$, tous les objets persistants chargés dans l'environnement sont pris en compte. Quand un ensemble d'objets est précisé, le deuxième argument détermine si ces objets sont les seuls à être considérés. Si *Deep* est vrai, les objets référencés par ceux qui sont indiqués sont également à écrire dans la base; dans le cas contraire, seuls les objets indiqués sont retenus.

4.2.1.2 Principe de l'algorithme général d'écriture

Le principe de l'algorithme général d'écriture est le suivant :

Dans un premier temps sont consultées dans la base les données relationnelles correspondant aux objets sélectionnés. Par comparaison entre ces données et les objets en mémoire vont pouvoir être identifiés ceux qui ont subi des modifications et ceux qui n'ont pas encore de correspondance relationnelle.

Les objets sont ensuite examinés un par un. Selon les cas sont effectuées dans la base des mises-à-jour ou des insertions de nouveaux n-uplets relationnels. Parfois, mises-à-jour et insertions sont conjointement réalisées pour un même objet (cf. §4.2.3).

Afin de limiter le coût de consultation des données relationnelles, les instances de chaque classe sont d'abord déterminées et regroupées. Chaque classe n'est interrogée qu'une seule fois pour l'ensemble de ses instances. L'interrogation n'est bien sûr faite que si elle est pertinente, c'est-à-dire si la classe possède au moins un objet déjà persistant parmi ceux à écrire.

La consultation préalable de la valeur des objets dans la base avant leurs éventuelles mises-à-jour est indispensable (cf. §4.2.3. On peut d'ores et déjà souligner qu'elle est très rentable en termes de coûts en requêtes car elle permet de minimiser le nombre de requêtes d'écriture adressées à la base. Pour un surcoût de $(1 + N)$ requêtes de consultation par classe[7] (cf. §4.1.2.1) est déterminé l'état des objets avant leurs éventuelles écritures dans

[7] N est le nombre de champs de type "ensemble" de la classe.

la base. Grâce à cette connaissance, le SGBD ne reçoit de requêtes d'insertion ou de mise-à-jour qu'à bon escient. Si, par exemple, aucun objet n'a été modifié depuis la dernière validation, aucune nouvelle requête n'est adressée à la base.

4.2.1.3 Détails de l'algorithme

commit-objects initialise trois dictionnaires donnant accès à diverses informations utilisées à différents stades du processus d'écriture :

- Le premier, appelé "saving-buffer", est un dictionnaire donnant accès à des objets système de la classe Driver-saving (cf. §4.2.3.2). Chacun d'entre eux contient des informations relatives à l'écriture de l'un des objets en cours de sauvegarde. La clé permettant d'accéder à chacun d'entre eux dans ce dictionnaire est d'ailleurs l'objet en cours de sauvegarde qui leur correspond.

  initialise-saving-buffer prépare un dictionnaire vide qui sera complété ultérieurement.

- Le second, appelé "tuple-buffer", est le dictionnaire des n-uplets relationnels correspondant aux objets déjà persistants en cours de validation. Ce dictionnaire est à deux niveaux. Au premier, les clés sont les classes; elles donnent accès à des sous-dictionnaires appelés "class-tuple-buffer". Chaque sous-dictionnaire contient les n-uplets des objets qui sont au moins instances de la classe correspondante. Ces n-uplets sont la réponse du SGBD à la requête principale de la classe, restreinte aux objets sélectionnés (cf. §4.1.2.1). Dans un sous-dictionnaire, le n-uplet d'un objet y est accessible par la clé relationnelle de l'objet correspondant.

  initialise-tuple-buffer prépare également un dictionnaire vide qui sera complété ultérieurement.

- La dernière, appelée "object-buffer", est le dictionnaire des objets à écrire. Ici, les clés d'accès sont les différentes classes décrites dans le schéma de correspondances courant, et les valeurs, les objets à écrire qui en sont au moins instances. Ce dictionnaire constitue aussi un indicateur sur le degré d'avancement de l'écriture. En effet, un objet validé dans la base ou en cours de validation en a été enlevé. À tout moment n'y figurent que ceux qui sont en attente d'être écrits. La validation globale est terminée quand le dictionnaire est vide d'objet.

  build-object-buffer initialise et construit le dictionnaire. Après cette construction, aucun nouvel objet n'y est généralement ajouté, sauf en cas d'incohérence dans le réseau d'objets à écrire (cf. §4.2.2.2).

L'appel à check-class-constraints-on-objects permet ensuite de vérifier que les objets respectent effectivement les contraintes imposées par leur classe. Si ce n'est pas le cas, selon la valeur de l'indicateur driver-writing-help, une erreur est générée ou une modification pertinente de l'objet est proposée.

Cet examen est important. Il garantit qu'après écriture, un nouveau chargement dans l'environnement de ces objets les conduira à être de la même classe que celle qui est la leur au moment de l'écriture.

On ignore si le modèle objet client utilisé gère les contraintes de classes et s'il est en mesure d'effectuer lui-même ces vérifications. Pour cette raison, DRIVER ne peut pas se dispenser d'effectuer lui-même ce contrôle.

Il est ensuite procédé à l'écriture des instances de chaque classe au moyen d'appels à commit-class-objects.

Une fois toutes les écritures terminées peuvent être définitivement validées les clés relationnelles des objets nouvellement persistants. Cette validation n'est faite qu'au dernier moment de façon à éviter toute rémanence d'allocation de clé en cas d'erreur lors de l'exécution de driver-commit-objects, auquel cas DRIVER doit restaurer l'environnement et la base dans l'état où ils étaient lorsque driver-commit-objects a été exécuté (cf. §4.2.6).

commit-class-objects

Le rôle de commit-class-objects est de procéder à l'écriture de toutes les instances d'une classe donnée. Elle retourne en résultat la liste des clés relationnelles des objets correspondants.

Dans un premier temps, le sous-dictionnaire "class-tuple-buffer" de la classe est construite à l'aide de build-class-tuple-buffer.

Une fois le sous-dictionnaire des n-uplets construit, chaque instance de la classe se trouvant encore dans le dictionnaire *Class_object_buffer* est ensuite confiée à commit-object pour écriture. L'appel à next-object retourne l'objet suivant dans le dictionnaire et l'en retire. Ainsi, si commit-class-objects est appelée récursivement pour la même classe, l'objet en question ne se trouve plus dans le dictionnaire et n'est donc pas considéré à nouveau.

get-object-tuple recherche le n-uplet correspondant à l'objet traité dans le "class-tuple-buffer" de la classe considérée.

Si un n-uplet est retrouvé, l'objet est au moins instance de cette classe dans la base. commit-object oriente donc la suite de l'exécution vers une éventuelle mise-à-jour de l'objet dans la base en faisant appel à update-object.

Dans le cas contraire, une alternative apparaît :

- L'objet n'a pas encore de représentation relationnelle dans la base. Il doit donc y être inséré.
- L'objet relationnel n'est pas de la classe considérée dans la base. Deux raisons d'échec à la requête de consultation existent :
  - La première est que l'objet relationnel ne vérifie tout simplement pas les contraintes imposées par la classe. Une simple mise-à-jour de l'objet dans la base s'impose alors.
  - La seconde raison peut être qu'une jointure non utilisée dans les correspondances des classes ancêtres permet d'atteindre, dans une table secondaire, la correspondance d'un champ propre à la classe. Si l'objet relationnel n'a jamais

été instance jusqu'à présent de la présente classe, la jointure ne peut qu'échouer. Dans ce cas de figure, l'écriture de l'objet dans la base se traduit par des insertions de n-uplets dans les tables secondaires nouvellement atteinte et des mises-à-jour des n-uplets qu'on pouvait déjà atteindre au niveau des classes ancêtres (cf. §4.2.3).

On constate ici que l'absence de n-uplet dans la réponse à une requête de classe ne signifie pas nécessairement que l'objet correspondant n'était pas persistant avant l'écriture. DRIVER détermine si un objet donné est persistant en consultant un dictionnaire des objets persistants qu'il gère dans chaque schéma de correspondances (cf. §3.3.2.1). Dans ce dictionnaire, un objet est accessible par sa clé relationnelle.

Quand aucun n-uplet n'est retourné par la base, write-object est appelé.

À titre d'exemple, créons dans l'environnement un nouvel objet `Vendeur`. Appelons-le `johnson`. En tant que vendeur, il dépend du département `sales` et son chef est `blake`. Comme il est nouveau dans l'entreprise, on demande à un autre vendeur `james` de l'encadrer dans un premier temps. Cette responsabilité est officialisée en plaçant l'objet `johnson` dans le champ `responsable-de` de `james`. Le surplus de travail engendré est compensé par une majoration de 500. de la commission qu'il touche. On profite enfin de l'occasion pour rectifier son numéro de téléphone qui a récemment changé et préciser qu'il habite maintenant à la même adresse que son collègue `martin`. De ce fait, il utilise également sa voiture.

La figure 4.20 présente la valeur de l'objet `johnson` et les valeurs du vendeur `james` avant puis après modification.

L'ordre driver-commit-objects({Object("`james`")}, $\top$) déclenche l'écriture dans la base de l'objet `james` et de ceux qu'il référence.

objects-to-save détermine l'ensemble des d'objets à écrire, soit :

{ Object("`james`"), Object("`blake`"), Object("`king`"), Object(`dept/10`), Object(`address/7839`), Object(`vehicle/1232`), Object("`sales`"), Object(`hardware/53`), Object(`software/743`), Object(`software/874`), Object("`allen`"), Object(`address/7499`), Object(`vehicle/2004`), Object("`ward`"), Object(`address/7521`), Object(`vehicle/1253`), Object("`martin`"), Object(`address/7654`), Object(`vehicle/3005`), Object(`address/7698`), Object(`vehicle/1429`), Object("`johnson`") }

Un certain nombre d'entre eux sont des défauts d'objet. On les reconnaît dans l'ensemble précédent au fait qu'ils sont présentés référencés par une clé relationnelle alors que les objets jumeaux sont référencés par leurs noms entre guillemets (""). Un objet encore représenté par un défaut ne peut pas avoir été modifié en mémoire. Pour cette raison, les défauts d'objet ne sont pas pris en compte et ne figurent pas dans les dictionnaires.

Objet de classe `Vendeur`	
nom	`james`
prénom	`peter`
num-ss	`K4G262`
qualif	`salesman`
chef	Object(“`blake`”)
responsable-de	()
salaire	1350.
grade	2
dpt	Object(“`sales`”)
adresse	Object(`address/7655`)
tel-privé	`40-449323`
véhicule	()
situ-famille	`separate`
commission	1000.
sal-total	2350.

$\Longrightarrow$

Objet de classe `Vendeur`	
nom	`james`
prénom	`peter`
num-ss	`K4G262`
qualif	`salesman`
chef	Object(“`blake`”)
responsable-de	Object(“`johnson`”)
salaire	1350.
grade	2
dpt	Object(“`sales`”)
adresse	Object(`address/7654`)
tel-privé	`40-445843`
véhicule	Object(`vehicle/3005`)
situ-famille	`separate`
commission	1500.
sal-total	2850.

Objet de classe `Vendeur`			
nom	`johnson`	**dpt**	Object(“`sales`”)
prénom	()	**adresse**	()
num-ss	()	**tel-privé**	()
qualif	()	**véhicule**	()
chef	Object(“`blake`”)	**situ-famille**	()
responsable-de	()	**commission**	()
salaire	()	**sal-total**	()
grade	()		

Figure 4.20 : Objets à mettre à jour dans la base

Après initialisation des deux dictionnaires "saving-buffer" et "tuple-buffer" est construit l'"object-buffer", le dictionnaire des objets à écrire. Son contenu est le suivant :

Object-buffer initial pour driver-commit-objects({Object("johnson")}, ⊤)	
Clés	**Valeurs**
Employé	{ Object("james"), Object("blake"), Object("king"), Object("allen"), Object("ward"), Object("martin"), Object("johnson") }
Vendeur	{ Object("james"), Object("allen"), Object("ward"), Object("martin"), Object("johnson") }
Cadre	{ Object("blake"), Object("king")}
Président	{ Object("king") }
Chef-de-service	{ Object("blake") }
Département	{ Object("sales") }
Projet	{ }
Adresse	{ }
Véhicule	{ }
Software	{ }
Hardware	{ }

Il est ensuite vérifié que les objets respectent les contraintes imposées par leurs classes. En examinant les instances de la classe Employé, DRIVER détecte que la valeur du champ salaire de l'objet johnson ne vérifie pas la contrainte qui lui est imposée (salaire compris entre 700. et 9999.). Si l'indicateur système driver-writing-help vaut ⊤, une nouvelle valeur est demandée à l'utilisateur et est affectée au champ. Choisissons par exemple la valeur 1000.

En examinant les instances de la classe Vendeur, DRIVER détecte à nouveau qu'un champ de l'objet johnson ne vérifie pas la contrainte qui lui est imposée. Il s'agit cette fois du champ qualif, auquel il est imposé de valoir salesman. Une nouvelle fois, en supposant que driver-writing-help vaille ⊤, l'utilisateur est mis à contribution pour indiquer une valeur compatible avec la contrainte imposée. La seule valeur possible est salesman.

A priori, DRIVER ne détecte plus ensuite aucun problème. La valeur de l'objet johnson avant écriture est finalement :

Objet de classe Vendeur			
nom	johnson	**dpt**	Object("sales")
prénom	()	**adresse**	()
num-ss	()	**tel-privé**	()
qualif	salesman	**véhicule**	()
chef	Object("blake")	**situ-famille**	()
responsable-de	()	**commission**	()
salaire	1000.	**sal-total**	1000.
grade	()		

On peut dès à présent signaler que les modifications faites aux objets PENDANT la validation de transaction (comme les modifications précédentes des champs `salaire` et `qualif`) ne seront définitives que si l'écriture se termine normalement. En cas d'erreur ou d'interruption de l'écriture provoquée par l'utilisateur, toute modification éventuellement effectuée sur les objets est défaite de façon à restituer l'environnement tel qu'il était avant le début de la validation (cf. §4.2.5 et §4.2.6).

Le premier class_object_buffer non vide est celui de la classe `Employé`. Ce sera donc le premier à être confié à commit-class-objects. Le sous-dictionnaire des n-uplets de la classe `Employé` qui n'était pas encore établi est construit. Sa valeur est :

Tuple-buffer de la classe `Employé`	
Clés	**Valeurs**
emp/7499	[7499 allen jack 76373Y salesman 7698 software/743 1600. 3 30 78-383726 2004]
emp/7521	[7521 ward peter 128347 salesman 7698 hardware/53 1250. 2 30 40-378467 1253]
emp/7654	[7654 martin georges 234FS1 salesman 7698 () 1250. 2 30 40-276951 3005]
emp/7655	[7655 james peter K4G262 salesman 7698 () 1350. 2 30 40-449323 ()]
emp/7698	[7698 blake harold A473SE manager 7839 dept/30 2850. 4 30 44-183211 1429]
emp/7839	[7839 king paul A34F4 president () () 5000. 5 10 40-223233 1232]

Chaque objet contenu dans le class_object_buffer est ensuite confié à commit-object pour écriture. Juste avant l'appel, l'objet traité est retiré du class_object_buffer.

Mis à part `johnson`, DRIVER a pu retrouver dans la base le n-uplet de chaque objet contenu dans le buffer. L'écriture de chacun d'entre eux est donc réalisée par un appel à update-object, exceptée celle de `johnson`, réalisée en ce qui la concerne par un appel à write-object.

À chaque fois, la clé relationnelle retournée par commit-object est recueillie. La liste de ces clés est finalement restituée par commit-objects.

Chaque class_object_buffer est ainsi traité. Une fois l'object-buffer vide d'objets, l'écriture est terminée.

Nous allons maintenant détailler les algorithmes de mise-à-jour et d'insertion d'objet dans la base. Nous reprendrons et poursuivrons alors l'exemple précédent pour les illustrer.

```
algorithme update-object(Class, Object, Tuple)
soit Select ← class-select(Class)
et   Key ← key-from-select(Select, Tuple)
et   Object_operator ← build-object-filter(Class, {Key})
et   Updates ← make-object-updates(Select, Object_operator);
faire build-updates(Updates, Class, Object, Tuple)
et    do-writings-if-any(Updates);
retourner Key;
fin
```

```
algorithme build-updates(Updates, Class, Object, Tuple)
pour tout F ∈ fields.Class
    faire si mappedp.F = ⊤
        et read-only.F = ⊥
        alors faire info-for-update(Updates, Object, Tuple).F;
fin
```

```
algorithme do-writings-if-any(Writings)
pour tout W ∈ Writings
    faire si info-to-writep(W)= ⊤
        alors faire write-database().W;
fin
```

Figure 4.21 : Algorithme de mise-à-jour d'un objet dans la base

4.2.2 Mise-à-jour des objets dans la base

4.2.2.1 Principe de l'algorithme

L'algorithme update-object (figure 4.21) permet de reporter dans la base les éventuelles modifications effectuées dans l'environnement sur un objet jumeau. Il permet aussi de redonner à un objet relationnel dont l'objet jumeau a été construit une valeur qu'il a perdue à la suite d'une modification concurrente.

update-object est appliqué à un objet jumeau pour une classe donnée. À chaque appel ne sont considérés de l'objet que les champs propres à la classe donnée. Ne sont donc prises en compte que les modifications effectuées sur ceux-là. L'examen des autres champs est fait lors d'autres appels à update-object ou write-object, où la classe traitée est celle à laquelle ils appartiennent.

Avant la consultation des champs sont créés et initialisés un certain nombre d'objets Driver-update. Les instances de la classe Driver-update sont la représentation en DRIVER des requêtes de mise-à-jour (cf. §4.2.2.3). Une requête SQL de mise-à-jour ne pouvant s'adresser qu'à une table spécifique, sont préparés autant d'objets Driver-update que la classe comporte de tables élémentaires. Ainsi, lors de l'examen d'un objet, si deux champs modifiés ont une correspondance portant sur deux tables différentes, deux requêtes seront nécessaires pour reporter les modifications dans la base.

Les champs sont ensuite consultés les uns après les autres. Par comparaison entre leurs valeurs dans l'objet et leurs valeurs déduites du n-uplet relationnel, on détermine ceux qui ont été modifiés dans l'environnement ou dans la base. Quand une différence est détectée, l'objet Driver-update correspondant est complété.

Une fois tous les champs consultés, une mise-à-jour de la base est réalisée si nécessaire par la fonction do-writings-if-any. Chaque objet Driver-update est examiné. Quand l'un d'entre eux comporte une information à reporter dans la base, l'envoi de message write-database est effectué sur cet objet et se traduit par l'envoi au SGBD de la requête SQL appropriée.

4.2.2.2 Méthodes de mise-à-jour des différentes classes de champ

Un champ qui possède une correspondance et qui est autorisé à écrire dans la base est consulté par un envoi de message info-for-update. Selon sa classe, telle ou telle méthode est atteinte et exécutée.

La plupart d'entre elles sont présentées figure 4.22. Seules manquent celles des champs de type "ensemble". Leur étude sera présentée ultérieurement (cf. §4.2.4). Disons seulement que comme le traitement des champs de type "ensemble" est identique lors de l'insertion ou de la mise-à-jour d'un objet dans la base, les méthodes en insertion et en mise-à-jour sont également les mêmes.

```
algorithme Driver-field::info-for-update(Updates, Object, Tuple).Field
fin
```

```
algorithme Atom-field::info-for-update(Updates, Object, Tuple).Field
soit Select ← class-select(ownerclass.Field)
et   Env_field_value ← field-value(Field, Object)
et   Db_field_value ← field-value-from-tuple(F, Select, Tuple);
si Env_field_value ≠ Db_field_value
alors soit Update ← find-writing(table.attr.Field, Updates)
       et Attr_value ← attr-value-to-database(attr.Field, Field, Object);
       faire add-into-writing(attr.Field, Attr_value, Update);
fin
```

```
algorithme Object2-field::info-for-update(Updates, Object, Tuple).Field
soit Select ← class-select(ownerclass.Field)
et   Env_contained_object ← field-value(Field, Object)
et   Db_contained_object ← field-value-from-tuple(F, Select, Tuple);
si Env_contained_object ≠ Db_contained_object
alors soit Update ← find-writing(table.attr.Field, Updates)
       et Attr_value ← do-get-object-key(Env_contained_object, {}, {}, Field);
       faire add-into-writing(attr.Field, Attr_value, Update);
fin
```

```
algorithme Object-field::info-for-update(Updates, Object, Tuple).Field
soit Select ← class-select(ownerclass.Field)
et   Env_contained_object ← field-value(Field, Object)
et   Db_contained_object ← field-value-from-tuple(F, Select, Tuple);
si Env_contained_object ≠ Db_contained_object
alors soit Table ← joined-table(joins.Field)
       et   Update ← find-writing(Table, Updates)
       et   Key_attrs ← key.deftable.table.class.Field
       et   Key ← contained-object-key(Object, Field, Select, Tuple);
       pour tout Attr ∈ Key_attrs
            soit Attr_value ← attr-value-from-key(Attr, Key_attrs, Key);
            faire add-into-writing(Attr, Attr_value, Update);
fin
```

Figure 4.22 : Méthodes info-for-update

Toutes les méthodes déterminent la valeur du champ dans l'objet relationnel et la comparent à la valeur du même champ dans l'objet jumeau. Si ces valeurs sont identiques, il n'y a rien à faire; les données relationnelles qui correspondent au champ reflètent correctement la valeur du champ en mémoire. Si elles diffèrent, une mise-à-jour de la base est alors nécessaire.

La comparaison des deux valeurs est faite au sein de chaque méthode. On pourrait s'en étonner, le mécanisme permettant de les retrouver étant le même à chaque fois. En fait, c'est le test de comparaison qui change. Bien qu'il soit toujours représenté par le signe = dans les algorithmes, il désigne des tests d'égalité différents. Il exprime un test d'égalité de valeurs dans la méthode des champs atomiques, un test d'égalité de référence pour les champs "objet" et "objet 2", enfin des tests d'égalité d'ensembles pour les champs de type "ensemble". Ces derniers varient eux-mêmes selon le type de l'élément de l'ensemble.

Les méthodes Atom-field::info-for-update et Object2-field::info-for-update se ressemblent beaucoup. Quand une mise-à-jour de la base est nécessaire, elles déterminent toutes deux l'objet Driver-update concerné, la valeur d'attribut à insérer dans la base, et complètent pareillement l'objet Driver-update sélectionné.

Elles divergent par contre dans le calcul de la valeur à donner à l'attribut correspondant au champ. La méthode des champs atomiques détermine cette valeur à l'aide de la fonction attr-value-to-database :

algorithme attr-value-to-database($Attr, Field, Object$)
soit $Field\_value \leftarrow$ field-value($Field, Object$);
retourner driver-saving-type-filter($Attr, Field, Object, Field\_value$);
fin

La valeur du champ est confiée à la fonction driver-saving-type-filter qui la fait passer dans un filtre driverwriting avant de la retourner.

Pour les champs de type "objet 2", la valeur donnée à l'attribut est la clé de l'objet contenu. Leur méthode établit cette clé en faisant appel à la fonction do-get-object-key. Si l'objet contenu est déjà persistant, do-get-object-key retourne simplement sa clé. Si ce n'est pas le cas, elle provoque son écriture et récupère la clé qui lui a été allouée (cf. §4.23).

La méthode des champs de type "objet" est un peu plus complexe car c'est un ensemble d'attributs qu'il faut mettre à jour dans la base quand le contenu d'un champ de ce type a été modifié.

Dans sa version actuelle, DRIVER impose à tous ces attributs d'appartenir à la même relation. Cette limitation est très peu restrictive; elle l'est d'autant moins qu'on s'aperçoit, à l'usage, que la plupart des clés utilisées dans les schémas comprennent très rarement plusieurs attributs.

```
algorithme contained-object-key(Object, Field, Select, Tuple)
soit Contained_object ← field-value(Field, Object)
et   Object_key_attrs ← key.deftable.table.ownerclass.Field
et   Read_attrs ← ra-compil.Field
et   Constrained_attrs ← Object_key_attrs ∩ Read_attrs
et   Constrained_attr_values ← attr-values-from-tuple(Constrained_attrs, Select, Tuple)
et   Key ← do-get-object-key(Contained_object,
                             Constrained_attrs, Constrained_attr_values, Field);
faire check-contained-object-key(Key, Read_attrs,
                                 Constrained_attrs, Constrained_attr_values,
                                 (Read_attrs ⊂ Object_key_attrs), Object, Field);
retourner Key;
fin
```

```
algorithme check-contained-object-key(Key, Read_attrs, Const_attrs, Const_attr_values,
                                      Fixedp, Object, Field)
tant que compatible-object-key-p(Read_attrs, Key, Const_attrs, Const_attr_values) = ⊥
    si driver-writing-help = ⊤
    alors soit Cont_object ← affect-new-object(Object, Field, Const_attrs,
                                               Const_attr_values, Fixedp);
        faire complete-buffers-if-unread-object(Cont_object)
        et Key ← do-get-object-key(Cont_object, Const_attrs, Const_attr_values, Field);
    sinon faire fatal-error("Unauthorised object field value")
fin
```

```
algorithme do-get-object-key(Cont_object, Const_attrs, Const_attr_values, Field)
si ∄ Cont_object
alors retourner {}
sinon soit Key ← object-key-while-saving(Cont_object);
    si ∃ Key
    alors retourner Key
    sinon soit Const_key_attrs ← constrained-key-attrs(Const_attrs, Field);
        retourner save-object(Cont_object, Const_key_attrs, Const_attr_values);
fin
```

Figure 4.23 : Algorithme de contained-object-key

algorithme save-object($Object, Const\_key\_attrs, Const\_attr\_values$)
faire set-initial-pairs($Object, Const\_key\_attrs, Const\_attr\_values$)
soit $Class\_object\_buffer \leftarrow$ object-main-class-object-buffer($Object$);
et $Keys \leftarrow$ commit-class-objects($Class\_object\_buffer$);
retourner tete($Keys$);
fin

Figure 4.24 : Algorithme de contained-object-key (suite)

Quand le contenu du champ doit être reporté dans la base, on recherche l'objet Driver-update de la table à mettre à jour. Cette table est celle qui comprend les attributs comparés aux attributs de clé de l'objet contenu puisque ce sont ces attributs qui vont être mis à jour.

L'algorithme de contained-object-key est présenté figures 4.23 et 4.24. Son rôle est de déterminer et retourner la clé de l'objet contenu.

On constate à son examen qu'il n'est pas aussi simple que ce à quoi on aurait pu s'attendre. Cette complexité est liée au traitement des jointures restreintes et des jointures fixées (cf. §3.2.3.2).

Afin d'éviter toute modification intempestive des attributs référentiels participant également à la clé de leur relation, l'algorithme de contained-object-key procède à la vérification du contenu des champs objet avant de retourner une référence.

Dans un premier temps (figure 4.23) sont déterminés les attributs référentiels "contraints", c'est-à-dire ceux participant à une clé. S'il y en a, leurs valeurs sont retrouvées dans le n-uplet de l'objet contenu. Une fois la référence de l'objet en question déterminée par la fonction `do-get-object-key`, la vérification de cohérence est faite par la fonction check-contained-object-key. Suite à cette vérification, la référence de l'objet est retournée.

Au sein de check-contained-object-key, le test de cohérence est effectué par le prédicat compatible-object-key-p. Quand ce test échoue, selon la valeur de l'indicateur DRIVER driver-writing-help, une erreur fatale est déclenchée ou une modification du champ objet visant à rétablir la cohérence est engagée. La procédure d'erreur fatale interrompt définitivement et annule la validation de transaction tandis qu'une modification adéquate du champ objet incriminé permet de poursuivre ensuite l'opération.

Le déclenchement d'erreur fatale est examiné en §4.2.6, la modification d'un objet pendant l'écriture en 4.2.5.

Dans le second cas, la modification du champ objet est assurée par la fonction affect-new-object. affect-new-object fait d'abord appel à la fonction allowed-objects (figure 4.25) pour connaître l'ensemble des objets de la couche virtuelle qui sont compatibles avec les contraintes imposées. L'utilisateur est ensuite mis à contribution pour en sélectionner un parmi eux. L'objet choisi est affecté dans le champ, est éventuellement rajouté dans

l'object-buffer et est enfin retourné.

En examinant l'algorithme de allowed-objects (figure 4.25), on constate que l'ensemble des objets autorisés comme valeur d'un champ objet pour un objet donné varie si la jointure objet associée à ce champ est fixée et si l'objet propriétaire du champ est persistant. Dans ce cas, l'ensemble se réduit à un seul car la référence de l'objet contenu est entièrement déterminée. Cet objet est alors retourné ($Objects2$). Dans les autres cas, l'ensemble des objets autorisés est constitué de tous ceux qui vérifient les contraintes, qu'ils soient persistants ou non ($Objects1 \cup Objects2$).

La clé relationnelle de l'objet contenu dans le champ est déterminée par do-get-object-key (figure 4.23). Si le champ est vide, la fonction retourne {} (absence de valeur). S'il y a un objet, elle cherche sa clé, y compris l'éventuelle clé provisoire lui ayant été affectée au cours de la même validation de transaction. Si une telle clé existe, elle la retourne. Dans le cas contraire, l'insertion de l'objet est déclenchée. Si la jointure objet associée au champ est restreinte, les attributs clés contraints de l'objet à insérer sont déterminés et communiqués avec leurs valeurs à save-object. Ces informations vont permettre que l'objet contenu ait une clé relationnelle compatible avec son référençage dans le présent champ.

save-object (figure 4.24) déclenche la validation du class_object_buffer de la classe principale de l'objet considéré. set-initial-pairs prend en compte les éventuelles contraintes d'attributs à propager et commit-class-objects amorce la validation.

Illustrons l'algorithme update-object complet en reprenant l'exemple de validation du vendeur `james` et des objets qu'il référence (cf. §4.2.1.3). Lors de cette validation, le premier appel à update-object est effectué pour l'objet `james` lors du traitement du class_object_buffer de la classe `Employé`.

La clé relationnelle de `james` (`emp/7655`) est retrouvée à partir de son n-uplet et utilisée pour déterminer l'opérateur relationnel de l'objet ("`emp.empno = 7655`"). Cet opérateur permettra de n'appliquer les éventuelles requêtes de mise-à-jour qu'aux n-uplets correspondant à l'objet considéré. make-object-updates crée deux objets Driver-update, un pour la table `emp` et le second pour la table `person`. Ces objets, initialement vides de modification, pourront être complétés par les méthodes info-for-update.

La méthode de la classe Object2-field est la première à identifier un champ modifié de `james`. Le champ en question est `responsable-de`, vide dans l'objet relationnel et référençant `johnson` dans l'objet jumeau. La modification du champ va se traduire côté relationnel par la modification de l'attribut `emp.responsible_for` associé. La valeur de cet attribut est la clé relationnelle de l'objet contenu. Comme l'objet `johnson` n'est pas encore persistant, sa clé n'est pas encore établie et do-get-object-key provoque son insertion dans la base par l'appel de save-object. Cet appel engendre une nouvelle validation du class_object_buffer de la classe `Employé`. En supposant que cette validation se solde par l'attribution à l'objet `johnson` de la clé `emp/7936`, l'objet Driver-update correspondant à `emp` est complété de la paire attribut-valeur [`emp.responsible_for`, "`emp/7936`"]. Ajoutons enfin que la correspondance du champ ne fait pas intervenir d'attribut contraint. De ce fait, set-initial-pairs n'a aucune action. Nous reviendrons sur l'insertion de l'objet `johnson` et la détaillerons en §4.2.3.6.

```
algorithme allowed-objects(Class, Attrs, Attr_values, Fixedp)
soit Objects1 ← non-persistent-compatible-objects(Class)
et   Objects2 ← persistent-compatible-objects(Class, Attrs, Attr_values);
si Fixedp = ⊤
et Objects2 ≠ {}
alors retourner Objects2;
sinon retourner Objects1 ∪ Objects2;
fin
```

```
algorithme non-persistent-compatible-objects(Class)
soit All_class_objects ← all-class-objects(Class);
et   Objects ← {};
pour tout Object ∈ All_class_objects
    soit Key ← object-key-while-saving(Object)
    si ∄ Key
    alors faire Objects ← Objects ∪ {Object};
retourner Objects;
fin
```

```
algorithme persistent-compatible-objects(Class, Attrs, Attr_values)
soit Select ← class-select(Class)
et   Operator ← make-attrs/values-operator(Attrs, Attr_values)
et   Tuples ← ask-database(Select, Operator)
et   Objects ← {};
pour tout Tuple ∈ Tuples
    soit Object ← get-object(object-key-from-select(Select, Tuple));
    faire Objects ← Objects ∪ {Object};
retourner Objects;
fin
```

Figure 4.25 : Algorithme de allowed-objects

La méthode Object-field::info-for-update identifie ensuite le champ `adresse` comme ayant également été modifié dans l'environnement. Cette fois, l'objet référencé dans le champ est déjà persistant et a pour clé `address/7654`. Seulement, la jointure objet J[`emp`→`address`] (de définition `emp.empno = address.empno`) qui est associée au champ est fixée car l'attribut référentiel `emp.empno` qui y participe constitue aussi la clé de la relation `emp`. Le test de compatibilité effectué par compatible-object-key-p échoue car la valeur `7654` d'`emp.empno` qui serait nécessaire pour que l'objet `Adresse` référencé puisse être retenue comme valeur du champ est différente de `7655`, valeur de clé de l'objet `james`. En supposant que driver-writing-help vaille ⊤, affect-new-object détermine dans la couche virtuelle l'ensemble des objets `Adresse` compatibles avec les contraintes, {Object(`address/7655`}, et affecte au champ sa seule valeur possible. Finalement, l'objet Driver-update correspondant à `emp` est complété de la paire [`emp.empno`, `7655`].

La méthode Atom-field::info-for-update identifie ensuite le champ `tel-prive` comme ayant également été modifié. L'attribut correspondance du champ est `person.phone`. L'application de la méthode complète donc l'objet Driver-update de `person` de la paire [`person.phone`, "`40-445843`"].

Enfin, la méthode Object-field::info-for-update détermine également que le champ `véhicule` a été modifié. L'objet `Véhicule` référencé dans le champ est déjà persistant et a pour clé `vehicle/3005`. L'attribut référentiel `person.car` n'étant pas élément de clé, il n'est pas contraint. contained-object-key ne détecte a fortiori aucune incohérence et retourne la clé de l'objet contenu. L'objet Driver-update correspondant à `person` est complété de la paire [`person.car`, `3005`].

`vehicule` est le dernier champ de `james` propre à la classe `Employé`. Son traitement termine l'exécution de build-updates. L'appel à do-writings-if-any qui suit se traduit par l'envoi au SGBD des requêtes correspondant aux deux objets Driver-update précédents et la mise-à-jour des données relationnelles dans la base. Ces requêtes et objets sont examinés dans la prochaine section (§4.2.2.3). L'appel à la fonction update-object se termine en retournant la clé de l'objet mis à jour, soit `emp/7655`.

Après l'examen de l'objet `james`, la validation du class_object_buffer de la classe `Employé` se poursuit. Cependant, lors du traitement du champ `responsable-de` de `james`, les objets qui se trouvaient encore dans ce class_object_buffer ont été validés et en ont donc été retirés. Ainsi, il est vide maintenant. L'exécution de commit-class-objects pour le class_object_buffer de la classe `Employé` se termine donc immédiatement en retournant la liste des clés des objets validés au cours de la même opération, soit {`emp/7655`}. La validation du class_object_buffer de la classe `Vendeur` est aussitôt enchaînée.

4.2.2.3 Les requêtes de mise-à-jour

DRIVER modifie dans la base un objet relationnel en adressant au SGBD une requête de mise-à-jour. Une telle requête est représentée par un objet de la classe Driver-update. Cet objet est progressivement construit par les méthodes info-for-update (cf. §4.2.2.2) et, une fois complet, permet de générer une requête SQL correspondante.

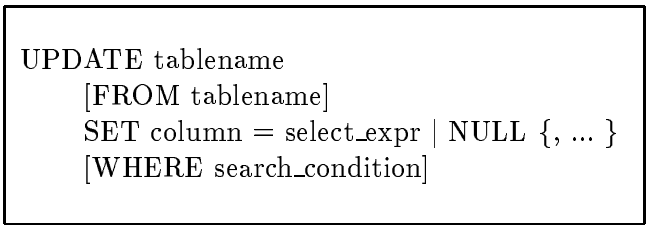

Figure 4.26 : Syntaxe d'une requête SQL de mise-à-jour de données

La syntaxe générale de la requête SQL de mise-à-jour est donnée figure 4.26. Disons, en quelques mots, que la requête est introduite par le mot-clé UPDATE et qu'elle s'adresse à une seule table. Les colonnes modifiées et leurs nouvelles valeurs sont précisées derrière le mot clé SET et les n-uplets concernés sont ceux qui vérifient les contraintes exprimées dans la `search_cond`.

La classe Driver-update (listing 21) présente, dans sa structure, les principales articulations d'une requête SQL de mise-à-jour de données. Nous verrons que les requêtes SQL d'insertion de données comportent ces mêmes articulations, moins la partie where (cf. §4.2.3.7). De ce fait, Driver-update est définie comme sous-classe de Driver-insert qui est la classe des requêtes d'insertion.

Driver-insert comprend quatre champs. Le premier reçoit la table logique à laquelle s'applique la requête. null-attributes contient l'ensemble des attributs auxquels on affecte la valeur NULL, et attributes l'ensemble des attributs auxquels on affecte une valeur différente de NULL. La liste de ces valeurs est précisée dans le champ values. Le champ where, propre à la classe Driver-update, contient pour sa part un ensemble de restrictions qui détermine l'ensemble des n-uplets auxquels s'appliquent les modifications.

Les classes Driver-insert et Driver-update répondent au message generate-sql. Leurs méthodes construisent et renvoient sous forme d'une chaîne de caractères la requête SQL correspondant à un objet.

Listing 21. Les classes de représentation des requêtes d'écriture

```
(driver-defclass Driver-insert ()
   (table           object     Driver-table)
   (null-attributes objectset  Driver-attribute)
   (attributes      ordobjset  Driver-attribute)
   (values          ordatomset ()))

(driver-defclass Driver-update Driver-insert
   (where objectset Driver-sql-op))
```

À titre d'exemple, la figure 4.27 présente les deux objets Driver-update construits dans la précédente section (§4.2.2.2) pour reporter dans la base les modifications effectuée en mémoire sur l'objet `james`.

Objet de classe Driver-update	
table	(emp)
null-attributes	()
attributes	(emp.responsible_for emp.empno)
values	(emp/7936 7655)
where	(emp.empno = 7655)

Objet de classe Driver-update	
table	(person)
null-attributes	()
attributes	(person.phone person.car)
values	(40-445843 3005)
where	(Subselect[J[emp→person], emp.empno = 7655])

Figure 4.27 : Objets Driver-update de mise-à-jour de l'objet `james`

Les deux requêtes correspondant à ces objets sont :

- ```
  update emp
  set    responsible_for = 'emp/7936', empno = 7655
  where  empno = 7655;
  ```

- ```
  update person
  set    phone = '40-445843', car = 3005
  where  exists(select * from emp
                where  emp.empno = 7655 and
                       (emp.ename = person.pname and emp.fname = person.fname));
  ```

4.2.3 Insertion d'objets dans la base

De façon générale, l'algorithme write-object (figure 4.31) permet d'insérer dans la base un objet qui n'est pas encore persistant. Mais du fait du critère qui sélectionne son appel, à savoir l'existence dans la base d'un n-uplet pour l'objet et la classe considérés (cf. figure 4.19), il est également sollicité :

- pour insérer des parties d'objet déjà persistant qui ne figurent pas encore dans la base. Ce cas de figure se produit quand un objet a été inséré alors qu'il était instance

d'une classe donnée et qu'il est validé à nouveau en étant instance d'une classe plus précise pour laquelle des correspondances ont été définies dans une table secondaire non utilisée dans la correspondance des classes ancêtres.

- quand l'objet ne vérifie pas encore dans la base les contraintes imposées par la classe considérée. L'application de write-object peut alors dans ce cas n'engendrer que des mises-à-jour de n-uplets existants.

Par souci de simplification, nous allons dans un premier temps ne considérer que le cas général d'utilisation de write-object, à savoir l'insertion d'un nouvel objet dans la base. Nous n'examinerons la résolution des autres cas de figure qu'ultérieurement.

4.2.3.1 Difficultés posées par l'insertion d'objet

Un objet persistant a pour correspondance relationnelle un ensemble de n-uplets répartis dans les tables élémentaires de sa classe. Plus exactement, sa correspondance est un unique n-uplet virtuel résultant de l'assemblage des dits n-uplets. Leur rapprochement, leur mise "bord à bord" en quelque sorte, résulte des jointures élémentaires définies entre les tables de la classe.

L'insertion d'un nouvel objet dans la base ne se réduit donc pas seulement à un problème d'insertion de n-uplets qui contiendraient les valeurs des champs. L'opération doit également faire en sorte que les jointures qui permettent d'accéder aux différentes parties de l'objet dans la base n'échouent pas lors des relectures ultérieures.

L'insertion d'un objet dans la base nécessite donc la résolution d'un système de contraintes où les variables sont les attributs des tables élémentaires de la classe considérée et où les équations sont les définitions des jointures et des contraintes de classe exprimées sous forme de contraintes sur attributs.

Afin de garantir l'accès ultérieur depuis l'environnement à tout objet relationnel auquel il offre la persistance, DRIVER affecte, au cours de la résolution, une valeur à chacun des attributs participant au système de contraintes. Grâce à ces valuations, tout échec de jointure ou de restriction est rendu impossible lors d'une future relecture.

Par définition, un attribut est dit *fondamental* s'il doit être nécessairement valué dans tout objet relationnel de la classe considérée.

Les attributs de tables élémentaires qui participent au système de contraintes sont donc fondamentaux. On peut faire la remarque qu'un attribut doit nécessairement appartenir à une table élémentaire de la classe pour être fondamental pour cette classe. Il existe en effet des attributs qui participent au système de contraintes et qui ne sont pas fondamentaux. Ce sont tous ceux qui n'appartiennent pas à l'une des tables élémentaires de la classe. Ces attributs proviennent de jointures sur des tables "externes" sur lesquelles ont été définis des champs de type read-only, accessibles en lecture seulement. Ces attributs ne sont pas considérés pendant la résolution.

Enfin, les relations élémentaires d'une classe comportent d'autres attributs fondamentaux que ceux qui sont impliqués dans le système de contraintes. Ce sont tous ceux qui

font partie de la clé de l'une d'entre elles.

Pour résumer, l'opération d'insertion d'un objet dans la base implique, avant insertion des n-uplets, de valuer l'ensemble des attributs fondamentaux de sa classe. Leurs valeurs sont contraintes par les équations issues des jointures et des restrictions.

À titre d'exemple, l'insertion d'un objet de la classe `Employé` donne lieu à la résolution, pour cet objet, du système suivant :

$$\begin{cases} \texttt{emp.sal} >= 700. \\ \texttt{emp.sal} <= 9999. \\ \texttt{emp.ename} = \texttt{person.pname} \\ \texttt{emp.fname} = \texttt{person.fname} \\ \texttt{emp.sal} >= \texttt{salgrade.losal} \\ \texttt{emp.sal} <= \texttt{salgrade.hisal} \end{cases}$$

Les deux dernières équations sont issues d'une jointure sur la table `salgrade`, table qui n'est pas élémentaire pour la classe `Employé`. Cette jointure participe à la correspondance d'un champ atomique read-only. Dans ces deux équations, `emp.sal` est fondamental car issu d'une table élémentaire de `Employé`. Par contre, `salgrade.losal` et `salgrade.hisal` ne le sont pas et ne seront donc pas valués lors de la résolution.

Les clés de `emp` et `person` étant respectivement { `emp.empno` } et { `person.pname`, `person.fname` }, l'ensemble des attributs fondamentaux de la classe `Employé` est finalement :

{ `emp.empno`, `emp.sal`, `emp.ename`, `emp.fname`, `person.pname`, `person.fname` }.

4.2.3.2 Les classes système de gestion d'écriture

L'ensemble des attributs fondamentaux et le système de contraintes à résoudre pour l'insertion d'objets sont uniques et invariants pour une classe donnée. Dans le même temps, leur résolution est spécifique pour chaque objet.

DRIVER définit deux classes, Driver-class-saving et Driver-saving, pour gérer ces informations et leurs résolutions (listing 22).

Listing 22. Les classes système de gestion d'écriture d'objets

```
(driver-defclass Driver-class-saving ()
  (cc-operators objectset  Driver-sql-op)
  (cc-joins     objectset  Driver-join)
  (operators    objectset  Driver-sql-op)
  (attributes   objectset  Driver-attribute)
  (attr-ops     dictionary ((object Driver-attribute) (objectset Driver-sql-op)))
```

```
  (joins        objectset  Driver-join))

(driver-defclass Driver-saving Driver-class-saving
  (object           object     ())
  (initial-key      atom       symbol)
  (new-key          atom       symbol)
  (object-class     object     Driver-class)
  (status           atomset    symbol)
  (unsolved-attrs   objectset  Driver-attribute)
  (initial-pairs    dictionary ((object Driver-attribute) (atom ())))
  (user-pairs       dictionary ((object Driver-attribute) (atom ())))
  (prev-pairs       dictionary ((object Driver-attribute) (atom ())))
  (pairs            dictionary ((object Driver-attribute) (atom ())))
  (prev-transitions ordobjset  Driver-transition)
  (transitions      ordobjset  Driver-transition))
```

La première regroupe toutes les informations sur le système de contraintes et l'ensemble des attributs fondamentaux d'une classe donnée. Un objet Driver-class-saving propre à une classe est construit au besoin et stocké dans l'objet Driver-class correspondant où il est disponible dès qu'il a été construit (cf. §3.3.2.3).

La classe Driver-class-saving est structurée de la façon suivante :

- cc-operators contient l'ensemble des équations issues des contraintes de la classe, et cc-joins, les jointures permettant d'atteindre les attributs participant à ces équations.
- operators contient l'ensemble des équations issues des jointures élémentaires de la classe.
- attributes référencent l'ensemble des attributs fondamentaux de la classe, exceptés ceux issus des contraintes de classe. En effet, ces derniers ont déjà été résolus par l'appel check-class-constraints-on-objects (cf. figure 4.19) au moment où write-object est appelé. Leur présence ici ne présente donc pas d'intérêt.

  Chaque attribut contenu dans attributes participe éventuellement à des équations. attr-ops contient un dictionnaire qui permet d'accéder rapidement, à partir d'un attribut, à l'ensemble des équations auxquelles il participe. Ce dictionnaire est utilisé pour accélérer les propagations de valeur lors de la résolution du système de contraintes.
- Enfin, joins référence l'ensemble des jointures participant au système de contraintes.

À titre d'exemple, la figure 4.28 présente les objets Driver-class-saving des classes `Employé` et `Vendeur`.

Objet de classe Driver-class-saving	
cc-operators	([emp.sal >= 700.] [emp.sal <= 9999.])
cc-joins	()
operators	([emp.ename = person.pname] [emp.fname = person.fname])
attributes	(emp.empno person.pname person.fname emp.ename emp.fname)
attr-ops	(...)
joins	(J[emp→person])

Objet de classe Driver-class-saving	
cc-operators	([emp.job = 'salesman'] [emp.sal >= 700.] [emp.sal <= 9999.])
cc-joins	()
operators	([emp.ename = person.pname] [emp.fname = person.fname])
attributes	(emp.empno person.pname person.fname emp.ename emp.fname)
attr-ops	(...)
joins	(J[emp→person])

Figure 4.28 : Objets Driver-class-saving des classes Employé et Vendeur

La classe Driver-saving a pour sa part été définie comme support d'aide à l'insertion d'objet dans la base. Sa structure est adaptée au calcul des attributs fondamentaux et autres, et à la construction des requêtes d'insertion et/ou de mise-à-jour qui s'en suit.

Un objet Driver-saving est défini pour chaque objet à écrire dans la base. Construits de façon opportuniste, ils sont, après construction, accessibles par l'intermédiaire du dictionnaire saving-buffer qui est initialisé en début de validation de transaction (cf. §4.2.1.3).

Définie comme sous-classe de Driver-class-saving, Driver-saving hérite de ses champs. Les équations et les attributs fondamentaux de la classe considérée sont ainsi directement disponibles dans ses instances. On notera cependant que les champs cc-operators et cc-joins contiennent toujours les informations correspondant à la classe précise de l'objet à écrire. De cette façon, dès la première résolution, on peut faire en sorte que l'objet vérifie ses contraintes qui subsument celles des classes ancêtres et qui sont les plus restrictives.

Elle ajoute un certain nombre d'autres champs, plus particulièrement destinés, pour leur part, à recueillir les données spécifiques à l'objet à insérer associé (listing 22) :

- Le champ object référence cet objet à insérer associé.
- Quand l'objet en question est déjà persistant au moment de la création de son Driver-saving, sa clé relationnelle est reportée dans le champ initial-key.

  Dans le cas contraire, une clé relationnelle lui est affectée à un certain moment de la résolution. Elle est alors reportée dans le champ new-key de son Driver-saving. Une clé nouvellement attribuée n'est pas directement enregistrée dans les dictionnaires de DRIVER. Ainsi, en cas d'erreur, le simple abandon des objets Driver-saving annule simplement les attributions de clé qui ont pu être faites au cours de la validation. Si

cette validation se termine normalement, DRIVER procède alors à l'enregistrement définitif des clés attribuées.

- Le champ object-class recueille la classe précise de l'objet à insérer.
- Le champ status contient un ensemble de symboles renseignant sur les états courants du système. Ces symboles permettent d'orienter la résolution de l'objet en cours d'écriture. Les différents états possibles existant dans la version actuelle de DRIVER sont adding-initial-pairs, initial-pairs, no-initial-pairs et user-pairs. Nous reviendrons ultérieurement sur le rôle joué par chacun d'entre eux (cf. §4.2.3.4).
- Le champ unsolved-attrs contient l'ensemble des attributs à valuer nécessairement pour lesquels aucune valeur n'a encore été arrêtée. Au début de la résolution, unsolved-attrs contient donc les mêmes objets que le champ attributes. Puis, progressivement, il se vide. La résolution se poursuit tant que ce champ référence des attributs.
- Avant de déclencher la résolution d'un objet, il est parfois nécessaire de fixer la valeur de certains attributs. Ce cas se produit quand le champ objet d'un objet déjà persistant référence un objet non persistant et que la correspondance de ce champ objet est une jointure objet restreinte. Dans ce cas, avant l'appel de write-object pour l'objet contenu, on précise les valeurs des attributs de sa clé qui vont permettre son référençage dans le champ objet précédent. Les paires attribut-valeur sont placées dans le champ initial-pairs de son objet Driver-saving lors du traitement du champ objet.
- Pendant la résolution d'un objet, l'utilisateur peut être sollicité pour donner une valeur à des attributs participant à sa correspondance. Les paires attribut-valeur correspondantes sont placées dans le champ user-pairs de son objet Driver-saving.
- Au cours de la résolution d'un objet, des valeurs sont affectées aux différents attributs qui participent à sa correspondance. Les paires attribut-valeur sont systématiquement ajoutées au contenu du champ pairs. Comme nous l'avons vu, si la valuation de l'attribut est faite par l'utilisateur ou précède la résolution effective de l'objet, la paire est également ajoutée, selon les cas, aux contenus des champs initial-pairs ou user-pairs. Quand une paire est ajoutée au contenu de pairs, l'attribut correspondant est retiré de unsolved-attrs.
- Si, au cours de la validation de transaction, un objet doit être modifié, l'ancienne valeur du champ concerné est conservée de façon à pouvoir défaire la modification, par exemple en cas d'erreur ou d'incohérence. Les modifications successives effectuées sur un objet sont stockées dans le champ transitions de son Driver-saving sous forme d'objets transitions. Une transition est en DRIVER un objet de la classe Driver-transition, défini par un triplet T[objet, champ, ancienne valeur] (cf. §4.2.5).
- Pour un objet donné, write-object peut être appelé plusieurs fois avec des classes différentes. Au début de chaque nouvel appel, les paires attribut-valeur déterminées

pour la classe précédente sont ajoutées au champ prev-pairs tandis que pairs est réinitialisé. De la même façon, les transitions effectuées pour la classe précédente sont ajoutées au champ prev-transitions, puis transitions est réinitialisé.

L'intérêt de séparer transitions et paires attribut-valeur de l'application courante de write-object et leurs homologues des appels précédents sera mis en évidence en §4.2.3.4.

La figure 4.29 présente l'objet Driver-saving de l'employé `johnson`. Il a précisément cette valeur au début de l'appel de write-object pour `johnson` et la classe `Employé`, immédiatement après son initialisation pour cette classe.

Objet de classe Driver-saving	
cc-operators	([emp.job = 'salesman'] [emp.sal >= 700.] [emp.sal <= 9999.])
cc-joins	()
operators	([emp.ename = person.pname] [emp.fname = person.fname] [emp.job = 'salesman'] [emp.sal >= 700.] [emp.sal <= 9999.])
attributes	(emp.empno person.pname person.fname emp.ename emp.fname emp.sal emp.job)
attr-ops	(...)
joins	(J[emp→person])
object	Object("johnson")
initial-key	()
new-key	()
object-class	Driver-class(Vendeur)
status	()
unsolved-attrs	(emp.empno person.pname person.fname emp.ename emp.fname emp.sal emp.job)
initial-pairs	()
user-pairs	()
prev-pairs	()
pairs	()
prev-transitions	(T[Object("johnson"), qualif, ()] T[Object("johnson"), salaire, ()])
transitions	()

Figure 4.29 : Objet Driver-saving initial de `johnson` pour la classe `Employé`

4.2.3.3 Principe de résolution

Résoudre un objet consiste donc à calculer les attributs référencés dans le champ unsolved-attrs de son Driver-saving en respectant les équations contenues dans le champ operators du même objet.

Dans cette optique, les champs de l'objet propres à la classe considérée sont tour à tour interrogés et leurs valeurs permettent de déduire de nouvelles paires [attribut, valeur].

Chaque paire est traitée de la même façon. DRIVER vérifie dans un premier temps qu'elle est compatible avec les équations partiellement instanciées et qu'aucune autre valeur n'a déjà été affectée à l'attribut considéré. S'il y a incohérence, l'exécution se poursuit par un appel à inconsistency-management, sur lequel nous allons revenir. Dans le cas contraire, la paire est ajoutée au contenu du champ **pairs** et l'ensemble des expressions faisant intervenir l'attribut en question est déterminé à l'aide du dictionnaire attr-ops.

Si l'attribut participe à des expressions, on essaie ensuite de propager sa nouvelle valeur dans ces expressions de façon à déduire autant de nouvelles paires que possible. Pour chacune d'entre elles, un appel récursif à la routine d'ajout de nouvelle paire est effectué.

Parfois, l'interrogation des champs n'est pas suffisante pour valuer l'ensemble des attributs fondamentaux de la classe. Dans ce cas, selon leurs types et la valeur de l'indicateur système driver-fix-key-generation-p, des valeurs générées leur sont automatiquement allouées ou l'utilisateur est sollicité pour leur en fournir.

4.2.3.4 Gestion des incohérences

En cas d'incohérence, inconsistency-management est appelé. Son algorithme est présenté figure 4.30. Son rôle est de rattraper l'erreur si c'est possible et si l'indicateur DRIVER driver-writing-help vaut $\top$, ou de déclencher une erreur fatale dans le cas contraire.

Rattraper l'erreur consiste à annuler la résolution en cours et à la recommencer.

undo-current-transitions réinitialise l'objet Driver-saving en défaisant les éventuelles modifications faites sur les objets pendant cette résolution, en replaçant dans l'ensemble des attributs à résoudre l'ensemble des attributs fondamentaux de la classe et en remettant à {} l'ensemble des paires déterminées. On remarque au passage l'intérêt de séparer les paires et les transitions obtenues et effectuées au cours de la présente résolution de celles résultant des précédents appels de write-object. En cas d'incohérence, seules celles de la résolution courante sont défaites. Les autres restent accessibles dans les champs prev-... de l'objet Driver-saving.

Enfin, relancer la résolution est obtenu en déclenchant une sortie de write-object1 qui retourne {}. Cette valeur engendre un nouvel appel de write-object1.

Le contenu du champ status de l'objet Driver-saving renseigne sur le déroulement de la résolution et oriente le traitement de l'incohérence s'il en apparaît une. Dans la version actuelle de DRIVER, status est susceptible de contenir quatre symboles différents :

- **adding-initial-pairs**. Au début de la résolution, il est possible d'introduire dans le système des paires imposées par la résolution d'un autre objet. De telles paires existent quand l'objet en cours d'écriture n'est pas encore persistant, qu'il est référencé dans un champ à objet et que la correspondance de ce champ objet est une jointure objet restreinte voire fixée. L'écriture en cours a été déclenchée récursivement, lors du traitement du champ objet d'un autre objet. Le lien existant entre des attributs de clé des deux objets est exprimé par ces paires.

```
algorithme inconsistency-management(Saving)
soit Fatal_error ← ⊥
et   Message1 ← "Inconsistency in writing constraints"
et   Message2 ← "";
selon
  cas status.Saving = {}
    faire Fatal_error ← ⊤;
  cas adding-initial-pairs ∈ status.Saving
    faire Fatal_error ← ⊤
    et    Message1 ← "Incompatible key and class constraints";
  cas status.Saving = {initial-pairs}
    si driver-writing-help = ⊥
    alors faire Fatal_error ← ⊤;
    sinon faire Message2 ← "Key constraints forsaked"
          et    status.Saving ← {no-initial-pairs};
  cas user-pairs ∈ status.Saving
    si driver-writing-help = ⊥
    alors faire Fatal_error ← ⊤;
    sinon faire Message2 ← "User inputs forsaked"
          et    status.Saving ← status.Saving \ {user-pairs};
  cas sinon
    faire Fatal_error ← ⊤;
si Fatal_error = ⊤
alors faire fatal-error(Message1);
sinon faire driver-warning(Message1)
      et    driver-warning(Message2)
      et    undo-saving-resolution(Saving)
      et sortir de write-object1() en retournant {};
fin
```

```
algorithme undo-saving-resolution(Saving)
faire undo-current-transitions(Saving)
et    unsolved-attrs.Saving ← attributes.Saving
et    user-pairs.Saving ← {}
et    pairs.Saving ← {};
fin
```

Figure 4.30 : Gestion d'incohérence dans la résolution d'écriture d'objet

Avant d'introduire la première d'entre elles, le statut adding-initial-pairs est ajouté au champ status de l'objet Driver-saving, et en est retiré après introduction de la dernière. Ce symbole identifie donc précisément la phase d'insertion des contraintes d'attributs pendant la résolution.

S'il y a incohérence, les paires qui sont indiquées sont incompatibles avec les contraintes de la classe traitée, c'est-à-dire avec celles de la classe principale de l'objet. Cela signifie que l'objet propriétaire du champ référençant l'objet ne peut contenir aucun objet instance de la classe principale considérée. Ce champ ayant été déclaré pour référencer des objets de cette classe, l'incohérence est grave et peut provenir d'une mauvaise définition du schéma de correspondances. Elle déclenche invariablement une erreur fatale.

- **initial-pairs**. Cet indicateur est ajouté au champ status de l'objet Driver-saving aussitôt après l'introduction de paires initiales, si l'opération s'est bien passée. Sa présence dans le statut indique donc que l'ensemble des paires résolues comprend des paires initiales.

  Si, par la suite, une incohérence est détectée, dans le cas où le statut du système vaut précisément {initial-pairs} et si driver-writing-help vaut $\top$, inconsistency-management tente de rattraper l'erreur et de recommencer la résolution en ne tenant plus compte des paires initiales.

  L'apparition d'une incohérence alors que le statut ne comprend que l'indicateur initial-pairs signifie que l'objet en cours d'écriture n'est pas une valeur autorisée pour le champ objet dans lequel il est référencé et qui est à l'origine des paires initiales stipulées. Rattraper l'erreur veut donc dire qu'on choisit d'écrire l'objet dans la base indépendamment de son référençage dans le champ objet. Une fois l'écriture terminée, l'exécution va revenir au traitement de l'écriture du champ objet incriminé, lequel va détecter l'incompatibilité entre les contraintes du champ et l'objet contenu. L'erreur a donc été déplacée à cet endroit-là. L'utilisateur est prévenu de cette incompatibilité et il lui est demandé de choisir une nouvelle valeur pour le champ objet parmi celles que le système lui propose. Les objets proposés sont tous les objets de la couche virtuelle qui vérifient les contraintes du champ.

  Si le statut comporte d'autres indicateurs que initial-pairs, une erreur fatale est générée.

- **no-initial-pairs**. Lors de la tentative de rattrapage d'erreur précédente, inconsistency-management demande la non-insertion de ces paires initiales lors de la nouvelle résolution en réinitialisant le statut à {no-initial-pairs}.

  La présence de cet indicateur dans le statut signale donc qu'une tentative de rattapage d'incohérence est en cours.

- **user-pairs**. Au cours de la résolution, il peut arriver, quand l'indicateur DRIVER driver-writing-help vaut $\top$, que l'utilisateur soit mis à contribution pour préciser une valeur de champ ou d'attribut qui se révèle nécessaire pour résoudre le système. Si

cela se produit, l'indicateur user-pairs est ajouté au champ status de l'objet Driver-saving. Sa présence dans le statut indique donc que l'ensemble des paires résolues comprend des paires directement indiquées par l'utilisateur.

Si une incohérence est détectée alors que cet indicateur est présent dans le statut, toujours dans le cas où driver-writing-help vaut $\top$, inconsistency-management tente de rattraper l'erreur.

En effet, comme l'utilisateur est intervenu dans la résolution, on peut supposer que l'incohérence provient d'une erreur de manipulation de sa part. Si c'est le cas, quand il est sollicité à nouveau, il lui suffit de rectifier sa réponse pour que la validation se poursuive correctement. S'il n'a pas fait d'erreur, le problème est alors ailleurs. Il peut alors interrompre et annuler la validation en cours au moment où il est interrogé en cliquant sur le bouton "Annulation transaction" prévu à cet effet.

4.2.3.5 Détails de l'algorithme

Le début de l'algorithme write-object (figure 4.31) est consacré à la recherche du Driver-saving de l'objet en cours d'écriture et à son initialisation pour la classe considérée en vue de la résolution.

Cette résolution à proprement parler est effectuée par write-object1. Quand elle se termine normalement, write-object1 retourne la clé de l'objet écrit et l'écriture de l'objet pour la classe considérée est terminée. Quand une incohérence est détectée et jugée rattrapable par inconsistency-management (cf. §4.2.3.4), cette même procédure provoque la sortie de write-object1 en retournant la valeur {}. Ceci a pour effet de re-déclencher un nouvel appel de write-object1 pour une nouvelle tentative de résolution.

write-object1 débute par l'appel de add-key-pairs qui assure l'introduction dans le système des paires déduites de la clé de l'objet. Bien sûr, ceci n'a de sens que quand l'objet à écrire est déjà persistant au moment où write-object1 (et write-object) est exécuté. Comme nous l'avons vu, ce cas de figure correspond à l'écriture d'un objet dont la classe dans l'environnement est plus précise que celle qu'il a dans la base. L'écriture en cours est alors effectuée pour l'une de ces classes dont l'objet est instance et qui sont plus précises que celle de l'objet dans la base.

Les paires initiales sont ensuite introduites dans le système à condition que l'indicateur no-initial-pairs ne figure pas dans le statut de l'objet Driver-saving

build-saving gère l'interrogation des champs de l'objet.

Les paires fournies par les champs à objet sont déduites des clés des objets référencés. Quand ces objets référencés ne sont pas encore persistants, leurs clés ne sont pas déterminées et aucune paire ne peut être immédiatement déduite. Il n'est pas non plus possible de déclencher récursivement leur écriture avant d'avoir défini et enregistré la clé de l'objet en cours de résolution. En effet, en cas de cycle dans le réseau d'objets, l'algorithme bouclerait. Selon l'état d'avancement de la résolution, les champs à objet peuvent donc ne pas pouvoir immédiatement fournir de nouvelles paires [attribut, valeur] au moment

```
algorithme write-object(Class, Object)
soit Saving ← do-get-object-saving(Object)
et   Key ← {};
faire init-object-saving(Saving, Object, Class);
répéter
    faire Key ← write-object1(Saving, Class) jusqu'à ∃ Key;
retourner Key;
fin
```

```
algorithme write-object1(Saving, Class)
faire add-key-pairs(Saving);
si no-initial-pairs ∉ status.Saving
alors faire add-initial-pairs(Saving);
faire build-saving(Saving, Class);
soit Writings ← build-writings(Saving, Class);
faire do-writings-if-any(Writings)
et    initial-key.Saving ← object-key-while-saving(object.Saving);
retourner initial-key.Saving;
fin
```

```
algorithme build-saving(Saving, Class)
soit Fields ← fields.Class
et Passnum ← 1;
tant que Fields ≠ {}
      faire Fields ← ask-fields-for-saving(Fields, Saving, Passnum);
      si Passnum = 2
         ou (Passnum = 1 et Fields = {})
      alors faire make-up-saving(Saving);
      faire Passnum ← Passnum + 1;
fin
```

Figure 4.31 : Algorithme d'insertion d'un objet dans la base

```
algorithme ask-fields-for-saving(Fields, Saving, Passnum)
soit Unsolved_fields ← {};
pour tout F ∈ Fields
    faire si mappedp.F = ⊤
        et read-only.F = ⊥
        alors soit UF ← info-for-writing(Saving, Passnum).F;
            faire Unsolved_fields ← Unsolved_fields ∪ UF;
retourner Unsolved_fields;
fin
```

```
algorithme make-up-saving(Saving)
soit Unsolved_attrs ← {};
pour tout Attr ∈ unsolved-attrs.Saving
    si attribute-pair(Attr) ∉ prev-pairs.Saving
    faire Unsolved_attrs ← ∪ {Attr};
faire solve-attributes(Unsolved_attrs, Saving);
si ∄ initial-key.Saving
alors faire solve-object-key(Saving);
```

Figure 4.32 : Algorithme d'insertion d'un objet dans la base (suite)

où ils sont interrogés. Il est parfois nécessaire de les reconsulter après que la clé de l'objet en cours d'écriture ait été déterminée.

La consultation des champs se fait en plusieurs passes. Pour chacun d'entre eux, la méthode info-for-writing correspondant à leur type est invoquée. Quand un champ peut fournir, pour la résolution, toutes les paires [attribut, valeur] dont il est capable et qu'il n'est pas nécessaire de le consulter une seconde fois, sa méthode renvoie {}. Par contre, s'il lui manque des informations pour répondre et qu'il doit être ultérieurement ré-interrogé, elle retourne l'objet Driver-field qui lui correspond.

À la première passe, les champs atomiques sont les seuls à fournir des paires [attribut, valeur] pour la résolution. À la seconde, les champs à objet sont interrogés mais seuls ceux qui référencent des objets déjà persistants sont en mesure de déduire de nouvelles paires.

Après la seconde, les attributs fondamentaux non encore résolus sont valués par l'appel à make-up-saving, soit de façon automatique, soit en sollicitant l'utilisateur. make-up-saving vérifie aussi que les attributs fondamentaux participant à une clé de relation se voient affecter des valeurs cohérentes avec les valeurs de clé déjà attribuées dans la base. Cette vérification est effectuée en interrogeant le SGBD par requête SQL. En cas d'incohérence, inconsistency-management est appelé. Dans le cas où l'objet en cours d'écriture n'est pas

encore persistant, make-up-saving se termine en faisant appel à solve-object-key. Cette procédure prend en compte la nouvelle clé de l'objet dans les dictionnaires de DRIVER. Cet enregistrement peut déjà avoir eu lieu s'il a fallu déclencher récursivement l'écriture d'un autre objet référencé non encore persistant.

Avant la troisième passe, les attributs constituant la clé de la relation principale de la classe de l'objet ont donc nécessairement reçu une valeur. La clé de l'objet est déterminée et les champs à objet référençant des objets non encore persistants vont pouvoir déclencher l'écriture récursive de leur contenu sans risquer de boucler à cause de cycles dans le réseau d'objets. Suite à ces écritures, les dernières paires [attribut, valeur] de l'objet sont déterminées et la fonction ask-fields-for-saving retourne {} : la résolution de l'objet est terminée.

build-writings construit ensuite les objets Driver-update et les objets Driver-insert nécessaires pour écrire l'objet dans la base. do-writings-if-any adresse ensuite les requêtes correspondantes au SGBD après les avoir générées.

La clé de l'objet est ensuite reportée dans le champ initial-key de l'objet Driver-saving où elle sera disponible pour un éventuel nouvel appel de write-object pour le même objet mais avec une classe différente.

4.2.3.6 Méthodes d'insertion des différentes classes de champ

Les méthodes info-for-writing des différents types de champs sont présentées figure 4.33. Seules manquent les méthodes des champs de types "ensemble" qui seront présentées ultérieurement (cf. §4.2.4).

La méthode Atom-field::info-for-writing des champs atomiques déduit la valeur de l'attribut associé au champ de la valeur du champ lui-même. Elle introduit ensuite la paire [attribut, valeur] ainsi formée dans le système de résolution. Cette paire est toujours calculable, quelle que soit la valeur du champ; aussi, la méthode retourne-t'elle toujours {}.

La méthode Object2-field::info-for-writing des champs objet de classe inconnue ne consulte la valeur du champ qu'à la seconde passe. Si le champ référence un objet déjà persistant, la clé de cet objet est retrouvée par la fonction object-key-while-saving et la paire [attribut associé au champ, Clé de l'objet référencé] est introduite dans le système.

Si l'objet n'est pas persistant, ce n'est qu'à la troisième passe, quand la clé de l'objet référençant est déterminée, que l'objet référencé est écrit en appelant récursivement les routines d'écriture. Une fois sa clé récupérée, la paire [attribut, clé] constituée est prise en compte.

La méthode Object-field::info-for-writing des champs de type objet est articulée selon le même principe. À la première passe, le champ objet n'est pas consulté. À la seconde, il est examiné et les paires sont déterminées par write-object-field-if-possible (figure 4.34) dans deux cas :

```
algorithme Driver-field::info-for-writing(Saving, Passnum).Field
retourner {};
fin
```

```
algorithme Atom-field::info-for-writing(Saving, Passnum).Field
soit Attribute_value ← attr-value-to-database(attr.Field, Field, object.Saving);
faire add-pair(Saving, [attr.Field, Attribute_value]);
retourner {};
fin
```

```
algorithme Object2-field::info-for-writing(Saving, Passnum).Field
selon
  cas Passnum = 1
      retourner {Field};
  cas Passnum = 2
      soit Contained_object ← field-value(Field, object.Saving);
      et   Key ← object-key-while-saving(Contained_object);
      si ∃ Key
      alors faire add-pair(Saving, [attr.Field, Key])
           et     retourner {};
      sinon retourner {Field};
  cas sinon
      soit Contained_object ← field-value(Field, object.Saving);
      et   Key ← do-get-object-key(Contained_object, {}, {}, Field);
      faire add-pair(Saving, [attr.Field, Key])
      et     retourner {};
fin
```

```
algorithme Object-field::info-for-writing(Saving, Passnum).Field
selon
  cas Passnum = 1
      retourner {Field};
  cas Passnum = 2
      retourner write-object-field-if-possible(Field, Saving);
  cas sinon
      retourner write-object-field(Field, Saving);
fin
```

Figure 4.33 : Méthodes info-for-writing

- **Quand le champ est vide :**
  S'il est associé à une jointure restreinte, tous les attributs contraints doivent déjà avoir été résolus. Il faut en plus que l'absence de valeur pour le champ soit autorisée et, pour cela, que la jointure objet restreinte fasse également intervenir des attributs référentiels non contraints.

- **Quand le champ référence un objet :**
  Il faut que cet objet soit déjà persistant. Si le champ est associé à une jointure restreinte, il faut également que la clé de l'objet soit compatible avec les attributs contraints déjà valués.

Si l'un de ces deux ensembles de conditions est vérifié, les paires déduites de la valeur du champ sont insérées dans le système. Dans le premier cas, les attributs référentiels non contraints se voient affecter la valeur NULL. Dans le second, tous les attributs référentiels sont valués en fonction de la clé de l'objet référencé.

Dans les autres cas, le traitement du champ est reporté à la troisième passe.

Quand débute cette troisième passe, tous les attributs fondamentaux de la classe sont valués et la clé de l'objet en cours de résolution a été déterminée. Les traitements des champs objet n'ayant pu être faits lors de la deuxième passe sont effectués par write-object-field (figure 4.35). Cette fonction obtient la clé de l'objet référencé par la fonction contained-object-key2 et en déduit les paires [attribut référentiel, valeur] correspondantes.

Le principe de contained-object-key2 est similaire à celui de contained-object-key (figure 4.23). En fait changent uniquement les outils d'obtention des attributs contraints et de leurs valeurs. Tandis que contained-object-key les tirait de l'objet Driver-select et du n-uplet de l'objet écrit, contained-object-key2 les trouve pour sa part dans l'objet Driver-saving qui gère ici la résolution de l'objet.

Après détermination des attributs référentiels contraints et de leurs valeurs, contained-object-key2 fait appel à do-get-object-key pour déclencher la recherche ou le calcul de la clé de l'objet référencé. Comme dans contained-object-key, la vérification de la validité de cette clé est réalisée par la procédure check-contained-object-key.

La clé de l'objet référencé peut avoir déjà été attribuée au cours de cette troisième passe, lors du traitement d'autres champs à objet. Nous avons vu en effet que l'écriture d'un objet effectué par save-object entraînait, dans la même opération, l'écriture de tous ceux qui étaient également référencés dans le même class-object-buffer que lui.

L'écriture de l'objet référencé est déclenchée quand clé demeure inaccessible. Si la jointure objet associée au champ est restreinte, constrained-key-attrs détermine les attributs clé qui leur y sont comparés. Ce sont ces attributs et leurs valeurs imposées qui vont former les éventuelles paires initiales intervenant dans la résolution de l'objet référencé.

Illustrons l'algorithme write-object complet en reprenant l'exemple de validation du vendeur `james` et des objets qu'il référence (cf. §4.2.1.3). Lors de cette validation, le premier appel à write-object est effectué pour l'objet `johnson` et la classe `Employé`. Après

algorithme write-object-field-if-possible($Field, Saving$)
soit $Object\_key\_attrs \leftarrow$ key.deftable.table.ownerclass.$Field$
et $Read\_attrs \leftarrow$ ra-compil.$Field$
et $Const\_attrs \leftarrow Read\_attrs \cap$ solved-attributes($Saving$)
et $Not\_const\_attrs \leftarrow Read\_attrs \setminus Const\_attrs$
et $Const\_attrs2 \leftarrow Read\_attrs \cap Object\_key\_attrs$
et $Const\_attr\_values \leftarrow$ saving-attribute-values($Saving, Const\_attrs$)
et $Cont\_object \leftarrow$ field-value($Field$, object.$Saving$)
et $Key \leftarrow$ object-key-while-saving($Cont\_object$)
et $Attr\_values \leftarrow$ attr-values-from-key(Key);
selon
  cas $\nexists\ Cont\_object$
    et $Const\_attrs2 \subset Const\_attrs$
    et $Not\_const\_attrs \neq \{\}$
    faire build-and-add-pairs($Not\_const\_attrs, \{\}, Saving$);
  cas $\exists\ Cont\_object$
    et $\exists\ Key$
    et compatible-object-key-p($Read\_attrs, Attr\_values, Const\_attrs, Const\_attr\_values$)
    faire build-and-add-pairs($Read\_attrs, Attr\_values, Saving$);
  cas sinon
    retourner $\{Field\}$;
fin

Figure 4.34 : Algorithmes de write-object-field-if-possible

algorithme write-object-field($Field, Saving$)
soit $Read\_attrs \leftarrow$ ra-compil.$Field$
et $Key \leftarrow$ contained-object-key2($Field, Saving$)
et $Attr\_values \leftarrow$ attr-values-from-key(Key);
faire build-and-add-pairs($Read\_attrs, Attr\_values, Saving$);
retourner {};
fin

algorithme contained-object-key2($Field, Saving$)
soit $Object\_key\_attrs \leftarrow$ key.deftable.table.ownerclass.$Field$
et $Read\_attrs \leftarrow$ ra-compil.$Field$
et $Const\_attrs \leftarrow Read\_attrs \cap$ solved-attributes($Saving$)
et $Const\_attr\_values \leftarrow$ saving-attribute-values($Saving, Const\_attrs$)
et $Cont\_object \leftarrow$ field-value($Field$, object.$Saving$)
et $Key \leftarrow$ do-get-object-key($Cont\_object, Const\_attrs, Const\_attr\_values, Field$);
faire check-contained-object-key($Key, Read\_attrs, Const\_attrs, Const\_attr\_values,$
($Read\_attrs \subset Object\_key\_attrs$), object.$Saving, Field$);
retourner Key;
fin

Figure 4.35 : Algorithmes de write-object-field

avoir été retrouvé, l'objet Driver-saving de `johnson` est initialisé par l'appel à init-object-saving. Sa valeur est ensuite celle qui a été présentée figure 4.29.

`johnson` n'étant pas encore persistant, l'appel à add-key-pairs est sans effet. Aucune contrainte n'ayant été posée sur les attributs de clé comme en témoigne le champ initial-pairs vide dans l'objet Driver-saving, celui à add-initial-pairs n'en a pas non plus. La résolution de l'objet est ensuite engagée.

La plupart des champs de la classe `Employé` sont traités lors de la première passe puisqu'au cours de celle-ci, la liste des champs non encore traités passe de {`name`, `prenom`, `num-ss`, `qualif`, `chef`, `responsable-de`, `salaire`, `grade`, `dpt`, `adresse`, `tel-prive`, `vehicule`} à {`chef`, `responsable-de`, `dpt`, `adresse`, `vehicule`}. À l'issue de la première passe, l'objet Driver-saving a pour valeur celle présentée figure 4.36.

Objet de classe Driver-saving	
...	...
object	Object("`johnson`")
initial-key	()
new-key	()
object-class	Driver-class(`Vendeur`)
status	()
unsolved-attrs	(`emp.empno person.pname person.fname emp.ename emp.fname`)
initial-pairs	()
user-pairs	()
prev-pairs	()
pairs	([`person.ssnum`, `NULL`] [`emp.job`, `salesman`] [`emp.sal`, `1000.`] [`person.phone`, `NULL`])
prev-transitions	(T[Object("`johnson`"), `qualif`, `()`] T[Object("`johnson`"), `salaire`, `()`])
transitions	()

Figure 4.36 : Objet Driver-saving de `johnson` (classe `Employé`) après la première passe

`emp.ename` et `emp.fname`, bien que correspondances des champs `nom` et `prenom`, sont toujours non résolus. La raison en est que ces champs sont read-only et que les champs read-only ne sont pas consultés pour la résolution (cf. figure 4.32).

À la seconde passe, La plupart des champs sont traités : le champ `responsable-de`, qui ne référence aucun objet, et les champs `chef`, `dpt`, `vehicule` qui référencent des objets persistants. Ces trois champs objet n'étant pas associés à des jointures restreintes, il n'y a aucune incompatibilité entre eux et leurs valeurs.

Le champ `adresse` par contre n'a pas encore pu être pris en compte dans la résolution. En effet, il est vide, associé à une jointure objet J[`emp`→`address`] (de définition `emp.empno = address.empno`) qui est fixée, donc restreinte a fortiori, et tous les attributs contraints ({`emp.empno`}) ne sont pas valués. Il sera traité à la troisième passe.

À la fin de la seconde passe, les attributs fondamentaux non encore résolus sont déter-

minés. Ce sont {emp.empno, person.pname, person.fname, emp.ename, emp.fname}. Les attributs correspondances de champs sont considérés en premier. Pour ceux-là, DRIVER regarde d'abord si une valeur n'est pas déjà disponible dans le champ. C'est le cas du champ nom qui contient "johnson". Sinon, il demande à l'utilisateur de valuer le champ. Dans notre exemple, le système demande une valeur pour le champ prenom, question à laquelle on répond par "ben". person.pname et person.fname sont déterminés par propagation de valeurs dans les contraintes. Si driver-fix-key-generation-p vaut ⊤, une valeur est générée pour emp.empno, par exemple, 7936.

À l'issue de la seconde passe, l'objet Driver-saving a pour valeur celle présentée figure 4.37.

Objet de classe Driver-saving	
...	...
object	Object("johnson")
initial-key	()
new-key	()
object-class	Driver-class(Vendeur)
status	(user-pairs)
unsolved-attrs	()
initial-pairs	()
user-pairs	([emp.fname, ben])
prev-pairs	()
pairs	([person.ssnum, NULL] [emp.job, salesman] [emp.sal, 1000.] [person.phone, NULL] [emp.mgr, 7698] [emp.responsible_for, NULL] [emp.deptn, 30] [person.car, NULL] [emp.fname, ben] [person.fname, ben] [emp.ename, johnson] [person.pname, johnson] [emp.empno, 7936])
prev-transitions	(T[Object("johnson"), qualif, ()] T[Object("johnson"), salaire, ()])
transitions	(T[Object("johnson"), prenom, ()])

Figure 4.37 : Objet Driver-saving de johnson (classe Employé) après la seconde passe

Le champ adresse reste le seul en lice pour la troisième passe. La méthode info-for-writing des champs objet est invoqué et contained-object-key2 est appelé pour déterminer la clé de l'objet référencé. Or, il n'y en a aucun. do-get-object-key retourne donc {} et check-contained-object-key teste ensuite la validité de cette valeur.

Or, ce champ est associé à une jointure objet fixée. Le seul attribut référentiel qu'elle comporte, emp.empno, est contraint et n'a pas de valeur : {} n'est donc pas une valeur autorisée pour adresse. Si driver-writing-help vaut ⊤, le système demande à l'utilisateur de choisir une autre valeur pour le champ. Comme la jointure associée est fixée, et qu'aucun objet Adresse de clé address/7936 n'existe, il n'y a pas de solution parmi les objets existants et DRIVER propose de valuer le champ avec un nouvel objet à créer. Si l'utilisateur valide ce choix, l'écriture de l'objet Adresse créé est aussitôt enchaînée par do-get-object-key. Cette fois, l'ensemble des attributs référentiels contraints de la jointure objet n'est pas vide puisqu'il vaut {emp.empno}. De cet ensemble et de la jointure elle-même est déduit

l'ensemble des attributs contraints de l'objet `Adresse` qui va être écrit. Cet ensemble est {`address.empno`}. La valeur contrainte de l'attribut `address.empno` est `7936`. La paire [`address.empno`, `7936`] sera, lors de l'écriture de l'objet, la seule paire initiale de la résolution.

L'objet Driver-saving de `johnson` pour la classe `Employé` ne subit plus ensuite de modification jusqu'à la fin de l'exécution de write-object. Sa valeur finale est donc identique à celle présentée figure 4.37.

Il permet finalement, par l'intermédiaire de la procédure build-writings, de construire un certain nombre d'objets Driver-insert. Comme nous allons le voir maintenant, ces objets permettent de générer des requêtes SQL qui assurent l'insertion dans la base des n-uplets correspondant à l'objet en cours d'écriture.

4.2.3.7 Les requêtes d'insertion

DRIVER insère dans la base un objet relationnel en adressant au SGBD des requêtes d'insertion. De telles requêtes sont représentées en DRIVER par des objets de la classe Driver-insert. Ces objets sont construits par la fonction build-writings à partir des objets Driver-saving. Une fois complets, ils permettent de générer les requêtes SQL correspondantes.

```
INSERT INTO tablename [(column {, ... })]
    VALUES (select_expr {, ... })

INSERT INTO tablename [(column {, ... })] subselect
```

Figure 4.38 : Syntaxe d'une requête SQL d'insertion de données

La syntaxe générale de la requête SQL d'insertion est donnée figure 4.38. Disons, en quelques mots, que la requête est introduite par le mot-clé INSERT et qu'elle s'adresse à une seule table. La liste des attributs se voyant affecter une valeur est précisée derrière le nom de la table. Elle est suivie du mot clé VALUES et de la liste des valeurs associées à ces attributs.

La seconde syntaxe, bien que disponible, n'est pas utilisée par DRIVER.

La classe Driver-insert a déjà été décrite en §4.2.2.3.

À titre d'exemple, la figure 4.39 présente les deux objets Driver-insert construits dans la précédente section (§4.2.3.6) pour insérer dans la base les n-uplets correspondant à l'objet `johnson`.

Objet de classe Driver-insert	
table	(emp)
null-attributes	(emp.responsible_for)
attributes	(emp.job emp.sal emp.mgr emp.deptn emp.fname emp.ename emp.empno)
values	(salesman 1000. 7698 30 ben johnson 7936)

Objet de classe Driver-insert	
table	(person)
null-attributes	(person.ssnum person.phone person.car)
attributes	(person.fname person.pname)
values	(ben johnson)

Figure 4.39 : Objets Driver-insert de mise-à-jour de l'objet johnson

Les deux requêtes correspondant à ces objets sont :

- ```
  insert emp
         (responsible_for, job, sal, mgr, deptn, fname, ename, empno)
  values (NULL, 'salesman', 1000., 7698, 30, 'ben', 'johnson', 7936);
  ```

- ```
  insert person
         (ssnum, phone, car, fname, pname)
  values (NULL, NULL, NULL, 'ben', 'johnson');
  ```

4.2.4 Écriture des champs de type "ensemble"

Les méthodes d'écriture des champs de type "ensemble" n'ont pas encore été complètement étudiées. Elles seront détaillées dans une publication ultérieure.

4.2.5 Modification d'objet en cours d'écriture

Quand l'indicateur système DRIVER driver-writing-help, vaut $\top$, la validation de transaction peut engendrer des modifications d'objet.

DRIVER gère un historique de ces modifications de façon à pouvoir les défaire en cas d'incohérence ou d'erreur. Cet historique se présente sous la forme d'une liste d'objets transitions, instances de la classe Driver-transition.

Listing 23. La classe Driver-transition

```
(driver-defclass Driver-transition ()
   (object    object2)
   (fieldname atom symbol)
   (oldvalue  *)
```

Chaque transition comprend trois informations (listing 23) :

- l'objet modifié;
- le champ modifié;
- l'ancienne valeur de ce champ dans l'objet considéré. Cette valeur ne peut ête typée. En effet, selon le type du champ, elle pourra être un symbole, un entier, un objet, un ensemble, etc...

Défaire des modifications se fait simplement en réaffectant au champ d'un objet son ancienne valeur. Comme un même champ aura pu être modifié plusieurs fois, il est important de revenir sur ces modifications dans l'ordre inverse de celui dans lequel elles ont été effectuées.

L'historique de chaque objet est géré dans le champ transitions de son Driver-saving associé.

4.2.6 Erreur fatale en cours d'écriture

En cas d'erreur grave intervenue pendant la validation de transaction, DRIVER fait appel à la procédure fatal-error (figure 4.40).

Cette procédure interrompt l'opération de validation de transaction en générant une erreur. Cependant, avant de provoquer cette erreur, elle procède à la restauration de l'environnement. Il est en effet souhaitable que l'exécution de driver-commit-objects, explicitement demandée par l'utilisateur, rende l'environnement tel qu'il était au début de la validation dans le cas où cette validation ne peut être menée à son terme. En cas d'échec, tout doit se passer comme si la tentative de validation n'avait jamais eu lieu.

```
algorithme fatal-error(Message)
faire dbms-rollback(dbms.driver-current-cursor())
et    undo-transitions()
et    clear-saving-buffer()
et    error(Message);
fin
```

Figure 4.40 : Algorithme de traitement des erreurs fatales

La restauration de l'environnement comprend principalement deux opérations :

- **La restauration de la base de données**. Toutes les requêtes d'insertions et de mise-à-jour adressées au SGBD dans le cadre de cette validation sont annulées. La base est ainsi replacée dans l'état où elle était au début de la validation. Cette opération est réalisée en envoyant au SGBD une requête SQL **rollback work**.
- **La restauration de la couche objet en mémoire**. Toutes les modifications effectuées en mémoire sur les objets dans le cadre de la validation sont annulées. Tous les champs qui ont été modifiés sont ré-affectés de leurs précédentes valeurs. Cette opération est possible grâce à la gestion par DRIVER des éléments de transition (cf. §4.2.5).

On remarquera enfin que le dictionnaire des objets système Driver-saving est vidée au moyen de l'appel à clear-saving-buffer de façon à libérer immédiatement la mémoire d'objets système volumineux qui de toute façon ne seront plus utilisés.

4.3 Suppression des objets relationnels

La suppression d'objets relationnels dans la base n'a pas encore été implantée en DRIVER.

Cependant, l'étude de l'algorithme correspondant a été réalisée et aucun problème particulier n'a été identifié. La suppression d'un objet en DRIVER se décompose simplement en trois opérations principales successives (figure 4.41) :

```
algorithme driver-delete-object(Object)
faire delete-object-references-in-layer(Object)
et    record-object-deletion(Object);
fin
```

```
algorithme delete-object-references-in-layer(Object_to_delete)
pour tout Class ∈ classes.driver-current-mapping()
    faire delete-object-references-in-class-objects(Class, Object_to_delete);
fin
```

```
algorithme delete-object-references-in-class-objects(Class, Dobject)
soit Object_fields ← object-class-fields(Class, object-class(Dobject))
et   Object_filter ← referenced-object-filter(Object_fields)
et   Class_object_references ← driver-select-objects(Object_filter);
faire driver-load-class-objects(Class, Class_object_references);
pour tout Object ∈ instances.main-class(Class)
    si class-instance-p(Object, Class)
    alors faire delete-object-references-in-object(Object, Object_fields, Dobject);
fin
```

Figure 4.41 : Algorithme de suppression d'objet

- La première a pour but d'éliminer de l'ensemble des objets de la couche virtuelle toute référence à l'objet à supprimer.

- La seconde "marque" l'objet en question comme étant supprimé. Ceci est fait en ajoutant un objet Driver-deletion (listing 24) le concernant au contenu du champ deleted-objects du schéma de correspondances courant.

  Le champ deleted-object référence l'objet détruit et le champ uncommitted-objects, les objets qui le référençaient et qui n'ont pas encore été validés dans la base.

- La troisième opération n'est effectuée qu'après validation dans la base[8] de tous les objets qui référençaient l'objet détruit. Elle n'apparaît donc pas dans le détail de l'algorithme figure 4.41.

  Elle se traduit d'une part par la suppression dans la base de tous les n-uplets correspondant à l'objet supprimé. Il y en a un par table élémentaire de la classe de l'objet.

  Elle provoque, d'autre part, la suppression dans DRIVER de toute référence système à l'objet, en particulier son référençage ou le référençage de sa clé dans les dictionnaires.

Listing 24. Classe système de gestion de suppression d'objet

```
(driver-defclass Driver-deletion ()
  (deleted-object      object2)
  (uncommitted-objects object2set))
```

La première opération est la plus complexe, aussi nous allons la détailler.

Les références éventuelles à l'objet à supprimer sont contenues dans les champs à objet (objet, ensemble d'objets et ensemble ordonné d'objets). Pour chaque classe du schéma de correspondances sont donc déterminés dans un premier temps les champs à objet référençant des objets de classe une des classes ancêtres de la classe de l'objet à supprimer ou sa classe elle-même.

Quand une classe comporte au moins un tel champ, une requête objet est construite et adressée à la base au moyen de la fonction driver-select-objects. Son but est d'y retrouver les objets relationnels de la classe considérée référençant l'objet à supprimer. S'il y en a, il est procédé à la construction en mémoire de leurs objets jumeaux, ceci uniquement dans le cas où ces objets jumeaux n'y auraient pas déjà été construits.

Enfin, sont recherchés parmi l'ensemble des objets jumeaux tous ceux qui référencent l'objet à détruire. Pour chacun d'entre eux, le champ à objet concerné est modifié pour en enlever la référence de l'objet considéré.

[8] Par validation(s) de transaction(s) DRIVER.

Il convient d'insister sur le fait que ce troisième volet de la première opération implique TOUS les objets jumeaux, donc pas seulement ceux des objets relationnels filtrés précédemment. De cette façon, on tient également compte des modifications d'objets effectuées pendant la transaction courante. Si, au cours de cette transaction, un objet a été modifié de façon à ce qu'il référence ou ne référence plus l'objet à supprimer, c'est sa dernière version qui est ainsi considérée, et pas sa valeur dans la base.

La troisième opération ne peut être effectuée que quand tous les objets qui référençaient l'objet détruit ont été validés dans la base. En effet, si elle était déclenchée avant, la base de données se trouverait être incohérente car des objets relationnels, en l'occurrence ceux correspondant aux objets jumeaux non encore validés, référenceraient encore l'objet détruit alors que celui-ci n'existerait déjà plus dans la base. Il suffirait alors d'une annulation de transaction pour l'un de ces objets pour que la couche objet de DRIVER soit incohérente.

On déduira d'autre part de ce qui précède qu'il suffit en DRIVER d'une annulation de transaction pour l'un des objets qui référençaient l'objet détruit pour également annuler la destruction de l'objet référencé. Par contre, si aucune annulation de transaction n'est effectuée pour les autres, ils continuent à ne plus référencer l'ex-"objet détruit".

4.4 Sélection d'objets relationnels dans la base

4.4.1 Généralités

La recherche d'objets relationnels dans la base s'effectue par un appel à la fonction driver-select-objects. Le critère de sélection est décrit sous forme d'un filtre qui constitue l'unique argument de la fonction.

Le filtre est un prédicat à n variables où chacune d'entre elles représente une instance d'une classe précisée, décrite dans le schéma de correspondances. Le corps du prédicat est un ensemble de contraintes liant généralement ces n variables et pouvant être des comparaisons entre objets, entre champs, entre champs et valeurs.

Dans l'implantation LISP de DRIVER, le filtre prend la forme d'une lambda-expression exprimée dans un environnement où sont précisées les classes des objets susceptibles d'être unifiés à ses variables :

```
((LAMBDA (objet1 ... objetN)
     corps-lambda)
 classe-objet1 ...  classe-objetN)
```

Les accès composés aux champs de ces objets sont autorisés. À titre d'exemple, la requête présentée figure 4.42 sélectionne dans la base toutes les paires [employé - chef de l'employé] pour lesquelles l'employé a une voiture de même modèle que celle de son chef. Ne sont demandées que les paires où le chef est au moins instance de la classe `Cadre`.

Ici, une seule paire est solution. L'employé ayant un tel véhicule est celui dont l'`empno` vaut 7782, et son chef est le cadre dont l'`empno` vaut 7839.

```
? (driver-select-objects
            '((lambda (objet1 objet2)
                  (and (eq objet2 (send 'chef objet1))
                       (eq (send 'modele (send 'vehicule objet1))
                           (send 'modele (send 'vehicule (send 'chef objet1))))))
              Employe Cadre))
= (((7782) (7839)))
```

Figure 4.42 : Exemple de sélection d'objets relationnels

driver-select-objects se limite à renvoyer des références symboliques sur les objets relationnels solution au lieu de directement renvoyer les objets jumeaux ou défauts d'objet correspondants, éventuellement construits pour l'occasion. On peut s'en étonner, d'autant plus que le système fournit également une primitive, driver-get-object, qui retourne l'objet correspondant à la référence qu'on lui indique. Cette référence, qui est la valeur de clé du n-uplet principal de l'objet, est semblable à celles retournées par la fonction de filtrage.

La raison en est qu'un n-uplet d'objets relationnels peut très bien vérifier dans la base les conditions imposées par un filtre, au contraire du n-uplet d'objets correspondants en mémoire. Cette divergence peut se produire quand ces n-uplets solution impliquent des objets pour lesquels des jumeaux ont déjà été construits et modifiés dans l'environnement.

De ce fait, si driver-select-objects retournait les objets jumeaux des objets relationnels solution, certains n-uplets d'objets retournés ne vérifieraient paradoxalement même pas le critère de sélection précisé par l'utilisateur.

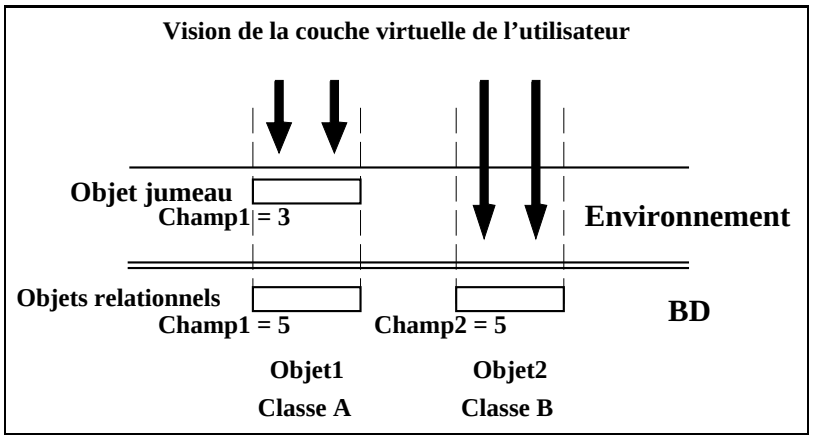

Figure 4.43 : Justification de la sélection d'objets en deux temps

La figure 4.43 présente un exemple qui illustre ce cas de figure. L'`Objet1` a été modifié dans l'environnement et n'a pas encore été validé dans la base. L'`Objet2` n'a, pour sa

part, fait l'objet d'aucune mise-à-jour.

Si on filtre dans la base les paires d'objets relationnels, l'un étant de classe `A`, le second de classe `B`, vérifiant $Objet\_class\_A.Champ1 = Objet\_class\_B.Champ2$, la paire [`Objet1, Objet2`] est solution. Pourtant, du fait des modifications effectuées dans l'environnement, la paire des objets jumeaux correspondants ne vérifie pas le critère de sélection : $Champ1.Objet1 = 3 \neq Champ2.Objet2 = 5$.

En conclusion, nous n'avons pas aujourd'hui de solution efficace permettant le filtrage d'objets sur l'ensemble de la couche objet virtuelle de DRIVER.

4.4.2 CLERIC

Les filtres de sélection d'objets sont compilés sous une forme objet par $CLERIC$, **C**ompilateur de **L**ambda **E**xpression en **R**eprésentation **I**ntermédiaire de **C**ontrainte. Ces représentations objet sont une Représentation Intermédiaire (RI) qui permet ensuite d'aisément générer du SQL.

CLERIC est également utilisé pour générer de semblables RI pour les jointures, les contraintes de classes sur un champ ou sur un attribut, à partir des déclarations de l'utilisateur. Les classes système des objets utilisés dans ces RI sont présentées figure 4.44.

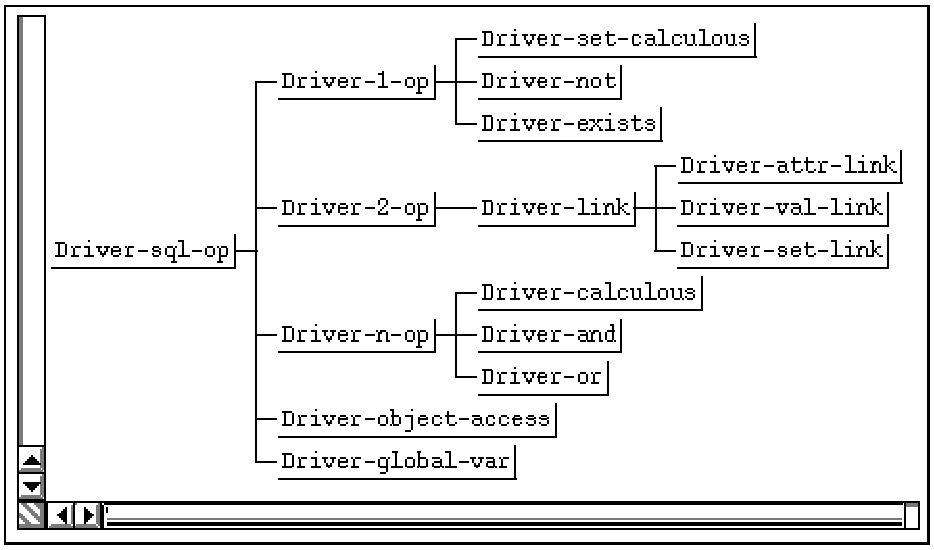

Figure 4.44 : Hiérarchie de la classe Driver-sql-op

Le compilateur fonctionne dans un contexte courant qui lui indique quelles sont les fonctions autorisées, leurs arguments, le type de ces arguments. Le contexte est composé d'un ensemble d'indicateurs système :

- **calculous**. Ce mode autorise l'utilisation des quatre opérations dans l'expression. Les arguments peuvent être de nouveaux appels fonctionnels, des variables ou des valeurs.

- **equation**. Ce mode autorise l'utilisation des prédicats and, or, not et null. Ce dernier permet de déterminer si un terme a une valeur ou non.
- **multi-tuple**. Ce mode indique que l'expression est un calcul sur un ensemble de n-uplets. Dans ce mode, les calculs de sommes et de moyennes sont autorisés.
- **attribute-manipulation**. Ce mode indique que les variables correspondent à des attributs.
- **object-manipulation**. Ce mode indique que les variables correspondent à des objets. Les accès aux champs de ces objets sont autorisés.
- **global-variables**. Enfin, ce dernier mode autorise l'usage de variables globales dans l'expression. Il est particulièrement utile pour l'usage de driver-select-objects, afin de permettre la définition de filtres d'objets dont les critères de sélection évoluent à chaque nouvelle évaluation de la même règle.

À titre d'exemple, la requête SQL générée pour la sélection d'objets relationnels présentée figure 4.42 est :

```
select distinct var126.empno,var127.empno
from emp var126,emp var127
where exists(select * from emp var128
             where var126.mgr = var128.empno and var127.empno = var128.empno)
      and exists(select * from vehicle var129,person var130,emp var131,
                              vehicle var132,person var133
                 where var126.ename = var130.pname
                       and var126.fname = var130.fname
                       and var130.car = var129.vnum
                       and var126.mgr = var131.empno
                       and var131.ename = var133.pname
                       and var131.fname = var133.fname
                       and var133.car = var132.vnum
                       and var129.model = var132.model)
      and var126.sal >= 700. and var126.sal <= 9999.
      and var127.job in ('president','manager')
      and var127.sal >= 700. and var127.sal <= 9999.
```

CONCLUSION

Tout en proposant une solution au problème de l'"Impedance Mismatch", nous avons montré, dans cette thèse, qu'il était possible de construire un système proposant des correspondances entre des concepts communs à de très nombreux modèles objet et ceux existant dans le modèle relationnel.

Nous avons ainsi le moyen d'associer une représentation objet à toute base relationnelle. Notre principal résultat est de pouvoir transformer aux yeux de l'utilisateur un SGBD relationnel classique en SGBD Orienté Objet.

Le SGBD objet ainsi produit présente deux originalités majeures :

- *L'utilisateur garde son modèle objet.*

  En effet, DRIVER peut prendre en compte tout modèle objet par simple définition d'un jeu de méthodes permettant de réaliser les opérations classiques de manipulation d'objet.

- *Les bases exploitées continuent à fonctionner en relationnel pur.*

  Grâce aux correspondances systématiques, nous sommes capables de prendre en compte une base relationnelle quelconque, lui attribuer une représentation pertinente dans un modèle objet choisi, et l'exploiter aussitôt comme une base objet. Toute application relationnelle disponible sur le marché est ainsi immédiatement exploitable sous forme objet.

  Inversement, tout objet, produit de raisonnement ou de calcul, sauvé dans une base DRIVER, est instantanément disponible sous la forme relationnelle, la plus répandue. La communication de données à un partenaire ne disposant pas de DRIVER ne nécessite aucune conversion ou traduction. Bien sûr, si ce partenaire utilise lui-même DRIVER, il peut immédiatement exploiter ces données dans le modèle objet de son choix et avec la représentation qui lui convient le mieux pour son application.

Perspectives

On peut d'ores et déjà envisager d'enrichir cette première version de DRIVER de nouvelles fonctionnalités.

- **Modèle objet canonique**. L'ensemble des concepts objet pour lesquels ont été identifiées des correspondances relationnelles définissent ou, plus précisément, font émerger un *modèle objet canonique*. Toute représentation objet exprimée dans ce modèle peut être rendue complètement persistante.

  Ce modèle canonique est extensible : la définition de nouvelles correspondances peut permettre la prise en compte de nouveaux concepts. Cependant, dans la version actuelle du système, si un client souhaite utiliser, dans une base de connaissance DRIVER, une caractéristique de modèle objet non encore gérée, sa prise en compte peut se révéler difficile. Rendre le modèle objet canonique de DRIVER extensible par l'utilisateur est un axe de recherche à part entière.

- **Typage des champs**. Le typage des champs doit être réexaminé et épuré. Les constructeurs de types déjà disponibles sont à généraliser de façon à les rendre orthogonaux. La composition des constructeurs de type doit en effet pouvoir être réalisée librement. Par exemple, il serait souhaitable de ne plus être limité aux seuls ensembles d'atomes et d'objets et de pouvoir définir des ensembles d'ensembles, etc.

  Actuellement, seuls les constructeurs "ensemble" et "liste" (ensemble ordonné) sont disponibles. Il serait intéressant de les compléter des constructeurs "n-uplet", "sac", "dictionnaire" existant par exemple dans O2.

- **Filtrage complet**. Un filtrage des objets relationnels a été proposé. Il reste cependant à étudier une fonction de filtrage pouvant s'appliquer à l'ensemble de la couche objet virtuelle.

- **Sous-traitance**. Dans certaines applications, l'évaluation de règles de production pourrait être directement réalisée au sein du SGBD sous forme de requêtes SQL. Le contrôle de ce type d'inférence est une autre voie d'approfondissement possible.

- **Gestion de versions**. DRIVER ne gère pas encore les versions d'objet. Actuellement, deux validations successives d'un même objet persistant se soldent nécessairement par l'écrasement dans la base de l'ancienne valeur de l'objet par sa nouvelle valeur.

  Or, au cours d'un raisonnement effectué par un système d'IA, un même objet peut transiter par une succession d'états qui marquent les différentes étapes de sa construction. S'il est persistant, il peut être intéressant de pouvoir sauver dans la base de données des versions différentes de cet objet qui correspondent à des états par lesquels il a transité ou à des solutions alternatives.

- **Persistance des méthodes**. La persistance des méthodes dans la base est un autre sujet d'étude : la forme sous laquelle elles y seraient stockées est à définir. Il serait en effet intéressant de préserver au mieux l'indépendance des données vis-à-vis de tout langage.

- **Distribution**. L'exploitation de DRIVER par réseau est un autre de nos objectifs. Nous travaillons actuellement sur un projet d'environnement d'IA distribué et nous aimerions pouvoir utiliser DRIVER dans cet environnement pour gérer nos bases de connaissances de grande taille.

  Une extension du langage assurant la communication au sein d'un environnement objet distribué peut être réalisée dans cette perspective.

- DRIVER **multi-utilisateurs**. Actuellement, DRIVER ne gère pas les fonctionnalités du SGBD liées à son exploitation en multi-utilisateurs, par exemple les droits d'accès. Cette extension pourrait être intéressante pour certaines applications.

- Enfin, DRIVER a été implanté en Le_Lisp v15. Il serait intéressant de le réécrire dans d'autres langages comme C++ pour le rendre disponible à un plus grand nombre d'utilisateurs.

Références

[AB84] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organised data. In *Proceedings of 3th Symposium on Principles of database systems*, ACM SIGACT-SIGMOD, Waterloo (California), April 1984, pp. 191–200.

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 1983, vol. 26, no. 4, pp. 360–365.

[ABD+89] M. P. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. Le Chesnay (France) : INRIA - LRI, August 1989, 19 pages, Technical Report ALTAIR 30-89.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. GALILEO : A strongly-typed, interactive conceptual language. *ACM Transactions on database Systems*, June 1985, vol. 10, no. 2, pp. 230–260.

[AG87] S. Abiteboul and S. Grumbach. Bases de données et objets structurés. *Technique et Science Informatiques*, 1987, vol. 6, no. 5, pp. 383–404.

[AG89a] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment) : The language and the data model. In *Proceedings of the SIGMOD international conference on Management of Data*, ACM SIGMOD, Portland (Oregon), May-June 1989, pp. 36–45.

[AG89b] R. Agrawal and N. H. Gehani. Rationale for the design of persistence and query processing facilities in the database programming language O++. In *Proceedings of 2nd International workshop on database programming languages*, Gleneden Beach (Oregon), June 1989, pp. 1–16.

[AH87] T. Andrew and C. Harris. Combining language and database advances in an object-oriented development environment. In *Proceedings of the international conference on Object-Oriented Programming Systems, Languages and Applications*, Orlando (Florida), October 1987, ACM SIGPLAN Notices, vol. 22, no. 12, pp. 430–440.

[Aid92] *AÏDA Version 1.65, Manuel de référence*, Gentilly (France) : Ilog, Mars 1992, 498 pages.

[ANSI86] American National Standards Institute, editor. *American National Standard for Information systems, database language, SQL*, vol. X3.l35. New-York : American National Standard for Information systems, 1986.

[Asq89] *ASQUELL Version 2.02, Manuel de référence*, Gentilly (France) : Ilog, Juin 1989, 23 pages.

[Atw85] T. M. Atwood. An object-oriented DBMS for design support applications. In *Proceedings of the international conference on Computer-Aided Technologies*, Montreal (Canada), September 1985, pp. 299–307.

[BA87] C. Beeri and S. Abiteboul. An algebra and a calculus for complex objects - Abstract. In *International workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt (Germany), April 1987, pp. 56–57.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD : A powerful and simple database language. In *Proceedings of 13th Very Large Databases conference*, Brighton (England), September 1987, pp. 97–105.

[BC85] A. L. Brown and W. P. Cockshot. The CPOMS Persistent Object Management System. Universities of Glasgow & St Andrew (Scotland), 1985, Technical Report 13-85.

[BCG+87] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk, and N. Ballou. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, January 1987, vol. 5, no. 1, pp. 3–26.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an object-oriented database system : the story of O2.* San Mateo (California) : Morgan Kaufmann Publishers, 1992, 625 pages.

[BDN+86] J. Bocca, H. Decker, J.-M. Nicolas, L. Vieille, and M. Wallace. Some steps towards a DBMS-based KBMS. In *IFIP 10th World Computer Congress*, Dublin (Ireland), September 1986, pp. 1061–1067.

[Bee87] C. Beeri et al. Sets and negation in a logical database language (LDL1). In *Proceedings of 6th Symposium on Principles of database systems*, ACM SIGMOD-SIGACT-SIGART, San Diego (California), March 1987, pp. 21–37.

[BFKM85] L. Brownston, R. Farrel, E. Kant, and N. Martin. *Programming expert systems in OPS5 : An introduction to ruled-based programming.* Wokingham (England) : Addison-Wesley, 1985, 471 pages.

[Bid86] N. Bidoit. Efficient evaluation of relational queries using nested relations. Le Chesnay (France) : INRIA, 1986, 55 pages, Technical Report 480.

[BK89] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *Journal of Computer System Sciences*, 1989, vol. 38, no. 2, pp. 326–340.

[BKKK87] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *International conference on Management of Data*, ACM SIGMOD, San Francisco (california), May 1987, vol. 16, no. 3, pp. 311–322.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, December 1986, vol. 18, no. 4, pp. 323–364.

[BM92] S. Boyera and S. Maria. Une interface graphique pour DRIVER. Sophia-Antipolis (France) : ESSTIN, Juin 1992, 13 pages. Rapport de stage - rapport technique.

[Bor88] H. Boral. Parallelism in BUBBA. In *Proceedings of International symposium on databases in parallel and distributed systems*, IEEE, Austin (Texas), December 1988, pp. 68–71.

[BR84] M. L. Brodie and D. Ridjanovic. On the design and specification of database transactions. In M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, editors, *On conceptual modelling*, Perspectives from Artificial Intelligence, Databases and Programming Languages, Berlin (Germany) : Springer, 1984, pp. 277–306.

[Bro88] M. L. Brodie. Future intelligent information systems : AI and database technologies working together. In M. L. Brodie and J. Mylopoulos, editors, *Readings in Artificial Intelligence and Databases*, San Mateo (California) : Morgan Kaufmann, 1988, pp. 623–644.

[BRS82] F. Bancilhon, P. Richard, and M. Scholl. On line processing of compacted relations. In *Proceedings of 8th Very Large Databases conference*, Mexico (Mexique), December 1982, pp. 263–269.

[But87] M. H. Butler. *Persistent LISP : Storing Interobject References in a Database.* PhD thesis : Berkeley, University of California, November 1987, 142 pages.

[BV92] E. Borelli-Vittori. *Modèle de données B-rel et approche logique B-log.* Thèse de doctorat, Université de Nice - Sophia-Antipolis (France), Février 1992, 227 pages.

[CAC+84] W. P. Cockshott, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. POMS : A Persistent Object Management System. *Software Practice and Experience*, January 1984, vol. 14, no. 1, pp. 49–71.

[Car87] M. Caruso. A conceptual overview of the VISION object-oriented database management system. In *Workshop on Database Programming Languages*, Roscoff (France), September 1987, pp. 67–70.

[CCRG+88] S. Ceri, S. Crespi-Reghizzi, G. Gottlob, F. Lamperti, L. Lavazza, L. Tanca, and R. Zicari. The ALGRES project. In *International Conference on Extending Database Technology*, in Lecture Notes in Computer Science, vol. 303, Venice (Italy) : Springer, 1988, pp. 551–555.

[CCRZ+90] S. Ceri, S. Crespi-Reghizzi, R. Zicari, F. Lamperti, and L. A. Lavazza. ALGRES : An advanced database system for complex applications. *IEEE Software*, July 1990, vol. 7, no. 4, pp. 68–78.

[CD88] F. Cuppens and R. Demolombe. A prolog-relational DBMS interface using delayed evaluation. In *Proceedings of the 3rd International Conference on Data and Knowledge bases : Improving usability and responsiveness*, Jerusalem (Israel), June 1988, pp. 135–148.

[CDF+86] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. E. Richardson, and E. J. Shekita. The architecture of EXODUS extensible DBMS. In *Proceedings of the International workshop on object-oriented database systems*, Pacific Grove (California), September 1986, pp. 52–65.

[CDRS86] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th Very Large Databases conference*, Kyoto (Japan), August 1986, pp. 91–100.

[CDV88] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In *Proceedings of the SIGMOD international conference on Management of Data*, ACM SIGMOD, Chicago (Illinois), June 1988, vol. 17, pp. 413–423.

[CFK+86] P. Ciscia, P. Franceschi, J. Kouloumdjian, G. Levi, G. H. Moll, C. Simonelli, G. Sardu, and L. Torre. The EPSILON knowledge base management system : Architecture and database access optimization. In *Proceedings of the Workshop on Integration of logic programming and databases*, Venice (Italy), December 1986.

[CGK+90] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering*, March 1990, pp. 76–90.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, March 1989, vol. 1, no. 1, pp. 146–166.

[Cha89] J. Chailloux. *LE_LISP de l'INRIA Version 15.22, Manuel de référence.* Le Chesnay (France) : INRIA, Janvier 1989, 430 pages.

[Che76] P. P. Chen. The entity-relationship model – towards a unified view of data. *ACM Transactions on database systems*, March 1976, vol. 1, no. 1, pp. 9–36.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the ACM SIGMOD conference*, ACM SIGMOD record, Boston (Massachusetts), June 1984, vol. 14, no. 2, pp. 316–325.

[Cod70] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 1970, vol. 13, no. 6, pp. 377–387.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on database systems*, December 1979, vol. 4, no. 4, pp. 397–434.

[Cod90] E. F. Codd. *The Relational model for database management : version 2.* Wokingham (England) : Addison-Wesley, 1990, 538 pages.

[CW84] C. L. Chang and A. Walker. PROSQL : A prolog programming interface with SQL/DS. San Jose (California) : IBM Research Laboratory, May 1984, 23 pages, Technical Report RJ 4314.

[DDGO87] U. Dayal, M. DeWitt, D. Goldhirsch, and J. Orenstein. PROBE final report. Computer Corporation of America, 1987, Technical Report CCA-87-02.

[DKA+86] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch. A DBMS prototype to support extended NF2 relations : An integrated view on flat tables and hierarchies. In *International conference on Management of Data*, SIGMOD record, Washington D. C., 1986, vol. 15, no. 2, pp. 356–366.

[DKL85] N. Derret, W. Kent, and P. Lyngbaek. Some aspects of operations in an object-oriented database. *IEEE Data Engineering*, December 1985, vol. 8, no. 4, pp. 302–310.

[DPS86] U. Deppisch, H. B. Paul, and H. J. Schek. A storage system for complex objects. In *Proceedings of the 1st international workshop on Object-Oriented Database Systems*, Asilomar (California), 1986, pp. 183–195.

[DRV86] E. Denoël, D. Roelants, and M. Vauclair. Query translation for coupling prolog with a relational database management system. In *ESPRIT'86*, Brussels (Belgique), September–October 1986, pp. 595–605.

[EMS88] J. Eliot, B. Moss, and S. Sinosky. Managing persistent data with Mneme : designing a reliable, shared object interface. In *Proceedings of the 2nd Object-oriented database systems workshop*, in Lecture Notes in Computer Science, vol. 334, Bad Munster Am Stein (Germany) : Springer, September 1988, pp. 298–316.

[FBC+87] D. Fishman, D. Beech, H. P. Cate, E.-C. Chow, T. Conners, J. W. Davis, N. Denett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. IRIS: an object-oriented database management

system. *ACM Transactions on Office and Information Systems*, January 1987, vol. 5, no. 1, pp. 46–69.

[FK85] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 1985, vol. 28, no. 9, pp. 904–920.

[FT83] P. Fischer and S. Thomas. Operators for non first normal form relations. In *7th International computer software and applications conference*, IEEE, Chicago (Illinois), November 1983, pp. 464–475.

[Fuc84] K. Fuchi. Revisiting original philosophy of fifth generation computer project. In *Proceedings of the International conference on 5th Generation computer systems*, Tokyo (Japan), November 1984, pp. 1–2.

[GMN84] H. Gallaire, J. Minker, and J.-M. Nicolas. Logic and databases : a deductive approach. *ACM Computing Surveys*, June 1984, vol. 16, no. 2, pp. 153–185.

[Gra86a] *G-BASE version 3.0, Reference guide*, Compiègne (France) : Graphael S.A., 1986, 56 pages.

[Gra86b] *G-BASE version 3.0, Introductory guide*, Compiègne (France) : Graphael S.A., 1986, 30 pages.

[GV90] G. Gardarin and P. Valduriez. *SGBD AVANCÉS Bases de données objets, déductives, réparties.* Paris (France) : Eyrolles, 1990, 255 pages.

[HK87a] S. E. Hudson and R. King. Object-oriented database support for software environment. In *International conference on Management of Data*, SIGMOD record, San Fransisco (California), May 1987, pp. 491–503.

[HK87b] R. Hull and R. King. Semantic database modeling : Survey, applications, and research issues. *ACM Computing Surveys*, September 1987, vol. 19, no. 3, pp. 201–260.

[HL90] D. Hsieh and T. Lunt. A logic to unify semantic network knowledge systems with object-oriented database models. Menlo Park (California) : SRI International, Computer Science Laboratory, December 1990, 48 pages, Technical Report SRI-CSL-90-15.

[HM81] M. Hammer and D. McLeod. Database description with SDM : A semantic database model. *ACM Transactions on database Systems*, September 1981, vol. 6, no. 3, pp. 351–386.

[HY84] R. Hull and C. K. Yap. The Format model. *ACM Transactions on database Systems*, July 1984, vol. 31, no. 3, pp. 518–537.

[HZ87] M. F. Hornick and S. B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office and Information systems*, January 1987, vol. 5, no. 1, pp. 70–95.

[Ito86] H. Itoh. Research and development on knowledge base systems at ICOT. In *Proceedings of the 12th Very Large Databases conference*, Kyoto (Japan), August 1986, pp. 437–445

[JS82] B. Jaeschke and H. J. Schek. Remarks on the algebra of non first normal form relations. In *ACM Symposium on Principles of database systems*, ACM SIGMOD record, Los Angeles (California), March 1982, pp. 124–138.

[KBB+87a] W. Kim, N. Ballou, J. Banerjee, H. T. Chou, J. F. Garza, and D. Woelk. Composite object support in an object-oriented database system. In *Proceedings of the international conference on Object-Oriented Programming Systems, Languages and Applications*, Orlando (Florida), October 1987, ACM SIGPLAN Notices, vol. 22, no. 12, pp. 118–125.

[KBB+87b] W. Kim, N. Ballou, J. Banerjee, H. T. Chou, J. F. Garza, and D. Woelk. Features of the ORION object-oriented database system. Austin (Texas) : Microelectronics and Computer technology Corporation (MCC), September 1987, Technical Report ACA-ST-308-87.

[KBC+88] W. Kim, N. Ballou, H. T. Chou, J. F. Garza, D. Woelk, and J. Banerjee. Integrating an object-oriented programming system with a database system. In *Proceedings of the international conference on Object-Oriented Programming Systems, Languages and Applications*, San Diego (California), September 1988, ACM SIGPLAN Notices, vol. 23, no. 11, pp. 142–152.

[KEE87] IntelliCorp Inc. KEE. KEEconnection : A bridge between databases and knowledge bases. Mountain View (California) : IntelliCorp Inc., 1987, 33 pages, Technical report.

[Kie89] G. Kiernan. *Intégration des types abstraits de données dans un SGBD relationnel déductif.* Thèse de doctorat, Université Pierre et Marie Curie - Paris VI (France), Septembre 1989, 218 pages.

[Kim90] W. Kim. Architectural issues in Orion object-oriented database. *Journal of Object-Oriented Programming*, March-April 1990, vol. 2, no. 6, pp. 29–38.

[Kim92] W. Kim. Introduction to SQL/X. In *Proceedings of Future Databases 1992, 2nd Far-East Workshop on Future Database Systems*, Kyoto (Japan), April 1992, pp. 2–7.

[KM85] R. King and D. McLeod. A unified model and methodology for conceptual database design. *ACM Transactions on Information Systems*, January 1985, vol. 3, no 1, pp. 2–21.

[Kou85] J. Kouloumdjian. The interface between the database management system and the inference machine. ESPRIT Project P-530, November 1985, Technical Report EPSILON 4.

[KP76] L. Kerschberg and J. E. S. Pacheco. A functional database model. Rio de Janeiro (Brazil) : Pontificia University Catolica, 1976, 29 pages, Technical report.

[KSD+90] B. Koch, T. Schunke, A. Dearle, F. Vaughan, C. Marlin, R. Fazakerley, and C. Barter. Cache coherency and storage management in a persistent object management. In *Implementing persistent object bases, principles and practice, Proceedings of the 4th international workshop on persistent object systems*, Martha's Vineyard (Massachusetts), September 1990, pp. 103–113.

[Kup85] G. M. Kuper. *The Logical Data Model : A new approach to database logic.* PhD thesis, Computer Science Department, Stanford University, September 1985, 104 pages.

[KV84] G. M. Kuper and M. Y. Vardi. The Logical Data Model. In *Proceedings of 3th Symposium on Principles of database systems*, ACM SIGACT-SIGMOD, Waterloo (California), April 1984, pp. 86–96.

[LeM90] S. Le Ménec. Interface graphique d'aide à la construction du schéma de correspondances DRIVER. Caen (France) : Université de Caen, Septembre 1990, 96 pages. Rapport de DEA Intelligence Artificielle et Applications.

[LeT86] N. Le Thanh. *Contribution à l'étude de la généralisation et de l'association dans une base de données relationnelle : les iso-dépendances et le modèle B-relationnel.* Thèse de docteur d'État Es-sciences, Université de Nice (France), Avril 1986, 470 pages.

[Leb90a] F. Lebastard. DRIVER : A persistent virtual object layer for reasoning. In *ISMIS-90, 5th International Symposium on Methodologies for Intelligent Systems*, Knoxville (Tennessee), October 1990, pp. 74–81.

[Leb90b] F. Lebastard. DRIVER : A persistent virtual object layer for reasoning. In *AAAI Workshop on Knowledge Base Management Systems*, Boston (Massachusetts), July 1990, multiple p..

[Leb90c] F. Lebastard. DRIVER : Couche objet virtuelle persistante et raisonnement. In *Les systèmes experts et leurs applications*, Avignon (France), Mai 1990, pp. 857–871.

[Leb92a] F. Lebastard. DRIVER : Une couche objet pour les bases de données relationnelles. Sophia-Antipolis (France) : CERMICS-INRIA, Octobre 1992, 19 pages, Technical Report 92-6.

[Leb92b] F. Lebastard. DRIVER version 1.34, manuel de référence. Sophia-Antipolis (France) : CERMICS-INRIA, Octobre 1992, 119 pages, Technical Report 92-7.

[Leb93] F. Lebastard. Bases de données expertes et grandes bases de connaissances : l'État de l'Art. *Génie logiciel & Systèmes experts*, Juin 1993, vol. 31. À paraître.

[Lev92] M. Levene. *The nested Universal Relation Database Model*, *Lecture Notes in Computer Science*, vol. 595. Berlin (Germany) : Springer, 1992, 177 pages.

[LG91] K.-W. Larry Lai and L. Guzenda. How to benchmark an OODBMS. *Journal of Object-Oriented Programming*, July-August 1991, vol. 4, no. 4, pp. 12–15.

[LLOW70] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, October 1970, vol. 34, no. 10, pp. 50–63.

[Man87] F. Manola. PDM : An object-oriented model for PROBE. Computer Corporation of America, September 1987, Technical Report CCA-87-03.

[Mas92] *MASAÏ Version 1.24, Manuel de référence*, Gentilly (France) : Ilog, août 1992, 538 pages.

[MAT92] *MATISSE : Open semantic database - product overview*, Saint-Quentin-en-Yvelines (France) : Intellitic International, 1992, 16 pages.

[MBG+87] J. Mylopoulos, A. Borgida, S. Greenspan, C. Meghini, and B. Nixon. Knowledge representation in the software development process : A case study. University of Toronto (Canada), 1987, 22 pages, Technical report Q/336/K56/1987.

[MD86] F. Manola and U. Dayal. PDM : An object-oriented data model. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove (California), September 1986, pp. 18–25.

[ME89] J. Moss and B. Elliot. The MNEME persistent object store. Amherst (Massachusetts) : University of Massachusetts, October 1989, Technical Report COINS-TR-89-107.

[Mir90] S. Miranda. *L'art des bases de données - Tome 3 - Comprendre et évaluer SQL.* Paris (France) : Eyrolles, 1990, 205 pages.

[Mis84] N. Mishkin. *Managing Permanent objects.* PhD thesis, New Haven (Connecticut) : Yale University, 1984, 90 pages.

[MKM+83] K. Murakami, T. Kakuta, N. Miyazaki, S. Shibayama, and H. Yokota. A relational database machine, first step to knowledge base machine. In *Proceedings of the 10th Annual international conference on Computer architecture*, Stockholm (Sweden), June 1983, pp. 423–425.

[MM91] G. Mopolo-Moke. *NICE-C++ : Une extension C++ pour la programmation persistante à partir d'un serveur de bases d'objets.* Thèse de doctorat, Université de Nice - Sophia-Antipolis (France), Décembre 1991, 295 pages.

[MNS+87] K. Morris, J. Naughton, Y. Saraiya, J. D. Ullman, and A. Van Gelder. YAWN! (Yet Another (W)indow on NAIL!). *Special issue on Databases and Logic, IEEE Data Engineering*, December 1987, vol. 10, no. 4.

[Mol87] G.-H. Moll. *Un langage pivot pour le couplage de Prolog avec des bases de données : formalisation et environnement opérationnel.* Thèse de doctorat, Université Claude Bernard - Lyon I, Octobre 1987, 221 pages.

[MS86] D. Maier and J. Stein. Development of an object-oriented DBMS. In *Proceedings of the international conference on Object-Oriented Programming Systems, Languages and Applications*, Portland (Oregon), September 1986, ACM SIGPLAN Notices, vol. 21, no. 11, pp. 472–482.

[MS87] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In *Research directions in object-oriented programming*, Cambridge (Massachusetts) : MIT Press, 1987, pp. 355–392.

[MS91] L. Marinos and R. A. Smit. From relations to objects : A translation methodology for an object oriented front-end to RDBMS. In *The next generation of information systems : From data to knowledge*, *Lecture Notes in Computer Science*, vol. 611, (Germany) Berlin (Germany) : Springer, 1991, pp. 148–167.

[MUG86] K. Morris, J. D. Ullman, and A. Van Gelder. Design overview of the NAIL! system. In *Proceedings of the 3th International conference on logic programming*, *Lecture Notes in Computer Science*, vol. 225, New-York : Springer, 1986, pp. 554–568.

[MVGP92] S. Miranda, P. Valduriez, G. Gardarin, and E. Penny. Next-generation DBMS. Université de Nice (France) : Support du cours EUROPACE, Mai 1992.

[MW91] P. Moore and A. E. Wade. An approach to standard DDL for OODBMSs. *Computer Standards & Interfaces*, October 1991, vol. 13, nos. 1-3, pp. 139–143.

[Nev86] B. Neveu. Une nouvelle génération de systèmes experts pour les domaines de conception. In *8ème Séminaire Tuniso-Français d'Informatique*, Tunis (Tunisie), mai 1986, p. multiple.

[Nic79] J.-M. Nicolas. *Contribution à l'étude théorique des bases de données - Apports de la logique mathématique.* Thèse de docteur d'État Es-sciences, Toulouse (France) : Université Paul Sabatier, Décembre 1979, 265 pages.

[NT89] S. Naqvi and S. Tsur, editors. *A logical language for data and knowledge bases.* New-York : Computer science press, 1989, 288 pages.

[OBS86] P. O'Brien, B. Bullis, and C. Shaffert. Persistent and shared objects in Trellis/Owl. In *Proceedings of the International workshop on object-oriented database systems*, Pacific Grove (California), September 1986, pp. 113–123.

[Ont86] Ontologic Inc. VBASE functional specifications. Billerica (Massachusetts) : Ontologic publications, 1986, Technical report.

[Ont88] Ontologic Inc. VBASE for object applications. Cambridge (Massachusetts) : Ontologic publications, 1988, Technical report.

[Ont91] Ontos Inc. ONTOS : Présentation générale. Paris (France) : BIM systèmes, Septembre 1991, 9 pages, Technical report.

[PA86] P. Pistor and F. Andersen. Designing a generalized NF2 model with an SQL-type language interface. In *Proceedings of the 12th Very Large Databases conference*, Kyoto (Japan), August 1986, pp. 278–288.

[PD89] P. Pistor and P. Dadam. The advanced information management prototype. In *Nested Relations and Complex Objects in Databases*, *Lecture Notes in Computer Science*, vol. 361, Berlin (Germany) : Springer, 1989, pp. 3–26.

[PK86] W. D. Potter and L. Kerschberg. A unified approach to modeling knowledge and data. In *Proceedings of the conference on knowledge and data*, Albufeira (Portugal), November 1986, pp. 265–291.

[PSS+87] H. B. Paul, H. J. Schek, M. H. Sholl, G. Weikum, and U. Deppisch. Architecture and implementation of Darmstadt Database Kernel System. In *Proceedings of the annual SIGMOD conference*, SIGMOD record, vol. 16, no. 3, San Francisco (California), 1987, pp. 196–207.

[PT86] P. Pistor and R. Traunmueller. A database language for sets, lists and tables. *Information systems*, 1986, vol. 11, no. 4, pp. 323–336.

[PT89] P. Pucheral and J.-M. Thevenin. A graph based data structure for efficient implementation of main memory DBMS's. Le Chesnay (France) : INRIA, February 1989, 24 pages, Research Report 978.

[PTE89] W. D. Potter, R. P. Trueblood, and C. M. Eastman. Hyper-semantic data modeling. *Data & Knowledge Engineering*, 1989, vol. 4, pp. 69–90.

[Puc89] P. Pucheral. *Extensibilité et performance d'un gérant d'objets pour applications bases de données.* Thèse de doctorat, Université Pierre et Marie Curie - Paris VI (France), 1989, 225 pages.

[Rei84] R. Reiter. Towards a logical reconstruction of relational database theory. In M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, editors, *On conceptual modelling*, Perspectives from Artificial Intelligence, Databases and Programming Languages, Berlin (Germany) : Springer, 1984, pp. 191–234.

[RK87] J. Rosenberg and J. L. Keedy. Object management and addressing in the MONADS architecture. In *Proceedings of the International Workshop on Persistent Object Systems*, Appin (Scotland), 1987.

[Ros90] J. Rosenberg. The MONADS architecture : a layered view. In *Implementing persistent object bases, principles and practice, Proceedings of the 4th international workshop on persistent objet systems*, Martha's Vineyard (Massachusetts), September 1990, pp. 215–225.

[RS87] L. A. Rowe and M. R. Stonebraker. The POSTGRES data model. In *Proceedings of 13th Very Large Databases conference*, Brighton (England), September 1987, pp. 83–96.

[Rus89] D. M. Russinoff. PROTEUS : A frame-based nonmonotonic inference system. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented concepts, Databases, and Applications*. New-York : ACM Press, 1989, pp. 127–150.

[SAB+89] M. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Verroust. VERSO : A database machine based on nested relations. In *Nested Relations and Complex Objects in Databases*, *Lecture Notes in Computer Science*, vol. 361, Berlin (Germany) : Springer, 1989, pp. 27–49.

[SCB+86] C. Schaffer, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the 1st international conference on Object-Oriented Programming Systems, Languages and Applications*, Portland (Oregon), September 1986, ACM SIGPLAN Notices, vol. 21, no. 11, pp. 9–16.

[Shi81] D. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on database Systems*, March 1981, vol. 6, no. 1, pp. 140–175.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, October 1991, vol. 34, no. 10, pp. 78–93.

[SKL88] S. Y. W. Su, V. Krishnamurthy, and H. Lam. An object-oriented semantic association model (OSAM*). In *AI in industrial engineering and manufacturing : Theoretical issues and applications*. Norcross (Ga) : American Institute of Industrial Engineers, 1988.

[Sme92] *SMECI Version 1.65, Manuel de référence*, Gentilly (France) : Ilog, Mai 1992, 470 pages.

[SPS87] M. H. Scholl, H. -B. Paul, and H. -J. Schek. Supporting flat relations by a nested relational kernel. In *Proceedings of 13th Very Large Databases conference*, Brighton (England), 1987, pp. 137–146.

[SS89] H. J. Schek and M. H. Scholl. The two roles of nested relations in the DASDBS project. In *Nested Relations and Complex Objects in Databases*,

Lecture Notes in Computer Science, vol. 361, Berlin (Germany) : Springer, 1989, pp. 49–68.

[Sto90] M. Stonebraker. Architectures for dbms-oriented expert systems. In *AAAI Workshop on Knowledge Base Management Systems*, Boston (Massachusetts), July 1990, multiple p..

[Su83] S. Y. W. Su. SAM* : A semantic association model for corporate and scientific statistical databases. *Information Sciences*, 1983, vol. 29, nos. 2-3, pp. 151–199.

[Su86] S. Y. W. Su. Modeling integrated manufacturing data with SAM*. *IEEE Computer Magazine*, January 1986, vol. 19, no. 1, pp. 34–49.

[SZ90] E. Shekita and M. Zwilling. Cricket : a mapped, persistent object store. In *Implementing persistent object bases, principles and practice, Proceedings of the 4th international workshop on persistent objet systems*, Martha's Vineyard (Massachusetts), September 1990, pp. 89–102.

[TYF86] T. J. Teorey, D. Yang, and J. P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, June 1986, vol. 18, no. 2, pp. 197–222.

[TZ84] S. Tsur and C. Zaniolo. An implementation of GEM – supporting a semantic data model on a relational back-end. In *ACM SIGMOD International Conference on the Management of Data*, New-York, 1984, pp. 286–295.

[TZ86] S. Tsur and C. Zaniolo. LDL : A logic-based query language. In *Proceedings of the 12th Very Large Databases conference*, Kyoto (Japan), August 1986, pp. 33–41.

[Ulm86] J. D. Ulmann. Implementation of logical query languages for databases. *ACM Transactions on database Systems*, 1986, vol. 10, no. 3, pp. 289–321.

[WEM79] G. Wiederhold and R. El-Masri. The structural model for database design. In *International conference on Entity-Relationship Approach to Systems Analysis and Design*. Los-Angeles (California) : North-Holland, December 1979, pp. 237–257.

[WW90] I. Williams and M. Wolczko. An object-based memory architecture. In *Implementing persistent object bases, principles and practice, Proceedings of the 4th international workshop on persistent objet systems*, Martha's Vineyard (Massachusetts), September 1990, pp. 114–130.

[WWH87] I. W. Williams, M. I. Wolczko, and T. P. Hopkins. Dynamic grouping in an object oriented virtual memory hierarchy. In *Proceedings of the European conference on object-oriented programming*, *Lecture Notes in Computer Science*, vol. 276, Paris (France) : Springer, June 1987, pp. 79–88.

[Zan83] C. Zaniolo. The database language GEM. In *ACM SIGMOD International Conference on the Management of Data*, New-York, 1983, pp. 207–217.

[Zan85] C. Zaniolo. The representation and deductive retrieval of complex objects. In *Proceedings of the 11th Very Large Databases conference*, Stockholm (Sweden), August 1985, pp. 458–469.

Annexe A

ANNEXES

A.1 ASQUELL

DRIVER (version Le_Lisp) utilise ASQUELL [Asq89] pour accéder aux SGBD relationnels et les interroger.

ASQUELL est une interface virtuelle du langage Le_Lisp aux SGBDR. Cette interface est multi-SGBDR, multi-utilisateurs (multi-bases) et multi-requêtes : elle offre en Le_Lisp des connexions simultanées à des SGBDR différents, comme INGRES et ORACLE, et permet d'établir plusieurs canaux de communication avec chacun d'entre eux. Ces canaux sont appelés des *curseurs*.

Elle repose sur :

- une représentation objet du schéma et du dictionnaire des données de la base à laquelle l'utilisateur est connecté;
- des méthodes définies pour les classes des objets représentant les fonctions de communication avec le SGBDR. Ces méthodes rendent transparente à l'utilisateur la nature du SGBDR utilisé.

Les fonctions de l'interface peuvent être regroupées en deux catégories principales :

- *Les fonctions générales d'accès au SGBD*
  Elles comprennent les fonctions de connexion/déconnexion, de création/suppression de curseurs, de validation/annulation de transaction, de chargement et consultation du schéma des bases connectées.
- *Les fonctions de l'interface SQL*
  Elles permettent d'adresser des requêtes aux bases connectées et de recevoir les réponses dans l'environnement Le_Lisp. Les requêtes doivent leur être fournies sous forme de chaînes de caractères si bien qu'il est nécessaire de connaître SQL pour utiliser ASQUELL.

A.2 Exemple de jointure objet restreinte

La présente annexe propose un exemple de correspondance dans laquelle un champ objet est associé à une jointure objet restreinte.

Considérons les classes `Modèle_voiture` et `Pièce_détachée` suivantes :

```
(driver-defclass Modèle_voiture ()
   (marque atom symbol)
   (type   atom symbol)
   (nom    atom string)
   ...)
```

```
(driver-defclass Pièce_détachée ()
   (nom    atom   symbol)
   (modèle object Modèle_voiture)
   ...)
```

associées respectivement aux relations `model` et `parts` :

model			
type	*mark*	designation	...
456B	Renault	Twingo GTL	...
R338	Renault	Clio 16S	...
456B	Peugeot	106SR	...

parts				
mark	*pid*	pname	type	...
Renault	59843	aile-av-g	456B	...
Renault	98435	aile-av-d	456B	...
Renault	4387	capot-av	R338	...
Peugeot	D7E43	aile-av-g	456B	...

La clé de la relation `model` est composée des attributs `type` et `mark`, celle de la relation `parts`, des attributs `mark` et `pid`. Les correspondances peuvent être les suivantes :

```
(driver-defclassmap Modèle_voiture model
   (fields (marque (model mark)        ())
           (type   (model type)        ())
           (nom    (model designation) ())
           ...))
(driver-defclassmap Pièce_détachée parts
   (letjoins
       ((objoin
            (parts model
             ((lambda (a b c d)
                 (and (eq a c) (eq b d)))
              (parts mark) (parts type) (model mark) (model type)))))
       (fields (nom    (parts pname) ())
               (modèle ()            (objoin))
       ...)))
```

Considérons en particulier la correspondance du champ `modèle` de la classe `Pièce_détachée`, à savoir la jointure objet `objoin` :

```
parts.mark = model.mark and parts.type = model.type
```

Dans cette jointure objet, l'attribut `parts.mark` possède le double rôle que nous avons souligné. C'est un élément de clé de la relation `parts` et il est utilisé comme attribut référentiel dans la jointure objet.

Examinons par exemple l'instance de `Pièce_détachée` qui correspond au premier n-uplet présenté dans la relation `parts` :

Objet de classe `Pièce_détachée`	
nom	aile-av-g
modèle	Object("`Twingo GTL`")
...	...

Cet objet a pour référence "`parts/Renault/59843`", valeur qui a été déterminée quand l'objet est devenu persistant. Une fois la référence fixée, elle ne peut plus être changée et les attributs qui la composent sont tenus de conserver la même valeur. Ainsi, pour l'objet considéré, `mark` et `pid` vaudront toujours "`Renault`" et "`59843`". En conséquence, la valeur du champ `modèle` ne peut plus être quelconque. Le n-uplet d'un objet candidat à être affecté dans le champ `modèle` de l'objet considéré doit vérifier la restriction `model.mark = 'Renault'`. L'objet "`106SR`" correspondant au troisième n-uplet de la relation `model` est par exemple une valeur qui n'est plus admise, l'attribut `model.mark` du n-uplet lui correspondant valant "`Peugeot`". Si on effectuait malgré tout l'affectation, la mise-à-jour du n-uplet "`parts/Renault/59843`" dans la base nécessiterait une modification des attributs référentiels `mark` et `type` (valeurs "`Peugeot`" et "`456B`"), ce qui n'est pas autorisé (pour `mark`).

A.3 L'interface graphique de DRIVER

Une première maquette d'interface graphique a été réalisée en 1990 par [LeM90]. Une autre, complètement nouvelle, a été proposée par S. Boyera et S. Maria au cours de leur stage de fin d'école d'ingénieur [BM92]. C'est cette dernière qui est ici rapidement présentée au travers de ses principaux panneaux.

Enfin, il est utile de signaler que ces interfaces ont été élaborées à l'aide des outils AÏDA [Aid92] et MASAÏ [Mas92] de la société Ilog.

La figure A.1 présente le panneau principal de l'interface. Y sont précisés le nom du schéma de correspondances, le modèle objet à utiliser pour ce schéma et le curseur courant sur le SGBD si la connexion a été réalisée.

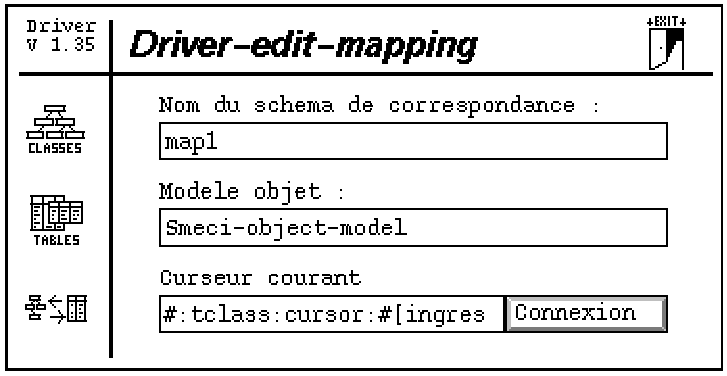

Figure A.1 : Panneau principal de l'interface graphique de DRIVER

Les trois icones sur la gauche permettent d'accéder aux informations sur les classes, les tables et les correspondances.

Si la connexion avec le SGBD n'a pas été établie, elle peut être obtenue en cliquant sur le bouton "Connexion". Le panneau présenté figure A.2 apparaît alors.

L'utilisateur doit choisir un SGBD parmi ceux proposés dans un menu et préciser le

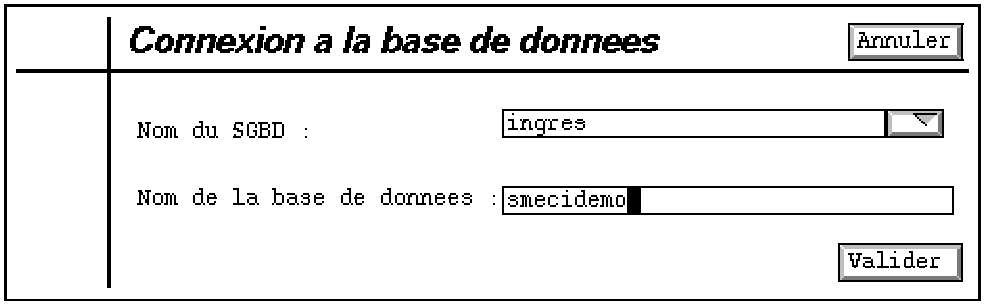

Figure A.2 : Panneau de connexion à la base

nom de la base sur laquelle est défini le schéma de correspondances.

Le panneau des classes est présenté figure A.3. Il permet de définir, éditer et effacer une classe dans le schéma. Le bouton "créer" permet de créer la classe sélectionnée dans le modèle objet client qui a été spécifié dans le panneau principal.

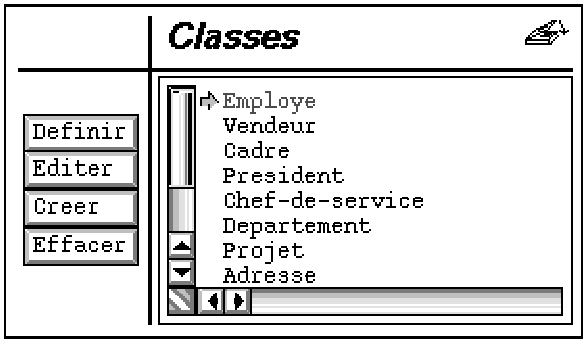

Figure A.3 : Panneau des classes

Le déclenchement d'une définition de classe appelle le panneau présenté figure A.4 où le concepteur du schéma indique le nom de la classe à définir et précise sa classe mère. Ses champs seront décrits lors de l'édition.

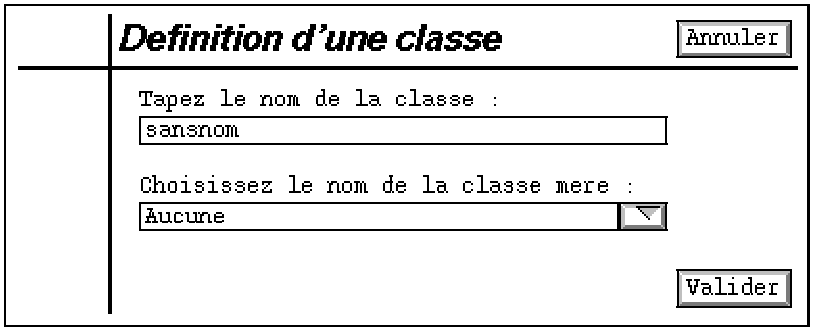

Figure A.4 : Définition d'une classe

La figure A.5 présente l'éditeur de classe. La partie gauche permet d'accéder et d'éditer la classe mère et les classes filles[1] de la classe considérée.

La partie centrale permet de définir, d'éditer et effacer les champs propres à la classe. Il est possible de visualiser et d'éditer (sans toutefois les modifier) les champs hérités des classes ancêtres en plaçant localp à ☐ (au lieu de ☒).

Enfin, la partie droite permet de créer, d'éditer et d'effacer des contraintes de classe sur les champs. On peut également visualiser ici les contraintes héritées des classes ancêtres.

[1] En fait, toutes les classes descendantes de la classe considérée.

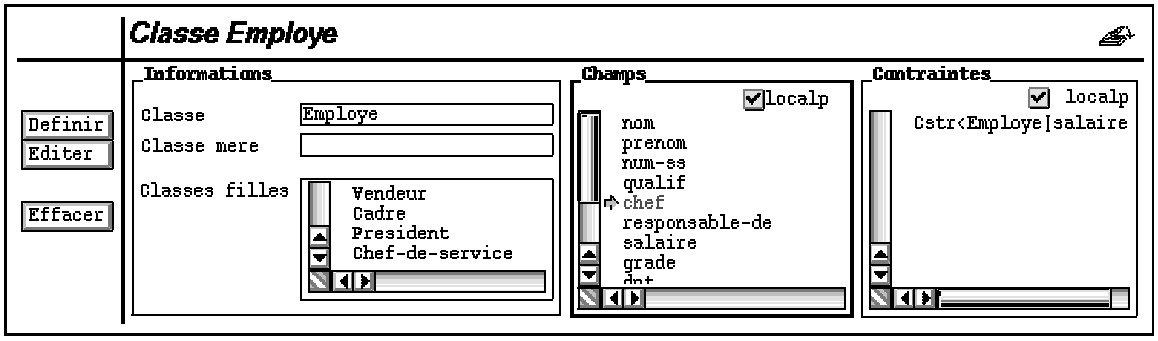

Figure A.5 : Édition d'une classe

L'édition d'un champ provoque l'apparition du panneau présenté figure A.6. Les différentes caractéristiques d'un champ reconnues par DRIVER y sont accessibles : Son nom, la classe à laquelle il appartient, son type, choisi parmi {atom, object, object2, atomset, ordatomset, objectset, ordobjset, object2set, ordobj2set, monotexpr, multitexpr}, son domaine, dont la sémantique est variable selon le type (sous-type pour le type atom, classe pour le type object, etc...), et l'expression permettant de calculer l'éventuelle valeur initiale du champ à affecter à la création d'une nouvelle instance.

Le booléen localp précise si le champ appartient également à la même classe dans le modèle objet client ou s'il est au contraire emprunté à une classe ancêtre. Nous avons vu que ce dernier type de déclaration permettait par exemple de définir une correspondance au champ d'une classe ancêtre non définie dans le schéma de correspondances.

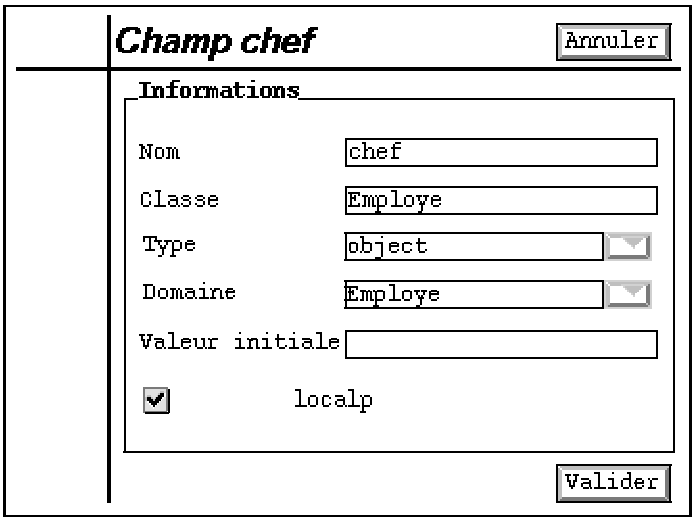

Figure A.6 : Édition d'un champ

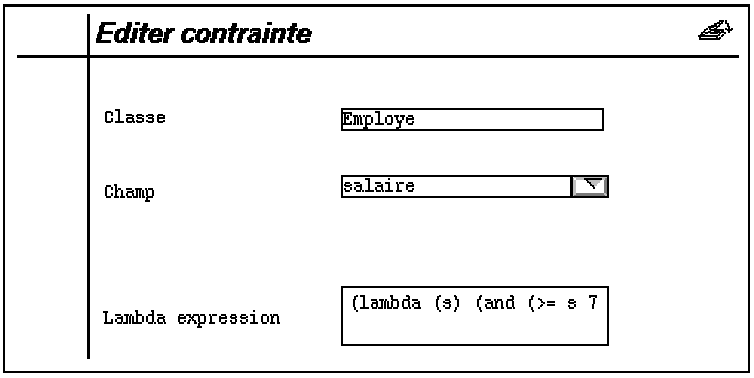

Figure A.7 : Édition d'une contrainte de classe

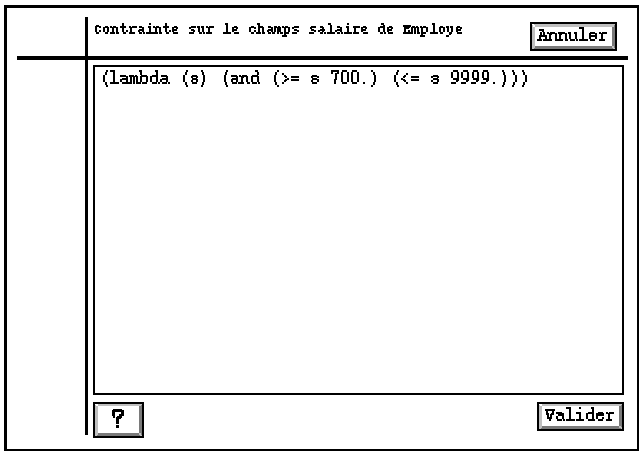

Figure A.8 : Édition de corps de contrainte

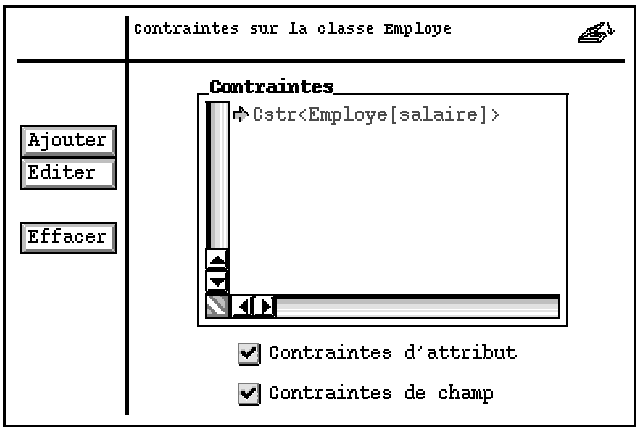

Figure A.9 : Éditeur de contraintes de classe

L'éditeur de classe donne également accès aux éventuelles contraintes définies pour la classe éditée et/ou ses classes ancêtres.

L'édition d'une contrainte provoque l'apparition du panneau présenté figure A.7. La classe et le champ sur lesquels est définie la contrainte sont précisés, suivi de l'expression de la contrainte.

Pour une classe donnée, il est possible de naviguer de contrainte en contrainte en choisissant un nouveau champ dans le sélectionneur prévu à cet effet.

Pour éditer l'expression d'une contrainte, il suffit de cliquer sur la fenêtre où elle apparaît. Ceci appelle alors l'éditeur présenté figure A.8.

L'expression éditée peut alors être sujette à toute modification. Les commandes de l'éditeur proposé sont celles d'*Emacs*; elles sont consultables en cliquant sur le bouton **?**, ce qui provoque l'apparition d'un panneau d'aide.

Après modification, la validation de la nouvelle expression déclenche sa vérification. En cas d'erreur, l'interface indique à l'utilisateur l'origine du problème et le renvoie ensuite à l'édition. Il n'est ainsi possible de quitter cet éditeur que quand tout est en ordre. Au besoin, on peut interrompre et annuler la modification amorcée pour retrouver l'ancienne expression de la contrainte.

Depuis le panneau des correspondances que nous allons voir par la suite, il est également possible d'accéder aux contraintes d'une classe donnée. À partir de ce panneau, on arrive alors sur celui de la figure A.9 qui, cette fois, donne aussi accès aux contraintes de classe directement définies sur les attributs sans correspondance. De ce panneau, il est donc possible de définir des contraintes sur attributs et/ou sur champs atomiques, de les éditer et de les supprimer.

Le panneau des tables est présenté figure A.10. De façon comparable à celui des classes, il permet de définir, éditer et effacer une table dans le schéma de correspondances. Le bouton "créer" permet de créer la table sélectionnée dans la base accessible par le curseur affiché dans le panneau principal.

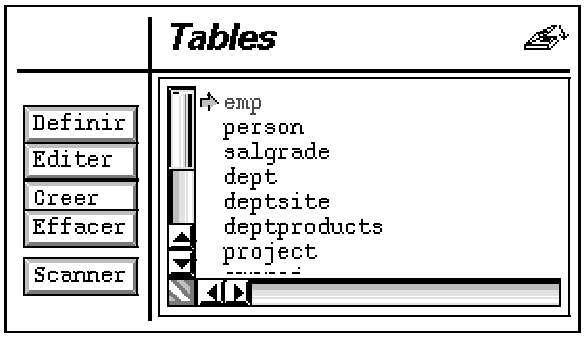

Figure A.10 : Panneau des tables

Le bouton “scanner” permet d’aller reconnaître les tables déjà existantes dans la base et de proposer leur utilisation totale ou partielle dans le schéma. Déclencher l’action correspondante provoque l’interrogation de la base au sujet des structures qu’elle comprend et l’apparition du panneau présenté figure A.11.

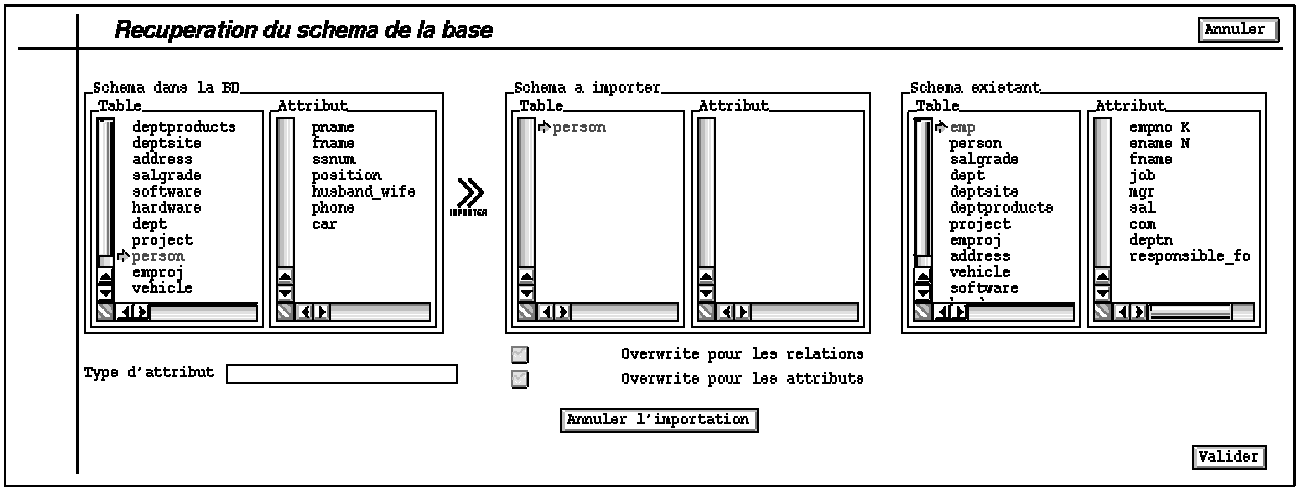

Figure A.11 : Importation du schéma de la base

Ce panneau présente trois ensembles de tables. Le plus à gauche est l’ensemble des structures de la base connectée. Le plus à droite correspond aux tables déjà définies dans le schéma de correspondances courant. Enfin, celui du centre regroupe les tables en cours d’importation dans ce schéma courant.

Les tables et les attributs sont importés de façon indépendante. Ainsi, on peut choisir très précisément les structures relationnelles qu’on souhaite définir dans le schéma de correspondances courant. Si on importe une table ou un attribut qui y existe déjà, l’ancienne définition est écrasée si le booléen overwrite des tables ou des attributs est à $\top$.

Les importations en cours sont disponibles dans le schéma et exploitables immédiatement après la validation. Par contre, tant que cette validation n’a pas encore été faite, toutes peuvent être annulées en actionnant le bouton “Annuler”.

L’éditeur de table est présenté figure A.12. Il permet de mettre à jour le nom de la table et de définir, éditer et effacer ses attributs.

Les noms des attributs étant soumis à une contrainte d’intégrité sont marqués d’un indicateur informant de la nature de cette contrainte :

- K identifie un attribut faisant partie de la clé de la table;
- N identifie un attribut soumis à une contrainte “NOT NULL”;
- U identifie un attribut soumis à une contrainte “UNIQUE”.

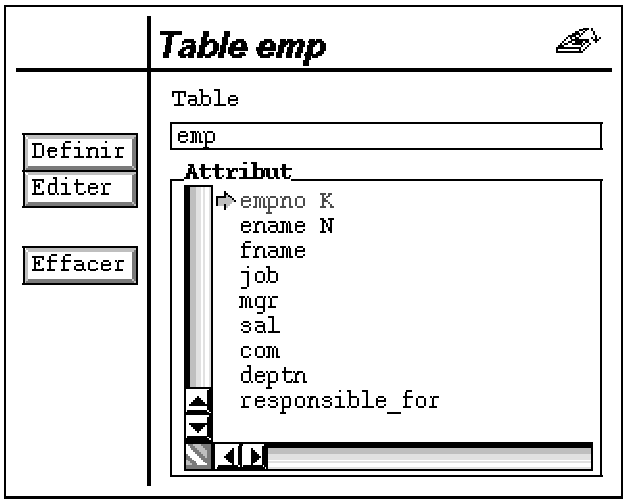

Figure A.12 : Édition d'une table

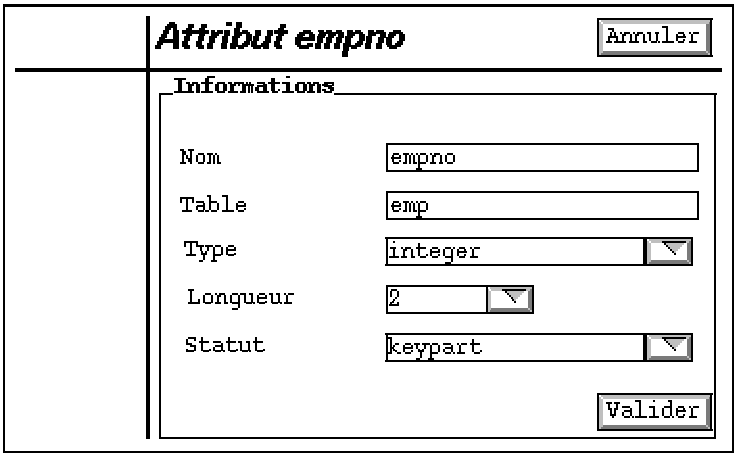

Figure A.13 : Édition d'un attribut

L'éditeur d'attribut est présenté figure A.13. Il visualise le nom de la table à laquelle appartient l'attribut et le nom de l'attribut lui-même et permet la mise-à-jour de ses trois autres caractéristiques qui sont son type, sa taille dans ce type, et son statut.

Les types possibles sont ceux retournés par la fonction driver-attribute-types, à savoir par défaut integer, float et string.

Quatre statuts existent : keypart, not-null, unique et void. Ils indiquent, pour les trois premiers, la nature de la contrainte d'intégrité qui s'applique à l'attribut considéré, et, pour le dernier, l'absence de contrainte sur l'attribut.

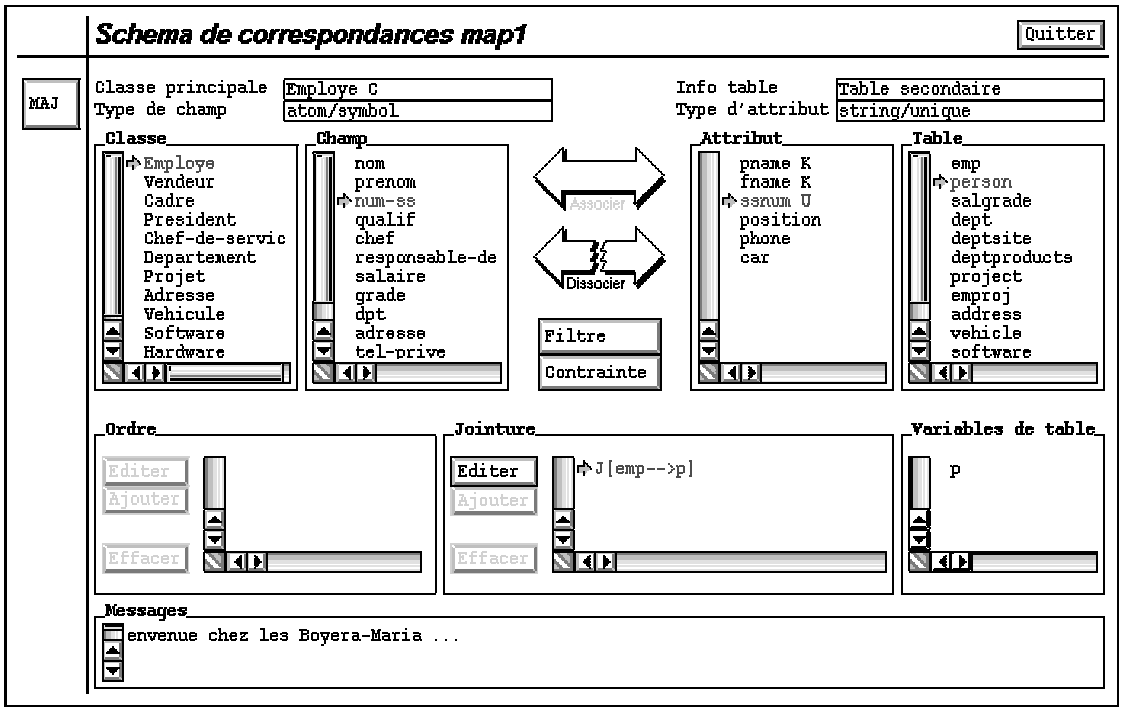

Figure A.14 : Panneau des correspondances (1)

La troisième icone du panneau principal de l'interface graphique donne accès à l'éditeur de correspondances présenté figure A.14.

Dans cet éditeur, les classes et leurs champs apparaissent dans la partie gauche du panneau, les tables et leurs attributs, dans la partie droite. En dessous sont successivement présentés les critères d'ordre participant aux correspondances des champs de type "ensemble ordonné", les jointures et les variables de table.

Le fait de sélectionner une classe, une table, un champ ou un attribut avec la souris affiche automatiquement sa correspondance sur le panneau si elle existe. Dans l'exemple présenté figure A.14, L'utilisateur a d'abord cliqué sur la classe `Employé`, ce qui a sélectionné la table `emp`. Il a ensuite cliqué sur le champ `num-ss`, ce qui a sélectionné la table `person` et l'attribut `ssnum`, affiché la jointure J[`emp`→`p`] et provoqué la mise-à-jour d'un certain nombre d'informations. Au demeurant, le même affichage sur le panneau peut être obtenu en cliquant sur la table `person` puis sur l'attribut `ssnum`.

Quand on sélectionne un élément sans correspondance, les autres scrollers ne réagissent pas. On peut ainsi choisir simultanément des informations dans chacun d'entre eux et créer une correspondance entre elles en cliquant sur le bouton "Associer". Les dissocier s'obtient simplement en sélectionnant l'une d'entre elles et en cliquant sur le bouton "Dissocier".

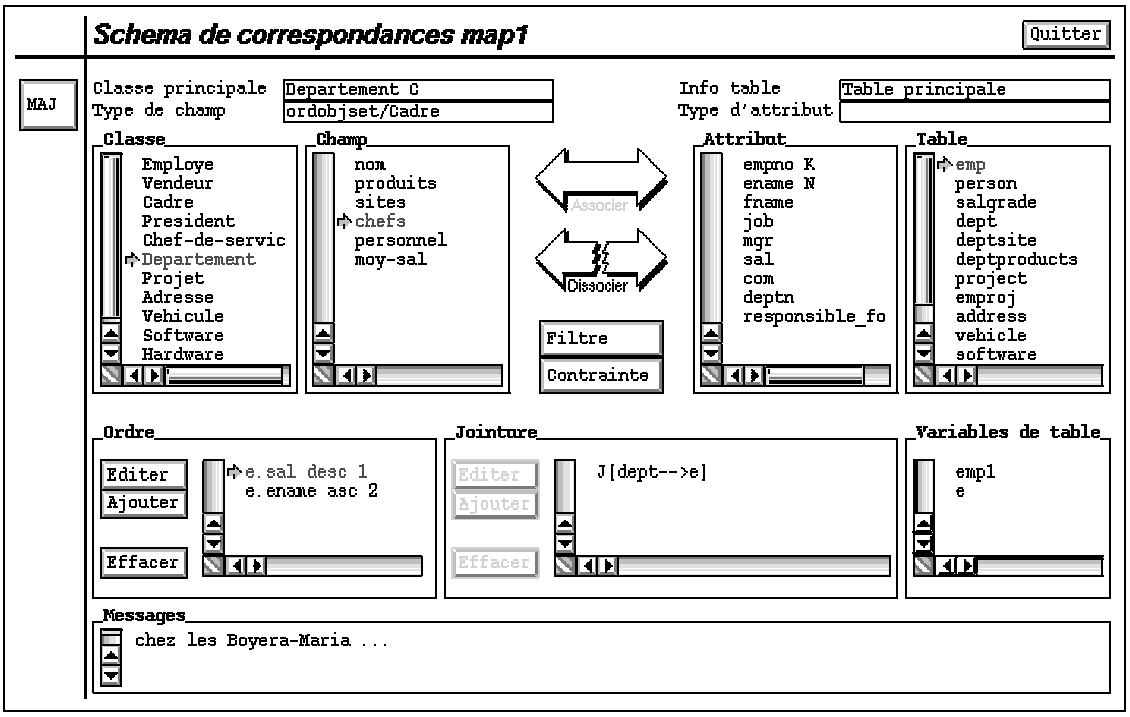

Figure A.15 : Panneau des correspondances (2)

Enfin, ajoutons qu'on peut accéder aux contraintes de classes au moyen du bouton "Contrainte" qui déclenche l'apparition du panneau présenté figure A.9. Par ailleurs, ce panneau donne également accès au gestionnaire de filtres qui est activé par le bouton "Filtre".

La figure A.15 présente un autre exemple d'association au travers de ce même panneau. Cette fois, le champ choisi est le champ `chefs` de la classe `Département`, de type "ensemble ordonné d'objets". La correspondance d'un champ de type "ensemble" étant une chaîne de jointures, aucun attribut n'est sélectionné. Pour les champs objet, la table retenue est la table principale de la classe des objets référencés. Ici, c'est `emp` puisqu'elle est table principale de la classe `Cadre`.

Deux critères d'ordre ont été spécifiés. L'ordre primaire se fait selon les valeurs de l'attribut `sal` par ordre décroissant, et l'ordre secondaire, selon les valeurs de l'attribut `ename` par ordre croissant.
Une demande d'édition d'un ordre appelle l'éditeur présenté figure A.16.

Enfin, une demande d'édition d'une jointure appelle l'éditeur présenté figure A.17.

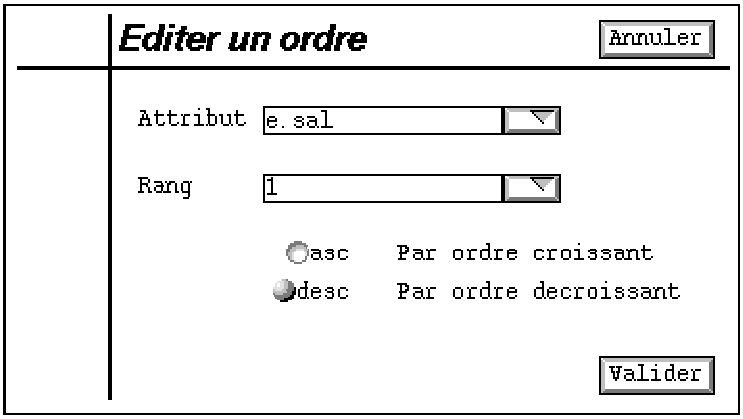

Figure A.16 : Édition d'un critère d'ordre de champ "ensemble"

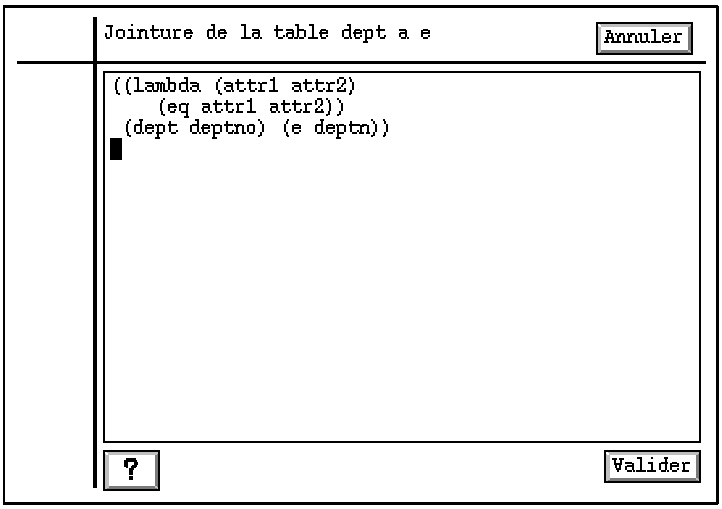

Figure A.17 : Édition d'une jointure

A.4 Conventions utilisées dans l'exposé des algorithmes

Classes
: Variable de nom "Classes" (Nom en italique commençant par une majuscule).

function($Arg_1, \ldots, Arg_N$)
: Fonction de nom "function" à N arguments (Nom en écriture droite suivi d'une liste d'arguments, éventuellement vide).

$Var \leftarrow value$
$Var \leftarrow Var2$
$Var \leftarrow$ function($Arg_1, \ldots, Arg_N$)
: Affectation de la valeur "$value$" (resp. de la valeur référencée par la variable "$Var2$", resp. de la valeur retournée par l'appel fonctionnel function($Arg_1, \ldots, Arg_N$)) à la variable de nom "Var".

  - Si cette affectation est faite dans l'environnement **soit**, elle est précédée de la création de la variable locale de même nom et c'est cette variable locale qui subit l'affectation. Une variable locale peut ainsi très bien masquer une autre variable, locale ou globale. La portée d'une variable locale est la fonction dans laquelle elle est créée.
  - Sinon, l'affectation concerne une variable déjà existante, locale ou globale.

Class::method($Arg_1, \ldots, Arg_N$).*Object*
: Méthode de nom "method" définie pour la classe de nom "Class" (N arguments). L'objet auquelle on applique la méthode est référencé par la variable *Object*.

message($Arg_1, \ldots, Arg_N$).*Object*
: Envoi du message "message" à l'objet référencé par la variable *Object* (N arguments).

field.*Object*
: Accès au champ de nom "field" de l'objet référencé par la variable *Object*.

Annexe B

MANUEL DE RÉFÉRENCE (DRIVER v1.35)

Ce document présente l'interface fonctionnelle de DRIVER version 1.35 :

- Sont présentées dans une première partie toutes les fonctions de construction du schéma de correspondances, de manipulation et de suppression de ses composants. Les trois macros DRIVER-DEFTABLE, DRIVER-DEFCLASS et DRIVER-DEF CLASSMAP sont essentielles pour l'utilisateur débutant. Elles permettent à elles seules la description d'un schéma complet (voir l'exemple de schéma §B.6).
- Sont ensuite présentées toutes les primitives de gestion de la couche objet virtuelle, la connexion aux SGBD, le filtrage d'objets dans la base, le chargement et l'écriture des objets.
- La prise en compte d'un nouveau modèle objet est décrite en §B.4, une illustration complète pour SMECI en §B.8.
- Un exemple de base de données, de schéma de correspondances et de session utilisateur utilisant cette base et ce schéma suivent.
- L'index des erreurs DRIVER est donné en §B.9.
- Enfin, l'index des fonctions et variables est donné en §B.10.

DRIVER **[FEATURE]**

Ce trait indique que le module DRIVER est chargé en mémoire. Si ce n'est pas le cas, on le charge par `^Ldriver`. Le fichier `driver.ll` doit se trouver dans le répertoire courant.

Exemple :

```
? (FEATUREP 'driver)
= ()
? ^Ldriver
Chargement d'ASQUELL...
Chargement d'ASQUELL oK
Chargement de DRIVER...
DRIVER v1.35
Chargement de DRIVER oK
= driver.ll
? (FEATUREP 'driver)
= t
```

B.1 Construction d'un schéma de correspondances

Les fonctions suivantes permettent de construire un schéma de correspondances à partir de la description des éléments connus des représentations relationnelle et objet à associer. Si les deux représentations et leur couplage sont entièrement décrites, le schéma est complet et directement exploitable. Si leur description n'est que partielle, des primitives permettent de générer les éléments manquants et de compléter le schéma.

B.1.1 Création et recherche d'un schéma de correspondances

(DRIVER-CREATE-MAPPING <mapping-name> . <overwritep>) **[SUBR à 1 ou 2 arguments]**

Crée et retourne en valeur un nouveau schéma de correspondances de nom <mapping-name>. Ce schéma doit être complété au moyen des fonctions du paragraphe suivant avant toute utilisation. Si le schéma est le premier créé lors de la session, il devient automatiquement le schéma courant.

Si la variable système DRIVER-AFFECT-CURSOR-P vaut t (valeur par défaut) et si au moins une connexion vers une base a été établie, un curseur est affecté au

nouveau schéma. En cas de redéfinition d'un schéma existant, le curseur courant de ce schéma est préservé.

En cas de tentative de redéfinition d'un schéma, l'erreur "schéma déjà existant" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, l'ancienne définition est perdue, écrasée par la nouvelle, ainsi que toutes les définitions qui y étaient rattachées (sauf son éventuel curseur). Toutes les structures qui la contenaient pointent maintenant sur la nouvelle.

Exemple:

```
? (DRIVER-CREATE-MAPPING 'map1)
= map1
? (DRIVER-CREATE-MAPPING 'map1)
** driver-create-mapping : existent mapping : map1
? (DRIVER-CREATE-MAPPING 'map1 t)
= map1
```

DRIVER-AFFECT-CURSOR-P **[Variable]**

Si une connexion à une base de données a été préalablement établie au moyen de la fonction DRIVER-CONNECT, la valeur de DRIVER-AFFECT-CURSOR-P conditionne l'affectation d'un curseur courant aux schémas de correspondances au moment de leurs créations. Si elle vaut t (valeur par défaut), l'affectation est faite. Le curseur choisi est le curseur courant du schéma courant s'il existe, ou celui retourné par la première exécution de DRIVER-CONNECT dans le cas contraire.

(DRIVER-FIND-MAPPING <mapping-name>) **[SUBR à 1 argument]**

Retourne le schéma de correspondances de nom <mapping-name> s'il existe.

Exemple:

```
? (DRIVER-FIND-MAPPING 'map1)
= map1
? (DRIVER-FIND-MAPPING 'map2)
= ()
```

(DRIVER-MAPPING-P <mapping>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <mapping> si ce dernier est un schéma de correspondances.

Exemple:

```
? (DRIVER-MAPPING-P 'map1)
= ()
? (DRIVER-MAPPING-P (DRIVER-FIND-MAPPING 'map1))
= map1
```

(DRIVER-CURRENT-MAPPING <mapping-name>)
(DRIVER-CURRENT-MAPPING0 <mapping>) [SUBR à 0 ou 1 argument]

Variables-fonctions permettant d'accéder au schéma de correspondances courant si <mapping-name> (resp. l'objet <mapping>) n'est pas fourni, et de le définir dans le cas contraire.

Exemple:

```
? (DRIVER-CURRENT-MAPPING)
= map1       ; initialise par le premier DRIVER-CREATE-MAPPING
? (DRIVER-CREATE-MAPPING 'map2)
= map2
? (DRIVER-CURRENT-MAPPING)
= map1
? (DRIVER-CURRENT-MAPPING 'map2)
= map2
? (DRIVER-CURRENT-MAPPING)
= map2
? (DRIVER-CURRENT-MAPPING0 (DRIVER-FIND-MAPPING 'map1))
= map1
? (DRIVER-CURRENT-MAPPING0)
= map1
```

(DRIVER-ALL-MAPPINGS <all-p>) [SUBR à 0 ou 1 argument]

Sans argument, cette fonction renvoie la liste de tous les schémas de correspondances de l'utilisateur créés depuis le début de la session. Si le paramètre t est précisé, tous les schémas seront renvoyés, y compris les schémas système (le méta-schéma, par exemple).

Exemple:

```
? (DRIVER-ALL-MAPPINGS)
= (map1 map2)
? (DRIVER-ALL-MAPPINGS t)
= (map1 map2)
? (DRIVER-METAMAPPING)
= driver-metamapping
? (DRIVER-ALL-MAPPINGS t)
= (map1 map2 driver-metamapping)
```

(DRIVER-COMPILE-MAPPING <mapping-name>)
(DRIVER-COMPILE-MAPPING0 <mapping>) **[SUBR à 1 argument]**

Compile le schéma de correspondances de nom <mapping-name> (resp. l'objet <mapping>). Cette opération, qui accélère les échanges d'objets entre l'environnement et la base de données, ne doit être faite que quand le schéma est terminé. Toute nouvelle modification du schéma annihile sa compilation.

Exemple:

```
? (DRIVER-COMPILE-MAPPING 'map1)
= t
```

(DRIVER-DELETE-MAPPING <mapping-name>)
(DRIVER-DELETE-MAPPING0 <mapping>) **[SUBR à 1 argument]**

Supprime le schéma de correspondances de nom <mapping-name> (resp. l'objet <mapping>) dans l'environnement. Toutes les données qu'il contient sont également perdues.

Exemple:

```
? (DRIVER-DELETE-MAPPING 'map2)
= t
```

B.1.2 Description de la représentation relationnelle

Les fonctions suivantes s'appliquent toutes au schéma de correspondances courant.

Elles permettent de décrire la représentation relationnelle et de définir dans le schéma des tables et des attributs.

(DRIVER-DEFTABLE <table-name> <attr1> ... <attrN>) **[MACRO]**

définit dans le schéma courant une table de nom <table-name> (symbole), comportant les attributs <attr1> ... <attrN>. L'ordre utilisé pour décrire ces attributs sera repris pour l'éventuelle création de la table considérée. Cette macro crée une table et chacun de ses attributs spécifiés.

<attr1> ... <attrN> décrivent les attributs de la table. Ce sont des listes à 4 éléments ayant la structure suivante :

(<attribute-name> <attr-type> <type-length> <status>)

Le type de l'attribut, la longueur de ce type ainsi que le statut de l'attribut doivent être spécifiés.

Les types d'attribut possibles sont par défaut *integer*, *float* et *string*. Cette liste peut être complétée en fonction de la base de données utilisée et des types qu'elle reconnaît au moyen de la fonction DRIVER-ADD-ATTRIBUTE-TYPE. Elle est consultée à chaque création d'attribut.

Les différents statuts possibles sont :

unique Il ne peut exister deux n-uplets de la table ayant même valeur (différente de *null*) pour l'attribut en question.

not-null L'attribut ne peut valoir *null*. Un nouveau n-uplet qu'on insère dans la table doit nécessairement définir une valeur pour l'attribut en question.

keypart L'attribut fait partie de la clé de la table.

void ou () L'attribut est classique. Il n'a pas de statut particulier.

Si la variable globale DRIVER-TABLE-MAKE vaut t (la valeur par défaut est nil), la fonction DRIVER-MAKE-TABLE0 est appelée et se traduit par la définition dans le SGBD courant de la relation correspondante.

DRIVER-DEFTABLE peut être décrit en lisp de la manière suivante (les fonctions DRIVER-CREATE-TABLE et DRIVER-CREATE-ATTRIBUTE0 sont explicitées ci-après) :

```
(DEFMACRO DRIVER-DEFTABLE (name . attr-list)
    `(LET ((table (DRIVER-CREATE-TABLE ',name t)))
          ,@(MAPCAR (LAMBDA (attr)
                      `(DRIVER-CREATE-ATTRIBUTE0 table
                                                 ,@(MAPCAR 'KWOTE attr)))
                    attr-list)
          (IF DRIVER-TABLE-MAKE
              (DRIVER-MAKE-TABLE0 table))
          table))
```

Exemple:

```
? (DRIVER-DEFTABLE emp
?                  (empno integer  2 keypart)
?                  (ename string  20 not-null)
?                  (fname string  18 ())
?                  (job   string  20 ())
?                  (sal   float    8 ())
?                  (mgr   integer  2 ())
?                  (dpt   integer  2 ()))
= emp
```

```
? (DRIVER-DEFTABLE vehicle
?                  (vnum    integer  2 keypart)
?                  (model   string  15 ())
?                  (licence string  15 unique))
= vehicle
```

DRIVER-TABLE-MAKE **[Variable]**

Si la variable globale DRIVER-TABLE-MAKE vaut t (valeur par défaut), la fonction DRIVER-MAKE-TABLE0 est exécutée à la fin de la déclaration d'une table par DRIVER-DEFTABLE et se traduit par la définition dans le SGBD courant de la relation correspondante.

(DRIVER-CREATE-TABLE <table-name> . <overwritep>)
[SUBR à 1 ou 2 arguments]

Définit la table de nom <table-name> dans le schéma courant. Cette définition est initialement vide d'attribut. Les attributs correspondants devront être créés au moyen de la fonction DRIVER-CREATE-ATTRIBUTE.

En cas de tentative de redéfinition d'une table, l'erreur "table déjà existante" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, l'ancienne définition est perdue, écrasée par la nouvelle, ainsi que toutes les définitions d'attributs qui l'utilisaient. Toutes les structures qui la contenaient pointent maintenant sur la nouvelle.

Exemple:

```
? (DRIVER-CREATE-TABLE 'person)
= person
? (DRIVER-CREATE-TABLE 'address)
= address
? (DRIVER-CREATE-TABLE 'dept)
= dept
? (DRIVER-CREATE-TABLE 'company)
= company
```

(DRIVER-FIND-TABLE <table-name>) **[SUBR à 1 argument]**

Retourne la table de nom <table-name> si elle existe dans le schéma courant.

Exemple:

```
? (DRIVER-FIND-TABLE 'person)
= person
```

(DRIVER-TABLE-P <table>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <table> si ce dernier est une table.

Cette fonction est strictement identique à la fonction DRIVER-TABLE-VAR-P.

Exemple:

```
? (DRIVER-TABLE-P (DRIVER-FIND-TABLE 'person))
= person
```

(DRIVER-DELETE-TABLE <table-name>)
(DRIVER-DELETE-TABLE0 <table>) **[SUBR à 1 argument]**

Détruit la table de nom <table-name> (resp. l'objet <table>) dans le schéma courant (définition et variables). Par effet de bord, les attributs définis dans cette table sont également détruits. De même, toutes les références à la table sont retirées du schéma de correspondances.

Cette fonction est strictement identique à la fonction DRIVER-DELETE-TABLE-DEF.

Exemple:

```
? (DRIVER-DELETE-TABLE 'company)
= t
```

(DRIVER-CREATE-ATTRIBUTE <table-name> <attr-name>
<attr-type> <typ-len> <status> . <overwritep>)

(DRIVER-CREATE-ATTRIBUTE0 <table> <attr-name>
<attr-type> <typ-len> <status> . <overwritep>)
[SUBR à 5 ou 6 arguments]

Définit l'attribut de nom <attr-name> dans la table de nom <table-name> (resp. à l'objet <table>). Le type de l'attribut, la longueur de ce type ainsi que le statut de l'attribut doivent être spécifiés.

Les types d'attribut possibles sont par défaut *integer*, *float* et *string*. Cette liste peut être complétée en fonction de la base de données utilisée et des types qu'elle reconnaît au moyen de la fonction DRIVER-ADD-ATTRIBUTE-TYPE. Elle est consultée à chaque création d'attribut.

Les différents statuts possibles sont :

unique Il ne peut exister deux n-uplets de la table ayant même valeur (différente de *null*) pour l'attribut en question.

not-null L'attribut ne peut valoir *null*. Un nouveau n-uplet qu'on insère dans la table doit nécessairement définir une valeur pour l'attribut en question.

keypart L'attribut fait partie de la clé de la table.

void ou () L'attribut est classique. Il n'a pas de statut particulier.

En cas de tentative de redéfinition d'un attribut, l'erreur "Attribut déjà existant" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, l'ancienne définition est perdue, écrasée par la nouvelle. Toutes les structures qui le contenaient pointent maintenant sur la nouvelle.

Exemple:

```
? (DRIVER-CREATE-ATTRIBUTE 'person 'pname 'string 20 'keypart)
= person.pname
? (DRIVER-CREATE-ATTRIBUTE 'person 'fname 'string 18 'keypart)
= person.fname
? (DRIVER-CREATE-ATTRIBUTEO
?       (DRIVER-FIND-TABLE 'person) 'ssnum 'string 15 'unique)
= person.ssnum
? (DRIVER-CREATE-ATTRIBUTE 'person 'position 'string 10 ())
= person.weight
? (DRIVER-CREATE-ATTRIBUTE 'person 'car 'integer 2 ())
= person.car
? (DRIVER-CREATE-ATTRIBUTE 'address 'empno 'integer 2 'keypart)
= address.empno
? (DRIVER-CREATE-ATTRIBUTE 'address 'state 'string 5 ())
= address.empno
? (DRIVER-CREATE-ATTRIBUTE 'dept 'deptnum 'integer 2 'keypart)
= dept.deptnum
```

(DRIVER-ATTRIBUTE-TYPES) [SUBR sans argument]

Retourne la liste des types d'attribut reconnus par DRIVER.

```
? (DRIVER-ATTRIBUTE-TYPES)
= (integer float string)
```

(DRIVER-ADD-ATTRIBUTE-TYPE <newtype>) [SUBR à 1 argument]

Définit un nouveau type d'attribut. Cette fonction permet de compléter la liste des types utilisables disponibles dans le SGBD utilisé.

ATTENTION : cette fonction ne vérifie pas que le SGBD reconnaît effectivement le type rajouté.

Exemple:

```
? (DRIVER-CREATE-ATTRIBUTE 'vehicle 'vdate 'date () ())
*** driver-create-attribute : Bad attribute type : date
? (DRIVER-ADD-ATTRIBUTE-TYPE 'date)
= date
? (DRIVER-CREATE-ATTRIBUTE 'vehicle 'vdate 'date () ())
= vehicle.vdate
```

(DRIVER-FIND-ATTRIBUTE <table-name> <attr-name>)
(DRIVER-FIND-ATTRIBUTE0 <table> <attr-name>)
[SUBR à 2 arguments]

Retourne l'attribut de nom <attr-name> de la table de nom <table-name> (de la table <table>) si elle existe.

La fonction DRIVER-FIND-ATTRIBUTE est strictement identique à la fonction DRIVER-FIND-ATTRIBUTE-VAR.

Exemple:

```
? (DRIVER-FIND-ATTRIBUTE 'emp 'empno)
= emp.empno
? (DRIVER-FIND-ATTRIBUTE 'address 'empno)
= address.empno
```

(DRIVER-ATTRIBUTE-P <attr>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <attr> si ce dernier est un attribut.

Cette fonction est strictement identique à la fonction DRIVER-ATTRIBUTE-VAR-P.

Exemple:

```
? (DRIVER-ATTRIBUTE-P (DRIVER-FIND-ATTRIBUTE 'emp 'empno))
= emp.empno
```

(DRIVER-DELETE-ATTRIBUTE <table-name> <attr-name>)
[SUBR à 2 arguments]

(DRIVER-DELETE-ATTRIBUTE0 <attr>) **[SUBR à 1 argument]**

Détruit l'attribut de nom <attr-name> dans la table de nom <table-name> (resp. l'objet <attr>) dans le schéma courant (définition et variables). Toutes les références à l'attribut sont retirées du schéma de correspondances.

La fonction DRIVER-DELETE-ATTRIBUTE est strictement identique à la fonction DRIVER-DELETE-ATTRIBUTE-DEF.

Exemple:

```
? (DRIVER-DELETE-ATTRIBUTE 'vehicle 'vdate)
= t
```

(DRIVER-TABLE-ATTRIBUTES <table-name>)
(DRIVER-TABLE-ATTRIBUTES0 <table>) **[SUBR à 1 argument]**

Renvoie dans l'ordre de leurs créations les attributs associés à une table. Ce même ordre est utilisé pour une éventuelle création de la relation correspondante dans la base de données relationnelle.

Les fonctions DRIVER-TABLE-ATTRIBUTES[0] sont strictement identiques aux fonctions DRIVER-TABLEVAR-ATTRVARS[0].

Exemple:

```
? (DRIVER-TABLE-ATTRIBUTES 'person)
= (person.pname person.fname person.ssnum person.position person.car)
```

B.1.3 Description de la représentation objet

Les fonctions suivantes s'appliquent toutes au schéma de correspondances courant.

Elles permettent de décrire la représentation objet et d'ajouter au schéma des définitions de classes et de champs.

(DRIVER-DEFCLASS <class-name> <super-class-name>
<field1> ... <fieldN>)
[MACRO]

définit dans le schéma courant une classe de nom <class-name>, sous-classe de la classe de nom <super-class-name> (seul l'héritage simple est autorisé), comportant les champs <field1> ... <fieldN>. L'ordre utilisé pour décrire ces champs sera repris pour l'éventuelle création de la classe considérée.

<field1> ... <fieldN> décrivent les champs de la classe. Ce sont des listes d'au moins 3 éléments ayant la structure suivante :

(<field-name> <field-type> <specific-type-info> . <options>)

Le type du champ doit être un symbole appartenant à la liste des types possibles : (atom atomset ordatomset object object2 objectset ordobjset object2set ordobj2set monotexpr multitexpr). L'élément <specific-type-info> possède une sémantique qui varie en fonction du type du champ. (voir la fonction DRIVER-CREATE-FIELD).

Une option se présente sous la forme d'une liste à au moins deux éléments dont le premier est le nom de l'option et le ou les suivants donnent son expression ou sa valeur. Elles permettent par exemple de préciser comment calculer la valeur initiale

du champ à la création de l'instance, ou encore, de contraindre un champ atomique. Voir la fonction DRIVER-CREATE-FIELD.

Il est également possible de contraindre un champ atomique ou de renforcer une contrainte posée sur un champ atomique défini dans une classe mère. <fieldM> est alors une liste à deux éléments dont le premier est le nom du champ et le second l'option contrainte, à savoir la liste composée du symbole *constraint* et de l'expression de la contrainte (lambda) telle qu'elle est spécifiée dans la description de la fonction DRIVER-CREATE-FIELD-CONSTRAINT.

Si la variable globale DRIVER-CLASS-MAKE vaut t (valeur par défaut), la fonction DRIVER-MAKE-CLASS0 est appelée et se traduit par la définition dans le langage objet client de la classe correspondante.

DRIVER-DEFCLASS peut être décrit en lisp de la manière suivante (les fonctions DRIVER-CREATE-CLASS, DRIVER-CREATE-FIELD0 et DRIVER-CREATE-CLASS-CONSTRAINT sont explicitées ci-après) :

```
(DEFMACRO DRIVER-DEFCLASS (name super-class-name . field-list)
    `(LET ((class (DRIVER-CREATE-CLASS ',name
                                       ',super-class-name t)))
          ,@(MAPCAR
               (LAMBDA (field)
                  (IF (AND (CONSP (CADR field))
                           (EQ (CAADR field) 'constraint))
                      `(DRIVER-CREATE-FIELD-CONSTRAINT ',name
                                ',(CAR field) ',(CADADR field))
                      `(DRIVER-CREATE-FIELD0 class
                                ,@(MAPCAR 'KWOTE field))))
               field-list)
          (IF DRIVER-CLASS-MAKE
              (DRIVER-MAKE-CLASS0 class))
          class))
```

Exemple:

```
? (DRIVER-DEFCLASS Employe ()
?            (nom      atom   symbol)
?            (prenom   atom   symbol)
?            (num-ss   atom   string)
?            (qualif   atom   symbol)
?            (chef     object Employe)
?            (salaire  atom   float)
?            (grade    atom   fix (initval 0))
?            (adresse  object Adresse)
?            (vehicule object Vehicule))
```

```
= Employe
? (DRIVER-DEFCLASS Vendeur Employe
?            (qualif (constraint
?                          (LAMBDA (qual) (EQ qual 'salesman))))
?            (situ-famille atom      symbol)
?            (commission   atom      float)
?            (sal-total    monotexpr float))
= Vendeur
? (DRIVER-DEFCLASS Departement ()
?            (nom       atom symbol)
?            (site      atom symbol (initval 'New-York)
?                       (constraint (LAMBDA (s) (MEMQ s
?                              '(New-York Washington Boston)))))
?            (chefs     ordobjset Employe)
?            (personnel objectset Employe)
?            (moy-sal   multitexpr float))
= Departement
```

DRIVER-CLASS-MAKE **[Variable]**

Si la variable globale DRIVER-CLASS-MAKE vaut t (valeur par défaut), la fonction DRIVER-MAKE-CLASS0 est exécutée à la fin de la déclaration d'une classe par DRIVER-DEFCLASS et se traduit par la définition dans le langage objet client de la classe correspondante.

(DRIVER-CREATE-CLASS <class-name>
<super-class-name> . <overwritep>)
[SUBR à 2 ou 3 arguments]

Ajoute la description de la classe de nom <class-name>, sous-classe de la classe de nom <super-class-name>, dans le schéma courant. Cette description est initialement vide de champ. Les champs correspondants devront être créés au moyen de la fonction DRIVER-CREATE-FIELD. En cas de tentative de redéfinition d'une classe, l'erreur "classe déjà existante" est provoquée, sauf si <overwritep> est précisé à t. Dans ce cas, l'ancienne définition est perdue, écrasée par la nouvelle, ainsi que toutes les informations qu'elle contenait. Toutes les structures qui l'utilisaient pointent maintenant sur la nouvelle.

Exemple:

```
? (DRIVER-CREATE-CLASS 'Adresse ())
= Adresse
? (DRIVER-CREATE-CLASS 'Vendeur-informatique 'Vendeur)
= Vendeur-informatique
```

(DRIVER-FIND-CLASS <class-name>) **[SUBR à 1 argument]**

Retourne la description de classe de nom <class-name> si elle existe dans le schéma courant.

Exemple:

```
? (DRIVER-FIND-CLASS 'Employe)
= Employe
? (DRIVER-FIND-CLASS 'Projet)
= ()
```

(DRIVER-FIND-MAIN-CLASS <class-name>)
(DRIVER-FIND-MAIN-CLASS0 <class>) **[SUBR à 1 argument]**

Renvoie la classe principale de la classe de nom <class-name> (de l'objet <class>). Une classe principale est une classe qui n'a pas de classe mère décrite dans le schéma.

Exemple:

```
? (DRIVER-FIND-MAIN-CLASS 'Vendeur-informatique)
= Employe
```

(DRIVER-MAIN-CLASSES) **[SUBR sans argument]**

Retourne l'ensemble des classes principales décrites dans le schéma de correspondances courant.

Exemple:

```
? (DRIVER-MAIN-CLASSES)
= (Employe Departement Adresse)
```

(DRIVER-CLASS-SUBCLASSES <class-name>)
(DRIVER-CLASS-SUBCLASSES0 <class>)
(DRIVER-DIRECT-SUBCLASSES <class-name>)
(DRIVER-DIRECT-SUBCLASSES0 <class>) **[SUBR à 1 argument]**

Les fonctions DRIVER-CLASS-SUBCLASSES(0) renvoient toutes les sous-classes de la classe de nom <class-name> (resp. l'objet <class>); les fonctions DRIVER-DIRECT-SUBCLASSES(0) ne renvoient que les sous-classes directes.

Exemple:

```
? (DRIVER-CLASS-SUBCLASSES 'Employe)
= (Vendeur Vendeur-informatique)
? (DRIVER-DIRECT-SUBCLASSES 'Employe)
= (Vendeur)
```

(DRIVER-CLASS-P <class>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <class> si ce dernier est une description de classe.

Exemple:

```
? (DRIVER-CLASS-P (DRIVER-FIND-CLASS 'Employe))
= Employe
```

(DRIVER-DELETE-CLASS <class-name>)
(DRIVER-DELETE-CLASS0 <class>) **[SUBR à 1 argument]**

Détruit la description de classe de nom <class-name> (resp. l'objet <class>) dans le schéma courant. Par effet de bord, les champs définis dans cette classe sont également détruits. De même, toutes les références à la classe sont retirées du schéma de correspondances.

Exemple:

```
? (DRIVER-DELETE-CLASS 'Vendeur-informatique)
= t
```

(DRIVER-CREATE-FIELD <class-name> <field-name> <ftype>
<options> . <overwritep>)

(DRIVER-CREATE-FIELD0 <class> <field-name> <ftype>
<options> . <overwritep>)
[SUBR de 3 à 6 arguments]

Ajoute la description du champ de nom <field-name> à la classe de nom <class-name> (resp. l'objet <class>) dans le schéma courant. Le type du champ doit être spécifié ainsi que certaines informations propres à chaque type. L'ensemble des types possibles est (atom atomset ordatomset object object2 objectset ordobjset object2set ordobj2set monotexpr multitexpr) :

atom Type *atomique*. Ce type caractérise le champ atomique destiné à contenir une information élémentaire. Il doit être précisé immédiatement après dans la liste des paramètres par un sous-type. Six valeurs existent par défaut; elles sont `symbol`, `string`, `fix` (ou `integer`), `float` et `()` (divers ou inconnu). Cette liste peut être complétée en fonction du modèle objet utilisé et des types qu'il reconnaît au moyen de la fonction DRIVER-ADD-ATOMIC-FIELD-TYPE. Elle est consultée à chaque création de champ atomique.

Des paramètres optionnels peuvent être précisés. Ils se présentent sous forme de listes à au moins deux éléments dont le premier est le nom de l'option et le ou les suivants donnent son expression ou sa valeur. Ces options sont :

initval Ce mot-clé introduit une forme Lisp qui va permettre de calculer la valeur initiale du champ. La forme est évaluée lors de la création de chaque instance. Attention donc de bien "coter" les constantes symboliques ou les listes qui ne sont pas des appels fonctionnels. Cette option peut être utilisée avec tous les types de champ.

constraint Cet autre mot-clé introduit une forme Lisp qui va permettre de contraindre les champs de type atomique (et uniquement les champs de ce type). L'expression est une lambda à un paramètre qui sera évaluée pour toute valeur candidate du champ (voir la description de DRIVER-CREATE-FIELD-CONSTRAINT).

localp Ce mot-clé permet de déclarer le champ comme ne faisant pas partie en fait de la classe <class-name>, mais d'une de ses classes mère. Il ne présente donc à proprement parler aucun intérêt pour la déclaration de la classe. Il permet simplement de signaler l'existence du champ <field-name> plus "haut" dans la hiérarchie, de façon à pouvoir lui définir une correspondance relationnelle au niveau de la classe <class-name> au moyen des fonctions DRIVER-DEFCLASSMAP et DRIVER-MAP-FIELD pour le cas où la classe mère le possédant ne serait pas persistante. Par défaut, un champ est considéré comme local.

atomset Type *ensemble d'atomes*. Ce type doit également être immédiatement précisé d'un argument qui définit lui le type de l'élément de l'ensemble. Les remarques effectuées pour le type atom restent valables.

ordatomset Type *ensemble ordonné d'atomes*. Ce type se rapproche du précédent. Néanmoins, l'ordre des éléments est ici important, ce qui n'était pas le cas pour le type atomset.

object Type *objet*. Ce type caractérise les champs destinés à recevoir un objet. Il doit être précisé immédiatement après dans la liste des paramètres par le nom de la classe des objets à contenir.

object2 Type *objet 2*. Il peut arriver qu'un champ objet soit amené à contenir des objets dont on ne puisse pas à l'avance connaître la classe. Cette seconde catégorie de champ objet est prévue pour ceux-là.

objectset Type *ensemble d'objets*. Ce type caractérise les champs destinés à recevoir un ensemble d'objets. Comme pour le type *objet*, il doit être immédiatement suivi par le nom de la classe commune aux objets éléments de l'ensemble.

ordobjset Type *ensemble ordonné d'objets*. Ce type se rapproche du précédent. Néanmoins, l'ordre des éléments est ici important, ce qui n'était pas le cas pour le type objectset.

object2set Type *ensemble d'objets 2*. Ce type caractérise les champs destinés à recevoir un ensemble d'objets de classes différentes et indépendantes (de classe principale différente).

ordobj2set Type *ensemble ordonné d'objets 2*. Ce type se rapproche du précédent. Néanmoins, l'ordre des éléments est ici important, ce qui n'était pas le cas pour le type object2set.

monotexpr Type *expression mono n-uplet*.

multitexpr Type *expression multi n-uplet*.

Ces deux derniers types permettent de caractériser une catégorie particulière de champ atomique. Ils ne peuvent être utilisés avec intérêt que dans la perspective du rapprochement de la représentation objet avec une représentation relationnelle. Les champs correspondants ne peuvent être consultés qu'en lecture. Ils sont destinés à recevoir le produit d'un calcul effectué dans la base de données.

Le premier des deux effectue ce calcul sur un seul n-uplet, associé généralement à l'objet considéré; Somme ou différence d'attributs du n-uplet sont des exemples d'expressions possibles.

Le second effectue le calcul sur un ensemble de n-uplets, correspondant éventuellement à un ensemble d'objets. Ce genre de calcul permet de calculer par exemple une moyenne ou la somme d'un ensemble de valeurs.

En cas de tentative de redéfinition d'un champ, l'erreur "champ déjà existant" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, l'ancienne définition est perdue, écrasée par la nouvelle. Toutes les structures qui la contenaient pointent maintenant sur la nouvelle.

Exemple:

```
? (DRIVER-CREATE-FIELD 'Adresse 'numero 'atom 'fix)
= numero
? (DRIVER-CREATE-FIELD 'Adresse 'ville 'atom 'symbol
?                      '(initval 'New-York))
= ville
? (DRIVER-CREATE-FIELDO
?            (DRIVER-FIND-CLASS 'Adresse) 'etat 'atom 'string
?            '(constraint (LAMBDA (val)
?                           (MEMQ val '(NY MA CA)))))
```

```
= etat
? (DRIVER-CREATE-FIELD 'Employe 'vehicule2 'object 'Vehicule
?                      '(initval (OMAKEQ #:tclass:Vehicule)))
= vehicule2
```

(DRIVER-FIELD-TYPE <class-name> <field-name>)

[SUBR à 2 arguments]

(DRIVER-FIELD-TYPE0 <field>) **[SUBR à 1 argument]**

Retourne le type du champ de nom <field-name> de la classe de nom <class-name> (resp. la classe <class-name>).

Exemple :

```
? (DRIVER-FIELD-TYPE 'Employe 'chef)
= object
```

(DRIVER-ATOMIC-FIELD-TYPES) **[SUBR sans argument]**

Retourne la liste des types atomiques de champ reconnus par DRIVER.

```
? (DRIVER-ATOMIC-FIELD-TYPES)
= (() integer fix float string symbol)
```

(DRIVER-ADD-ATOMIC-FIELD-TYPE <newtype>) **[SUBR à 1 argument]**

Définit un nouveau type de champ atomique. Cette fonction permet de compléter la liste des types utilisables disponibles dans le modèle objet utilisé.

ATTENTION : cette fonction ne vérifie pas que le modèle objet reconnaît effectivement le type rajouté.

Exemple:

```
? (DRIVER-CREATE-FIELD 'Adresse 'rue 'atom 'doublestring)
*** driver-create-field : Bad field subtype : doublestring
? (DRIVER-ADD-ATOMIC-FIELD-TYPE 'doublestring)
= doublestring
? (DRIVER-CREATE-FIELD 'Adresse 'rue 'atom 'doublestring)
= rue
```

(DRIVER-FIND-FIELD <class-name> <field-name>)
(DRIVER-FIND-FIELD0 <class> <field-name>)
(DRIVER-FIND-HERITED-FIELD <class-name> <field-name>)
(DRIVER-FIND-HERITED-FIELD0 <class> <field-name>)
(DRIVER-FIND-FIELD-IN-BRANCH <classname> <fieldname>)
(DRIVER-FIND-FIELD-IN-BRANCH0 <class> <fieldname>)
[SUBR à 2 arguments]

Les fonctions DRIVER-FIND-FIELD(0) retournent la description du champ de nom <field-name> de la classe de nom <class-name> (resp. l'objet <class>) si elle existe.

Les fonctions DRIVER-FIND-HERITED-FIELD(0) retournent la description du champ de nom <field-name> de la classe de nom <class-name> (resp. l'objet <class>) ou de l'une de ses classes mère, si cette description existe.

Les fonctions DRIVER-FIND-FIELD-IN-BRANCH(0) retournent la description du champ de nom <field-name> si elle existe dans la branche de la classe de nom <class-name> (resp. l'objet <class>). On appelle branche d'une classe l'ensemble de ses classes ancêtres jusqu'à la classe principale incluse, plus toutes ses classes descendantes (classes filles).

Exemple:

```
? (DRIVER-FIND-FIELD 'Employe 'nom)
= nom
? (DRIVER-FIND-FIELD 'Vendeur 'qualif)
= ()
? (DRIVER-FIND-HERITED-FIELD 'Vendeur 'qualif)
= qualif
? (DRIVER-FIND-FIELD-IN-BRANCH 'Employe situ-famille)
= situ-famille
```

(DRIVER-FIND-FIELD-CLASS-IN-HIERARCHY <classname> <fieldname>)
(DRIVER-FIND-FIELD-CLASS-IN-HIERARCHY0 <class> <fieldname>)
[SUBR à 2 arguments]

Recherche la classe dans laquelle se trouve le champ de nom <fieldname>. La recherche est effectuée dans la hiérarchie de la classe de nom <classname> (resp. l'objet <class>). Si plusieurs champs ont ce même nom, retourne la première classe qu'elle trouve qui possède un champ de ce nom.

Exemple :

```
? (DRIVER-FIND-FIELD-CLASS-IN-HIERARCHY 'Vendeur 'prenom)
= Employe
```

(DRIVER-FIELD-P <field>)
(DRIVER-ATOM-FIELD-P <field>)
(DRIVER-ATOMSET-FIELD-P <field>)
(DRIVER-ORDATOMSET-FIELD-P <field>)
(DRIVER-OBJECT-FIELD-P <field>)
(DRIVER-OBJECT2-FIELD-P <field>)
(DRIVER-SET-FIELD-P <field>)
(DRIVER-OBJECTSET-FIELD-P <field>)
(DRIVER-OBJECT2-FIELD-P <field>)
(DRIVER-ORDOBJSET-FIELD-P <field>)
(DRIVER-ORDOBJ2SET-FIELD-P <field>)
(DRIVER-MONOTEXPR-FIELD-P <field>)
(DRIVER-MULTITEXPR-FIELD-P <field>) **[SUBR à 1 argument]**

Le prédicat DRIVER-FIELD-P est vrai et renvoie l'objet <field> si ce dernier est un champ. Même chose pour ses pendants, avec un test portant sur les différents types de champs.

Exemple:

```
? (DRIVER-FIELD-P (DRIVER-FIND-FIELD 'Employe 'nom))
= nom
? (DRIVER-SET-FIELD-P (DRIVER-FIND-FIELD 'Employe 'nom))
= ()
```

(DRIVER-DELETE-FIELD <class-name> <field-name>)

[SUBR à 2 arguments]

(DRIVER-DELETE-FIELD0 <field>) **[SUBR à 1 argument]**

Détruit la description de champ de nom <field-name> de la classe de nom <class-name> (resp. l'objet <field>) dans le schéma courant. Par effet de bord, toute correspondance définie sur le champ est également détruite.

Exemple:

```
? (DRIVER-DELETE-FIELD 'Employe 'vehicule2)
= t
```

(DRIVER-CLASS-FIELD-P <class> <field>)
(DRIVER-HERITED-FIELD-P <class> <field>) **[SUBR à 2 arguments]**

Le prédicat DRIVER-CLASS-FIELD-P est vrai et renvoie l'objet <field> si ce dernier est une description de champ de la classe <class>. Le prédicat DRIVER-HERITED-FIELD-P est vrai et renvoie l'objet <field> si ce dernier est une description de champ de la classe <class> ou de l'une de ses classes mère.

Exemple:

```
? (DRIVER-CLASS-FIELD-P (DRIVER-FIND-CLASS 'Employe)
?                       (DRIVER-FIND-FIELD 'Employe 'prenom))
= prenom
? (DRIVER-CLASS-FIELD-P (DRIVER-FIND-CLASS 'Vendeur)
?                       (DRIVER-FIND-FIELD 'Employe 'prenom))
= ()
? (DRIVER-HERITED-FIELD-P (DRIVER-FIND-CLASS 'Vendeur)
?                         (DRIVER-FIND-FIELD 'Employe 'prenom))
= prenom
```

(DRIVER-CREATE-FIELD-CONSTRAINT <class-name>

<field-name> <lambda> . <overwrite>)
[SUBR à 3 ou 4 arguments]

(DRIVER-CREATE-FIELD-CONSTRAINT0 <field>

<lambda> . <overwrite>)
[SUBR à 2 ou 3 arguments]

Cette fonction permet de poser une contrainte simple sur un champ atomique.

Cette contrainte s'exprime sous la forme d'une lambda à un argument qui représente la valeur à affecter au champ. Si l'application de la lambda avec une valeur possible pour le champ renvoie vrai au sens lisp, la contrainte est respectée; elle ne l'est pas dans le cas contraire. L'expression (unique) constituant le corps de la lambda peut faire appel aux primitives lisp suivantes : $and, or, not, =, eq, neq, neqn, null, <>$ $, <, <=, >, >=, memq$ et $nmemq$ $((notmemq))$. Les primitives à deux arguments doivent être utilisées avec la variable comme premier argument et une constante comme second.

L'usage de l'ensemble des primitives lisp de l'interprète n'est pas autorisé car DRIVER devra pouvoir compiler l'expression en SQL.

En cas de tentative de redéfinition d'une contrainte déjà définie sur le champ dans la même classe, l'erreur "contrainte déjà existante" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, l'ancienne définition est perdue et écrasée par la nouvelle. Toutes les descriptions qui l'utilisaient pointent maintenant sur la nouvelle.

Exemple:

```
? (DRIVER-CREATE-FIELD-CONSTRAINT 'Employe 'qualif
?                  '(LAMBDA (value)
?                       (OR (NULL value)
?                           (MEMQ value
                                  '(chairman director salesman
?                                        engineer secretary)))))
= Cstr<Employe[qualif]>
? (DRIVER-CREATE-FIELD-CONSTRAINT 'Employe 'salaire
?                  '(LAMBDA (val)
?                       (AND (>= val 1000.) (< val 10000.))))
= Cstr<Employe[salaire]>
? (DRIVER-CREATE-FIELD-CONSTRAINT 'Vendeur 'commission
?                  '(LAMBDA (comm)
?                       (OR (MEMQ comm '(100. 200. 300. 500.))
?                           (>= comm 800.))))
= Cstr<Vendeur[commission]>
```

(DRIVER-FIND-CLASS-CONSTRAINT <class-name> <field-name>)
(DRIVER-FIND-CLASS-CONSTRAINT0 <class> <field>)

[SUBR à 2 arguments]

Retourne la contrainte définie pour la classe de nom <class-name> (resp. l'objet <class>) sur le champ de nom <field-name> (resp. l'objet <field>) si elle existe.

Complément d'information sur ces deux fonctions dans la partie *Description des correspondances.*

Exemple:

```
? (DRIVER-FIND-CLASS-CONSTRAINT 'Employe 'qualif)
= Cstr<Employe[qualif]>
```

(DRIVER-FIND-FIELD-CONSTRAINTS-ON-CLASS <class-name>)
(DRIVER-FIND-FIELD-CONSTRAINTS-ON-CLASS0 <class>)

[SUBR à 1 argument]

Renvoie l'ensemble des contraintes qui s'appliquent aux champs de la classe de nom <class-name> (de l'objet <class>). Une contrainte définie sur le champ d'une classe masque l'eventuelle contrainte posée sur le même champ dans une classe mère.

Exemple:

```
? (DRIVER-FIND-FIELD-CONSTRAINTS-ON-CLASS 'Vendeur)
= (Cstr<Vendeur[qualif]> Cstr<Vendeur[commission]>
    Cstr<Employe[salaire]>)
```

(DRIVER-CLASS-CONSTRAINT-P <constraint>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <constraint> si ce dernier est une contrainte de classe.

Exemple:

```
? (DRIVER-CLASS-CONSTRAINT-P
?           (DRIVER-FIND-CLASS-CONSTRAINT 'Vendeur 'qualif))
= Cstr<Vendeur[qualif]>
```

(DRIVER-DELETE-CLASS-CONSTRAINT <class-name> <field-name>)
(DRIVER-DELETE-CLASS-CONSTRAINT0 <class> <field>)
[SUBR à 2 arguments]

Supprime la contrainte définie pour la classe de nom <class-name> (resp. l'objet <class>) sur le champ de nom <field-name> (resp. l'objet <field>).

Complément d'information sur ces deux fonctions dans la partie *Description des correspondances*.

Exemple:

```
? (DRIVER-DELETE-CLASS-CONSTRAINT 'Vendeur 'commission)
= t
```

B.1.4 Description des correspondances

Les fonctions suivantes s'appliquent toutes au schéma de correspondances courant.

(DRIVER-DEFCLASSMAP <class-name> [<table-name>]
[<class-options>] [(fields <field1-map> ... <fieldN-map>)])

(DRIVER-DEFCLASSMAP <class-name> [<table-name>]
[<class-options>] [(letjoins (<join1> ... <joinM>)
(fields <field1-map> ... <fieldN-map>))])
[MACRO]

Définit dans le schéma courant une correspondance entre la classe de nom <class-name> et la relation <table-name>. Si la classe <class-name> est la sous-classe d'une classe dont la correspondance a déjà été précisée, <table-name> est optionnel.

Un certain nombre d'options sont disponibles pour préciser la correspondance de la classe. Une option se présente sous la forme d'une liste à au moins deux éléments dont le premier est le nom de l'option et le ou les suivants donnent son expression ou sa valeur. Deux possibilités existent ici; elles sont *mappedp* et *attrconstraint*. Voir la description de la fonction DRIVER-MAP-CLASS.

<field1-map> ... <fieldN-map> décrivent la correspondance des champs de la classe. Ce sont des listes de 3 à 7 éléments ayant la structure suivante :

(<field-name> <field-map> <join-vars> . <options>)

Selon le type du champ (atomique, objet, ...), sa correspondance <field-map> est un attribut, l'expression d'un calcul sur la base (lambda appliquée) ou nil, complété par <join-vars> qui identifient les jointures nécessaires pour retrouver les informations.

<join-vars> est en fait une liste de noms de variables parmi celles définies au sein du *letjoins*. Il est possible de référencer une jointure définie pour la correspondance d'une classe ancêtre au moyen d'une *référence directe de jointure*. On la présente sous forme d'une liste de trois éléments, dont le premier est le nom de la classe pour laquelle la jointure a été définie et les deux autres, la table de départ et la table d'arrivée de la jointure. On ne peut pas référencer une jointure définie pour la correspondance d'une autre classe par le nom de variable qu'on lui avait associé alors car ce dernier n'a plus aucune signification en dehors du letjoins.

L'ensemble des options disponibles pour chaque type de champ est présenté dans la description de la fonction DRIVER-MAP-FIELD.

Quand la correspondance d'un champ comprend au moins une jointure, l'expression (fields <field1-map> ... <fieldN-map>) qui contient cette correspondance doit être incluse dans l'environnement

(letjoins (<join1> ... <joinM>) (fields ...))

qui permet de définir localement des jointures.

<join1> ... <joinM> sont des listes à deux éléments. le premier est un nom de variable (symbole) qui permettra de référencer la jointure dans le corps du (fields ...). Le second est une liste à trois éléments ayant la structure suivante :

(<table1-name> <table2-name> <applied-lambda>)

Pour une plus grande facilité d'usage ou par nécessité (définition d'auto-jointure), il est possible de renommer un nom de table localement à la description de la correspondance d'une classe. Il suffit de remplacer le nom de la table par la liste comprenant le nouveau nom suivi de la table (physique) dont on parle.

Consulter la description de la fonction DRIVER-CREATE-JOIN pour plus d'informations sur la création de jointure.

La description fonctionnelle de cette macro est trop complexe pour être donnée ici. DRIVER-DEFCLASS s'expanse en appels aux fonctions DRIVER-CREATE-TABLE-VAR, DRIVER-CREATE-ATTRIBUTE-VAR, DRIVER-MAP-CLASS, DRIVER-CREATE-ATTR-CONSTRAINT, DRIVER-CREATE-JOIN et DRIVER-MAP-FIELD.

Exemple:

```
? (DRIVER-DEFCLASSMAP Employe emp
?    (attrconstraint ((LAMBDA (no) (>= no 1000))
?                     (emp empno)))
?    (LETJOINS
?        ((j1 (emp (p person)
?              ((LAMBDA (a1 a2 a3 a4)
?                  (AND (EQ a1 a3) (EQ a2 a4)))
?               (emp ename) (emp fname)
?               (p pname) (p fname))))
?         (j2 (emp (emp1 emp)
?              ((LAMBDA (a1 a2) (EQ a1 a2))
?                 (emp mgr) (emp1 empno))))
?         (j3 (emp address
?              ((LAMBDA (a1 a2) (EQ a1 a2))
?               (emp empno) (address empno))))
?         (j4 (p vehicle
?              ((LAMBDA (a1 a2) (EQ a1 a2))
?               (p car) (vehicle vnum)))))
?        (FIELDS (nom       (emp ename)    ())
?                (prenom    (emp fname)    ())
?                (num-ss    (p ssnum)      (j1))
?                (qualif    (emp job)      ())
?                (chef      ()             (j2))
?                (salaire   (emp sal)      ())
?                (adresse   ()             (j3))
?                (vehicule  ()             (j1 j4)
?                                       (filter veh-filtre))))
= t
? (DRIVER-DEFCLASSMAP Vendeur
?    (FIELDS (situ-famille (p position)      ((Employe emp p)))
            (commission   (emp com)          ())
?           (sal-total    ((LAMBDA (a1 a2) (+ a1 a2))
?                          (emp sal) (emp com)) ())))
= t
```

(DRIVER-MAP-CLASS <class-name> [<table-name>]
[<options>] . <overwritep>)

(DRIVER-MAP-CLASS0 <class> [<table>] [<options>] . <overwritep>)
[SUBR de 1 à 4 arguments]

Définit une correspondance entre classe et relation. La classe de nom <class-name> (resp. l'objet <class>) devient la représentation de la relation de nom <table-name>

(resp. l'objet <table>) dans l'environnement objet, et inversement, la relation devient la correspondance relationnelle de la classe qu'on lui associe. Par défaut, tout n-uplet de la relation est un objet potentiel instance de la classe associée et tout objet persistant de la classe est un représentant dans l'environnement d'un n-uplet de la relation associée.

La relation associée à la classe est appelée sa *relation principale*. Une relation peut être choisie comme relation principale d'une classe principale que si elle n'est pas déjà relation principale d'une autre classe principale ou relation secondaire d'une autre classe. Si la classe <class-name> est la sous-classe d'une classe dont la correspondance a déjà été précisée, <table-name> n'a pas à être nécessairement re-précisé.

Les options sont toujours des listes à au moins deux éléments dont le premier est le nom de l'option et le ou les suivants donnent son expression ou sa valeur. Deux possibilités sont ici proposées, *mappedp* et *attrconstraint* :

mappedp Cette option permet de déclarer une classe comme étant sans correspondance relationnelle (alors même que l'on est en train de l'expliciter). Il y a deux intérêts à cela. D'une part, cela permet, en phase de mise au point du schéma par exemple, d'interrompre temporairement les échanges avec la base concernant une classe. Enfin, et c'est là l'intérêt majeur, cela permet de déclarer une classe sans correspondance au milieu d'une hiérarchie mappée. Toute création d'instance de la classe en question ou d'une de ses sous-classes s'effectuera ensuite correctement, les instances bénéficiant et héritant normalement des champs de la classe intermédiaire.

attrconstraint Cette option permet de poser une contrainte sur un attribut sans correspondance (non couplé à un champ atomique de la classe associée) de la relation considérée. La contrainte contribuera à filtrer les n-uplets "instances potentielles" de la classe en question qui devront nécessairement la vérifier.

Le mot-clé introduit une forme Lisp qui va permettre de contraindre un attribut. Une forme est une lambda à arguments typés, à savoir, une liste à deux éléments dont le premier est l'expression de la contrainte (lambda) telle qu'elle est spécifiée dans la description de la fonction DRIVER-CREATE-ATTR-CONSTRAINT ci-après, et le second est le "typage" de l'argument, à savoir la liste nom de la relation, nom de l'attribut dans la relation.

Exemple:

```
? (DRIVER-MAP-CLASS 'Dept 'dept)
= t
? (DRIVER-MAP-CLASS 'Adresse 'address
?     '(attrconstraint ((LAMBDA (num) (>= num 1000))
?                       (address empno))))
= t
```

(DRIVER-CLASS-MAP <class-name>)
(DRIVER-CLASS-MAP0 <class>) **[SUBR à 1 argument]**

Renvoie la table associée à la classe de nom <class-name> (resp. l'objet <class>) si elle a été spécifiée.

Exemple:

```
? (DRIVER-CLASS-MAP 'Adresse)
= address
? (DRIVER-CLASS-MAP 'Vendeur)
= emp
```

(DRIVER-TABLE-MAIN-CLASS <table-name>)
(DRIVER-TABLE-MAIN-CLASS0 <table>) **[SUBR à 1 argument]**

Renvoie la classe principale dont la correspondance relationnelle est la table de nom <table-name> (resp. l'objet <table>).

Exemple:

```
? (DRIVER-TABLE-MAIN-CLASS 'emp)
= Employe
```

(DRIVER-DELETE-CLASS-MAP <class-name>)
(DRIVER-DELETE-CLASS-MAP0 <class>) **[SUBR à 1 argument]**

Détruit la correspondance établie entre la classe principale de nom <class-name> (resp. l'objet <class>) et la table qu'on lui a associée. Cette correspondance est également détruite pour toutes ses sous-classes. Détruit également les correspondances des champs de la classe et des sous-classes.

Exemple:

```
? (DRIVER-DELETE-CLASS-MAP 'Adresse)
= t
```

(DRIVER-CREATE-ATTR-CONSTRAINT <class-name>
(<table-name> <attribute-name>) <lambda> <joins> . <overwrite>)

(DRIVER-CREATE-ATTR-CONSTRAINT0 <class> <attribute>
<lambda> <joins> . <overwrite>)
[SUBR à 4 ou 5 arguments]

Cette fonction permet de poser une contrainte simple sur un attribut. Cette contrainte s'exprime sous la forme d'une lambda à un argument où cet argument représente l'attribut à contraindre.

Exemple:

```
? (DRIVER-CREATE-ATTR-CONSTRAINT 'Adresse '(address state)
?                  '(lambda (val) (memq val '(NY CA))) ())
= Cstr<Adresse[address.state]>
```

(DRIVER-FIND-CLASS-CONSTRAINT <class-name>
(<table-name> <attribute-name>))

(DRIVER-FIND-CLASS-CONSTRAINT0 <class> <attribute>)
(DRIVER-DELETE-CLASS-CONSTRAINT <class-name>
(<table-name> <attribute-name>))

(DRIVER-DELETE-CLASS-CONSTRAINT0 <class> <attribute>)
[SUBR à 2 arguments]

Ces fonctions permettent donc également de rechercher ou détruire une contrainte posée sur un attribut. Voir les précédentes descriptions dans le paragraphe B.1.3.

Exemple:

```
? (DRIVER-FIND-CLASS-CONSTRAINT 'Adresse '(address state))
= Cstr<Adresse[address.state]>
```

(DRIVER-FIND-ATTR-CONSTRAINTS-ON-CLASS <class-name>)
(DRIVER-FIND-ATTR-CONSTRAINTS-ON-CLASS0 <class>)
[SUBR à 1 argument]

Renvoie l'ensemble des contraintes s'appliquant à la classe de nom <class-name> (de l'objet <class>) et posées sur des attributs. Une contrainte définie pour une classe sur un attribut masque l'eventuelle contrainte posée sur le même attribut dans une classe mère.

Exemple:

```
? (DRIVER-FIND-ATTR-CONSTRAINTS-ON-CLASS 'Vendeur
= (Cstr<Employe[emp.empno]>)
```

(DRIVER-FIND-CONSTRAINTS-ON-CLASS <class-name>)
(DRIVER-FIND-CONSTRAINTS-ON-CLASS0 <class>)
[SUBR à 1 argument]

Renvoie l'ensemble des contraintes s'appliquant à la classe de nom <class-name> (de l'objet <class>) (champs et attributs). Une contrainte définie sur le champ d'une classe masque l'eventuelle contrainte posée sur le même champ dans une classe mère.

Exemple:

```
? (DRIVER-FIND-CONSTRAINTS-ON-CLASS 'Vendeur)
= (Cstr<Vendeur[qualif]> Cstr<Employe[salaire]> Cstr<Employe[emp.empno]>)
```

(DRIVER-CREATE-JOIN <table1-name> <table2-name>
<applied-lambda> . <overwrite>)

(DRIVER-CREATE-JOIN0 <table1> <table2>
<applied-lambda> . <overwrite>)
[SUBR à 3 ou 4 arguments]

Définit un chemin de la table de nom <table1-name> (resp. l'objet <table1>) vers la table de nom <table2-name> (resp. l'objet <table2>). Ce chemin peut ensuite être utilisé pour définir la correspondance d'un champ atomique de la classe de table principale <table1-name> sur un attribut de la table <table2-name>. Il peut également être utilisé pour définir la correspondance des champs objet, ensemble, et autres.

Le chemin doit être explicité sous forme d'une lambda où chaque attribut qui y participe s'exprime sous forme d'une variable passée en argument. Le chemin est établi entre deux n-uplets des deux relations quand l'application de la lambda renvoie une valeur différente de nil.

<applied-lambda> est une liste dont le premier élément est une lambda, et dont les autres représentent, sous forme de listes nom de la relation, nom de l'attribut dans la relation, l'ensemble des attributs qu'on associe à ses arguments.

Si l'application de la lambda à un n-uplet de valeurs d'attributs renvoie une valeur vraie (au sens lisp), la jointure est établie pour ce n-uplet. Elle ne l'est pas dans le cas contraire. L'expression (unique) constituant le corps de la lambda peut faire appel aux primitives lisp suivantes : $and, or, not, =, eq, neq, neqn, null, <>, <, <=, >, >=, memq$ et $nmemq$ ($(not\ memq)$). Les primitives à deux arguments doivent être utilisées avec la variable comme premier argument et une constante comme second. L'usage de l'ensemble des primitives lisp de l'interprète n'est pas autorisé car DRIVER doit ensuite pouvoir compiler l'expression en SQL pour l'exploiter.

En cas de tentative de redéfinition d'une jointure, l'erreur "jointure déjà existante" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, l'ancienne définition est perdue, écrasée par la nouvelle. Toutes les structures qui la contenaient pointent maintenant sur la nouvelle.

Exemple:

```
? (DRIVER-CREATE-JOIN 'dept 'emp
?          '((LAMBDA (a b) (EQ a b)) (dept deptno) (emp deptn)))
= J[dept->emp]
```

(DRIVER-FIND-JOIN <table1-name> <table2-name>)
(DRIVER-FIND-JOIN0 <table1> <table2>) **[SUBR à 2 arguments]**

Retourne la jointure de la table de nom <table1-name> (resp. l'objet <table1>) vers la table de nom <table2-name> (resp. l'objet <table2> si elle existe.

Exemple:

```
? (DRIVER-FIND-JOIN 'dept 'emp)
= J[dept->emp]
```

(DRIVER-JOIN-MATCHING <table1-name> <table2-name> . <joins>)
(DRIVER-JOIN-MATCHING0 <table1> <table2> . <joins>)
[SUBR à 2 ou 3 arguments]

Retourne l'ensemble des jointures de la table de nom <table1-name> (resp. l'objet <table1>) vers la table de nom <table2-name> (resp. l'objet <table2>. L'une des deux tables (ou les deux) peut ne pas être précisée; sa référence doit alors être remplacée par ().

Si une liste de jointures <joins> est précisée, la recherche s'effectuera dedans. Sinon, l'ensemble des jointures du schéma courant sera considéré.

Exemple:

```
? (DRIVER-JOIN-MATCHING 'dept ())
= (J[dept->emp])
```

(DRIVER-JOIN-P <join>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <join> si ce dernier est une jointure.

Exemple:

```
? (DRIVER-JOIN-P (DRIVER-FIND-JOIN 'dept 'emp))
= J[dept->emp]
```

(DRIVER-DELETE-JOIN <table1-name> <table2-name>)
(DRIVER-DELETE-JOIN0 <join>) **[SUBR à 1 et 2 arguments]**

Supprime la jointure de la table de nom <table1-name> vers la table de nom <table2-name> (resp. l'objet jointure <join>).

Exemple:

```
? (DRIVER-DELETE-JOIN 'dept 'emp)
= t
```

(DRIVER-MAP-FIELD <class-name> <field-name> <field-map> <join-descriptions> [<options>] . <overwritep>)
[SUBR de 4 à 8 arguments]

(DRIVER-MAP-FIELD0 <field> <field-map> <joins> [<options>] . <overwritep>)
[SUBR de 3 à 7 arguments]

Définit la correspondance du champ <field-name> (resp. l'objet <field>) de la classe <class-name> (resp. l'objet <class>) dans le monde relationnel.

En fonction du type du champ, sa correspondance peut être un attribut, une expression, un ensemble de jointures, un attribut accessible depuis la table principale par jointure.

<field-map> est la correspondance du champ, sauf si cette correspondance est un ensemble de jointures. Si la correspondance est ou comprend un ensemble de jointures, cet ensemble doit être déclaré par <join-descriptions> (resp. <joins>). <field-map> peut donc être une description d'attribut (resp. un objet attribut) ou une expression calculée (voir ci-après). Si la correspondance du champ est uniquement un ensemble de jointures, <field-map> vaut nil.

atom Les champs atomiques sont associés à un attribut. Si l'attribut fait partie de la relation principale de la classe, aucune jointure ne doit être spécifiée (<join-descriptions> (resp. <joins>) vaut nil). S'il fait partie d'une table secondaire, un ensemble de jointures reliant la table secondaire à la table principale doit être précisé.

L'option **filter** est disponible pour tous les types de champs.

filter permet de définir des filtres par lesquels passeront les données pendant les échanges avec la base. La valeur du champ lu dans la base passe dans le filtre avant écriture dans l'objet. La valeur à écrire dans la base et provenant du champ de l'objet transite également par le filtre. La définition et la description des méthodes de filtrage sont présentées au paragraphe B.3.

atomset Les champs de type "ensemble d'atomes" sont associés à un attribut d'une table secondaire, relié à la relation principale par un ensemble de jointures. Cet ensemble de jointures doit permettre de sélectionner un ensemble de valeurs de l'attribut en question.

ordatomset Même chose que pour les champs de type *atomset*. En plus, l'ordre des éléments de l'ensemble est précisé par la valeur d'attributs de la même table secondaire. Le premier attribut définit l'ordre primaire, le second l'ordre secondaire, et ainsi de suite. L'option **orders** permet de préciser ces attributs et, pour chacun d'entre eux, l'ordre (ascendant ou descendant) dans lequel doit se faire le tri.

La syntaxe de l'option est la suivante :

```
(orders (attr1 ord1) ... (attrN ordN))
```

où chaque attrI est un attribut pour la fonction DRIVER-MAP-FIELD0 et une liste (<table-name> <attr-name>) pour la fonction DRIVER-MAP-FIELD, et où chaque ordI prend une valeur parmi **asc** et **desc**.

object Les champs de type "objet" représentent des liens entre objets dans la représentation objet. Ils ont naturellement comme correspondances un ensemble de jointures reliant leurs relations principales respectives. <field-map> vaut nil.

object2 Les champs de type "objet 2" retrouvent leurs valeurs en décodant une chaîne de caractères contenue dans un attribut. La correspondance de ce type de champ est donc l'attribut en question, complété de l'ensemble des jointures qui permettent de l'atteindre depuis la relation principale de la classe contenant le champ.

objectset Les champs de type "ensemble d'objets" représentent des liens entre un objet et un ensemble d'objets. Ils ont comme correspondances un ensemble de jointures reliant leurs relations principales respectives. A la différence des champs de type objet, ces jointures sont vérifiées par un ensemble de n-uplets au lieu d'un seul. <field-map> vaut nil.

ordobjset Même chose que pour les champs de type *objectset*, l'ordre des éléments de l'ensemble étant en plus précisé par la valeur d'attributs de la table secondaire. voir la description du type **ordatomset**.

object2set Les champs de type "ensemble d'objets 2" sont associés à un attribut d'une table secondaire, à partir des valeurs duquel seront décodés des pointeurs d'objet. Cet attribut doit être accessible par un ensemble de jointures. A la différence du champ de type objet2, ces jointures sont vérifiées par un ensemble de n-uplets au lieu d'un seul.

ordobj2set Même chose que pour les champs de type *ordobjset*, l'ordre des éléments de l'ensemble étant en plus précisé par la valeur d'attributs de la table secondaire. voir la description du type **ordatomset**.

monotexpr Le type "calcul sur un n-uplet" permet d'associer à un champ l'expression d'un calcul effectué à partir des n-uplets constituant l'objet, celui de la relation principale et ceux des relations secondaires. La correspondance du champ est donc un calcul s'exprimant sous forme d'une lambda appliquée à des attributs, complété de l'ensemble des jointures permettant d'accéder à ses attributs.

Sont autorisées toutes combinaisons des fonctions $+$, $-$, $*$ et $/$.

multitexpr Le type "calcul sur un ensemble de n-uplets" permet d'associer à un champ l'expression d'un calcul effectué sur un ensemble de n-uplets. La correspondance de ce type de champ est également une lambda appliquée, complétée de l'ensemble des jointures permettant d'accéder à l'ensemble des n-uplets en relation avec chaque objet.

Sont autorisées, en plus des fonction de calcul précédentes, les fonctions *avg* (*average* et *sum* permettant de calculer la moyenne et la somme des valeurs d'attributs accessibles par le biais de l'ensemble de jointures indiqué.

En cas de tentative de redéfinition d'une correspondance, l'erreur "correspondance déjà existante" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, l'ancienne définition est perdue, écrasée par la nouvelle.

Exemple:

```
? (DRIVER-MAP-FIELD 'Dept 'personnel ()
?        '((dept emp ((LAMBDA (a1 a2) (EQ a1 a2))
?                     (dept deptnum) (emp empno)))))
= t
? (DRIVER-MAP-FIELD 'Dept 'moy-sal
?               '((LAMBDA (a1) (AVG a1)) (emp sal))
?               '((dept emp)))
= t
```

(DRIVER-FIELD-MAP <class-name> <field-name>)
(DRIVER-FIELD-MAP0 <field>) **[SUBR à 2 et 1 argument]**

Renvoie la correspondance du champ de nom <field-name> de la classe de nom <class-name> (resp. l'objet <field>). Cette correspondance se présente sous la forme d'une liste à deux éléments. Le premier est l'attribut (la lambda appliquée pour les champs calculés) s'il y en a un, le second est la liste des jointures, s'il y en a une.

Exemple:

```
? (DRIVER-FIELD-MAP 'Employe 'nom)
= (emp.ename ())
? (DRIVER-FIELD-MAP 'Employe 'num-ss)
= (t115@p(person).ssnum (J[emp->t115@p(person)]))
? (DRIVER-FIELD-MAP 'Employe 'voiture)
= (() (J[emp->t115@p(person)] J[t115@p(person)->t115@vehicle(vehicle)]))
```

(DRIVER-ATTRIBUTE-MAP-FIELD <table-name> <attr-name>)
[SUBR à 2 arguments]

(DRIVER-ATTRIBUTE-MAP-FIELD0 <attribute>) [SUBR à 1 argument]

Retourne le champ ayant pour correspondance l'attribut de nom <attr-name> dans la table de nom <table-name> (resp. à l'objet <table>) s'il existe.

(voir les fonctions DRIVER-ATTRIBUTE-DEF-MAP-FIELDS(0))

Exemple :

```
? (DRIVER-ATTRIBUTE-MAP-FIELD 'emp 'fname)
= prenom
? (DRIVER-ATTRIBUTE-MAP-FIELD 'person 'position)
= ()
? (DRIVER-ATTRIBUTE-MAP-FIELD 't115@p 'position)
= situ-famille
```

(DRIVER-DELETE-FIELD-MAP <class-name> <field-name>)
(DRIVER-DELETE-FIELD-MAP0 <field>) [SUBR à 2 ou 1 argument]

Détruit la correspondance établie sur le champ de nom <field-name> de la classe de nom <class-name> (resp. l'objet <field>).

Exemple:

```
? (DRIVER-DELETE-FIELD-MAP 'Dept 'personnel)
= t
```

(DRIVER-MAIN-CLASS-TABLES <class-name> . <allp>)
(DRIVER-MAIN-CLASS-TABLES0 <class> . <allp>)
[SUBR à 1 ou 2 arguments]

Retourne l'ensemble des tables élémentaires de la classe de nom <class-name> (resp. l'objet <class>) si <allp> n'est pas précisé ou vaut (), ou toutes les tables en utilisées dans la correspondance de la classe si <allp> vaut t

Exemple :

```
? (DRIVER-MAIN-CLASS-TABLES 'Employe)
= (t115@emp1(emp) emp t115@p(person))
? (DRIVER-MAIN-CLASS-TABLES 'Employe t)
= (t115@emp1(emp) t115@sg(salgrade) t115@dept(dept) t115@address(address)
  t115@vehicle(vehicle) emp t115@p(person))
```

(DRIVER-MAIN-TABLE-P <table>)
(DRIVER-SUB-TABLE-P <table>) [SUBR à 1 argument]

Le prédicat DRIVER-MAIN-TABLE-P est vrai et renvoie l'objet <table> si la table en question est table principale d'une classe. Le prédicat DRIVER-SUB-TABLE-P est vrai et renvoie également l'objet si la table en question est table secondaire d'une classe.

Une table ne peut pas être simultanément principale et secondaire. Elle peut aussi être ni l'une, ni l'autre.

Exemple:

```
? (DRIVER-MAIN-TABLE-P (DRIVER-FIND-TABLE 'emp))
= emp
? (DRIVER-SUB-TABLE-P (DRIVER-FIND-TABLE 'person))
= person
```

(DRIVER-CLASS-LOADING-ORDER <class-names>)
(DRIVER-CLASS-LOADING-ORDER0 <classes>)

[SUBR à 0 ou 1 argument]

Permet de spécifier l'ordre de chargement des classes à utiliser si des objets de différentes classes doivent être chargées au même moment. Cet ordre ne peut être quelconque : on devra forcément charger une classe avant ses filles.

Si cette fonction n'est pas employée, l'ordre de déclaration des classes est utilisé par défaut.

Si seule une partie des classes définies dans le schéma de correspondances est indiquée, ces classes seront chargées d'abord, suivies des autres selon l'ordre dans lequel elles ont été déclarées. Attention à ce que l'ordre de l'ensemble vérifie bien la restriction qui est imposée.

Renvoie l'ensemble des classes (éventuellement complété) dans l'ordre spécifié. Si aucun argument n'est passé, renvoie l'ordre courant de chargement des classes.

Exemple:

```
? (DRIVER-CLASS-LOADING-ORDER)
= (Employe Vendeur Dept Adresse)
? (DRIVER-CLASS-LOADING-ORDER '(Employe Dept Vendeur))
= (Employe Dept Vendeur Adresse)
```

(DRIVER-FIELD-LOADING-ORDER <class-name> . <field-names>)
(DRIVER-FIELD-LOADING-ORDER0 <class> . <fields>)

[SUBR à 1 ou 2 arguments]

Permet de spécifier l'ordre de chargement des champs de la classe indiquée. Si cette fonction n'est pas employée, l'ordre de déclaration des champs est utilisé par défaut.

Si seule une partie des champs définis dans la classe est indiquée, ces champs seront chargés d'abord, suivies des autres selon l'ordre dans lequel ils ont été déclarés.

Renvoie l'ensemble des champs (éventuellement complété) dans l'ordre spécifié. Si seule la classe est spécifiée, renvoie l'ordre courant de chargement de ses champs.

Exemple:

```
? (DRIVER-FIELD-LOADING-ORDER 'Employe)
= (nom prenom num-ss qualif grade chef dpt salaire adresse)
? (DRIVER-FIELD-LOADING-ORDER 'Employe
?                             '(nom prenom num-ss adresse))
= (nom prenom num-ss adresse qualif grade chef dpt salaire)
```

B.1.5 Description relationnelle : fonctions de bas niveau

Les fonctions présentées en B.1.2 permettent de créer simplement des tables et leurs attributs pour décrire le schéma relationnel d'une base de données. Les définitions et variables de tables et d'attributs y sont gérées de manière complètement transparente pour l'utilisateur qui ne manipule en fait que des variables. Ces fonctions sont suffisantes pour l'utilisateur qui ne décrit les correspondances avec le monde objet qu'au moyen de la macro-fonction DRIVER-DEFCLASSMAP. Elles sont insuffisantes sans l'utilisation de DRIVER-DEFCLASSMAP pour exprimer les auto-jointures ou pour définir plusieurs jointures entre deux mêmes tables.

Les fonctions suivantes, de plus bas niveau, donnent accès aux objets définitions et variables de façon indépendante. Elles permettent la création de variables de tables et d'attributs directement utilisables avec les primitives de description des correspondances, en particulier pour la définition de jointure.

(DRIVER-CREATE-TABLE-DEF <table-def-name> . <overwritep>)
[SUBR à 1 ou 2 arguments]

Ajoute la définition de la table de nom <table-def-name> dans le schéma courant. Cette définition est initialement vide d'attribut. Les attributs correspondants devront être créés au moyen de la fonction DRIVER-CREATE-ATTRIBUTE-DEF.

La création d'une définition de table provoque automatiquement la création de la variable de table de même nom.

En cas de tentative de redéfinition d'une table, l'erreur "table déjà existante" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, l'ancienne définition est perdue, écrasée par la nouvelle, ainsi que toutes les définitions d'attributs, les variables de table et d'attribut qui l'utilisaient. Toutes les structures qui le contenaient pointent maintenant sur la nouvelle.

Cette fonction ne se différencie de la fonction DRIVER-CREATE-TABLE que par le résultat qu'elle retourne qui est ici une définition de table plutôt qu'une (variable de) table dans l'autre cas.

Exemple:

```
? (DRIVER-CREATE-TABLE-DEF 'project)
= <project>
? (DRIVER-CREATE-TABLE-DEF 'car)
= <car>
```

(DRIVER-FIND-TABLE-DEF <table-def-name>) **[SUBR à 1 argument]**

Retourne la définition de table de nom <table-def-name> si elle existe dans le schéma courant.

Exemple:

```
? (DRIVER-FIND-TABLE-DEF 'project)
= <project>
```

(DRIVER-TABLE-DEF-P <table-def>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <table-def> si ce dernier est une définition de table.

Exemple:

```
? (DRIVER-TABLE-DEF-P (DRIVER-FIND-TABLE-DEF 'project))
= <project>
```

(DRIVER-DELETE-TABLE-DEF <table-def-name>)
(DRIVER-DELETE-TABLE-DEF0 <table-def>) **[SUBR à 1 argument]**

Détruit la définition de table de nom <table-def-name> (resp. l'objet <table-def>) dans le schéma courant. Par effet de bord, les attributs définis dans cette table sont également détruits, ainsi que toute variable associée à la table ou à l'un de ses attributs. De même, toutes les références à la table sont retirées du schéma de correspondances.

Exemple:

```
? (DRIVER-DELETE-TABLE-DEF 'car)
= t
```

(DRIVER-CREATE-ATTRIBUTE-DEF <table-def-name>
<attr-def-name> <attr-type> <typ-len> <status> . <overwritep>)

(DRIVER-CREATE-ATTRIBUTE-DEF0 <table-def> <attr-def-name>
<attr-type> <typ-len> <status> . <overwritep>)
[SUBR à 4 ou 5 arguments]

Ajoute la définition de l'attribut de nom <attr-def-name> à la table de nom <table-def-name> (resp. à l'objet <table-def>) dans le schéma courant. Le type de l'attribut, la longueur de ce type ainsi que le statut de l'attribut doivent être spécifiés. Les différents statuts possibles sont *unique*, *not-null*, *keypart*, *void* ou *()* (voir la fonction DRIVER-CREATE-ATTRIBUTE).

Les types d'attribut possibles sont par défaut *integer*, *float* et *string*. Cette liste peut être complétée en fonction de la base de données utilisée et des types qu'elle reconnaît au moyen de la fonction DRIVER-ADD-ATTRIBUTE-TYPE. Elle est consultée à chaque création d'attribut.

La création d'une définition d'attribut provoque automatiquement la création de la variable d'attribut associée.

En cas de tentative de redéfinition d'un attribut, l'erreur "définition d'attribut déjà existante" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, l'ancienne définition est perdue, écrasée par la nouvelle, ainsi que toutes les variables d'attribut qui l'utilisaient. Toutes les structures qui le contenaient pointent maintenant sur la nouvelle.

Ces fonctions ne se différencient des fonctions DRIVER-CREATE-ATTRIBUTE et DRIVER-CREATE-ATTRIBUTE0 que par le résultat qu'elles retournent qui est ici une définition d'attribut plutôt qu'un (une variable d') attribut dans l'autre cas.

Exemple:

```
? (DRIVER-CREATE-ATTRIBUTE-DEF 'project 'projno
?                              'integer 8 'keypart)
= <project.projno>
? (DRIVER-CREATE-ATTRIBUTE-DEF 'project 'pname 'string 20 ())
= <project.pname>
```

(DRIVER-FIND-ATTRIBUTE-DEF <table-def-name> <attr-def-name>)
(DRIVER-FIND-ATTRIBUTE-DEF0 <table-def> <attr-def-name>)
[SUBR à 2 arguments]

Retourne la définition d'attribut de nom <attr-def-name> de la table de nom <table-def-name> (de la table <table-def>) si elle existe.

Exemple:

```
? (DRIVER-FIND-ATTRIBUTE-DEF 'project 'pname)
= <project.pname>
```

(DRIVER-ATTRIBUTE-DEF-P <attr-def>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <attr-def> si ce dernier est une définition d'attribut.

Exemple:

```
? (DRIVER-ATTRIBUTE-DEF-P
?            (DRIVER-FIND-ATTRIBUTE-DEF 'project 'projno))
= <project.projno>
```

(DRIVER-DELETE-ATTRIBUTE-DEF <table-def-name> <attr-def-name>)
[SUBR à 2 arguments]

(DRIVER-DELETE-ATTRIBUTE-DEF0 <attr-def>) **[SUBR à 1 argument]**

Détruit la définition d'attribut de nom <attr-def-name> de la table de nom <table-def-name> (l'objet <attr-def>) dans le schéma courant. Par effet de bord, toute variable associée à l'attribut est également détruite. De même, toutes les références à l'attribut sont retirées du schéma de correspondances.

Exemple:

```
? (DRIVER-DELETE-ATTRIBUTE-DEF 'project 'pname)
= t
```

(DRIVER-TABLEDEF-ATTRIBUTEDEFS <table-def-name>)
(DRIVER-TABLEDEF-ATTRIBUTEDEFS0 <table-def>)
[SUBR à 1 argument]

Renvoie dans l'ordre de leurs créations les définitions d'attribut associées à une définition de table. Ce même ordre est utilisé pour une éventuelle création de la relation correspondante dans la base de données relationnelle.

Exemple:

```
? (DRIVER-TABLEDEF-ATTRIBUTEDEFS 'project)
= (<project.projno>)
```

(DRIVER-ATTRIBUTE-DEF-MAP-FIELDS <table-def-name> <attr-name>)
[SUBR à 2 arguments]

(DRIVER-ATTRIBUTE-DEF-MAP-FIELDS0 <attribute-def>)
[SUBR à 1 argument]

Renvoie la liste des champs associés à une variable d'attribut dont la définition d'attribut est de nom <attr-def-name> et de table de nom <table-def-name> (resp. la table <table-def>).

Exemple :

```
? (DRIVER-ATTRIBUTE-DEF-MAP-FIELDS 'person 'position)
= (situ-famille)
```

Fonctions portant sur les variables de structure

Les variables de table et d'attribut sont utilisées pour construire les correspondances. Les définitions ne peuvent en effet en aucun cas être utilisées directement.

(DRIVER-CREATE-TABLE-VAR <table-def-name> <var-name>)
(DRIVER-CREATE-TABLE-VAR0 <table-def> <var-name>)
[SUBR à 2 arguments]

Fonctions définissant une variable associée à la définition de table de nom <table-def-name> (resp. à l'objet <table-def>) dans le schéma courant.

Exemple:

```
? (DRIVER-CREATE-TABLE-VAR 'project 'project)
** driver-create-table-var : existent table variable : project
? (DRIVER-CREATE-TABLE-VAR 'project 'proj1)
= proj1(project)
? (DRIVER-CREATE-TABLE-VAR 'project 'proj2)
= proj2(project)
```

(DRIVER-FIND-TABLE-VAR <var-name>) **[SUBR à 1 argument]**

Retourne la variable de table de nom <var-name> si elle existe dans le schéma courant.

Exemple:

```
? (DRIVER-FIND-TABLE-VAR 'project)
= project
? (DRIVER-FIND-TABLE-VAR 'proj1)
= proj1(project)
```

(DRIVER-DO-FIND-TABLE-VAR <table-def-name> <var-name>)
(DRIVER-DO-FIND-TABLE-VAR0 <table-def> <var-name>)
[SUBR à 2 arguments]

Retourne la variable de table de nom <var-name> si elle existe dans le schéma courant. Sinon, la crée associée à la définition de table de nom <table-def-name> (resp. l'objet <table-def>).

Exemple:

```
? (DRIVER-DO-FIND-TABLE-VAR 'project 'proj1)
= proj1(project)
? (DRIVER-DO-FIND-TABLE-VAR 'project 'proj2)
= proj2(project)
```

(DRIVER-TABLE-VAR-P <table-var>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <table-var> si celui-ci est une variable de table.

Exemple:

```
? (DRIVER-TABLE-VAR-P (DRIVER-FIND-TABLE-VAR 'proj1))
= proj1(project)
```

(DRIVER-DELETE-TABLE-VAR <table-var-name>)
(DRIVER-DELETE-TABLE-VAR0 <table-var>) **[SUBR à 1 argument]**

Détruit la variable de table de nom <table-var-name> (resp. l'objet <table-var>) dans le schéma courant. Par effet de bord, les variables d'attributs associées à cette variable de table sont également détruites. De même, toutes les références à la table sont retirées du schéma de correspondances.

ATTENTION: Il n'est pas autorisé de détruire la variable de table automatiquement créée en même temps que la définition de table, celle portant en fait le même nom.

Exemple:

```
? (DRIVER-DELETE-TABLE-VAR 'project)
** driver-delete-table-var :
             not an erasable table variable : project
? (DRIVER-DELETE-TABLE-VAR 'proj2)
= t
```

(DRIVER-TABLEDEF-TABLEVARS <table-def-name>)
(DRIVER-TABLEDEF-TABLEVARS0 <table-def>) **[SUBR à 1 argument]**

Renvoie la liste des variables associées à la définition de table de nom <table-def-name> (resp. l'objet <table-def>).

Exemple:

```
? (DRIVER-TABLEDEF-TABLEVARS 'project)
= (project proj1(project))
```

(DRIVER-CREATE-ATTRIBUTE-VAR <tablevar-name>
<attrdef-name> . <overwritep>)

(DRIVER-CREATE-ATTRIBUTE-VAR0 <table-var>
<attribute-def> . <overwritep>)
[SUBR à 2 ou 3 arguments]

Fonctions créant une variable d'attribut associée à la variable de table de nom <tablevar-name> (resp. à l'objet <table-var>) et à la définition d'attribut de nom <attrdef-name> (resp. et à l'objet <attribute-def>).

En cas de tentative de redéfinition d'une variable d'attribut, l'erreur "variable d'attribut déjà existante" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, il n'y a pas d'erreur. L'objet reste le même puisqu'ayant même variable de table et même définition d'attribut. L'intérêt de ce bouléen est est de pouvoir générer automatiquement des créations de variables d'attribut sans se soucier si elles existent déjà.

Exemple:

```
? (DRIVER-CREATE-ATTRIBUTE-VAR 'proj1 'projno)
= proj1(project).projno
? (DRIVER-CREATE-ATTRIBUTE-VAR 'project 'projno)
** driver-create-attribute-var :
              existent attribute variable : (project . projno)
? (DRIVER-CREATE-ATTRIBUTE-VAR0
?                   (DRIVER-FIND-TABLE-VAR 'proj1)
?                   (DRIVER-FIND-ATTRIBUTE-DEF 'Address 'city))
** driver-create-attribute-var :
                 not one of the attribute table : (project . city)
```

(DRIVER-FIND-ATTRIBUTE-VAR <table-var-name> <attr-def-name>)
(DRIVER-FIND-ATTRIBUTE-VAR0 <table-var> <attribute-def>)
[SUBR à 2 arguments]

Retourne la variable d'attribut associée à la variable de table de nom <table-var-name> (resp. à l'objet <table-var>) et à la définition d'attribut de nom <attr-def-name> (resp. et à l'objet <attribute-def>) si elle existe dans le schéma courant.

Exemple:

```
? (DRIVER-FIND-ATTRIBUTE-VAR 'project 'projno)
= project.projno
? (DRIVER-FIND-ATTRIBUTE-VAR 'proj1 'projno)
= proj1(project).empno
```

(DRIVER-DO-FIND-ATTRIBUTE-VAR <table-var-name> <attr-def-name>)
(DRIVER-DO-FIND-ATTRIBUTE-VAR0 <table-var> <attribute-def>)
[SUBR à 2 arguments]

Retourne la variable d'attribut associée à la variable de table de nom <table-var-name> (resp. à l'objet <table-var>) et à la définition d'attribut de nom <attr-def-name> (resp. et à l'objet <attribute-def>) si elle existe dans le schéma courant. Sinon, la crée et la renvoie.

Exemple:

```
? (DRIVER-DO-FIND-ATTRIBUTE-VAR 'proj1 'projno)
= proj1(project).projno
? (DRIVER-DO-FIND-ATTRIBUTE-VAR 'proj2 'projno)
= proj2(project).empno
```

(DRIVER-ATTRIBUTE-VAR-P <attribute-var>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <attribute-var> si celui-ci est une variable d'attribut.

Exemple:

```
? (DRIVER-ATTRIBUTE-VAR-P
?           (DRIVER-FIND-ATTRIBUTE-VAR 'proj1 'projno))
= proj1(project).projno
```

(DRIVER-ATTRDEF-ATTRVARS <table-def-name> <attribute-def-name>)
[SUBR à 2 arguments]

(DRIVER-ATTRDEF-ATTRVARS0 <attribute-def>) **[SUBR à 1 argument]**

Renvoie la liste des variables associées à la définition d'attribut de nom <attribute-def-name> de la table de nom <table-def-name> (resp. à l'objet <attribute-def> pour la seconde fonction).

Exemple:

```
? (DRIVER-ATTRDEF-ATTRVARS 'project 'projno)
= (project.projno proj1(project).projno)
```

(DRIVER-TABLEVAR-ATTRVARS <table-var-name>)
(DRIVER-TABLEVAR-ATTRVARS0 <table-var>) **[SUBR à 1 argument]**

Renvoie la liste des variables d'attribut définies à partir de la variable de table de nom <table-var-name> (resp. l'objet <table-var>).

Exemple:

```
? (DRIVER-TABLEVAR-ATTRVARS 'proj1)
= (proj1(project).projno)
```

(DRIVER-DELETE-ATTRIBUTE-VAR <table-var-name> <attr-def-name>)
(DRIVER-DELETE-ATTRIBUTE-VAR0 <attribute-var>)

[SUBR à 1 argument]

Détruit la variable d'attribut associée à la variable de table de nom <table-var-name> et à la définition d'attribut de nom <attr-def-name> (resp. l'objet <attribute-var>) si elle existe dans le schéma courant. Par effet de bord, toutes les références à l'attribut sont retirées du schéma de correspondances. Par exemple, l'apparition de la variable dans une jointure provoquera la destruction de la jointure.

ATTENTION: Il n'est pas autorisé de détruire la variable d'attribut automatiquement créée en même temps que la définition d'attribut, celle dont la variable de table associée porte le même nom que la définition de table correspondante.

Exemple:

```
? (DRIVER-DELETE-ATTRIBUTE-VAR 'proj1 'projno)
= t
? (DRIVER-DELETE-ATTRIBUTE-VAR 'project 'projno)
** driver-delete-table-var :
            not an erasable attribute variable : (project projno)
```

B.1.6 Création des classes et des relations

Ces fonctions permettent de définir les structures objet et relationnelles respectivement dans le langage objet et dans le SGBD connecté si elles n'y existent pas encore.

(DRIVER-MAKE-TABLE <table-name>)
(DRIVER-MAKE-TABLE0 <table>) **[SUBR à 1 argument]**

Définit dans le SGBD associé au schéma de correspondances courant la table de nom <table-name> (resp. l'objet <table>) précédemment décrite dans ce même schéma.

ATTENTION: Cette fonction ne vérifie pas l'inexistence de la table en question dans le SGBD avant la création demandée.

Exemple:

```
? (DRIVER-MAKE-TABLE 'emp)
= t
```

(DRIVER-MAKE-CLASS <class-name>)
(DRIVER-MAKE-CLASS0 <class>) **[SUBR à 1 argument]**

Définit dans le langage objet hôte la classe de nom <class-name> précédemment décrite dans le schéma de correspondances courant.

Exemple:

```
? (DRIVER-MAKE-CLASS 'Employe)
= t
```

B.1.7 Génération automatique de classes et de relations

Ces fonctions permettent de générer automatiquement, à partir d'une représentation objet ou relationnelle, celle qui lui correspond dans l'autre modèle. Les rapprochements sont également produits.

(DRIVER-GENERATE-TABLE-MAP <table-name>) **[MACRO]**

Définit dans le schéma courant une correspondance objet pour la table de nom <table-name>. La classe générée comporte un champ atomique par attribut, le type de l'attribut étant repris comme type du champ. Il conviendra donc de définir les méthodes de filtrage en lecture - écriture des champs avec la base pour les attributs ayant un type différent d'integer, float et string.

DRIVER-GENERATE-TABLE-MAP peut être décrit en lisp de la manière suivante :

```
(DEFMACRO DRIVER-GENERATE-TABLE-MAP (table-name)
   (LET ((table-def (SEND 'table
                          (DRIVER-FIND-TABLE table-name))))
    `(PROGN
       (DRIVER-DEFCLASS ,(SEND 'name table-def) ()
                ,@(MAPCAR (LAMBDA (attr-def)
                            `(,(SEND 'attrname attr-def)
                              atom
                              ,(SEND 'type attr-def)))
                          (DRIVER-TABLEDEF-ATTRIBUTEDEFS0
                                                  table-def)))
       (DRIVER-DEFCLASSMAP ,(SEND 'name table-def)
                           ,(SEND 'name table-def)
                (FIELDS
                   ,@(MAPCAR (LAMBDA (attr-def)
                               `(,(SEND 'attrname attr-def)
                                 (,(SEND 'name table-def)
                                  ,(SEND 'attrname attr-def))
                                 ()))
                             (DRIVER-TABLEDEF-ATTRIBUTEDEFS0
                                                table-def)))))))
```

Exemple:

```
? (DE essai () (DRIVER-GENERATE-TABLE-MAP emp))
= essai
? ^Pessai
(de essai ()
    (driver-generate-table-map emp))
= ()
? (essai)
** driver-map-class : Yet a main table : emp
? ; emp est deja table principale de la classe Employe !
? (driver-delete-class-map 'Employe)
= t
? (essai)
= t
? ^Pessai
(de essai ()
    (progn
      (let ((class (driver-create-class 'emp '() t)))
           (driver-create-field0 class 'empno 'atom 'integer)
           (driver-create-field0 class 'ename 'atom 'string)
           (driver-create-field0 class 'fname 'atom 'string)
```

```
              (driver-create-field0 class 'job 'atom 'string)
              (driver-create-field0 class 'mgr 'atom 'integer)
              (driver-create-field0 class 'sal 'atom 'float)
              (driver-create-field0 class 'com 'atom 'float)
              (driver-create-field0 class 'deptn 'atom 'integer)
              (if driver-class-make (driver-make-class0 class))
              class)
              (progn
                (driver-map-class 'emp 'emp t)
                (driver-map-field 'emp 'empno '(emp empno) '())
                (driver-map-field 'emp 'ename '(emp ename) '())
                (driver-map-field 'emp 'fname '(emp fname) '())
                (driver-map-field 'emp 'job '(emp job) '())
                (driver-map-field 'emp 'mgr '(emp mgr) '())
                (driver-map-field 'emp 'sal '(emp sal) '())
                (driver-map-field 'emp 'com '(emp com) '())
                (driver-map-field 'emp 'deptn '(emp deptn) '())
                t)))
= ()
```

(DRIVER-GENERATE-CLASS-MAP <class-name>) **[MACRO]**

Définit dans le schéma courant une correspondance relationnelle pour la class de nom <class-name>.

DRIVER-GENERATE-CLASS-MAP n'est pas encore implémentée dans cette version de DRIVER.

B.2 Les primitives utilisateur

B.2.1 Connexion et communication avec le SGBD

(DRIVER-CONNECT <dbms-name> <base-name>) [SUBR à 2 arguments]

Assure la connexion à la base de données de nom <base-name> définie dans le SGBD de nom <dbms-name>. Retourne le curseur résultant de cette connexion.

À la première utilisation de DRIVER-CONNECT lors d'une session, le curseur est stocké dans une variable système. Il est utilisé par défaut comme curseur courant dans tout schéma de correspondances ultérieurement créé au moyen de la fonction DRIVER-CREATE-MAPPING si la variable système DRIVER-AFFECT-CURSOR-P vaut t (valeur par défaut).

Après avoir établi la connexion et créé le curseur, DRIVER-CONNECT effectue un appel à la fonction DRIVER-DBMS-CONNECTION avec les deux arguments <dbms-name> et <base-name> avant de rendre la main.

Exemple de connexion sur la base `smecidemo` avec le SGBD ingres :

```
? (DRIVER-CONNECT 'ingres 'smecidemo)
= #:tclass:cursor:#[ingres 0 t ()]
```

(DRIVER-DBMS-CONNECTION <dbms-name> <user-name>)
[SUBR à 2 arguments]

Fonction appelée par DRIVER-CONNECT lors de la connexion à une base de données. Par défaut, cette fonction est vide et a comme définition :

```
(de driver-dbms-connection (dbms-name user-name))
```

L'utilisateur a tout loisir de la redéfinir pour ses besoins propres.

(DRIVER-CREATE-NEW-CURSOR <dbms-name> <base-name>)
(DRIVER-CREATE-NEW-CURSOR0 <dbms>) [SUBR à 1 ou 2 arguments]

Crée et renvoie un nouveau curseur sur la base de données de nom <base-name> définie dans le SGBD de nom <dbms-name>.

Exemple:

```
? (setq c2 (DRIVER-CREATE-NEW-CURSOR 'ingres 'smecidemo))
= #:tclass:cursor:#[ingres 0 t ()]
```

(DRIVER-MAPPING-CURSOR <mapping-name> . <cursor>)
(DRIVER-MAPPING-CURSOR0 <mapping> . <cursor>)
[SUBR à 1 ou 2 arguments]

Variables-fonctions permettant d'accéder en lecture au curseur courant du schéma de correspondances de nom <mapping-name> si <cursor> n'est pas fourni, et de le remplacer par <cursor> dans le cas contraire.

Exemple:

```
? (DRIVER-MAPPING-CURSOR 'map1)
= #:tclass:cursor:#[ingres 0 t ()]
```

(DRIVER-CURRENT-CURSOR <cursor>) **[SUBR à 0 ou 1 argument]**

Variable-fonction permettant d'accéder en lecture au curseur courant du schéma courant si <cursor> n'est pas fourni, et de le remplacer par <cursor> dans le cas contraire.

REMARQUE : Un curseur n'est courant que relativement à un schéma donné. Si le schéma courant change, le curseur courant est également susceptible de changer.

Exemple:

```
? (DRIVER-CURRENT-CURSOR)
= #:tclass:cursor:#[ingres 0 t ()]
```

DRIVER-FATAL-REQUEST-ERROR **[Variable]**

Si la variable système DRIVER-FATAL-REQUEST-ERROR vaut t (valeur par défaut), tout échec de requête pour cause d'erreur détectée par le SGBD provoque une erreur DRIVER. Si elle vaut (), l'exécution continue comme si la requête était valide.

B.2.2 Filtrage et manipulation des objets relationnels

DRIVER-COMMIT-AFTER-READING **[Variable]**

Si la variable système DRIVER-COMMIT-AFTER-READING vaut t (valeur par défaut), toutes les requêtes de consultation qui ne sont pas faites dans le cadre d'écritures d'objets dans la base sont suivies d'une validation SGBD (commit SQL) qui déverrouille les données et les rend accessibles aux autres utilisateurs. Dans le cas contraire, seul DRIVER-COMMIT-OBJECTS déclenche une validation SGBD.

(DRIVER-EXISTENT-KEY-P <class-name> <key-value>)
(DRIVER-EXISTENT-KEY-P0 <class> <key-value>)

[SUBR à 2 arguments]

Ce prédicat est vrai et renvoie t si la valeur de clé <key-value> (liste de valeurs correspondant à la liste des attributs composant la clé de la table principale associée à la classe) correspond à un objet relationnel au moins instance de la classe de nom <class-name> (resp. l'objet <class>).

Cette fonction n'est utilisable que quand la connexion à la base a été réalisée car elle engendre une requête SQL qui est immédiatement envoyée à la base pour tester l'existence d'un n-uplet relationnel.

Exemple:

Les réponses retournées par DRIVER-EXISTENT-KEY-P ci-après ont été obtenues alors que :

- le tuple de clé 50 dans la relation emp n'existe pas;
- le tuple de clé 7027 existe mais ne vérifie pas les contraintes imposées par la classe Employé;
- le tuple de clé 7698 existe et vérifie les contraintes imposées par la classe Employé sans vérifier celles imposées par la classe Vendeur;
- le tuple de clé 7499 existe et vérifie les contraintes imposées par la classe Vendeur.

```
? (DRIVER-EXISTENT-KEY-P 'Employe '(50))
= ()
? (DRIVER-EXISTENT-KEY-P 'Employe '(7027))
= ()
? (DRIVER-EXISTENT-KEY-P 'Employe '(7698))
= t
? (DRIVER-EXISTENT-KEY-P 'Vendeur '(7698))
= ()
? (DRIVER-EXISTENT-KEY-P 'Employe '(7499))
= t
? (DRIVER-EXISTENT-KEY-P 'Vendeur '(7499))
= t
```

(DRIVER-GET-OBJECT <class-name> <key-value>)
(DRIVER-GET-OBJECT0 <class> <key-value>) **[SUBR à 2 arguments]**

Renvoie l'objet jumeau associé à l'objet relationnel de classe de nom <class-name> (resp. l'objet <class>) dont la clé dans la base a pour valeur <key-value> (liste de valeurs correspondant à la liste des attributs composant la clé de la table principale associée à la classe), s'il a été construit, et sinon, le défaut d'objet le représentant.

Si la variable système DRIVER-EXISTENT-KEY-TEST vaut t (la valeur par défaut est nil), DRIVER vérifie que la valeur de clé existe.

Exemple:

```
? (DRIVER-GET-OBJECT 'Employe '(7499))
= driver-object-default-1
```

DRIVER-EXISTENT-KEY-TEST [Variable]

Si la variable système DRIVER-EXISTENT-KEY-TEST vaut t (la valeur par défaut est nil), DRIVER vérifie à l'appel de DRIVER-GET-OBJECT que la valeur de clé correspond effectivement à un objet relationnel en appelant la fonction DRIVER-EXISTENT-KEY-P. Cette option est relativement coûteuse puisqu'elle engendre une requête SQL qui est immédiatement envoyée à la base.

Exemple :

```
? DRIVER-EXISTENT-KEY-TEST
= ()
? (DRIVER-GET-OBJECT 'Employe '(1)) ; clef inexistante dans la base
= driver-object-default-2
? (SETQ DRIVER-EXISTENT-KEY-TEST t)
= t
? (DRIVER-GET-OBJECT 'Employe '(1))
** driver-get-object : Not an existent object key : (Employe 1)
```

(DRIVER-GET-OBJECT-IF-LOADED <class-name> <key-value>)
(DRIVER-GET-OBJECT-IF-LOADED0 <class> <key-value>)
[SUBR à 2 arguments]

Renvoie l'objet jumeau associé à l'objet relationnel de classe de nom <class-name> (resp. l'objet <class>) et de valeur de clé <key-value> (liste de valeurs correspondant à la liste des attributs composant la clé de la table principale associée à la classe), s'il a été construit.

Exemple:

```
? (DRIVER-GET-OBJECT-IF-LOADED 'Employe '(7499))
= ()
```

(DRIVER-SELECT-OBJECTS <applied-lambda>) [SUBR à 1 argument]

Renvoie la liste des ensembles de valeurs de clé correspondant aux ensembles d'objets qui, dans la base, vérifient le filtre exprimé sous la forme d'une lambda appliquée <applied-lambda> :

((LAMBDA (obj1 ... objN) corps-lambda) classe-obj1 ... classe-objN)

<applied-lambda> est une liste dont le premier élément est une lambda, et dont les autres précisent la classe des objets associés à ses arguments. Les arguments de la lambda seront chacun instanciés à un des objets des n-uplets à sélectionner.

Un n-uplet d'objets vérifie le filtre si l'application de la lambda à ce n-uplet renvoie une valeur vraie au sens lisp.

Le corps de la lambda doit être constituée d'une expression unique pouvant faire appel aux primitives lisp suivantes : $and, or, not, =, eq, neq, neqn, null, <>, <, <=, >, >=, memq$ et $nmemq$ ($(notmemq)$). Les primitives à deux arguments doivent être utilisées avec la variable comme premier argument et une constante comme second. Comme les variables représentent des objets, la fonction SEND peut en plus être utilisée pour accéder en lecture aux contenus de leurs champs.

De même, comme DRIVER-SELECT-OBJECTS peut en principe servir dans un environnement dynamique, les variables définies avant l'appel de la fonction (par LET ou DEFVAR par exemple) peuvent être utilisées dans le corps de la lambda.

Restrictions

L'accès aux champs de type calcul multi-tuples **multitexpr** n'est pas autorisé pour définir le filtre dans la version actuelle de cette fonction.

D'autre part, les méthodes *driverloading* destinées à modifier une valeur issue de la base avant de l'affecter dans un champ ne sont pas utilisées puisque le filtre est directement compilé en SQL. Seules les valeurs brutes des attributs seront donc exploitées.

Enfin, l'usage d'autres primitives lisp de l'interprète n'est pas autorisé car DRIVER devra pouvoir compiler l'expression en SQL.

Exemple:

```
? ; Filtrer les employes dont le salaire est superieur au seuil
? (DRIVER-SELECT-OBJECTS
?    '((LAMBDA (o1) (>= (SEND 'salaire o1) seuil))
?      Employe))
*** driver-select-objects : unbounded variable : seuil
? (LET ((seuil 1500.))
?      (DRIVER-SELECT-OBJECTS
```

```
?         '((LAMBDA (o1) (>= (SEND 'salaire o1) seuil))
?           Employe)))
= (((7839)) ((7566)) ((7698)) ((7782))
   ((7902)) ((7788)) ((7499)))
?
? ; Filtrer les employe ayant une voiture de meme modele
? ; que leur chef
? (DRIVER-SELECT-OBJECTS
?    '((LAMBDA (obj)
?         (EQ (SEND 'modele (SEND 'vehicule obj))
?             (SEND 'modele (SEND 'vehicule (SEND 'chef obj)))))
?      Employe))
= (((7782)))
?
? ; Filtrer les couples d'employes ayant meme chef et dont le
? ; salaire du premier est superieur a celui du second
? (DRIVER-SELECT-OBJECTS
?    '((LAMBDA (o1 o2)
?        (AND (EQ (SEND 'chef o1) (SEND 'chef o2))
?             (> (SEND 'salaire o1) (SEND 'salaire o2))))
?      Employe Employe))
= (((7655) (7521)) ((7655) (7654)) ((7499) (7521))
   ((7499) (7654)) ((7499) (7655)) ((7698) (7782))
   ((7566) (7782)) ((7566) (7698)))
? ; Filtrer les couples Vendeur-Employe, tous deux ayant meme
? ; chef, tel que le salaire du vendeur est superieur a celui
? ; de l'employe
? (DRIVER-SELECT-OBJECTS
?    '((LAMBDA (o1 o2)
?        (AND (EQ (SEND 'chef o1) (SEND 'chef o2))
?             (> (SEND 'salaire o1) (SEND 'salaire o2))))
?      Vendeur Employe))
= (((7499) (7521)) ((7499) (7654)) ((7499) (7655))
   ((7655) (7521)) ((7655) (7654)))
```

(DRIVER-LOAD-CLASS-OBJECTS <class-name> . <key-values>)
(DRIVER-LOAD-CLASS-OBJECTS0 <class> . <key-values>)
[SUBR à 1 ou 2 arguments]

Construit et renvoie les objets jumeaux associés aux objets relationnels de classe principale de nom <class-name> (resp. l'objet <class>) dont les clés dans la base ont pour valeur <key-values> (liste des listes de valeurs correspondant à la liste des attributs composant la clé de la table principale associée à la classe). Si <class-name> est le nom d'une classe qui n'est pas principale, sa classe principale est

automatiquement considérée. Si <key-values> vaut t, tous les objets jumeaux correspondant à la classe principale précisée sont construits.

Quand un objet existe déjà dans l'environnement, le contenu de ses champs est perdu, écrasé par les valeurs issues de la base. De même, sa classe est éventuellement mise à jour.

Exemple:

```
? (DRIVER-LOAD-CLASS-OBJECTS 'Employe '((7499)))
= (allen)
? (DRIVER-GET-OBJECT-IF-LOADED 'Employe '(7499))
= allen
? (DRIVER-LOAD-CLASS-OBJECTS 'Departement t)
= (accounting research sales)
```

(DRIVER-OBJECT-LOADING-BEGIN <main-class>) **[SUBR à 1 argument]**

Cette fonction est appelée par DRIVER avant tout chargement en mémoire d'un groupe d'objets. La classe principale des objets est passé en argument. Par défaut, elle ne fait rien, sa définition est :

```
(de driver-object-loading-begin (mainclass))
```

Elle peut bien sûr être redéfinie par l'utilisateur.

(DRIVER-OBJECT-LOADING-END <main-class>) **[SUBR à 1 argument]**

Cette fonction est appelée par DRIVER après tout chargement en mémoire d'un groupe d'objets. La classe principale des objets est passé en argument. Par défaut, elle ne fait rien, sa définition est :

```
(de driver-object-loading-end (mainclass))
```

Elle peut bien sûr être redéfinie par l'utilisateur.

(DRIVER-OBJECT-LOADED <object> <class>) **[SUBR à 2 arguments]**

DRIVER envoie le message driver-object-loaded avec l'argument <class> à tout objet jumeau dont les champs propres à la classe <class> viennent d'être chargés.

L'utilisateur a donc tout loisir de définir des méthodes pour les classes persistantes du schéma. La méthode appelée par défaut est la fonction DRIVER-OBJECT-LOADED qui ne fait rien. DRIVER-OBJECT-LOADED peut bien sûr être redéfinie.

Exemple de méthode avec le modèle objet MicroCeyx :

```
(de {Employe}:driver-object-loaded (object class)
    (print "Employe " object
           "charge (Champs propres a la classe " class ")"))
```

(DRIVER-LOAD-DEEP-OBJECTS <objects>) **[SUBR à 1 argument]**

Construit les objets jumeaux des objets passés en arguments et tous les objets qu'ils référencent directement ou indirectement.

ATTENTION : Seuls les objets passés en argument sont rechargés (anciennes valeurs écrasées). Les objets référencés qui ont déjà été construits dans l'environnement ne sont pas rechargés mais seulement retrouvés en mémoire.

Exemple:

```
? (DRIVER-LOAD-DEEP-OBJECTS (LIST (DRIVER-GET-OBJECT 'Adresse '(7499))))
= (adresse-23)
```

(DRIVER-ALL-PERSISTENT-OBJECTS-IN-MEMORY) **[SUBR sans argument]**

Retourne la liste de tous les objets persistants référencés dans l'environnement, que ce soient des objets jumeaux ou des défauts d'objet.

Exemple :

```
? (DRIVER-ALL-PERSISTENT-OBJECTS-IN-MEMORY)
= ()
```

B.2.3 Manipulation explicite des défauts d'objet

Les fonctions DRIVER-GET-OBJECT(0) permettent de contruire le défaut d'objet d'un objet relationnel qui n'aurait pas encore de référence dans l'environnement.

(DRIVER-OBJECT-DEFAULT-CLASS) **[SUBR sans argument]**

Renvoie l'objet classe DRIVER des défauts d'objet. La construit si elle n'existe pas déjà.

Exemple:

```
? (DRIVER-OBJECT-DEFAULT-CLASS)
= driver-object-default
```

(DRIVER-OBJECT-DEFAULT-P <object-default>) **[SUBR à 1 argument]**

Ce prédicat est vrai et renvoie l'objet <object-default> si ce dernier est un défaut d'objet.

Exemple:

```
? (DRIVER-OBJECT-DEFAULT-P (DRIVER-GET-OBJECT 'Employe '(7698)))
= driver-object-default-17
```

(DRIVER-LOAD-OBJECT-DEFAULT <object-default>)

[SUBR à 1 argument]

Construit et renvoie l'objet jumeau associé au défaut d'objet <object-default>. L'objet jumeau se substitue au défaut d'objet partout où ce dernier était référencé. Si le défaut d'objet correspond à un objet qui est manquant, la fonction DRIVER-MISSING-OBJECT-PROCESSING est appelée.

Exemple:

```
? (DRIVER-LOAD-OBJECT-DEFAULT
?           (DRIVER-GET-OBJECT 'Employe '(7698)))
= blake
```

(DRIVER-MISSING-OBJECT-PROCESSING <default>)

[SUBR à 1 argument]

En cas de tentative malheureuse de chargement d'un défaut d'objet, la fonction DRIVER-MISSING-OBJECT-PROCESSING est appelée avec, en argument, le défaut d'objet incriminé. Par défaut, elle déclenche une erreur "Objet manquant" :

```
(de DRIVER-MISSING-OBJECT-PROCESSING (default)
    (error 'driver-missing-object-processing
           "Missing relational object" default))
```

Elle peut être redéfinie.

B.2.4 Accès à l'objet relationnel

(DRIVER-SEND <message> <object> . <args>)

[SUBR à 2 et plus arguments]

Envoie le message <message> à l'objet <object> avec les arguments <args>. Si <object> est un défaut d'objet, le jumeau est construit et se substitue à lui avant l'envoi de message.

Exemple :

```
? (DRIVER-GET-OBJECT 'Employe '(7839))
= driver-object-default-30
? (DRIVER-SEND 'salaire (DRIVER-GET-OBJECT 'Employe '(7839)))
= 5000.
? (DRIVER-GET-OBJECT 'Employe '(7839))
= king
```

(DRIVER-READ-FIELD-VALUE <field-name> <object>) [SUBR à 2 arguments]

Renvoie le contenu du champ <field-name> de l'objet <object>. Si ce dernier est encore un défaut d'objet, le jumeau est construit et se substitue à lui avant lecture du champ.

Exemple:

```
? (DRIVER-GET-OBJECT 'Employe '(7654))
= driver-object-default-18
? (DRIVER-READ-FIELD-VALUE 'salaire
?                (DRIVER-GET-OBJECT 'Employe '(7654)))
= 1250
? (DRIVER-GET-OBJECT 'Employe '(7654))
= martin
```

(DRIVER-SET-FIELD-VALUE <field-name> <object> <value>) [SUBR à 3 arguments]

Affecte le champ <field-name> de l'objet <object> avec la valeur <value>. Si ce dernier est encore un défaut d'objet, le jumeau est construit et se substitue à lui avant écriture du champ.

Exemple:

```
? (DRIVER-GET-OBJECT 'Employe '(7329))
= driver-object-default-14
? (DRIVER-SET-FIELD-VALUE 'salaire
?               (DRIVER-GET-OBJECT 'Employe '(7329)) 1500.)
= 1500.
? (DRIVER-GET-OBJECT 'Employe '(7329))
= smith
```

DRIVER-UPDATE-DAEMON-P **[Variable]**

Si la variable système DRIVER-UPDATE-DAEMON-P vaut t (la valeur par défaut est ()), DRIVER met à jour un objet jumeau s'il a connaissance par l'une des fonctions DRIVER-READ-FIELD-VALUE-IN-DATABASE(0) ou DRIVER-SET-FIELD-VALUE-IN-DATABASE(0) que l'objet relationnel correspondant a une valeur différente.

Pour plus de détails, consulter la description de ces fonctions.

(DRIVER-READ-FIELD-VALUE-IN-DATABASE <field-name> <object>)
(DRIVER-READ-FIELD-VALUE-IN-DATABASE0 <field> <object>)
[SUBR à 2 arguments]

Renvoie le contenu du champ <field-name> (resp. l'objet champ <field>) de l'objet <object> en allant directement lire la valeur dans la base de données. La lecture du champ ne provoque pas la mutation d'un défaut d'objet en jumeau.

Quand la variable DRIVER-UPDATE-DAEMON-P vaut t (la valeur par défaut est nil), si l'objet jumeau est déjà constitué et que la valeur du champ lue dans la base est différente de celle se trouvant dans l'objet, le champ en question est mis à jour dans l'objet jumeau.

Exemple:

```
? (DRIVER-GET-OBJECT 'Employe '(7655))
= driver-object-default-19
? (DRIVER-READ-FIELD-VALUE-IN-DATABASE 'salaire
?                (DRIVER-GET-OBJECT 'Employe '(7655)))
= 1350
? (DRIVER-GET-OBJECT 'Employe '(7655))
= driver-object-default-19
```

(DRIVER-SET-FIELD-VALUE-IN-DATABASE <field-name>
<object> <value>)

(DRIVER-SET-FIELD-VALUE-IN-DATABASE0 <field> <object> <value>)
[SUBR à 3 arguments]

Affecte le champ <field-name> (resp. l'objet champ <field>) de l'objet <object> avec la valeur <value> directement dans la base de données. L'écriture du champ ne provoque pas la mutation d'un défaut d'objet en jumeau.

On ne vérifie pas que la valeur affectée dans la base respecte les constraintes de classe de l'objet. Ainsi, une affectation dans la base peut provoquer la mutation d'un objet vers une nouvelle classe. La classe de l'objet relationnel devient alors différente de la classe de l'objet jumeau correspondant.

Quand la variable DRIVER-UPDATE-DAEMON-P vaut t (la valeur par défaut est nil), si l'objet jumeau est déjà constitué et que la valeur du champ écrite dans la base est différente de celle se trouvant dans l'objet, le champ en question est mis à jour dans l'objet jumeau.

Quand la variable DRIVER-MESSAGES-P vaut t (valeur par défaut), un message est imprimé dans le terminal quand DRIVER modifie la base de données.

ATTENTION Ces fonctions permettent d'aller directement écrire dans la base une information. Elles violent donc les règles qui régissent les transactions DRIVER. Leur utilisation doit donc être circonspecte et pertinente.

Exemple:

```
? (DRIVER-GET-OBJECT 'Employe '(7655))
= driver-object-default-19
? (DRIVER-SET-FIELD-VALUE-IN-DATABASE 'salaire
?               (DRIVER-GET-OBJECT 'Employe '(7655)) 1400.)
Mise-a-jour de l'objet driver-object-default-19
= 1400.
? (DRIVER-GET-OBJECT 'Employe '(7655))
= driver-object-default-19
? (DRIVER-READ-FIELD-VALUE-IN-DATABASE 'salaire
?                (DRIVER-GET-OBJECT 'Employe '(7655)))
= 1400.
```

(DRIVER-OBJECT-CLASS-IN-DATABASE <object>) [SUBR à 1 argument]

Renvoie la classe que l'objet <object> a dans la base de données. Cette classe peut être différente de celle de l'objet <object>.

Exemple:

```
? (DRIVER-OBJECT-CLASS-IN-DATABASE (DRIVER-GET-OBJECT 'Employe '(7655)))
= Vendeur
```

(DRIVER-OBJECT-CLASS <object>) [SUBR à 1 argument]

Renvoie la classe (persistante) de l'objet <object>. Si <object> est encore un défaut d'objet, consulte alors la classe de l'objet relationnel associé dans la base de données (fait appel à DRIVER-OBJECT-CLASS-IN-DATABASE pour cela).

Exemple:

```
? (DRIVER-OBJECT-CLASS (DRIVER-GET-OBJECT 'Employe '(7655)))
= Vendeur
```

(DRIVER-OBJECT-MAIN-CLASS <object>) **[SUBR à 1 argument]**

Renvoie la classe principale (persistante) de l'objet <object>.

Exemple:

```
? (DRIVER-OBJECT-MAIN-CLASS (DRIVER-GET-OBJECT 'Employe '(7655)))
= Employe
```

(DRIVER-CLASS-INSTANCE-P-IN-DATABASE <class-name> <object>)
(DRIVER-CLASS-INSTANCE-P-IN-DATABASE0 <class> <object>)
[SUBR à 2 arguments]

Teste si l'objet <object> est au moins instance de la classe de nom <class-name> (resp. l'objet <class>) dans la base de données.

Exemple:

```
? (DRIVER-GET-OBJECT 'Employe '(7655))
= driver-object-default-19
? (DRIVER-CLASS-INSTANCE-P-IN-DATABASE 'Vendeur
?               (DRIVER-GET-OBJECT 'Employe '(7655)))
= t
```

B.2.5 Écriture d'objets dans la base de données

Une transaction concerne un objet au moins, l'ensemble des objets de la couche virtuelle au plus. Elle débute suite à la connexion à la base ou après la validation de la transaction précédente.

(DRIVER-COMMIT-OBJECTS <objects> . <deep>)
[SUBR à 1 ou 2 arguments]

Valide la transaction courante. Si <objects> vaut t, tous les objets persistants de la couche virtuelle sont validés dans la base. Si <objects> est une liste d'objets, la validation ne s'applique à eux.

Quand une liste d'objets est précisée, deux nouvelles possibilités apparaissent, selon la valeur de l'argument *deep* :

- Si *deep* n'est pas précisé ou s'il vaut t (valeur par défaut) sont non seulement sauvés dans la base les objets indiqués, mais également l'ensemble des objets qu'ils référencent directement ou indirectement par ses champs persistants.
- Si *deep* vaut () ne sont sauvés que les objets précisés.

  Cependant, dans ce second cas, sont également écrits dans la base les objets non encore persistants directement référencés par l'un au moins des objets à valider. Si l'un d'entre eux référence lui-même un nouvel objet non persistant, ce dernier est également écrit dans la base, et ainsi de suite.

Quand une liste d'objets est précisée, elle peut comprendre des objets non encore persistants. L'application de DRIVER-COMMIT-OBJECTS sur ces objets les rend alors persistants.

La variable système DRIVER-WRITING-HELP oriente le traitement des erreurs de validation de transaction. Voir sa description ci-après.

La variable DRIVER-MESSAGE-P contrôle l'affichage de messages. Si elle vaut t (valeur par défaut), des messages sont affichés lorsque la base de données est modifiée.

Exemple:

```
? (DRIVER-COMMIT-OBJECTS (LIST (DRIVER-GET-OBJECT 'Employe '(7329))))
Mise-a-jour de l'objet smith
= (smith)
```

DRIVER-FIX-KEY-GENERATION-P [Variable]

Quand un objet doit être inséré dans la base (objet nouvellement persistant), DRIVER doit lui affecter des valeurs de clé pour toutes les tables élémentaires de sa classe. Quand un attribut de clé est de type numérique et si la variable système DRIVER-FIX-KEY-GENERATION-P vaut t (valeur par défaut), DRIVER interroge la base pour connaître la valeur maximale de cet attribut et donne la valeur $max+1$ à l'attribut pour l'objet. Dans le cas contraire, DRIVER demande une valeur à l'utilisateur.

DRIVER-WRITING-HELP [Variable]

La variable système DRIVER-WRITING-HELP oriente le traitement des erreurs de validation de transaction. Si elle vaut t (valeur par défaut), les erreurs sont analysées et des corrections sont proposées à l'utilisateur. Si elle vaut (), l'erreur est simplement déclenchée.

(DRIVER-ROLLBACK-OBJECTS <objects>) [SUBR à 1 argument]

Annule toute modification effectuée pendant la transaction sur les objets précisés. Si <objects> vaut t, l'annulation porte sur l'ensemble des objets persistants de la couche virtuelle. L'annulation est obtenue en retransformant en défauts d'objet tous les objets persistants concernés.

Exemple:

```
? (DRIVER-ROLLBACK-OBJECTS (LIST (DRIVER-GET-OBJECT 'Employe '(7329))))
= (smith)
```

B.2.6 Persistance des schémas de correspondances

Les schémas de correspondances sont persistants grâce à l'existence d'un méta schéma qui détermine la correspondance relationnels des classes système de DRIVER.

(DRIVER-METAMAPPING) **[SUBR sans argument]**

Renvoie le méta-schéma de correspondances.

Exemple:

```
? (DRIVER-METAMAPPING)
= driver-metamapping
```

(DRIVER-LOAD-MAPPING <mapping-name> . <overwritep>)
[SUBR à 1 ou 2 arguments]

Charge dans l'environnement le schéma de correspondances de nom <mapping-name>. Ce schéma doit être contenu dans les tables système DRIVER définies dans la base de données relationnelle.

En cas de tentative de rechargement d'un schéma, l'erreur "schéma déjà existant" est provoquée, sauf si <overwrite> est précisé à t. Dans ce cas, l'ancienne définition est perdue, écrasée par la nouvelle, ainsi que toutes les définitions qui y étaient rattachées (sauf son éventuel curseur). Toutes les structures qui le contenaient pointent maintenant sur la nouvelle.

Exemple:

```
? (DRIVER-LOAD-MAPPING 'map2)
= map2
```

(DRIVER-SAVE-MAPPING <mapping-name>)
(DRIVER-SAVE-MAPPING0 <mapping>) **[SUBR à 1 argument]**

Écrit le schéma de correspondances de nom <mapping-name> dans la base de données. L'écriture a lieu dans les tables système DRIVER définies dans la base de données relationnelle.

Exemple:

```
? (DRIVER-SAVE-MAPPING 'map1)
= map1
```

B.2.7 Interaction avec l'utilisateur

DRIVER-MESSAGES-P **[Variable]**

Booléen. Quand cette variable vaut t (valeur par défaut), DRIVER est autorisé à afficher des messages pour signaler toute action importante qu'il entreprend (modification d'un objet, mise-à-jour ou insertion de données relationnelles dans la base, etc...).

(DRIVER-PRINT <message>) **[SUBR à 1 argument]**

Cette fonction est utilisée par DRIVER pour afficher un message <message> destiné à l'utilisateur.

Elle peut être redéfinie.

(DRIVER-WARNING <message>) **[SUBR à 1 argument]**

Cette fonction est utilisée par DRIVER pour prévenir l'utilisateur d'un problème. La teneur de l'avertissement est précisée dans le message <message>.

Cette fonction peut être redéfinie.

(DRIVER-ASK-VALUE <message> <lvalues> <exit>)
[SUBR à 3 arguments]

Cette fonction est utilisée par DRIVER pour demander à l'utilisateur de lui fournir une valeur. La demande est exprimée dans le message <message>. Éventuellement, une liste de valeurs possibles <lvalues> peut être précisée; si toute valeur est autorisée, <lvalues> vaut (). Quand DRIVER autorise l'annulation de la demande, il fournit par <exit> un nom d'échappement à utiliser pour ce cas.

Cette fonction peut être redéfinie.

B.2.8 Divers

(DRIVER-ALL-LINKED-OBJECTS <objects>) **[SUBR à 1 argument]**

Renvoie la liste des objets référencés directement ou indirectement par l'ensemble d'objets <objects>.

Exemple:

```
? (DRIVER-GET-OBJECT 'Employe '(7654))
= martin
? (DRIVER-ALL-LINKED-OBJECTS (LIST (DRIVER-GET-OBJECT 'Employe '(7654))))
= (martin blake driver-object-default-10 sales driver-object-default-8
driver-object-default-3 driver-object-default-9 allen driver-object-default-5
driver-object-default-6 driver-object-default-17 driver-object-default-19
driver-object-default-20 driver-object-default-22 driver-object-default-23
driver-object-default-24 driver-object-default-25)
```

DRIVER-COMPACT-KEY-SEPARATOR **[Variable]**

Contient le code du caractère utilisé comme séparateur dans les clés compactes en interne à DRIVER. Par défaut, DRIVER-COMPACT-KEY-SEPARATOR vaut #// (code du caractère " / ").

La macro DRIVER-DEFCLASSMAP génère de nouveaux noms de tables logiques pour la correspondance de chaque classe afin d'obtenir des espaces de nom indépendants.

Lors de la description des correspondances de la classe `nom-class`, DRIVER-DEFCLASSMAP substitue à la table `nom-table` une nouvelle table de nom :

```
(CATENATE (DRIVER-CREATE-SYNONYM nom-class) '@ nom-table)
```

(DRIVER-CREATE-SYNONYM <symbol>) **[SUBR à 1 argument]**

Crée et renvoie un synonyme pour le symbole <symbol>

Exemple :

```
? (DRIVER-CREATE-SYNONYM 'toto)
= t119
```

(DRIVER-FIND-SYNONYM <symbol>) **[SUBR à 1 argument]**

Retrouve le synonyme défini pour le symbole <symbol>

Exemple :

```
? (DRIVER-FIND-SYNONYM 'toto)
= t119
```

(DRIVER-FIND-SYNONYM-KEY <synonym>) **[SUBR à 1 argument]**

Retrouve le symbole pour lequel a été généré le synonyme <synonym>.

Exemple :

```
? (DRIVER-FIND-SYNONYM-KEY 't119)
= toto
```

B.3 Filtrage des champs en lecture et écriture

Des filtres peuvent être définis pour modifier des données issues de la base avant leur affectation dans le champ d'un objet. De manière similaire, le contenu d'un champ peut être modifié avant d'être reporté dans la base de données. Ces filtres sont mis en place sous forme de méthodes qui sont invoquées lors des transferts de données.

Les méthodes **driverloading** sont invoquées pour filtrer une donnée issue de la base et destinée à être placée dans le champ d'un objet. Inversement, les méthodes **driverwriting** sont invoquées pour filtrer un contenu de champ à reporter dans la base de données.

Le filtre est défini lors de la création du champ avec l'une des fonctions DRIVER-DEFCLASS ou DRIVER-CREATE-FIELD en utilisant l'option *filter* dont la syntaxe est :

(filter <filter-name>)
ou
(filter (<filter-name> [class] [field] [object]))

Cette déclaration implique la définition par l'utilisateur des deux méthodes #:driver loading:<filter-name> et #:driverwriting:<filter-name> qui assureront le filtrage des valeurs entre le champ correspondant et la base de données.

Si le mot-clé filter introduit le seul symbole <filter-name>, les méthodes seront invoquées avec comme unique argument la valeur à filtrer. Si au contraire il introduit une liste de paramètres, le premier doit être le nom du filtre <filter-name> suivi d'un ou plusieurs mots réservés parmi *class*, *field* et *object*. Dans ce cas, les méthodes seront invoquées avec comme premier argument la valeur à filtrer, puis, selon les mots réservés choisis, l'objet DRIVER classe de l'instance à laquelle appartient le champ, l'objet DRIVER field correspondant au champ concerné, ou encore l'instance concernée elle-même. Le passage de ces différents objets avec la valeur à filtrer permet de tenir compte du contenu de certains de leurs champs pour déterminer la valeur renvoyée par le filtre.

La fonction DRIVER-FIELD-LOADING-ORDER permet d'établir l'ordre de chargement des champs d'un objet. Un intérêt de pouvoir choisir cet ordre peut être d'utiliser le contenu de certains champs de l'objet en cours de chargement – champs à charger en priorité donc – pour en construire d'autres. Un des exemples illustre cette possibilité.

Un certain nombre de types de champ atomiques ont des filtres qui leur sont associés par défaut. Ces types sont *symbol*, *float*, *fix*, *integer* et *string*. Il n'est donc pas nécessaire de redéfinir un filtre par exemple pour un champ atomique de type symbole associé à un attribut relationnel de type string. Bien entendu, l'utilisateur est libre de le faire s'il le souhaite.

Exemple:

```
? (DRIVER-ADD-FIELD-TYPE 'date)
```

```
= date
? (DRIVER-CREATE-FIELD 'Employe 'date_embauche 'atom 'date)
= date_embauche
? (DRIVER-MAP-FIELD 'Employe 'date_embauche '(emp bdate)
?                   () '(filter date1))
= t
? (de #:driverloading:date1 (datestr)
?     (build-date-from-string datestr)) ; user function call
= #:driverloading:date1
? (de #:driverwriting:date1 (date)
?     (build-date-string date)) ; user function call
= #:driverwriting:date1
? (DRIVER-ADD-FIELD-TYPE 'misc-type)
= misc-type
? (DRIVER-CREATE-FIELD 'Adresse 'datatype 'atom 'symbol)
= datatype
? (DRIVER-CREATE-FIELD 'Adresse 'data 'atom 'misc-type)
= data
? (DRIVER-MAP-FIELD 'Adresse 'datatype '(address datatype) ())
= t
? (DRIVER-MAP-FIELD 'Adresse 'data '(address userdata)
?                   '(filter (datatyping object)))
= t
? (de #:driverloading:datatyping (data obj)
?     (selectq (send 'datatype obj)
?              (symbol (symbol () data))
?              (float (convert-string-to-float data))
?              (t data)))
= #:driverloading:datatyping
? (de #:driverwriting:datatyping (data obj)
?     (string data))
= #:driverwriting:datatyping
```

DRIVER-SYSTEM-FIELD-FILTERS [Variable]

Contient la liste des filtres système reconnus par DRIVER. Ces filtres particuliers ne sont pas gérés à l'aide de méthodes driverloading et driverwriting.

Exemple :

```
? DRIVER-SYSTEM-FIELD-FILTERS
= (() integer fix float string symbol)
```

(DRIVER-FILTERS) [SUBR sans argument]

Renvoie la liste des déclarations de filtres effectuées dans le schema courant. Cette liste comprend les filtres système utilisés.

Exemple :

```
? (DRIVER-FILTERS)
= (symbol float fix string integer)
```

(DRIVER-FIND-FILTER-FIELDS <filter-name>) [SUBR à 1 argument]

Retourne la liste des champs utilisant le filtre de nom <filter-name>.

Exemple :

```
? (DRIVER-FIND-FILTER-FIELDS 'symbol)
= (nom prenom num-ss qualif tel-prive situ-famille nom sites nom
  ville etat plaque modele nom genre nom type)
```

(DRIVER-VALID-FIELD-FILTER <filter-decl>) [SUBR à 1 argument]

Vérifie que la déclaration de filtre <filter-decl> est correcte. Les deux méthodes `driverloading` et `driverwriting` doivent avoir été créées. En cas d'incorrection, une erreur est déclenchée.

Exemple :

```
? (DRIVER-VALID-FIELD-FILTER 'symbol)
= symbol
? (DRIVER-VALID-FIELD-FILTER '(symbol object))
** driver-valid-field-filter : Filter function arguments and filter
 declaration mismatch : (symbol object)
```

(DRIVER-FIELD-FILTER-VALIDATION <filter-name>) [SUBR à 1 argument]

Vérifie toutes les déclarations d'utilisation du filtre de nom <filter-name>. En cas d'incorrection, une erreur est déclenchée.

Exemple :

```
? (DRIVER-FIELD-FILTER-VALIDATION 'symbol)
= symbol
```

(DRIVER-LOADING-TYPE-FILTER <field> <object> <value>)
[SUBR à 3 arguments]

Cette fonction fait passer la valeur <value> dans le filtre en lecture éventuellement défini pour le champ <field>. La valeur qu'elle retourne est ensuite affectée dans le champ de l'objet <object> en cours de chargement.

C'est elle qui gère l'application des filtres système ou des méthodes driverloading. Il est possible mais déconseillé de la redéfinir.

(DRIVER-SAVING-TYPE-FILTER <attr> <field> <object> <value>)
[SUBR à 4 arguments]

Cette fonction fait passer la valeur <value> dans le filtre en écriture éventuellement défini pour le champ <field>. La valeur qu'elle retourne est ensuite traitée pour écriture dans la base de données. Quand le champ a pour correspondance un attribut, <attr> est cet attribut, sinon elle vaut (). <object> est l'objet en cours d'écriture.

C'est cette fonction qui gère l'application des filtres système ou des méthodes driverwriting. Il est possible mais déconseillé de la redéfinir.

B.4 Utilisation d'un nouveau modèle objet

DRIVER peut être utilisé avec de nombreux modèles objets. La description du modèle choisi s'effectue en définissant un certain nombre de méthodes qui permettront au système de créer une classe, une instance, d'affecter un champ, etc.

B.4.1 Manipulation des modèles objet

Un modèle objet {USER-OBJECT-MODEL} à utiliser doit être déclaré sous forme d'une classe MicroCeyx, sous-classe de Driver-Object-model, classe racine des modèles objet.

Exemple avec SMECI :

```
(DEFTCLASS {Driver-Object-model}:Smeci-object-model)
(DE {Smeci-object-model}:prin (o) (prin "Smeci-object-model"))
```

DRIVER-ROOT-OBJECT-MODEL **[Variable]**

Cette variable contient un objet instance de la classe Driver-Object-model, classe racine des modèles objet. Elle ne doit pas être modifiée, sous peine de forte perturbation du système.

DRIVER-MICROCEYX-OBJECT-MODEL **[Variable]**

Cette variable contient un objet instance de la classe MicroCeyx, sous-classe de Driver-Object-model. Elle ne doit pas être modifiée, sous peine de forte perturbation du système.

DRIVER-USER-OBJECT-MODEL **[Variable]**

Cette variable doit contenir le modèle objet client courant. Elle est consultée à chaque création de schéma de correspondances pour déterminer de quel modèle seront les futurs objets à charger de la base correspondante. Par défaut, DRIVER-USER-OBJECT-MODEL contient un objet instance de la classe *MicroCeyx*. MicroCeyx est donc le modèle objet utilisé par défaut.

Exemple avec SMECI :

```
(setq DRIVER-USER-OBJECT-MODEL (OMAKEQ {Smeci-object-model}))
```

(DRIVER-CURRENT-OBJECT-MODEL) **[SUBR sans argument]**

Renvoie le modèle objet courant. Ce modèle courant est le modèle du schéma courant, s'il y en a un, ou le modèle référencé par DRIVER-USER-OBJECT-MODEL dans le cas contraire. C'est le modèle utilisé par DRIVER pour toutes ses opérations effectuées sur les objets client.

Exemple avec SMECI :

```
? (DRIVER-CURRENT-OBJECT-MODEL)
= Smeci-object-model
```

B.4.2 L'interface fonctionnelle objet

Les exemples de définition de méthode sont présentées pour une utilisation de DRIVER avec SMECI. La définition de l'interface fonctionnelle pour Microceyx est donnée en annexe.

({USER-OBJECT-MODEL}:DEFINE-CLASS <o-model> <class>) **[SUBR à 2 arguments]**

Cette fonction doit créer la classe correspondant à la classe DRIVER <class> dans le modèle objet USER-OBJECT-MODEL.

Exemple avec SMECI (La méthode complète, notamment les fonctions :smeci-field-def et :class-fields-to-define sont présentées en annexe §B.8) :

```
(de {Smeci-object-model}:define-class (object-model class)
    (tag bad-allocation
         (:define-class %class))
    t)

(de :define-class (class)
    (if (S-category-p (:class-name class))
        (S-kill-category (:class-name class)))
    (apply 'defScategory
           (mcons (:class-name class)
                  (or (if (send 'super class)
                          (send 'name (send 'super class)))
                      'S.Object)
                  (mapcar ':smeci-field-def
                          (:class-fields-to-define class)
                          (cirlist class))))
    t)

(de :class-name (class)
    (if (eq (send 'name class) 'driver-object-default)
        'S.Object-default
        (send 'name class)))
```

({USER-OBJECT-MODEL}:EXISTING-CLASS-P <o-model> <class>)
[SUBR à 2 arguments]

Méthode devant déterminer si la classe correspondant à la classe DRIVER <class> existe dans le modèle objet USER-OBJECT-MODEL.

Exemple avec SMECI :

```
(de {Smeci-object-model}:existing-class-p (object-model class)
    (S-category-p (:class-name class)))
```

({USER-OBJECT-MODEL}:OMAKE <o-model> <class>)
[SUBR à 2 arguments]

Méthode devant créer et renvoyer une nouvelle instance de la classe correspondant à la classe DRIVER <class> dans le modèle objet USER-OBJECT-MODEL.

Exemple avec SMECI :

```
(de {Smeci-object-model}:omake (object-model class)
    (S-create-object (:class-name class)))
```

({USER-OBJECT-MODEL}:READ-FIELD-VALUE <o-model>
<message> <object>)
[SUBR à 3 arguments]

Fonction devant retourner le contenu du champ <field-name> de l'objet <object>.

Exemple avec SMECI :

```
(de {Smeci-object-model}:read-field-value (object-model message object)
    (S-get-value message object))
```

({USER-OBJECT-MODEL}:SET-FIELD-VALUE <o-model> <message>
<object> <value>)
[SUBR à 4 arguments]

Fonction devant affecter le contenu du champ <field-name> de l'objet <object> avec la valeur <value>.

Exemple avec SMECI :

```
(de {Smeci-object-model}:set-field-value
                            (object-model message object value)
    (ifn (and (memq message '(name nom))
              (eq ({Smeci-object-model}:read-field-value
                                    object-model message object)
                  value))
         (S-set-value message object value))
    value)
```

({USER-OBJECT-MODEL}:SEND-MESSAGE <o-model> <message> <object> <args>)
[SUBR à 4 arguments]

Méthode devant envoyer le message <message> à l'objet <object> avec les arguments <args> dans le modèle objet USER-OBJECT-MODEL.

Exemple avec SMECI :

```
(de {Smeci-object-model}:send-message
                            (object-model message object args)
    (apply 'send (mcons message object args)))
```

({USER-OBJECT-MODEL}:CLASS-INSTANCE-P <o-model> <object> <class>)
[SUBR à 3 arguments]

Prédicat devant déterminer si l'objet <object> est instance de la classe correspondant à la classe DRIVER <class> dans le modèle objet USER-OBJECT-MODEL.

Exemple avec SMECI :

```
(de {Smeci-object-model}:class-instance-p (object-model object class)
    (S-isa-p object (:class-name class)))
```

({USER-OBJECT-MODEL}:CHANGE-OBJECT-CLASS <o-model> <newclass> <object>)
[SUBR à 3 arguments]

Fonction devant faire transiter l'objet <object> de son ancienne classe vers la classe correspondant à la classe DRIVER <newclass> dans le modèle objet USER-OBJECT-MODEL.

Exemple avec SMECI :

```
(de {Smeci-object-model}:change-object-class
                              (object-model newclass object)
    (if (or (driver-object-default-p object)
            (eq newclass (driver-object-default-class)))
        (S-change-category object 'S.Object))
    (S-change-category object (send 'name newclass)))
```

({USER-OBJECT-MODEL}:ALL-CLASS-OBJECTS <o-model> <class>) [SUBR à 2 arguments]

Fonction devant renvoyer l'ensemble des instances de la classe correspondant à la classe DRIVER <class> dans le modèle objet USER-OBJECT-MODEL.

Exemple avec SMECI :

```
(de {Smeci-object-model}:all-class-objects (class)
    (S-all-objects (:class-name class)))
```

B.5 Exemple de base relationnelle, smecidemo

Cette base relationnelle est celle utilisée dans l'exemple de session d'utilisation présenté ci-après. Elle a été définie sous INGRES Version 6.3/02 sur sun3 et sun4.

Sont présentées d'abord les structures des relations. Suivent ensuite leurs données, l'ensemble étant disposé sous forme de tableaux.

emp		Table des employés
empno	entier	Numéro de l'employé, clé de la relation emp
ename	20 caractères	Nom de l'employé
fname	20 caractères	Prénom
job	20 caractères	Emploi dans l'entreprise
mgr	entier	Numéro du chef de l'employé
sal	réel	Salaire
com	réel	Commission, pour certains employés
deptn	entier	Numéro du département de l'employé
responsible_for	20 caractères	référence d'un objet

person		Table des personnes
pname	20 caractères	Nom de la personne
fname	20 caractères	Prénom, nom et prénom forme la clé de la relation
ssnum	15 caractères	Numéro de sécurité sociale
position	10 caractères	Situation de famille
husband_wife	10 caractères	Prénom du conjoint (si marié)
phone	12 caractères	Numéro de téléphone
car	entier	Numéro de la voiture de la personne

dept		Table des départements
deptno	entier	Numéro de département, clé de la relation
dname	20 caractères	Nom du département

deptsite		Tables des sites des départements
deptno	entier	Numéro de département, clé de la relation
sname	20 caractères	Localisation

salgrade		Table donnant le grade en fonction du salaire
grade	entier	grade (nombre), clé de la relation
losal	entier	salaire inférieur
hisal	entier	salaire supérieur

project		Table des projets de recherche
projno	entier	Numéro de projet, clé de la relation
pname	20 caractères	Nom du projet
budget	entier	Son budget

emproj		Table gérant la relation n-m entre employés et projets, un projet étant constitués d'un certain nombre d'employés, et un employé pouvant appartenir à plusieurs projets
projno	entier	Numéro du projet
empno	entier	Numéro de l'employé
		projno et empno forment la clé de la relation

address		Table des adresses
empno	entier	Numéro de l'employé de l'adresse en question, clé de la relation
num	entier	Numéro dans la rue
street	20 caractères	Nom de la rue
zip	entier	Code postal
city	12 caractères	Nom de la ville
state	5 caractères	Nom de l'état

vehicle		Table des véhicules
vnum	entier	Numéro de véhicule, clé de la relation
model	15 caractères	Modèle du véhicule
licence	15 caractères	Immatriculation. valeurs uniques
year	entier	Année de construction

defproducts		Tables des produits gérés par les départements
deptno	entier	Numéro de département
pref	20 caractères	Référence d'objet produit

software		Tables des logiciels développé par la société
softnum	entier	Numéro de logiciel
sname	20 caractères	Nom du logiciel
category	20 caractères	Type de logiciel
year1	entier	Date de début de développement
s_year	entier	Date de commercialisation

hardware		Table du matériel développé par la société
hardnum	entier	Numéro de matériel
hname	20 caractères	Nom du produit
type	20 caractères	Type de matériel

Table **emp**

empno	ename	fname	job	mgr	sal	com	deptn	responsible_for
7839	king	paul	president		5000.		10	project/102
7566	jones	eric	manager	7839	2975.		20	dept/20
7698	blake	harold	manager	7839	2850.		30	dept/30
7782	clark	john	manager	7839	2450.		10	dept/10
7902	ford	john	analyst	7566	3000.		20	project/101
7788	scott	pit	analyst	7566	3000.		20	project/103
7499	allen	jack	salesman	7698	1600.	300.	30	software/743
7521	ward	peter	salesman	7698	1250.	1400.	30	hardware/53
7654	martin	georges	salesman	7698	1250.	1400.	30	
7655	james	peter	salesman	7698	1350.	1000.	30	
7934	miller	paul	clerk	7782	1300.		10	
7329	smith	john	clerk	7902	800.		20	

Table **person**

pname	fname	ssnum	position	husband_wife	phone	car
king	paul	A34F4	married	anita	40-223233	1232
jones	eric	B7C123	married		40-783539	1975
scott	helen	NJGOFBE	married	pit	44-003231	1928
king	anita	NVJS5	married	paul	40-223233	
blake	harold	A473SE	celibate		44-183211	1429
clark	john	462SQ1	married	laura	40-893723	1230
clark	laura	G35H89	married	john	40-893756	
miller	rita	9043TU	married	paul	43-127772	
ford	laura	J9845G2	married	john	40-374845	4328
ford	john	6D3210	married	laura	40-374845	4328
scott	pit	A435C	married	helen	44-003231	1928
allen	jack	76373Y	celibate		78-383726	2004
ward	peter	128347	married	susie	40-378467	1253
ward	suzie	OIRGNE	married	peter	40-378502	
james	peter	K4G262	separate		40-449323	
martin	georges	234FS1	celibate		40-276951	3005
miller	paul	95Z0H5	married	rita	43-127772	1276
smith	john	J3847Y2	celibate		44-332047	1550

Table **dept**	
deptno	dname
10	accounting
20	research
30	sales

Table **deptsite**	
deptno	sname
10	New-York
20	Boston
30	Chicago
10	Chicago
30	Los-Angeles
30	Austin

Table **deptproducts**	
deptno	pref
20	software/743
30	software/874
30	software/743
30	hardware/53

Table **salgrade**		
grade	losal	hisal
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

Table **project**		
projno	pname	budget
102	beta	175000
103	gamma	95000
101	alpha	250000

Table **emproj**	
projno	empno
101	7566
101	7788
101	7902
101	7329
101	7839
102	7698
102	7521
102	7654
102	7499
102	7566
103	7782
103	7934
103	7566

Table **software**				
softnum	sname	category	year1	s_year
743	Pay1	pay	1974	1975
874	Accountancy1	book-keeping	1975	1977

Table **hardware**		
hardnum	hname	type
53	hbd123	computer

Table **address**					
empno	num	street	zip	city	state
7839	345	5th avenue		New-York	NY
7566	34	Maple street		Boston	MA
7698	12	Vermont street		Chicago	
7782	125	86th street		New-York	NY
7902					
7788					
7521					
7654					
7655					
7934	24	56th street		Rego Park	NY
7329	3	Highgate road		Boston	MA
7499					

Table **vehicle**			
vnum	model	licence	year
1232	Ford-1300	FD3840	1977
1975	Chryler-303	HG8943	1973
1429	Peugeot-205	PO2354	1989
1230	Ford-1300	LD7646	
4328	Ekton-730	HY4738	
1928	Peugeot-305	AW8593	1985
2004	BMW-730	UU4853	1990
1253	Toyota-1900	PV3824	
3005	Ford-1300	GH4500	
1276	Chrysler-303	KJ4753	
1550	Toyota-1800	XD4007	

B.6 Exemple de schéma de correspondances

Ceci est un exemple de schéma de correspondances établi pour l'exploitation de la base de données smecidemo.

Ce schéma est contenu dans le fichier `example.ll`.

```
; ------------------------------------------------------
; Mapping Lisp et Bases de donnees relationnelles
; example.ll - Exemple de schema de correspondances
; INRIA/CERMICS - 17 juillet 1991
; ------------------------------------------------------

;------------------------------------------------------
; Name           : Mapping creation
;------------------------------------------------------

(driver-create-mapping 'map1 t)

;------------------------------------------------------
; Name           : Table definitions
;------------------------------------------------------

(driver-deftable emp
   (empno           integer 2 keypart)
   (ename           string 20 not-null)
   (fname           string 20 ())
   (job             string 20 ())
   (mgr             integer 2 ())
   (sal             float   8 ())
   (com             float   8 ())
   (deptn           integer 2 ())
   (responsible_for string 20 ()))

(driver-deftable person
   (pname    string 20 keypart)
   (fname    string 20 keypart)
   (ssnum    string 15 unique)
   (position string 10 ())
   (phone    string 12 ())
   (car      integer 2 ()))

(driver-deftable salgrade
   (grade integer 2 keypart)
   (losal integer 2 ())
   (hisal integer 2 ()))

(driver-deftable dept
   (deptno integer 2 keypart)
   (dname  string 20 ()))
```

```
(driver-deftable deptsite
   (dept  integer 2 keypart)
   (sname string 20 keypart))

(driver-deftable deptproducts
   (deptno integer  2 keypart)
   (pref   string  20 keypart))

(driver-deftable project
   (projno integer 2 keypart)
   (pname  string 20 ())
   (budget integer 2 ()))

(driver-deftable emproj
   (projno integer 2 keypart)
   (empno  integer 2 keypart))

(driver-deftable address
   (empno  integer 2 keypart)
   (num    integer 2 ())
   (street string 20 ())
   (zip    integer 2 ())
   (city   string 12 ())
   (state  string  5 ()))

(driver-deftable vehicle
   (vnum    integer 2 keypart)
   (model   string 15 ())
   (licence string 15 unique)
   (year    integer 2 ()))

(driver-deftable software
   (softnum  integer  2 keypart)
   (sname    string  20 ())
   (category string  20 ())
   (year1    integer  2 ())
   (s_year   integer  2 ()))

(driver-deftable hardware
   (hardnum integer  2 keypart)
   (hname   string  20 ())
   (type    string  20 ()))


;-----------------------------------------------------
; Name           : Class descriptions
;-----------------------------------------------------

(driver-defclass Employe ()
   (nom       atom   symbol (localp ()))
```

```
   (prenom    atom   symbol)
   (num-ss    atom   symbol)
   (qualif    atom   symbol)
   (chef      object Employe)
   (responsable-de object2)
   (salaire   atom   float
        (constraint (lambda (s) (and (>= s 700.) (<= s 9999.)))))
   (grade     atom   fix (initval 0))
   (dpt       object Departement)
   (adresse   object Adresse)
   (tel-prive atom   symbol)
   (vehicule  object Vehicule))

(driver-defclass Vendeur Employe
   (qualif (constraint (lambda (qual) (eq qual 'salesman))))
   (situ-famille atom      symbol)
   (commission   atom      float)
   (sal-total    monotexpr float))

(driver-defclass Cadre Employe
   (qualif (constraint (lambda (q) (memq q '(manager president))))))

(driver-defclass President Cadre
   (qualif (constraint (lambda (q) (eq q 'president)))))

(driver-defclass Chef-de-service Cadre
   (qualif (constraint (lambda (q) (eq q 'manager)))))

(driver-defclass Departement ()
   (nom       atom       symbol (localp ()))
   (produits  object2set)
   (sites     atomset    symbol)
   (chefs     ordobjset  Cadre)
   (personnel objectset  Employe)
   (moy-sal   multitexpr float))

(driver-defclass Projet ()
   (nom       atom       symbol (localp ()))
   (budget    atom       float)
   (employes  ordobjset  Employe)
   (sal-total multitexpr float))

(driver-defclass Adresse ()
   (numero atom fix)
   (rue atom string)
   (ville atom symbol)
   (etat  atom symbol))

(driver-defclass Vehicule ()
   (plaque atom symbol)
```

```
   (modele atom symbol))

(driver-defclass Software ()
   (nom         atom symbol (localp ()))
   (genre       atom symbol)
   (annee1      atom integer)
   (annee-vente atom integer))

(driver-defclass Hardware ()
   (nom  atom symbol (localp ()))
   (type atom symbol))

;-----------------------------------------------------
; Name           : Map definitions
;-----------------------------------------------------

(driver-defclassmap Employe emp
   (letjoins
       ((j1 (emp (p person)
             ((lambda (a1 a2 a3 a4)
                 (and (eq a1 a3) (eq a2 a4)))
              (emp ename) (emp fname) (p pname) (p fname))))
        (j2 (emp (emp1 emp)
             ((lambda (a1 a2) (eq a1 a2))
                (emp mgr) (emp1 empno))))
        (j3 (emp (sg salgrade)
             ((lambda (a1 a2 a3)
                (and (>= a1 a2) (<= a1 a3)))
              (emp sal) (sg losal) (sg hisal))))
        (j4 (emp dept
             ((lambda (a1 a2) (eq a1 a2))
              (emp deptn) (dept deptno))))
        (j5 (emp address
             ((lambda (a1 a2) (eq a1 a2))
              (emp empno) (address empno))))
        (j6 (p vehicle
             ((lambda (a1 a2) (eq a1 a2))
              (p car) (vehicle vnum)))))
       (fields (nom            (emp ename)           () (readonly t))
               (prenom         (emp fname)           () (readonly t))
               (num-ss         (p ssnum)             (j1))
               (qualif         (emp job)             ())
               (chef           ()                    (j2))
               (responsable-de (emp responsible_for) ())
               (salaire        (emp sal)             ())
               (grade          (sg grade)            (j3) (readonly t))
               (dpt            ()                    (j4))
               (adresse        ()                    (j5))
               (tel-prive      (p phone)             (j1))
               (vehicule       ()                    (j1 j6)))))
```

```
(driver-defclassmap Vendeur
   (fields (situ-famille (p position)      ((Employe emp p)))
           (commission   (emp com) ())
           (sal-total    ((lambda (a1 a2) (+ a1 a2))
                          (emp sal) (emp com)) ())))

(driver-defclassmap Departement dept
   (letjoins
       ((j1 (dept (e emp)
            ((lambda (attr1 attr2) (eq attr1 attr2))
             (dept deptno) (e deptn))))
        (j2 (dept (ds deptsite)
            ((lambda (a b) (eq a b))
             (dept deptno) (ds dept))))
        (j3 (dept (dp deptproducts)
            ((lambda (a b) (eq a b))
             (dept deptno) (dp deptno)))))
       (attrconstraint ((lambda (no) (> no 0)) (dept deptno))
                       ())
       (fields (nom       (dept dname)   ())
               (produits  (dp pref)      (j3))
               (sites     (ds sname)     (j2))
               (chefs     ()             (j1)
                  (orders ((e sal) desc)
                          ((e ename) asc))
                  (readonly t))
               (personnel ()             (j1) (readonly t))
               (moy-sal   ((lambda (a1) (avg a1)) (e sal))
                                         (j1)))))

(driver-defclassmap Projet project
   (letjoins
       ((j1 (project emp
            ((lambda (a1 a2 a3 a4)
               (and (eq a1 a2) (eq a3 a4)))
             (project projno) (emproj projno)
             (emproj empno) (emp empno)))))
       (fields (nom       (project pname)  ())
               (budget    (project budget) ())
               (employes  ()               (j1) (orders ((emp empno) asc)))
               (sal-total ((lambda (a) (sum a)) (emp sal)) (j1)))))

(driver-defclassmap Adresse address
   (fields (numero (address num)    ())
           (rue    (address street) ())
           (ville  (address city)   ())
           (etat   (address state)  ())))

(driver-defclassmap Vehicule vehicle
```

```
   (fields (plaque (vehicle licence) ())
           (modele (vehicle model)  ()))))

(driver-defclassmap Software software
   (fields (nom         (software sname)    ())
           (genre       (software category) ())
           (annee1      (software year1)    ())
           (annee-vente (software s_year)   ())))

(driver-defclassmap Hardware hardware
   (fields (nom  (hardware hname) ())
           (type (hardware type)  ())))


;-----------------------------------------------------
; Name           : Mapping compilation
;-----------------------------------------------------

(driver-compile-mapping 'map1)
```

B.7 Exemple de session d'utilisation

Cette exemple de session d'utilisation est effectuée avec le SGBD Ingres v6. La base utilisée est `smecidemo` dont les relations et leurs contenus sont donnés en annexe (cf. §B.5).

Le schéma de correspondances utilisé est `map1`, présenté précédemment. Il est contenu dans le fichier `example.ll` (cf. §B.6).

Après chargement d'un environnement Le_lisp (lelisp, aida, smeci, ...), ...

```
; Le-Lisp (by INRIA) version 15.24 ( 2/Janv/91)   [sun]
; Systeme Complice : mer 13 mars 91 10:48:22
= (31bitfloats abbrev callext compiler complice date debug defstruct
loader mc68881 messages microceyx pathname pepe pretty setf
virbitmap virtty)
?
?
? ; On se place dans le directory ou se trouve driver.ll
? !cd /u/crazy/0/secoia/lebastar/driver
= t
? ^Ldriver
Chargement d'ASQUELL...
Chargement d'ASQUELL oK
Chargement de DRIVER...
DRIVER v1.35
Chargement de DRIVER oK
= driver.ll
?
?
?
? ; Connexion a la base smecidemo
? (driver-connect 'ingres 'smecidemo)
= #:tclass:cursor:#[ingres 0 t ()]
?
? ; Chargement et definition d'un schema de correspondances
? ^Lexample
= example.ll
?
?
? (driver-all-mappings)
= (map1)
? (driver-current-mapping)
= map1
?
? (driver-existent-key-p 'Employe '(7698))
= t
? (driver-get-object 'Employe '(7698))
```

```
= driver-object-default-1
? (driver-get-object-if-loaded 'Employe '(7698))
= ()
? (driver-object-default-p
?           (driver-get-object 'Employe '(7698)))
= driver-object-default-1
? (driver-load-object-default
?           (driver-get-object 'Employe '(7698)))
= blake
? (driver-get-object 'Employe '(7698))
= blake
? (driver-get-object-if-loaded 'Employe '(7698))
= blake
?
? (driver-load-class-objects 'Departement t)
= (accounting research sales)
? (driver-get-object 'Employe '(7499))
= driver-object-default-16
? (driver-object-class-in-database (driver-get-object 'Employe '(7499)))
= Vendeur
? (driver-read-field-value-in-database 'commission
?                (driver-get-object 'Employe '(7499)))
= 400.
? (driver-set-field-value-in-database 'commission
?                (driver-get-object 'Employe '(7499)) 500.)
Mise-a-jour de l'objet S.Object-default-16
= 500.
? (driver-get-object 'Employe '(7499))
= driver-object-default-16
? (driver-read-field-value 'salaire
?                (driver-get-object 'Employe '(7499)))
= 1600.
? (driver-get-object 'Employe '(7499))
= allen
?
?
? ; Filtrer les employes parmi tous les tuples de emp
? (driver-select-objects '((lambda (obj) t) Employe))
= (((7329)) ((7499)) ((7521)) ((7566)) ((7654)) ((7655))
 ((7698)) ((7782)) ((7788)) ((7839)) ((7902)) ((7934)))
?
? ; Filtrer les vendeurs parmi tous les tuples de emp
? (driver-select-objects '((lambda (obj) t) Vendeur))
= (((7499)) ((7521)) ((7654)) ((7655)))
```

```
?
? ; Filtrer les employes qui ont une voiture de meme modele
? ; que leur chef
? (driver-select-objects
?    '((lambda (obj)
?         (eq (send 'modele (send 'vehicule obj))
?             (send 'modele (send 'vehicule (send 'chef obj)))))
?      Employe))
= (((7782)))
?
?
?
? (end)
Que Le_lisp soit avec vous.
```

B.8 Exemple de prise en compte d'un modèle objet

Voici un exemple de prise en compte du modèle objet de SMECI. Le listing suivant est en fait le contenu du fichier `smecidriver.ll`.

```
; ----------------------------------------------------------------------------
; Mapping Lisp et Bases de donnees relationnelles
; smecidriver.ll - Definition des methodes objet pour utilisation de Smeci V1.5
; INRIA/CERMICS - 4 septembre 1989 - Maj 21 avril 1992
; ----------------------------------------------------------------------------

(setq #:sys-package:colon 'smecidriver)

(eval-when (load eval)
(add-feature 'smecidriver)
)

(eval-when (load eval compile)
(deftclass {Driver-Object-model}:Smeci-object-model)
)

(eval-when (load eval compile)
(defvar driver-user-object-model (omakeq {Smeci-object-model}))
(defvar :dbid-sep #/@)
)

(de {Smeci-object-model}:prin (o) (prin "Smeci-object-model"))


;-----------------------------------------------------
; Name           : {Smeci-object-model}:define-class
; Date           : 09/04/89
; Author         : Franck LEBASTARD
; Arguments      : Driver-class instance
; Result         : t
; Description    : Creates Smeci categorie and declare it in the system.
; Side effects   : Does deftclass
; Export         : Yes (method)
; Document       : No
; Modified date  : No
;-----------------------------------------------------

(de {Smeci-object-model}:define-class (object-model class)
    (tag bad-allocation
         (:define-class class))
    t)

(de :define-class (class)
    (if (S-category-p (:class-name class))
        (S-kill-category (:class-name class)))
```

```
    (apply 'defScategory
           (mcons (:class-name class)
                  (or (if (ogytq Driver-class super class)
                          (:class-name (ogytq Driver-class super class)))
                      'S.Object)
                  (mapcar ':smeci-field-def
                          (:class-fields-to-define class)
                          (cirlist class)))))

(de :class-name (class)
    (if (eq (ogytq Driver-class name class) 'driver-object-default)
        'S.Object-default
        (ogytq Driver-class name class)))

(de :class-fields-to-define (class)
    (let ((fields (mapcan (lambda (field)
                            (and (ogytq Driver-field localp field)
                                 (list field)))
                          (ogytq Driver-class fields class))))
         (mapc (lambda (field)
                 (if (and (driver-field-p field)
                          (not (memq field fields)))
                     (setq fields (nconc1 fields field))))
               (mapcar 'car (ogytq Driver-class restrictions class)))
         fields))

(de :redefine-category-tree (main-class)
    (mapc '{Smeci-object-model}:define-class
          (cirlist ())
          (cons main-class (driver-class-subclasses0 main-class))))

;-------------------------------------------------------
; Name           : :smeci-field-def
; Date           : 09/04/89
; Author         : Franck LEBASTARD
; Arguments      : Driver-field object, Driver-class object
; Result         : Symbolic smeci field definition for creer-categorie funct.
; Description    : Calculates every field slot
; Side effects   : No
; Export         : No
; Document       : No
; Modified date  : No
;-------------------------------------------------------

; Les facettes inverse import unite type-lien extension ne sont pas remplies.
(de :smeci-field-def (field class)
    (mapcan (lambda (facette)
              (tag smeci-field-def
                   (list facette (send facette field class))))
            '(nom-ch allocation type domain range force-test?)))
```

```
;-----------------------------------------------------
; Name           : nom-ch defini type domain range
; Date           : 09/04/89
; Author         : Franck LEBASTARD
; Arguments      : Driver-field object, Driver-class object
; Result         : slot value
; Description    : Calculate field slots
; Side effects   : No
; Export         : Yes (methods)
; Document       : No
; Modified date  : No
;-----------------------------------------------------

(de {Driver-field}:nom-ch (field class)
    (exit smeci-field-def (list (ogytq Driver-field name field))))

; Warning : the p type can't be deducted with the only defined informations
(de {Driver-field}:allocation (field class)
    (ifn (memq field (ogytq Driver-class fields class))
         (exit smeci-field-def ())
         (if (:any-constraint-on-field class field)
             'constrained 'instance)))

(de :any-constraint-on-field (class field)
    (car (any (lambda (class1)
                 (driver-find-field-constraints-on-class0 class1 field))
              (cons class (driver-class-subclasses0 class)))))

(de {Object-field}:allocation (field class) 'instance)
(de {Object-set}:allocation (field class) 'instance)

(de {Atom-field}:type (field class)
    (ifn (memq field (ogytq Driver-class fields class))
         (exit smeci-field-def ())
         (let ((rest (:any-constraint-on-field class field)))
              (:smeci-type (ogytq Driver-field ftype field)
                           (and rest
                                (send 'domain-type
                                      (ogytq Driver-join0 operator rest)))))))

(de {Atom-set}:type (field class)
    (ifn (memq field (ogytq Driver-class fields class))
         (exit smeci-field-def ())
         (:smeci-type (ogytq Driver-field ftype field) 'set)))

(de {Object-field}:type (field class)
    (:smeci-type 'object ()))

(de {Object-set}:type (field class)
```

```
    (:smeci-type 'object 'set))

(de {Computed-field}:type (field class)
    (ifn (memq field (ogytq Driver-class fields class))
         (exit smeci-field-def ())
         (:smeci-type (ogytq Driver-field ftype field) ())))

(de :smeci-type (type1 type2)
    (selectq type1
             (string
               (selectq type2
                        (set 'l*)
                        (t '*)))
             (symbol
               (selectq type2
                        (enum 'enum-symbol)
                        (bounded 'bounded-symbol)
                        (set 'lsymbol)
                        (t 'symbol)))
             ((fix integer)
               (selectq type2
                        (enum 'enum-fix)
                        (bounded 'bounded-fix)
                        (set 'lfix)
                        (t 'fix)))
             (float
               (selectq type2
                        (enum 'enum-float)
                        (bounded 'bounded-float)
                        (set 'lfloat)
                        (t 'float)))
             (object
               (selectq type2
                        (set 'lobject)
                        (t 'object)))
             (()
               (selectq type2
                        (set 'l*)
                        (t '*)))
             (t (print "Not a good type : " type1) ())))

(de {Driver-field}:domain (field class) (exit smeci-field-def ()))
(de {Object-field}:domain (field class)
    (or (nlistp (:class-name (ogytq Object-field class field)))
        (car (:class-name (ogytq Object-field class field)))))

(de {Object-set}:domain (field class)
    (or (nlistp (:class-name (ogytq Object-set setobjclass field)))
        (car (class-name (ogytq Object-set setobjclass field)))))
```

```
(de {Driver-field}:range (field class)
    (let ((rest (cassq field (ogytq Driver-class restrictions class))))
         (ifn rest (exit smeci-field-def ())
              (or (and (ogytq Driver-class super class)
                       (eq (S-get-facet-value
                                    (:class-name
                                           (ogytq Driver-class super class))
                                    (ogytq Driver-field name field)
                                    'allocation)
                           'instance)
                       (exit bad-allocation
                             (:redefine-category-tree
                                    (driver-find-main-class0 class))))
                  (:build-smeci-range
                            (send 'smeci-range
                                  (ogytq Driver-join0 operator rest)
                                  field))))))

(de :build-smeci-range (lranges)
    (selectq (caar lranges)
             (min (list (cdar lranges) (cassq 'max lranges)))
             (max (list (cassq 'min lranges) (cdar lranges)))
             (lval (cdar lranges))
             (val (mapcan (lambda ((key . val))
                            (if (eq key 'val) (list val)))
                          lranges))))

; smeci ne gere pas les negations.
(de {Driver-sql-op}:domain-type (op))

(de {Driver-n-op}:domain-type (op)
    (any (lambda (op1) (send 'domain-type op1))
         (ogytq Driver-n-op args op)))

(de {Driver-val-link}:domain-type (op)
    (selectq (ogytq Driver-sql-op op op)
             (= 'enum)
             ((< <= > >=) 'bounded)))

(de {Driver-set-link}:domain-type (op)
    (if (eq (ogytq Driver-sql-op op op) 'in) 'enum))

(de {Driver-sql-op}:smeci-range (op field))

(de {Driver-or}:smeci-range (op field)
    (mapcan (lambda (r)
               (if (eq (car r) 'val) (list r)))
            (mapcan (lambda (op1)
                       (send 'smeci-range op1 field))
                    (ogytq Driver-n-op args op))))
```

```
(de {Driver-and}:smeci-range (op field)
    (mapcan (lambda (op1) (send 'smeci-range op1 field))
            (ogytq Driver-n-op args op)))

(de {Driver-val-link}:smeci-range (op field)
    (let ((val (driver-loading-type-filter field ()
                                           (ogytq Driver-2-op arg2 op))))
         (selectq (ogytq Driver-sql-op op op)
                  (= (list (cons 'val val)))
                  ((> >=) (list (cons 'min val)))
                  ((< <=) (list (cons 'max val))))))

(de {Driver-set-link}:smeci-range (op field)
    (if (eq (ogytq Driver-sql-op op op) 'in)
        (let ((lval (mapcar (lambda (val)
                              (driver-loading-type-filter field () val))
                            (ogytq Driver-2-op arg2 op))))
             (list (cons 'lval lval)))))

(de {Driver-field}:force-test? (field class)
    (exit smeci-field-def ()))

(de {Object-field}:force-test? (field class)
    (ifn (memq field (ogytq Driver-class fields class))
         (exit smeci-field-def ())))

(de {Object-set}:force-test? (field class)
    (ifn (memq field (ogytq Driver-class fields class))
         (exit smeci-field-def ())))

#|
(de {Driver-field}:inverse (field))

(de {Driver-field}:import (field))

(de {Driver-field}:unite (field))

(de {Driver-field}:type-lien (field))

(de {Driver-field}:extension (field))
|#


;-------------------------------------------------------
; Name           : {Smeci-object-model}:omake
; Date           : 09/11/89
; Author         : Franck LEBASTARD
; Arguments      : Driver-class object according to a Smeci categorie, from
;                : which a new instance will be created.
```

```
; Result         : A new class instance.
; Description    : Calls Smeci nouvel-objet function.
; Side effects   : On Smeci
; Export         : Yes (method)
; Document       : No
; Modified date  : No
;-----------------------------------------------------

(de {Smeci-object-model}:omake (object-model class)
    (S-create-object (:class-name class)))


;-----------------------------------------------------
; Name           : {Smeci-object-model}:set-field-value
; Date           : 03/90
; Author         : Franck LEBASTARD
; Arguments      :
; Result         :
; Description    :
; Side effects   :
; Export         : Yes (method)
; Document       :
; Modified date  :
;-----------------------------------------------------

(de {Smeci-object-model}:set-field-value (object-model msg obj value)
    (ifn (and (memq msg '(name nom))
              (eq ({Smeci-object-model}:read-field-value object-model
                                                         msg obj)
                  value))
        (S-set-value msg obj value))
    value)

(de {Smeci-object-model}:read-field-value (object-model msg obj)
    (S-get-value msg obj))


;-----------------------------------------------------
; Name           : {Smeci-object-model}:change-object-class
; Date           : 09/07/89
; Author         : Franck LEBASTARD
; Arguments      : Driver-class object, Any Smeci Objet object
; Result         : Objet object from Driver-class class.
; Description    : Changes the class of an object.
; Side effects   : On Smeci
; Export         : Yes (method)
; Document       : No
; Modified date  : 09/11/89
;-----------------------------------------------------
```

```
(de {Smeci-object-model}:change-object-class (object-model newclass obj)
    (if (or (driver-object-default-p obj)
            (eq newclass (driver-object-default-class)))
        (S-change-category obj 'S.Object))
    (S-change-category obj (:class-name newclass)))


;-----------------------------------------------------
; Name           : {Smeci-object-model}:existing-class-p
; Date           : 09/07/89
; Author         : Franck LEBASTARD
; Arguments      : Driver-class object
; Result         : t or ()
; Description    : Does a class exist
; Side effects   : No
; Export         : Yes (redefined)
; Document       : No
; Modified date  : No
;-----------------------------------------------------

(de {Smeci-object-model}:existing-class-p (object-model class)
    (S-category-p (:class-name class)))


;-----------------------------------------------------
; Name           : {Smeci-object-model}:class-instance-p
; Date           : 09/07/89
; Author         : Franck LEBASTARD
; Arguments      : An object, Driver-class object
; Result         : t or ()
; Description    : Is an object a class instance
; Side effects   : No
; Export         : Yes (redefined)
; Document       : No
; Modified date  : No
;-----------------------------------------------------

(de {Smeci-object-model}:class-instance-p (object-model object class)
    (S-isa-p object (:class-name class)))

;-----------------------------------------------------
; Name           : {Smeci-object-model}:all-class-objects
; Date           : 03/30/90
; Author         : Franck LEBASTARD
; Arguments      :
; Result         :
; Description    :
; Side effects   : No
; Export         : Yes
; Document       : No
```

```
; Modified date  : No
;-----------------------------------------------------

(de {Smeci-object-model}:all-class-objects (object-model class)
    (S-all-objects (:class-name class)))


;-----------------------------------------------------
; Name           : driver-dbms-connection
; Date           :
; Author         : Franck LEBASTARD
; Arguments      :
; Result         :
; Description    :
; Side effects   :
; Export         :
; Document       :
; Modified date  :
;-----------------------------------------------------

(de driver-dbms-connection (dbms-name user-name)
    (S-set-category-collector (:show-signature '-CAT dbms-name user-name))
    (S-set-object-collector (:show-signature '-OBJ dbms-name user-name)))

;-----------------------------------------------------
; Name           : driver-object-loading-begin, driver-object-loading-end
; Date           : 09/11/89
; Author         : Franck LEBASTARD
; Arguments      : Main class
; Result         : Void
; Description    : Updates Smeci Object Controle panel.
; Side effects   : No
; Export         : Yes (redefinition)
; Document       : No
; Modified date  : No
;-----------------------------------------------------

(de driver-object-loading-begin (mainclass)
    (:current-cursor-object-database))

(de driver-object-loading-end (mainclass)
;    (funcall '#:smecicontrol:update-interface)
)

(de :current-cursor-category-database ()
    (S-set-category-collector
         (:dbms-signature '-CAT
              (ogytq cursor dbms
                     (or (driver-current-cursor)
                         (error 'smeci-driver "No current cursor" ()))))))
```

```
(de :current-cursor-object-database ()
    (S-set-object-collector
         (:dbms-signature '-OBJ
              (ogytq cursor dbms
                     (or (driver-current-cursor)
                         (error 'smeci-driver "No current cursor" ()))))))

(de :dbms-signature (coltype dbms)
    (:show-signature coltype (ogytq dbms name dbms)
                     (ogytq database name (ogytq dbms database dbms))))

(de :show-signature (coltype nom-sgbd utilisateur)
    (catenate "SGBD" coltype (ascii :dbid-sep)
              nom-sgbd (ascii :dbid-sep)
              utilisateur))
```

B.9 Messages d'erreurs et autres de DRIVER

Codes	Messages d'erreur et autres de DRIVER
driverr11	"Pas de modèle objet courant"
	"No current object model"
driverr21	"Pas un schéma de correspondances"
	"Not a mapping"
driverr22	"Pas de schéma de correspondances courant"
	"No current mapping"
driverr23	"Schéma de correspondances existant"
	"Existent mapping"
driverr24	"Schéma de correspondances inconnu"
	"Unknown mapping"
driverr25	"Schéma de correspondances incomplet"
	"Mapping not complete"
driverr31	"nom de table existant"
	"Existent table name"
driverr32	"Définition de table inconnue"
	"Unknown table definition"
driverr33	"Pas une définition de table"
	"Not a table definition"
driverr34	"Table sans clé"
	"No table key"
driverr41	"Attribut existant"
	"Existent attribute"
driverr42	"Mauvais statut d'attribut"
	"Bad attribute status"
driverr43	"Mauvais type d'attribut"
	"Bad attribute type"
driverr44	"Type d'attribut existant"
	"Existing attribute type"
driverr45	"Définition d'attribut inconnue"
	"unknown attribute definition"
driverr46	"Pas une définition d'attribut"
	"Not an attribute definition"
driverr47	"Pas un des attribut de la table"
	"Not one of the table attributes"
driverr51	"Variable de table existante"
	"Existent table variable"

Codes	Messages d'erreur et autres de DRIVER
driverr52	"Variable de table inconnue"
	"Unknown table variable"
driverr53	"Pas une variable de table"
	"Not a table variable"
driverr54	"Pas une variable de table de base"
	"Not a basic table variable"
driverr55	"Variable de table non effacable"
	"Not an erasable table variable"
driverr61	"Variable d'attribut existante"
	"Existent attribute variable"
driverr62	"Variable d'attribut inconnue"
	"Unknown attribute variable"
driverr63	"Pas une variable d'attribut"
	"Not an attribute variable"
driverr64	"Pas une variable d'attribut effacable"
	"Not an erasable attribute variable"
driverr71	"Nom de classe existant"
	"Existent class name"
driverr72	"classe inconnue"
	"Unknown class"
driverr73	"Pas une classe"
	"Not a class"
driverr81	"Champ existant"
	"Existent field"
driverr82	"Type de champ inconnu"
	"Unknown field type"
driverr83	"Type de champ atomique mauvais"
	"Bad atomic field type"
driverr84	"Type de champ existant"
	"Existing field type"
driverr85	"Champ inconnu"
	"Unknown field"
driverr86	"Pas un champ"
	"Not a field"
driverr87	"Pas un champ atomique"
	"Not an atomic field"
driverr88	"Pas un champ de la classe"
	"Not a class field"

Codes	Messages d'erreur et autres de DRIVER
driverr89	"Pas un champ propre ou herité"
	"Not an own or herited field"
driverr90	"Pas un champ de la hiérarchie"
	"Not a hierarchy field"
driverr91	"Pas une variable d'attribut ou un champ propre ou hérité"
	"Not an attribute variable or an atomic own or herited field"
driverr101	"Contrainte de classe existante sur"
	"Existent class constraint on"
driverr102	"Contrainte inconnue"
	"Unknown constraint"
driverr103	"Expression invalide"
	"Invalid expression"
driverr111	"Jointure déjà existante"
	"Existent join"
driverr112	"Jointure inconsistante"
	"Inconsistent join"
driverr113	"Jointure inconnue"
	"Unknown join"
driverr114	"Pas une jointure"
	"Not a join"
driverr115	"Variable de jointure existante"
	"Existent join variable"
driverr116	"Variable de jointure indéfinie"
	"Undefined join variable"
driverr117	"Deux fois la même variable"
	"Two variables have same name"
driverr118	"Déjà un environnement 'fields' ou 'letjoins"'
	"Yet a 'fields' or 'letjoins' environment"
driverr119	"Noms trop proches"
	"Too close names"
driverr121	"Pas la table principale"
	"Not the main table"
driverr122	"Déjà une table principale"
	"Yet a main table"
driverr123	"Déjà une table secondaire"
	"Yet a sub-table"
driverr124	"Table principale indéfinie"
	"Undefined main table"

Codes	Messages d'erreur et autres de DRIVER
driverr125	"Table principale indéfinie dans la classe principale"
	"Undefined main table on main class"
driverr126	"Table déjà utilisée dans le schéma"
	"Table already used in mapping"
driverr127	"Attribut déjà utilisé dans le schéma"
	"Attribute already used in mapping"
driverr128	"Attribut de jointure déjà utilisé dans le schéma"
	"Join attribute already used in mapping"
driverr131	"Classe sans correspondance"
	"Class not mapped"
driverr132	"Classe avec correspondance déjà établie"
	"Already mapped class"
driverr133	"Pas une correspondance de classe"
	"Not mapped on a class"
driverr134	"Pas une classe principale"
	"Not a main class"
driverr135	"Pas de classe principale"
	"No main class"
driverr141	"Champ sans correspondance"
	"Field not mapped"
driverr142	"Champ avec correspondance déjà établie"
	"Already mapped field"
driverr143	"Aucun ordre spécifié"
	"No order specified"
driverr144	"Ordre specifié invalide"
	"Invalid order specification"
driverr145	"Ensemble de jointures incomplet"
	"Incomplete join set"
driverr146	"Trop de jointures"
	"Too many joins"
driverr147	"Table principale utilisée comme table secondaire dans une jointure"
	"Main table used as a sub-table in a join"
driverr148	"Jointure objet illégale"
	"Object join doesn't comply restrictions"
driverr149	"Champ read-only"
	"Read only field"
driverr150	"Ordre des classes invalide"
	"Invalid class order"

Codes	Messages d'erreur et autres de DRIVER
driverr161	"Pas un curseur"
	"Not a cursor"
driverr162	"Pas de curseur courant"
	"No current cursor"
driverr163	"Pas un objet dbms"
	"Not a dbms"
driverr164	"Objet dbms introuvable"
	"Can't find dbms"
driverr165	"Pas une clé d'objet valide"
	"Not a valid object key"
driverr166	"Pas une clé d'objet existante"
	"Not an existent object key"
driverr167	"Pas un défaut d'objet"
	"Not an object default"
driverr168	"Objet relationnel manquant"
	"Missing relational object"
driverr169	"Pas un objet persistant"
	"Not a persistent object"
driverr170	"Pas un champ de l'objet"
	"Not an object field"
driverr181	"Fonction de filtrage en chargement indéfinie"
	"Undefined loading filter function"
driverr182	"Fonction de filtrage en écriture indéfinie"
	"Undefined writing filter function"
driverr183	"Mauvais arguments des fonctions de filtrage en lecture et écriture"
	"Bad loading or writing filter function arguments"
driverr184	"Incompatibilité déclaration du filtre et arguments des fonctions de filtrage"
	"Filter function arguments and filter declaration mismatch"
driverr185	"Déclarations différentes d'utilisation du même filtre"
	"filter declaration mismatch"
driverr191	"Pas une lambda-expression"
	"Not a lambda expression"
driverr192	"Déclaration d'argument non autorisé"
	"Unauthorised argument declaration"
driverr193	"Trop de lambda-expression"
	"Too many lambda expressions"
driverr194	"Mauvais nombre de variables de lambda"
	"Bad lambda variable number"
driverr195	"Expression non autorisée"
	"Unauthorised expression"

Codes	Messages d'erreur et autres de DRIVER
driverr196	"Mauvais nombre ou type d'argument"
	"Bad argument types or number"
driverr197	"Fonction non autorisée"
	"Unauthorised function"
driverr198	"Variable non autorisée dans un not"
	"Variable unauthorised in a not"
driverr199	"Nil non autorisé"
	"Unexpected nil"
driverr200	"Second argument invalide"
	"Invalid 2nd argument"
driverr101	"Expression constante non autorisée"
	"Unauthorised constant expression"
driverr202	"Variable inconnue"
	"Unknown variable"
driverr203	"Pas un type numérique"
	"Not a numeric type"
driverr204	"Type incompatible"
	"Mismatch type"
driverr205	"Types incompatibles"
	"Incompatible types"
driverr206	"Comparaison entre un champ objet et un champ non objet"
	"Object and not object field(s) comparison"
driverr207	"Comparaison d'instances de classes incompatibles"
	"Incompatible class instance comparison"
driverr208	"Mauvais opérande"
	"Bad operand"
driverr209	"Mauvais comparateur d'objets"
	"Bad object comparator"
driverr210	"Type de champ non encore autorisé dans une expression de sélection d'objets"
	"Field type not supported yet in object selecting"
driverr211	"Variable indéfinie"
	"Unbounded variable"
driverr212	"Mauvais type de valeur de variable"
	"Bad variable value type"
driverr241	"Chargement de l'objet ~A"
	"Loading object ~A"
driverr242	"Écriture de l'objet ~A"
	"Inserting object ~A"
driverr243	"Mise-à-jour de l'objet ~A"
	"Updating object ~A"

Codes	Messages d'erreur et autres de DRIVER
driverr251	"Échec de requête de sélection : ~A"
	"DBMS select request error : ~A"
driverr252	"Échec de requête de lecture du max d'un attribut"
	"Maximum attribute value request error"
driverr253	"Échec de requête d'insertion : ~A"
	"DBMS insert request error : ~A"
driverr254	"Échec de requête de mise-à-jour : ~A"
	"DBMS update request error : ~A"
driverr261	"Valeur de mauvais type pour l'attribut ~A : ~A. Non sauvée."
	"Bad type value for ~A attribute : ~A. Not saved."
driverr262	"~A ne peut être sauvé comme valeur du champ ~A de l'objet ~A"
	"~A can't be saved in field ~A of ~A object"
driverr263	"Incompatibilité entre les contraintes de clé et de classe"
	"Incompatible key and class constraints"
driverr264	"Incohérence dans les contraintes d'écriture"
	"Inconsistency in writing constraints"
driverr265	"Incohérence dans les contraintes d'écriture : ~A"
	"Inconsistency in writing constraints : ~A"
driverr266	"Abandon des contraintes de clé : ~A"
	"Key constraints forsaked : ~A"
driverr267	"Abandon des saisies utilisateur : ~A"
	"User inputs forsaked : ~A"
driverr268	"La valeur du champ ne vérifie pas les contraintes"
	"Field value doesn't comply constraints"
driverr269	"La valeur du champ ~A de l'objet ~A ne vérifie pas les contraintes"
	"~A Field value of object ~A doesn't comply constraints"
driverr270	"Objet ~A, champ ~A, nouvelle valeur : ~A"
	"Object ~A, field ~A, new value : ~A"
driverr271	" Annulation du commit"
	" Commit abortion "
driverr301	"Nouveau"
	"New"
driverr302	"Sélectionnez un objet"
	"Select an object"
driverr303	"Donner une valeur au champ ~A de l'objet ~A vérifiant ~A"
	"Give a value for the ~A field of object ~A, complying ~A"
driverr304	"Donnez une valeur à l'attribut ~A de type ~A, vérifiant ~A"
	"Give a value for the ~A attribute (type ~A), complying ~A"
driverr305	" Entrer une valeur "
	" Enter a value "
driverr306	" D'accord "
	" OK "

Codes	**Messages d'erreur et autres de** DRIVER
driverr401	"Pas un symbole"
	"Not a symbol"
driverr402	"Pas une liste"
	"Not a list"
driverr403	"Argument incorrect"
	"Invalid argument"
driverr404	"Option invalide"
	"Invalid option"

B.10 Index des fonctions et des variables

Index

Résumé

Cette thèse présente DRIVER, une couche objet virtuelle persistante permettant d'utiliser dans un même formalisme objet choisi aussi bien l'information contenue dans les bases de données relationnelles que la connaissance d'un système de plus haut niveau, comme notre générateur de Systèmes Experts SMECI. Un schéma de correspondances, défini par l'utilisateur, associe aux données des bases connectées une représentation objet qui permet de les manipuler et de les exploiter de façon absolument transparente dans l'environnement du système expert, notamment dans le cadre du raisonnement. DRIVER permet également d'apporter la persistance aux objets de l'environnement, au gré de l'utilisateur.

Mots-clés :

Intelligence artificielle, base de donnée relationnelle, modèle relationnel, modèle objet, mapping objet-relationnel, filtrage, objet.

Abstract

This thesis presents DRIVER, a persistent virtual object layer, that permits to use, in a same chosen object formalism, both the information contained in relational databases and the knowledge of a higher-level system, such as our expert system shell SMECI. A user-defined mapping assigns an object representation to data of connected bases; it permits to handle and to utilize them exactly as other objects in the expert system environment, for example during reasoning. DRIVER can also supply some environment objects with persistency, according to user's wishes.

Keywords :

Artificial intelligence, relational database, relational model, object model, object-relational mapping, filtering, object.