Un modèle et un langage pour les bases de données généralisées: projet TIGRE

Fernando Velez Jara

THESE

présentée à

l'Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR INGENIEUR
«Informatique»

par

Fernando VELEZ

UN MODELE ET UN LANGAGE POUR LES BASES DE DONNEES GENERALISEES.

PROJET TIGRE.

Thèse soutenue le 5 septembre 1984 devant la commission d'examen.

C. DELOBEL	Président
M. ADIBA	Examinateurs
R. BALTER	
F. BANCILHON	
J. MOSSIERE	

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

ANCEAU François	E.N.S.I.M.A.G.
BARRAUD Alain	E.N.S.I.E.G.
BAUDELET Bernard	E.N.S.I.E.G.
BESSON Jean	E.N.S.E.E.G.
BLIMAN Samuel	E.N.S.E.R.G.
BLOCH Daniel	E.N.S.I.E.G.
BOIS Philippe	E.N.S.H.G.
BONNETAIN Lucien	E.N.S.E.E.G.
BONNIER Etienne	E.N.S.E.E.G.
BOUVARD Maurice	E.N.S.H.G.
BRISSONNEAU Pierre	E.N.S.I.E.G.
BUYLE BODIN Maurice	E.N.S.E.R.G.
CAVAIGNAC Jean-François	E.N.S.I.E.G.
CHARTIER Germain	E.N.S.I.E.G.
CHENEVIER Pierre	E.N.S.E.R.G.
CHERADAME Hervé	U.E.R.M.C.P.P.
CHERUY Arlette	E.N.S.I.E.G.
CHIAVERINA Jean	U.E.R.M.C.P.P.
COHEN Joseph	E.N.S.E.R.G.
COUMES André	E.N.S.E.R.G.
DURAND Francis	E.N.S.E.E.G.
DURAND Jean-Louis	E.N.S.I.E.G.
FELICI Noël	E.N.S.I.E.G.
FOULARD Claude	E.N.S.I.E.G.
GENTIL Pierre	E.N.S.E.R.G.
GUERIN Bernard	E.N.S.E.R.G.
GUYOT Pierre	E.N.S.E.E.G.
IVANES Marcel	E.N.S.I.E.G.
JAUSSAUD Pierre	E.N.S.I.E.G.
JOUBERT Jean-Claude	E.N.S.I.E.G.
JOURDAIN Geneviève	E.N.S.I.E.G.
LACOUME Jean-Louis	E.N.S.I.E.G.
LATOMBE Jean-Claude	E.N.S.I.M.A.G.

.../...

LESSIEUR Marcel	E.N.S.H.G.
LESPINARD Georges	E.N.S.H.G.
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G.
MAZARE Guy	E.N.S.I.M.A.G.
MOREAU René	E.N.S.H.G.
MORET Roger	E.N.S.I.E.G.
MOSSIERE Jacques	E.N.S.I.M.A.G.
PARIAUD Jean-Charles	E.N.S.E.E.G.
PAUTHENET René	E.N.S.I.E.G.
PERRET René	E.N.S.I.E.G.
PERRET Robert	E.N.S.I.E.G.
PIAU Jean-Michel	E.N.S.H.G.
POLOUJADOFF Michel	E.N.S.I.E.G.
POUPOT Christian	E.N.S.E.R.G.
RAMEAU Jean-Jacques	E.N.S.E.E.G.
RENAUD Maurice	U.E.R.M.C.P.P.
ROBERT André	U.E.R.M.C.P.P.
ROBERT François	E.N.S.I.M.A.G.
SABONNADIERE Jean-Claude	E.N.S.I.E.G.
SAUCIER Gabrielle	E.N.S.I.M.A.G.
SCHLENKER Claire	E.N.S.I.E.G.
SCHLENKER Michel	E.N.S.I.E.G.
SERMET Pierre	E.N.S.E.R.G.
SILVY Jacques	U.E.R.M.C.P.P.
SOHM Jean-Claude	E.N.S.E.E.G.
SOUQUET Jean-Louis	E.N.S.E.E.G.
VEILLON Gérard	E.N.S.I.M.A.G.
ZADWORNY François	E.N.S.E.R.G.

PROFESSEURS ASSOCIES

BASTIN Georges	E.N.S.H.G.
BERRIL John	E.N.S.H.G.
CARREAU Pierre	E.N.S.H.G.
GANDINI Alessandro	U.E.R.M.C.P.P.
HAYASHI Hirashi	E.N.S.I.E.G.

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

PROFESSEURS E.N.S. Mines de Saint-Etienne

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

ALLIBERT Michel	Maître de Recherche
ANSARA Ibrahim	Maître de Recherche
ARMAND Michel	Maître de Recherche
BINDER Gilbert	
CARRE René	Maître de Recherche
DAVID René	Maître de Recherche
DEPORTES Jacques	
DRIOLE Jean	Maître de Recherche
GIGNOUX Damien	
GIVORD Dominique	
GUELIN Pierre	
HOPFINGER Emil	Maître de Recherche
JOUD Jean-Charles	Maître de Recherche
KAMARINOS Georges	Maître de Recherche
KLEITZ Michel	Maître de Recherche
LANDAU Ioan-Dore	Maître de Recherche
LASJAUNIAS J.C.	
MERMET Jean	Maître de Recherche
MUNIER Jacques	Maître de Recherche
PIAU Monique	
PORTESEIL Jean-Louis	
THOLENCE Jean-Louis	
VERDILLON André	

CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)

LESBATS Pierre	Directeur de Recherche
BISCONDI Michel	Maître de Recherche
KOBYLANSKI André	Maître de Recherche
LE COZE Jean	Maître de Recherche
LALAUZE René	Maître de Recherche
LANCELOT Francis	Maître de Recherche
THEVENOT François	Maître de Recherche
TRAN MINH Canh	Maître de Recherche

PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)

ALLIBERT Colette	E.N.S.E.E.G.
BERNARD Claude	E.N.S.E.E.G.
BONNET Rolland	E.N.S.E.E.G.
CAILLET Marcel	E.N.S.E.E.G.
CHATILLON Catherine	E.N.S.E.E.G.
CHATILLON Christian	E.N.S.E.E.G.
COULON Michel	E.N.S.E.E.G.
DIARD Jean-Paul	E.N.S.E.E.G.
EUSTAPOPOULOS Nicolas	E.N.S.E.E.G.
FOSTER Panayotis	E.N.S.E.E.G.

.../...

GALERIE Alain	E.N.S.E.E.G.
HAMMOU Abdelkader	E.N.S.E.E.G.
MALMEJAC Yves	E.N.S.E.E.G. (CENG)
MARTIN GARIN Régina	E.N.S.E.E.G.
NGUYEN TRUONG Bernadette	E.N.S.E.E.G.
RAVAINE Denis	E.N.S.E.E.G.
SAINFORT	E.N.S.E.E.G. (CENG)
SARRAZIN Pierre	E.N.S.E.E.G.
SIMON Jean-Paul	E.N.S.E.E.G.
TOUZAIN Philippe	E.N.S.E.E.G.
URBAIN Georges	E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON)
GUILHOT Bernard	E.N.S. Mines Saint Etienne
THOMAS Gérard	E.N.S. Mines Saint Etienne
DRIVER Julien	E.N.S. Mines Saint Etienne
BARIBAUD Michel	E.N.S.E.R.G.
BOREL Joseph	E.N.S.E.R.G.
CHOVET Alain	E.N.S.E.R.G.
CHEHIKIAN Alain	E.N.S.E.R.G.
DOLMAZON Jean-Marc	E.N.S.E.R.G.
HERAULT Jeanny	E.N.S.E.R.G.
MONLLOR Christian	E.N.S.E.R.G.
BORNARD Guy	E.N.S.I.E.G.
DESCHIZEAU Pierre	E.N.S.I.E.G.
GLANGEAUD François	E.N.S.I.E.G.
KOFMAN Walter	E.N.S.I.E.G.
LEJEUNE Gérard	E.N.S.I.E.G.
MAZUER Jean	E.N.S.I.E.G.
PERARD Jacques	E.N.S.I.E.G.
REINISCH Raymond	E.N.S.I.E.G.
ALEMANY Antoine	E.N.S.H.G.
BOIS Daniel	E.N.S.H.G.
DARVE Félix	E.N.S.H.G.
MICHEL Jean-Marie	E.N.S.H.G.
OBLED Charles	E.N.S.H.G.
ROWE Alain	E.N.S.H.G.
VAUCLIN Michel	E.N.S.H.G.
WACK Bernard	E.N.S.H.G.
BERT Didier	E.N.S.I.M.A.G.
CALMET Jacques	E.N.S.I.M.A.G.
COURTIN Jacques	E.N.S.I.M.A.G.
COURTOIS Bernard	E.N.S.I.M.A.G.
DELLA DORA Jean	E.N.S.I.M.A.G.
FONLUPT Jean	E.N.S.I.M.A.G.
SIFAKIS Joseph	E.N.S.I.M.A.G.
CHARUEL Robert	U.E.R.M.C.P.P.
CADET Jean	C.E.N.G.
COEURE Philippe	C.E.N.G. (LETI)

.../...

DELHAYE Jean-Marc	C.E.N.G. (STT)
DUPUY Michel	C.E.N.G. (LETI)
JOUVE Hubert	C.E.N.G. (LETI)
NICOLAU Yvan	C.E.N.G. (LETI)
NIFENECKER Hervé	C.E.N.G.
PERROUD Paul	C.E.N.G.
PEUZIN Jean-Claude	C.E.N.G. (LETI)
TAIEB Maurice	C.E.N.G.
VINCENDON Marc	C.E.N.G.

LABORATOIRES EXTERIEURS

DEMOULIN Eric	C.N.E.T.
DEVINE	C.N.E.T. (R.A.B.)
GERBER Roland	C.N.E.T.
MERCKEL Gérard	C.N.E.T.
PAULEAU Yves	C.N.E.T.
GAUBERT C.	I.N.S.A. Lyon

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur	: Monsieur M. MERMET
Directeur des Etudes et de la formation	: Monsieur J. LEVASSEUR
Directeur des recherches	: Monsieur J. LEVY
Secrétaire Général	: Mademoiselle M. CLERGUE

Professeurs de 1ère Catégorie

COINDE	Alexandre	Gestion
GOUX	Claude	Métallurgie
LEVY	Jacques	Métallurgie
LOWYS	Jean-Pierre	Physique
MATHON	Albert	Gestion
RIEU	Jean	Mécanique - Résistance des matériaux
SOUSTELLE	Michel	Chimie
FORMERY	Philippe	Mathématiques Appliquées

Professeurs de 2ème catégorie

HABIB	Michel	Informatique
PERRIN	Michel	Géologie
VERCHERY	Georges	Matériaux
TOUCHARD	Bernard	Physique Industrielle

Directeur de recherche

LESBATS	Pierre	Métallurgie

Maîtres de recherche

BISCONDI	Michel	Métallurgie
DAVOINE	Philippe	Géologie
FOURDEUX	Angeline	Métallurgie
KOBYLANSKI	André	Métallurgie
LALAUZE	René	Chimie
LANCELOT	Francis	Chimie
LE COZE	Jean	Métallurgie
THEVENOT	François	Chimie
TRAN MINH	Canh	Chimie

Personnalités habilitées à diriger des travaux de recherche

DRIVER	Julian	Métallurgie
GUILHOT	Bernard	Chimie
THOMAS	Gérard	Chimie

Professeur à l'UER de Sciences de Saint-Etienne

VERGNAUD	Jean-Maurice	Chimie des Matériaux & chimie industrielle

****

Je tiens particulièrement à remercier :

Monsieur Claude DELOBEL, Professeur à l'Université Scientifique et Médicale de Grenoble, qui m'a toujours accordé sa confiance et son soutien. Je suis très reconnaissant pour l'honneur qu'il me fait de présider ce jury.

Monsieur Michel ADIBA, Professeur à l'Université Scientifique et Médicale de Grenoble, qui est à l'origine de cette thèse et qui m'a accueilli dans son équipe. Je suis particulièrement sensible à sa totale disponibilité à mon égard et à ses nombreux conseils et encouragements ; qu'il soit ici très sincèrement remercié.

Monsieur Roland BALTER, Responsable de l'antenne de Grenoble du Centre de Recherches BULL, qui m'a toujours aidé et conseillé au sein de son équipe. Je lui suis très reconnaissant de la confiance qu'il m'a toujours témoignée et d'avoir bien voulu participer au jury.

Monsieur François BANCILHON, Professeur à l'Université d'Orsay et Responsable Scientifique à l'INRIA, qui a bien voulu juger mon travail et participer au jury.

Monsieur Jacques MOSSIERE, Professeur à l'Institut Nationale Polytechnique de Grenoble, qui a bien voulu participer au jury.

J'aurais beaucoup à dire pour remercier Mauricio LOPEZ, qui a été depuis 1980 à mes côtés, soit à l'IMAG, soit au centre de Recherches BULL. A travers les très nombreuses discussions que nous avons eues, il a su me faire profiter de sa grande expérience et son appui amical m'a toujours été précieux.

Je voudrais aussi remercier tous les membres de l'équipe TIGRE et ceux du centre de Recherches BULL à Grenoble pour l'amitié qu'ils m'ont toujours témoignée. Marie-Laure WOROWSKI et Brigitte HUMBERT ont su déchiffrer un manuscrit mal écrit et utiliser avec virtuosité une certaine TTX 80 ; qu'elles en soient vivement remerciées.

Je ne saurais oublier Maria Giovanna, ma femme, pour son soutien moral et pour la patience dont elle a fait preuve pendant la réalisation du manuscrit.

A mis padres

RESUME

Cette thèse traite du problème de la gestion de bases de données où les informations à stocker et à manipuler sont de nature diverse (factuelles, textuelles, imagées...) et à structure interne éventuellement complexe. Ces données, dits "généralisées" apparaissent dans des applications bureautiques et de conception assistée par ordinateur. Nous présentons un modèle de données (TIGRE) capable de permettre la description de ces données, et le langage de définition et de manipulation associé (LAMBDA). Finalement, nous présentons le cadre de réalisation des énoncés de manipulation du langage.

mots-clé : Bases de données généralisées, modèle de données, langage de manipulation, serveur de données, documents multi-media, interfaces usager.

SOMMAIRE

N.B. Chaque Chapitre est précedé d'un sommaire plus détaillé

Chapitre 1

INTRODUCTION

1. INTRODUCTION

Les systèmes de bases de données ont été conçus pour gérer des grandes quantités de données alphanumériques formatées, principalement dans le domaine de la gestion. Cependant, de nouvelles applications informatiques ont de gros besoins en matière de gestion de données. Des exemples sont la bureautique, où l'on manipule des documents, la CAO qui traite des gros volumes d'information de structure et de nature très diverse, le génie logiciel, où l'on gère des programmes, la synthèse de la parole où l'on manipule de la voix digitalisée, la cartographie, le traitement d'images, le recherche documentaire, etc.

Pendant longtemps, ces types d'applications ont permis de développer des logiciels "ad-hoc" mais on ressent aujourd'hui, le besoin d'utiliser des logiciels qui offrent davantage de facilités. Il faut donc envisager le développement de systèmes de gestion des bases de données offrant des fonctionnalités nouvelles. Il s'agit de gérer de manière intégrée des données dites "généralisées" : ces données volumineuses et/ou à structure complexe, généralement multi-média (des textes, des images, des graphiques) qui apparaissent dans les applications mentionnées ci-dessus.

Une étude générale des nouvelles perspectives concernant les applications, fonctionnalités et technologies des bases de données a été développée par le groupe BD3. Ses résultats ont été publiés dans [BD3 83]. L'aspect modélisation et manipulation de données généralisées y est considéré comme une orientation de recherche qu'il faut privilégier. En effet, d'une part les nouvelles applications appèlent des modèles prenant en compte davantage d'aspects sémantiques de l'application ; d'autre part, elles demandent des outils de manipulation beaucoup plus puissants en raison de l'hétérogénéité des objets, structurés ou non, qui doivent être plus riches et des langages adaptés à leur manipulation.

C'est dans cette ligne de recherche que le présent travail se situe. Plus précisément, notre travail se place dans le cadre plus général du projet TIGRE (Traitement d'Informations Généralisées Réparties) mené

conjointement à Grenoble par le Centre de Recherche BULL et le Laboratoire de Génie Informatique de l'IMAG. L'objectif général du projet TIGRE est la manipulation des données généralisées dans un contexte bureautique. A terme, il doit conduire à la réalisation d'un système expérimental s'appuyant sur un serveur de bases de données généralisées et plusieurs postes de travail communiquant grâce à un réseau local.

Dans ce travail, nous nous sommes d'abord intéressés à un modèle de données capable de permettre la définition de ces données généralisées. La définition finale de ce modèle (appelé aussi TIGRE) a été un travail collectif auquel nous avons participé. Il intègre des concepts développés dans le domaine des bases de données (modélisation sémantique de données) et dans le domaine des langages de programmation (concept de types de données) pour constituer une extension du modèle "Entité Association" [Che 76].

Nous nous sommes ensuite intéressés à la conception d'un langage de définition et de manipulation de données généralisées associé au modèle TIGRE. Ce langage appelé LAMBDA est un langage de haut niveau orienté ensembliste mais non directement destiné à l'utilisateur final d'une base de données généralisée. En effet, la conception du langage a été influencée par le désir de faciliter son intégration dans des langages de programmation de haut niveau possédant la notion de type.

Dans le chapitre 2, nous présentons une synthèse des travaux réalisés sur le thème des bases de données généralisées. Celà nous permettra de mieux situer les objectifs et l'état actuel du projet TIGRE.

Au chapitre 3, nous présentons les principaux concepts du modèle TIGRE et les énoncés de définition de LAMBDA. La notion de "document généralisé" y est présentée et la spécificité des concepts du modèle TIGRE y est discutée par rapport à d'autres modèles sémantiques existants. Une définition formelle du modèle est également présentée.

Le chapitre 4 traite des énoncés d'interrogation de LAMBDA qui permettent d'extraire de la base des données de nature diverse. Les constructions du langage sont introduites à l'aide de nombreux exemples,

puis formalisées grâce à une notation mathématique.

Le chapitre 5 présente les énoncés de manipulation qui permettent la modification de tout ou partie de la base. Un accent tout particulier est mis sur deux points : d'abord sur le couplage entre l'outil de manipulation des documents généralisés (l'éditeur multi-média des postes de travail - actuellement en cours de développement dans le cadre du projet TIGRE) et la base de données. Deuxièmement, sur les propagations des mises à jour, l'approche que nous avons choisi pour préserver l'intégrité de la base lors des mises à jour.

Au cours du chapitre 6, nous évaluerons le pouvoir d'expression de la partie "Entité-Association" de LAMBDA et nous effectuons une comparaison de LAMBDA avec d'autres langages proposés dans la littérature. La complétude de LAMBDA par rapport à une algèbre "Entité-Association" définie dans [MR 83a] est démontrée.

Enfin, au chapitre 7, nous décrivons l'architecture du serveur de bases de données TIGRE. Nous nous concentrerons plus particulièrement sur le cadre de réalisation des opérations d'interrogation et de mises à jour de LAMBDA.

Chapitre 2

TRAITEMENT DES DONNEES GENERALISEES DANS LES SYSTEMES DE GESTION DE DONNEES

SOMMAIRE

2 - TRAITEMENT DE DONNEES GENERALISEES DANS LES SYSTEMES DE GESTION DE DONNEES

Par le terme "données généralisées", nous entendons des données volumineuses et éventuellement à structure interne complexe. Ce terme regroupe donc les données multi-média (textes, images, graphiques, voix digitalisée), la combinaison structurée des données multi-média (connue généralement sous le terme de "document généralisé" ou "document multi-média") et les données alphanumériques à structure interne complexe manipulées comme une unité. Ce sont donc des données qui apparaissent dans des applications comme la bureautique et la CAO.

Contrairement aux données alphanumériques, les données généralisées ne sont pas atomiques et ont une sémantique propre. Elles demandent des outils de manipulation puissants, tant matériels (interfaces d'entrée -sortie pour les données multi-média, par exemple) que logiciels (éditeurs de texte, de graphiques...), et du fait de leur taille importante, elles posent des nouveaux problèmes de stockage et d'accès dans les systèmes de gestion de données.

Dans ce chapitre, nous effectuons une synthèse des travaux (encore peu nombreux) réalisés sur le thème données généralisées et nous présentons l'approche prise dans le projet TIGRE pour traiter les données généralisées.

Les approches qui ont été pris pour manipuler des données généralisées diffèrent sur les points suivants :

- Le type de modélisation effectuée sur la structure logique des données généralisées.

- Le degré d'intégration de cette modélisation sur les données généralisées aux modèles de données "alphanumériques" existants (relationnel, Entité-association, par exemple).

- Les fonctionnalités offertes au niveau système pour manipuler les données généralisées (méthodes d'accès, représentation interne, contrôle de concurrence, confidentialité, support pour implanter des contraintes d'intégrité, etc.)

- Les fonctionnalités offertes à l'usager pour retrouver des données généralisées (type de langage, puissance du mécanisme de recherche -séquentiel, par attributs, par contenu -, aides à la formulation de requêtes, etc).

- Les fonctionnalités offertes au niveau "poste de travail" pour créer, modifier, présenter, diffuser, etc, les données généralisées.

- L'extension ou non d'un SGBD déjà existant.

2.1. L'ETAT DE L'ART

2.1.1. Extensions à System-R

Récemment, des extensions a system-R [ABCE 76] ont été proposées dans [HL-82], [LP 83], [LKMP 84]. Ces extensions visent à offrir des fonctionnalités pour des applications non traditionnelles qui :

(a) stockent des chaînes de caractères de longueur arbitraire.
(b) traitent des données structurées mais hétérogènes comme une unité
(c) utilisent la base de données de façon très interactive, chaque interaction pouvant durer très longtemps.

La première caractéristique correspond à une exigence des applications qui manipulent des données multi-media volumineux. Pour cela, on a étendu les méthodes d'accès dans System-R. Chaque "champ long" (selon leur terminologie) a un descripteur associé que le système connait et c'est lui qui gère l'allocation de l'espace physique. Cette approche constraste avec celle prise dans INGRES (cf section 2.1.1.2). Etant donné que les descripteurs de champs longs et les descripteurs

d'espace libre sont stockés comme des n-uplets dans des relations internes, les mécanismes de verrouillage et de reprise après panne sont utilisés pour maintenir la cohérence de la base.

Par ailleurs, on a étendu SQL de façon à pouvoir retrouver la valeur d'un champ long "par morceaux " car la taille de l'espace de travail mémoire d'un programme est limité. Les recherches par contenu à l'intérieur d'un champ long ne sont pas offertes.

Le concept d'objet complexe a été proposé pour pallier à la difficulté soulignée dans (b). Un objet complexe peut être considéré comme un agrégat de n-uplets potentiellement hétérogènes qui sont généralement utilisés ensemble. Cet agrégat est hiérarchique, ce qui reflète une constante que l'on retrouve dans beaucoup d'applications orientées vers l'ingénierie. Le système connait la structure d'un objet complexe. En effet, on a introduit trois nouveaux types d'attributs. Un attribut défini sur le type IDENTIFICATEUR constitue la clé primaire des relations contenant des objets complexes dont la valeur est unique dans le système. Le type COMPONENT-OF (R) effectue des liaisons hiérarchiques à l'intérieur de l'objet complexe. La valeur pour un n-uplet t est la valeur de l'attribut IDENTIFICATEUR du père de t qui se trouve dans la relation R. Finalement, le type REFERENCE (R) permet d'effectuer des liaisons autres que des liaisons hiérarchiques entre des n-uplets d'un objet complexe. Le système implante des contraintes d'intégrité sémantiques sur ces objets : on peut supprimer tout l'objet en une seule commande de suppression ; on ne peut pas insérer un composant d'un objet complexe sans père, on peut le copier, le verrouiller. Une méthode d'accès efficace aux composants d'un objet complexe est implantée. L'interface usager aux objets complexes a été améliorée avec la notion de curseur-complexe et de jointure implicite [LKMP 84].

L'exigence (c) est plutôt une caractéristique des environnements de conception qui partagent des objets complexes. Les transactions dans ce genre d'environnement peuvent s'étendre pendant une période de temps très longue durant laquelle la base est incohérente. Ceci pose

de gros problèmes liés au contrôle de concurrence et à la reprise. Un mécanisme de partage d'objets complexes appelé Transaction conversationnelle a été proposé. Il utilise la notion de base privée et base publique et le principe consiste en la séparation de la cohérence et la reprise après pannes. Récemment, on a étendu ce mécanisme de façon à supporter des environnements de conception plus flexibles dans lesquels plusieurs ingénieurs partagent des objets complexes incomplets d'une manière contrôlée. La notion de transactions imbriquées est incorporée au mécanisme [KLMP 83].

2.1.2 Extensions à INGRES

Dans [SSLK 84], on propose des extensions au SGBD INGRES [SWKH 76] orientées au traitement efficace des documents. Ces extensions sont les suivantes :

(a) chaînes de caractères de longueur variable (type 'texte')
(b) Relations "ordonnées"
(c) opérateurs de sous-chaînes de caractères
(d) opérateurs de rupture et concaténation

Avec (a), le texte d'un document arbitrairement long peut être regardé comme la valeur d'un attribut. Pour implanter cette fonctionnalité, le paradigme "Types abstraits" a été choisi : un code approprié est développé et intégré au SGBD et la chaîne de caractères est stockée dans un fichier hors de la base de données. L'extension (b) permet de traiter le document par "morceaux" (mots, lignes, paragraphes ou combinaisons de ces trois). Une structure de stockage et accès est proposée pour accéder rapidement aux lignes (morceaux) et pour éviter des problèmes tels que la renumérotation de lignes lors de mises à jour. Les extensions (c) permettent de faire des recherches "par contenu", c.a.d., des recherches de sous-chaînes à l'intérieur d'une chaîne, ainsi que de faire des remplacements d'une sous-chaîne par une autre, etc. Finalement, les opérateurs (d) permettent qu'un document puisse changer de format (par ex. de stockage par paragraphes à stockage "en

tas"). Ces opérateurs qui s'appliquent aux champs de type texte et type chaînes de caractères de longueur fixe peuvent être utilisés à l'intérieur d'une requête QUEL. Pour ce qui est de la recherche d'un document "par contenu", on n'a fait mention d'aucune méthode particulière pour accélérer la recherche (contrairement aux travaux menés à Toronto, cf. section 2.1.1.3) On suggère qu'avec ces extensions, on peut développer beaucoup plus aisément des éditeurs de texte comme un programe d'application sur INGRES. Cette approche permet d'étendre aux documents des services bien connus des SGBD : contrôle de concurrence, reprise après pannes, contrôle d'accès.

On a proposé récemment [SAHR 84] l'utilisation de QUEL comme type d'un attribut d'une relation. Autrement dit, la valeur d'un attribut de type QUEL est un énoncé QUEL que l'on peut exécuter. Des fonctionnalités intéressantes en résultent : définition de règles, actions prédéfinies ("triggers"), représentation d'objets complexes. Concernant ce dernier point, la différence essentielle avec l'approche prise par System-R est que ce dernier connait la structure d'un objet complexe (et de ce fait, il peut implanter des contraintes d'intégrité et des fonctions "globales" sur ces objets (cf. section 2.1.1.1) alors qu'ici, INGRES ne le connaît pas. C'est une requête QUEL, qui est la valeur d'un attribut de la n-uplet racine de l'objet complexe, qui "rassemble" les morceaux.

2.1.3 Travaux menés à l'université de Toronto.

A Toronto, on développe depuis deux ans un système bureautique de classement de messages multimédia [TCEF 83], [CVLL 84]. Il permet de stocker et de retrouver des messages composés de textes, images (ensemble de pixels), graphiques, tableaux, voix et des valeurs d'attributs (caractères alphanumériques). Le système est construit sur le système Unix de la machine SUN (il n'est pas construit sur un SGBD; il n'implémente pas un modèle de données comme on l'entend couramment mais plutôt un "modèle de messages"). Les messages sont retrouvés par leur contenu, par la valeur de leurs attributs et par la donnée d'expressions régulières de sous chaînes dans le texte. Un mé

canisme de recherche par contenu sur les graphiques est également offert où l'on exploite les relations spatiales entre les objets du graphique ; les requêtes de ce type sont construites en utilisant un éditeur graphique spécialisé. On peut également retrouver des messages par la valeur des attributs des tableaux, par l'existence ou non de voix dans le message et par la position des images dans la présentation du message.

L'interface usager pour exprimer une requête utilise à fond les fonctionnalités graphiques du poste de travail. Elle combine la spécification des conditions de sélection (filtrage) et le balayage ("browsing") des messages retrouvés. Le balayage est accompli en utilisant des "miniatures" et des abstractions vocales. Une miniature est une abstraction visuelle du message, rétréci de façon à ce que plusieurs miniatures puissent tenir au même temps à l'écran ; ces miniatures sont automatiquement générées à la création du message. Les abstractions vocales sont des suites courtes de mots qui expliquent le contenu du message. L'usager décide ou non de retrouver le message en fonction de ce qu'il voit ou entend. Le filtrage peut être affiné en cours de route, et on peut extraire une partie du contenu d'un message pour l'utiliser dans la reformulation de la requête. Ceci est particulièrement utile pour spécifier des conditions concernant les graphiques.

Les messages sont stockés dans un ou plusieurs fichiers (non reliés hiérarchiquement entre eux par des "directories"). Une méthode d'accès pour la recherche efficace de sous-chaînes de caractères dans un texte est utilisée. La méthode utilise une technique de signature : une chaîne de bits de longueur fixe est créée pour chaque bloc de texte (un ou plusieurs paragraphes). La signature est construite à la création du message. On prend chaque mot non trivial du bloc et on lui applique une fonction de hachage à chaque triplet. Les bits correspondants dans la chaîne sont mis à un. La taille des signatures et le nombre de bits par mot ont été déterminés de telle façon que le système est optimisé [CF 84]. Cette méthode contraste avec celle basée sur des signatures par mot non trivial et non par bloc proposée dans [Lar 83]

Pour déterminer si un mot apparaît dans un bloc, on lui applique la même transformation et on regarde la signature du bloc. Cette méthode retrouve un sur-ensemble de messages qualifiés ; la capacité de balayer les messages retrouvés permet à l'usager de prendre les messages qui vérifient la condition de filtrage.

Pour résumer, on peut dire qu'il s'agit d'un système où l'adressage par contenu aux objets multimédia est très poussé, cet adressage étant supporté par une méthode d'accès efficace, et où l'interface usager est adaptée à un environnement bureautique.

2.1.4 Le projet BIG à Toulouse.

Le projet BIG (Bases d'Informations Généralisées) [CCZ 83] est un projet de recherche développé à l'université Paul Sabatier de Toulouse, dont l'objet est de développer des concepts et des spécifications sur les bases de données intégrant des données alphanumériques, du texte, des graphiques et des images. Un modèle de données dérivé du modèle Entite- Association (ER) a été défini. Il inclut les notions de classe d'entités, classe d'associations, classe de textes, classe d'images et classe de documents.

Une classe de textes (resp. d'images) correspond en gros à une classe d'entités dans laquelle chaque entité en plus d'avoir des attributs la décrivant, a un contenu implicite de nature textuelle (resp. picturale). Les documents ont chacun des attributs les décrivant, une structure logique hiérarchique et un contenu dans lequel on peut trouver des textes, des images et des graphiques. Plusieurs présentations peuvent être associées au même document. Ces dernières interviennent dans la phase de production des documents : le même document peut être produit avec des présentations et des procédés différents.

Les documents sont représentés de façon interne en utilisant des classes de la manière suivante : un n-uplet "racine" d'une classe d'entité contient les attributs décrivant le document et plusieurs n-uplets représentent la structure du document (un n-uplet par noeud de la structure). Le contenu du document (les feuilles) se trouve dans des

n-uplets des classes d'entités, de textes et d'images. Cette méthode de représentation des documents rappelle le concept d'objet complexe dans system R.

Deux architectures du système de gestion de bases d'informations généralisées sont en cours de réalisation : la première étend le SGBD DMS II de Burroughs, la deuxième est développée sur l'ordinateur individuel PERQ. A notre connaissance, il n'existe pas à l'heure actuelle, de méthodes d'accès adaptée à la recherche efficace des sous-chaînes de caractères à l'intérieur d'un texte, ainsi que des spécifications d'un langage de requête associé au modèle proposé.

2.1.5 Le langage TQL, un langage d'interrogation textuel.

Les chercheurs de l'équipe VERSO à l'INRIA ont proposé [BR 82] une nouvelle approche à la conception de systèmes de gestion de données qui manipulent des données alphanumériques (relationnelles) et des données textuelles. Dans cette approche, l'usager voit deux bases de données, une base de données relationnelle et une base de données textuelle qu'il peut déclarer et interroger. Un langage de définition textuel (TDL) et un langage d'interrogation textuel (TQL) est défini. La communication entre les deux bases de données (génération du texte à partir de relations et l'interprétation du texte par des relations) n'est pas adressée à notre connaissance.

TQL permet de définir des textes structurés hiérarchiquement à l'aide d'une grammaire. TQL retrouve des sous-arborescences d'un ou plusieurs textes qui satisfont des conditions. La notion de base de TQL est la notion de curseur. Un curseur est une variable qui désigne un noeud dans la structure d'un texte. Les conditions s'effectuent sur des composants d'un curseur; elles sont du type recherche de sous-chaînes par contenu ou "jointure" entre deux structures de textes. Pour accélérer les recherches, on a prévu d'utiliser un filtre matériel qui effectuerait la plupart des opérations en temps linéaire.

2.2 - NOTRE APPROCHE : TIGRE

2.2.1. Introduction

Les documents que l'on manipule couramment comportent du texte, des photographies, des schémas géométriques. Ils circulent entre les correspondants d'une organisation accompagnés éventuellement de commentaires écrits ou oraux. Si l'on était capable de digitaliser et de stocker les différents constituants (l'alphanumérique, le géométrique, le photographique et le vocal) on savait encore mal exprimer leur intégration au sein d'une entité que l'on appelera le "document généralisée".

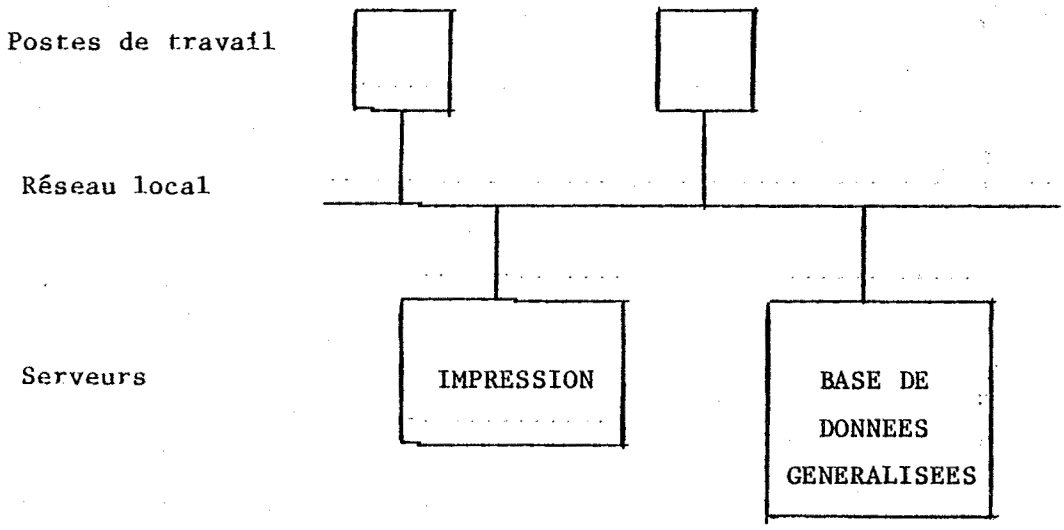
La définition précise d'un modèle de document généralisé comportant des outils pour définir sa structure et sa sémantique constituait un préalable à toute tentative de représentation informatique. C'est dans cet esprit que le projet TIGRE a été lancé en 1982. Il est mené conjointement par le Centre de Recherches BULL et le laboratoire IMAG à Grenoble. A partir d'une définition complète du document généralisé, le projet s'est articulé en deux thèmes d'étude :

(a) la conception et la réalisation des outils sur un poste de travail basé sur ordinateur individuel permettant l'acquisition et la manipulation de documents généralisés.

(b) la réalisation d'un serveur comprenant une base de données adaptée à la gestion et au stockage des nouveaux types d'objets multimédia rencontrés dans un document généralisé.

Les postes de travail constituent évidemment le moyen d'accès privilégié au serveur base de données via un réseau local. Dans ce contexte, on peut envisager que chaque utilisateur dispose d'une autonomie qui lui évite le recours à un site serveur central pour les traitements qui lui sont propres. Compte tenu du progrès de la micro-informatique, on peut envisager le développement sur les postes de travail, d'unités de traitement locales réalisant des fonctions propres au domaine d'application considéré: la rédaction de documents, la conservation de données privées, la tenue d'agenda, la messagerie et l'accès aux ser-

vices externes sont des exemples typiques. L'architecture retenue est donc la suivante :



Une importance particulière a donc été donnée aux applications bureautiques. Néanmoins, des études sont menés dans d'autres domaines comme la CAO et la programmation en logique [TIG 13], [TIG 7], [TIG 12]. L'ensemble de la recherche dans le cadre du projet TIGRE est présenté dans les différents rapports de recherche (voir bibliographie).

2.2.2. Aspect "poste de travail"

L'intérêt principal des recherches dans cet aspect du projet est celui de concevoir et réaliser un éditeur interactif de documents généralisés. Il est actuellement en cours de spécification. L'objectif est d'une part, d'assurer une bonne homogénéité dans la manipulation des objets de nature différente et d'autre part, d'offrir un degré d'extensibilité important, par l'ajout de nouveaux environnements ("medias"), par exemple, tableaux, formules mathématiques, graphiques, et par la définition de nouveaux types de documents structurés (rapports, contrats...). Pour ce faire, on se propose de réaliser un noyau d'éditeur universel capable de communiquer avec plusieurs environnements correspondants chacun à une nature d'objet. Des expériences ont été menées concernant les natures suivantes : les formules mathématiques [Qui 83], les graphiques [Jol 83] et les tableaux [Dar 84]. L'éditeur connaît la structure logique des documents. Il permet donc

que les usagers manipulent des documents par sa structure logique.

L'interface utilisateur est en soi un thème de recherche dans TIGRE. Elle sera clairement séparée du reste de l'éditeur ce qui permattra d'essayer différents types d'interface utilisateur et de garder un minimum d'indépendance vis-à-vis des matériels d'affichage.

2.2.3. Aspect bases de données

Dans la partie "serveur de données généralisées", nous nous sommes intéressés à plusieurs aspects.

D'abord, nous avons voulu intégrer le modèle de document généralisé à un modèle de données "alphanumériques", dans le but de pouvoir décrire des documents dans les environnements des applications où ils évoluent. Pour cela, notre modèle de données "alphanumériques hôte a été un modèle sémantique, le modèle "Entité-Association". Nous y avons intégré le concept de document généralisé comme un constructeur de types, les attributs d'une entité ou d'une association pouvant être définis sur un type-document, et nous avons encore étendu le modèle Entité-Association pour y inclure des abstractions comme la généralisation et l'agrégation développées dans d'autres modèles sémantiques. En particulier, le concept d'agrégation d'entités que nous avons défini permet le description d'une autre "catégorie" de données généralisées: les données alphanumériques à structure interne complexe manipulées comme une unité. Ces objets "moléculaires" apparaissent dans des applications orientées vers l'ingénierie (cf. section 2.1.1) comme par exemple la CAO de circuits intégrés. Le modèle de données résultant appelé TIGRE est décrit dans le chapitre 3.

Un des types de données avec une sémantique particulière c'est le temps. Des applications telles que la bureautique et les modèles économétriques ont besoin du concept de temps d'une façon explicite dans la base de données. Une extension au modèle TIGRE incluant ce concept est proposée dans [Pal 84].

Ensuite, nous nous sommes intéressés à la conception de LAMBDA, le language de manipulation associé à ce modèle. Il permet de retrouver des documents en fonction de leur contenu et de leurs liaisons sémantiques avec le reste d'objets de la base, pour les manipuler dans l'éditeur.

Il prend également en compte la sémantique explicite offerte par le modèle TIGRE. Il est destiné à être intégré dans un langage de programmation possédant la notion de type.

Le serveur de données TIGRE est construit sur un SGBD relationnel et sur des modules indépendants pour la gestion (stockage accès) des documents généralisés. L'interface entre les deux niveaux (niveau TIGRE et systèmes sous-jacents) se compose de primitives de l'algèbre relationnelle et de primitives spécifiques pour les documents généralisés. Dans la mesure du possible, le serveur TIGRE sera compatible avec tout SGBD relationnel offrant une interface algébrique, ce qui permettra, en particulier, d'employer la machine base de données VERSO, développée à l'INRIA [BS 80], [BS 82]. D'autres fonctionnalités concernant les données généralisées sont actuellement en cours d'étude : un mécanisme de reprise après pannes, et un mécanisme d'autorisation [Azr 84], [AA 84]. L'architecture du serveur TIGRE est décrite au chapitre 7.

Un effort important est également réalisé dans le thème "base de données" du projet TIGRE concernant la coopération entre le serveur TIGRE et un système de programmation en logique tel que PROLOG. L'objectif est de définir un outil puissant de vérification de contraintes sur la structure logique des données (alphanumériques et généralisées), sur les propriétés statiques et sur leur comportement dynamique. L'approche est basée sur la définition en PROLOG des concepts de base du modèle TIGRE. On peut ainsi représenter, contrôler et manipuler la sémantique associée à des applications évoluées sur des données généralisées. Ceci est entièrement sous-traité à PROLOG (la gestion des objets généralisés étant dévolue au serveur TIGRE). Pour plus de détails, voir [TIG7 83], [TIG12 83], [TIG15 84] et [TIG18 84].

Chapitre 3

LE MODELE DE DONNEES TIGRE

SOMMAIRE

3.1. APPROCHE DE MODELISATION

Un modèle de données est un formalisme (un ensemble cohérent de concepts) utilisé pour définir la sémantique des objets concernés par une application de bases de données. Cette sémantique correspond aux proprietés de structure et comportement des objets définis. En particulier, elle tient compte de l'effet que les opérateurs existants ont sur ceux objets. Ces proprietés des objets sont exprimées par ce qui est appelé le schéma conceptuel d'une application.

Le modèle de données TIGRE a été défini dans le but de fournir le formalisme nécessaire à la modélisation d'applications qui manipulent des données généralisées. Nous considerons le schéma conceptuel comme une collection de types de données (comme ceci a été fait dans [Bro 80], [Bro 81], [Sch 77], [LMWW 79], [SNF 79], [BRA 79]). Les types de données sont définis en utilisant une collection de structures de données qui, dans notre cas, correspondent à des catégories de types de données. Par ailleurs, les types de données obtenus peuvent être combinés à l'aide d'opérateurs de types pour produire des nouveaux types.

Finalement, un mécanisme de définition de contraintes peut être utilisé pour maintenir l'intégrité sémantique des données correspondant aux differents types de données. Ce mécanisme est necessaire pour définir et appliquer des contraintes d'intégrité qui ne sont pas inhérentes aux structures de données. Les données peuvent être créées, supprimées et modifiées grace a des opérateurs de données, associés aux types ou structures de données correspondantes. Notre approche est de considerer un modèle de données comme une collection de structures de données avec leurs opérateurs de types et de données correspondants, ainsi qu'un mécanisme de contraintes d'intégrité.

Les sections suivantes introduisent le modèle TIGRE comme une extension du modèle Entité-Association [Che 76] utilisant une approche typée. L'extension fournit des types de données correspondant aux données généralisées, un mécanisme de typage fort et un ensemble d'operateurs de types. Les opérateurs de types fournissent une capa-

cité d'abstraction au sens bases de données: généralisation, agrégation. La généralisation est obtenue par trois opérateurs: spécialisation, union et intersection. Deux formes d'agrégation sont offertes: agrégation d'entités et agrégation d'associations. Les énoncés de définition de LAMBDA servent à définir des types (eventuellement par application d'un opérateur sur un ou plusieurs types existants). Les opérateurs de données sont fournis par les énoncés de manipulation de LAMBDA. Nous n'avons pas abordé le mécanisme de définition de contraintes d'intégrité.

3.2 STRUCTURES DE DONNEES

Il existe trois structures de données dans le modèle TIGRE: celles correspondant aux types de base, aux types construits et aux types-classe.

3.2.1. Types de base

Les types de base sont soit simples: entier, réel, booléen, chaines de caractères, soit restreints: scalaire et intervalle, correspondant aux types "enumerated" et "subrange" de PASCAL. On les appelle restreints parce qu'ils spécifient implicitement une contrainte d'intégrité par rapport aux types de base simples sur lesquels ils sont définis. Par exemple, une valeur d'un type intervalle est un entier ou un réel qui doit être compris entre une limite inférieure et une limite supérieure données.

exemple: Type Argent : (3000..50000);

3.2.2. Types construits et le constructeur Document

Les types construits sont obtenus par application des constructeurs suivants : "Renommage", "Enregistrement", "Tableau" et "Document". Le constructeur de Renommage offre la notion forte de type en permettant

de renommer un type de base.

Ainsi, la distinction entre des types sémantiquement différents définis sur le même type de base est rendue possible en utilisant ce constructeur. Le constructeur enregistrement est similaire au "record" de Pascal, mais il n'est défini que sur des types de base ou renommés pour des raisons d'orthogonalité dans la construction de types. Ceci empêche par exemple de définir des types enregistrement ayant des constituants définis eux-mêmes comme des enregistrements. Les constituants des enregistrements peuvent prendre la valeur "null", mais on doit le déclarer dans la définition du type Enregistrement. Le constructeur Tableau (Array) produit des types dont les réalisations sont des listes d'un même type de base ou renommé.

Exemples: Type Hist-Travaux : Array (12) of String(15) ;

```
Type Adresse : Record
                    numéro : Integer (nulls allowed)
                    rue, ville : String(20)
               end;
```

Le constructeur Document est utilisé pour la définition de ce que l'on entend par "document généralisé" dans le cadre du projet TIGRE [TI3 83], [BRV 83]. Les types document définis à l'aide de ce constructeur spécifient une structure hiérarchique qui doit être satisfaite par toutes les réalisations (documents) du type. Celle-ci est satisfaite par les documents finis et partiellement par les documents en cours de création. Ces documents, quelque soit leur état de définition, pourront être stockés dans la base.

La structure hiérarchique des types document est construite à l'aide d'opérateurs de composition qui pourraient être vus comme des sous-constructeurs du constructeur Document. Ces sous-constructeurs construisent ce que l'on appelle des éléments composés (noeuds non-feuilles dans la hiérarchie) à partir des objets déjà définis. Ces

derniers peuvent être des composants élémentaires appellés unités, des éléments composés ou des structures hiérarchiques de types Document déjà définis.

Une unité possède l'une des natures suivantes: Texte, Graphique, Image, Formule, Tableau, Programme, Vocal ou Référence (à un élément associé - voir plus bas).

Les constructeurs de structure sont:

- Mise en séquence d'objets (agrégat) : "begin ... end"
- Liste d'objets d'une même classe : "list (min, max) of ..." . Min et max contraignent la cardinalité de la liste.

- Choix entre plusieurs objets : "case s of v_1: ..., v_n: ...; " s est appelé le sélecteur et $v_1,...,v_n$ sont les valeurs du sélecteur.

- Affectation de valeurs constantes : ":= nom-constante". La constante peut être une unité ou un élément associé declaré dans une partie reservée aux constantes.

Le constructeur document permet l'utilisation de certains éléments, appelés éléments associés. La position de ces derniers dans le document n'est pas toujours liée à la structure mais peut être liée à la présentation. Il peut être fait réference dans le document à ces éléments associés. Chaque élément associé appartient à une classe d'éléments associés. On distingue les classes d'éléments associés suivantes: illustration, note, énoncé bibliographique, marque interne, commentaire, paramètre et fonction.

Chaque classe d'élément associé définit une structure, des règles de cohérence et des règles de présentation logique associées. Par exemple, pour la classe énoncé bibliographique, les règles de cohérence (qui s'appliqueraient aux documents complets) pourraient être les suivantes:

- Correspondance bijective entre les énoncés et les documents décrits par les énoncés.

- Tout énoncé est cité au moins une fois (une citation étant une unité référence à un énoncé bibliographique).

- A toute citation dans un document correspond un et un seul énoncé.

Les présentations logiques attachées aux classes d'éléments associés doivent être compatibles: la présentation des éléments associés et de leurs citation doit permettre de les distinguer (par exemple, la référence à une note en bas de page est présentée différement de la référence à un énoncé bibliographique).
Les classes paramètre et fonction ont une importance particulière parce qu'ils font la liaison entre le modèle de données TIGRE et le modèle de document. Un paramètre désigne une partie variable d'un document, accesible directement par son nom, sans avoir besoin de passer par la structure. Une fonction désigne une règle de calcul précise, par exemple, des calculs arithmétiques, des références à des objets externes au document se trouvant dans la base (une partie d'un document, une réalisation d'un type quelconque dans la base). Toute fonction est nommée et éventuellement munie de paramètres formels et de conditions d'évaluation.

La liste des classes d'éléments associés n'est pas exhaustive. C'est le rôle du concepteur du modèle de document de définir de nouvelles classes d'éléments associés adaptés aux besoins de certains utilisateurs ou de documents specialisés. L'utilisateur ne peut pas créer ses propres classes d'éléments associés. Autrement dit, le constructeur "élément associé" n'est pas accessible à l'usager final. En conséquence, les classes d'éléments associés sont définies au même niveau que les types Document, et seront donc communes à tous les documents.

Les classes d'éléments associés ne seront liées à un document que si elles sont explicitement nommées dans la définition du type Document correspondant ou lorsque des éléments de ces classes seront créés dans le document .

Si l'on définit un élément associé lors de la définition d'un type Document D, sa présence est obligatoire dans tous les documents qui sont des instances de D. Dans la définition du type Document l'élément associé est identifié par un nom de constante qui est déclaré dans une partie reservée aux constantes (voir paramètre "titulaire" dans l'exemple ci-après).

Il est possible d'associer à chaque élément associé ou unité d'un document un ou plusieurs "attributs". Les attributs sont employés pour attacher une certaine présentation sur un élément donné. Certains attributs peuvent être fixés dans la définition du type document.

Les documents seront manipulés (crées, saisis, modifiés) dans l'éditeur interactif que l'on développe dans le cadre du projet TIGRE. Chaque nature d'unité (texte, graphique, formule,...) a un jeu d'opérateurs associés offerts par l'éditeur.

```
Type Clause : document
              structure

                begin
                |
                | Titre : texte
                | Contenu : Liste (1,*) of unité;
                |
                end;
              end;
```

```
Type  Contrat-Travail: document

                structure
                  begin
                  |
                  |  Contractants : begin
                  |                 |
                  |                 |  Employeur : Text:=TO;
                  |                 |  Employé : référence
                  |                 |                 := Titulaire ;
                  |                end;
                  |
                  |  Corps : List (1,*) of clause.structure;
                  |  Signature : image;
                  |
                  end;

                constantes
                  |
                  |  Unité TO := 'Compagnie ABC';
                  |  Paramètre Titulaire : texte ;
                  |            date-embauche : texte ;
                  |            établissement : texte;
                  |            Niveau : texte;
                  |  Fonction salaire-embauche :   SALAIRE(niveau,
                  |                                établissement);
                end;
```

Dans cet exemple, pour construire le corps du type document Contrat-travail, nous avons utilisé la structure hiérarchique du type document Clause (ce qui est représenté par "clause.structure"). Le type "unité" représente n'importe quelle unité (texte, image,...). Quatre paramètres et une fonction doivent être présentes dans n'importe quel document de type Contrat-travail; le paramètre "Titulaire" est contraint à apparaitre à un endroit spécifique dans la structure (bien qu'il peut apparaitre aussi ailleurs).

Dans ce qui suit, nous utilisons la notation VS(T) pour indiquer l'ensemble de valeurs du type T. Nous définirons VS(t_j) àù t_j est un type document.

$$VS(t_j) = VS(SOUS\text{-}ARBRE(t_j))$$

La notation VS(SOUS-ARBRE(t_j)) correspond à l'ensemble de valeurs que peuvent prendre les arborescences des documents qui sont des instances de t_j; nous considérons t_j comme étant le noeud racine de l'arborescence de tous les documents du type. Maintenant, pour un noeud n en général, nous définirons SOUSARBRE(n). On dénotera par CONSTRUCTEUR(n) le constructeur à l'aide duquel le noeud n a été défini. nous avons :

(a) Si CONSTRUCTEUR(n) = agrégat, et les constituants de n (les fils dans l'arborescence) sont les noeuds $n_1,\ldots,n_k$, alors

$$VS(SOUS\text{-}ARBRE(n)) = VS(SOUS\text{-}ARBRE(n_1)) \times \ldots \times VS(SOUS\text{-}ARBRE(n_k))$$

(b) Si CONSTRUCTEUR(n) = liste, et les constituants de n sont les noeuds $n_1,\ldots,n_k$, tous étant construits par le même constructeur et k étant compris entre les bornes minimale et maximale de la liste, nous avons :

$$VS(SOUS\text{-}ARBRE(n)) = VS(SOUS\text{-}ARBRE(n_1)) \times \ldots \times VS(SOUS\text{-}ARBRE(n_k))$$

(c) Si CONSTRUCTEUR(n) = choix, et n peut être construit suivant les valeurs $v_1,\ldots,v_k$ du sélecteur s, alors

$$VS\ (SOUS\text{-}ARBRE(n)) = \bigcup_{i=1}^{k} VS\ (SOUS\text{-}ARBRE(n \text{ suivant } v_i))$$

Ici nous considérons que le noeud "n suivant v_i" a la signification suivante: CONSTRUCTEUR(n suivant v_i) donne le constructeur qui construit le noeud n suivant la valeur v_i.

(d) Si n est construit suivant la structure d'un type document T alors

VS (SOUS-ARBRE(n)) = VS (SOUS-ARBRE(T))

(e) Si n est une unité de nature U, alors

VS (SOUS-ARBRE(n)) = VS (U)

Pour chaque nature U, VS (U) est l'ensemble de tous les objets de nature U que l'on pourra construire avec l'éditeur. Nous ne définirons donc pas cet ensemble ici; pour nous, ces ensembles sont considérés comme des ensembles de base pour définir les ensembles de valeurs d'un type document.

3.2.3. Types classe

Les types-classe peuvent être des types-entité ou des types-association. Les types-entité correspondent à des ensembles d'entités ("entity sets") et les types-association à des ensembles d'associations dans le modèle Entité-Association (ER) [Che 76]. Les sections suivantes donnent les définitions des types-classe au niveau conceptuel et au niveau des structures de données. Ensuite, nous donnons des examples de définitions de types-classe avec LAMBDA et, finalement, nous présentons la sémantique des variables correspondant aux types-classe.

3.2.3.1. Définitions au niveau conceptuel

En accord avec [Che 76], au niveau conceptuel, les entités existant dans notre esprit sont classées en differents ensembles d'entités tels que EMPLOYE, PROJET et DEPARTEMENT.

Un ensemble d'associations A_i est une relation mathématique entre n entités, chacune appartenant à un ensemble d'entités E_i:

$$\{(e_1,\ldots,e_n) \mid e_i \in E_i,\ i=1,\ldots,n\}$$

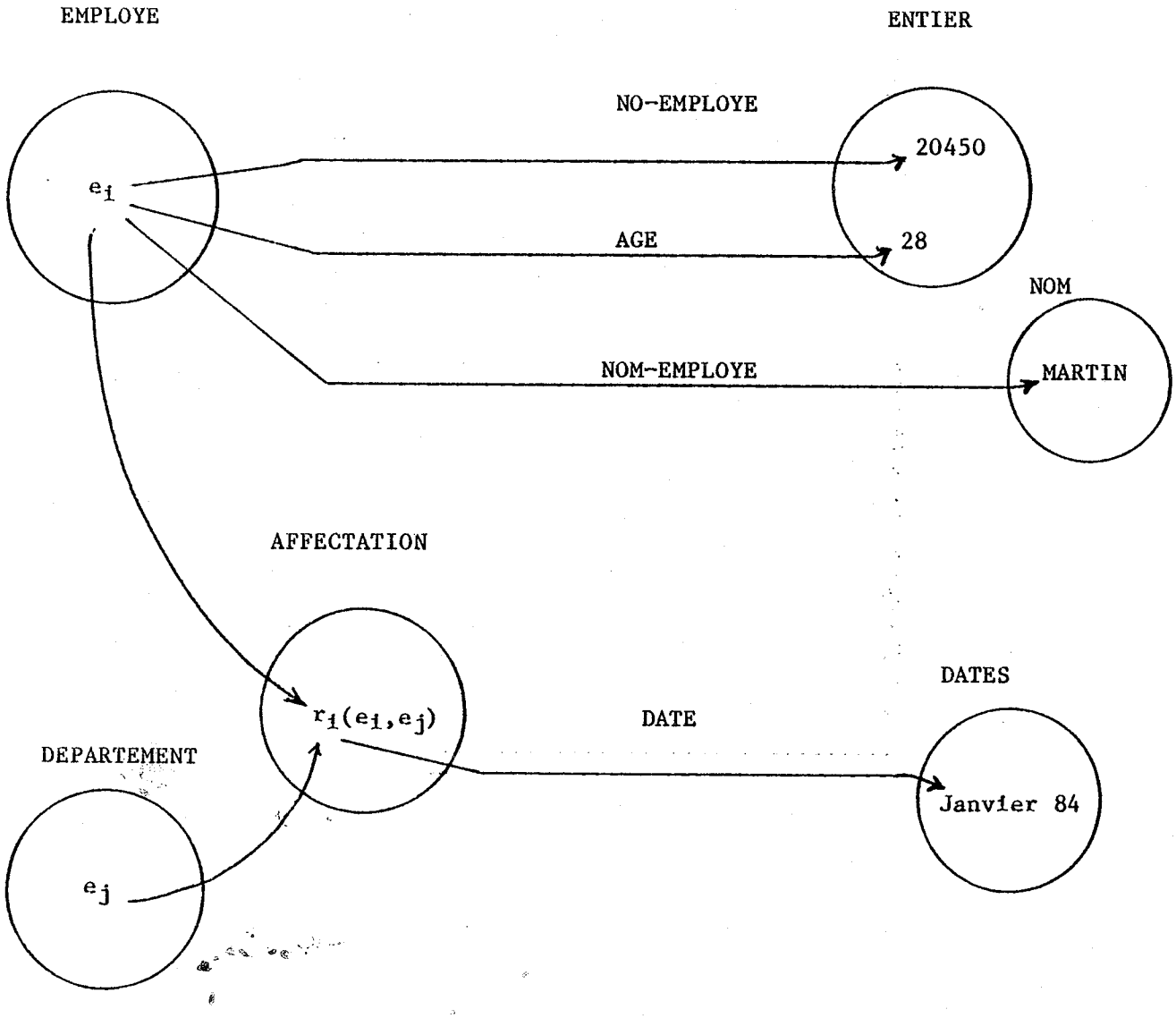
et chaque n-uplet d'entités $(e_1,\ldots,e_n)$ est une association.

Un attribut est définit comme une fonction entre un ensemble d'entités (ou un ensemble d'associations) et un ensemble de valeurs:

$$f: E_i \text{ ou } A_i \longrightarrow V_i$$

Nous considerons que des ensembles de valeurs comme ENTIER ou NOM sont les ensembles de valeurs des types de données correspondants.

ENSEMBLE D'ENTITES	ENSEMBLE D'ASSOCIATIONS	ATTRIBUT	ENSEMBLE DE VALEURS



Toujours au niveau conceptuel, nous définirons les types-classe en utilisant la notion de fait. Un fait regroupe une entité ou une association avec les valeurs de ses attributs. Dans l'exemple ci-dessus, $(e_i,20450,28,MARTIN)$ et $(r_i(e_i,e_j),janvier\ 1984)$ sont des faits. La signification du premier est: le NO-EMPLOYE de l'EMPLOYE e_i est 20450, son AGE est 28 et son NOM est MARTIN. La signification du second est: l'EMPLOYE e_i travaille dans le DEPARTEMENT e_j depuis Janvier 1984.

Un type-classe C, associé à un ensemble particulier E d'entités (ou d'associations), est défini comme l'ensemble de tous les ensembles possibles de faits à propos des entités (ou associations) dans E qui satisfont un ensemble eventuellement vide de contraintes d'intégrité:

$$VS(C)=\left\{F:\left\{(e,v_1,\ldots,v_n)\mid v_i=A_i(e)\in VS(t_i)\ \text{et}\ e\in E\right\}\mid IC(F)=vrai\right\}$$

où les A_i sont les attributs de E, $VS(t_i)$ est l'ensemble de valeurs du type t_i dans lequel A_i prend ses valeurs et IC est un prédicat qui exprime les contraintes d'intégrité imposées sur le type-classe C. Nous verrons plus tard que les t_i possibles sont restreints au niveau structures de données.

Il faut noter que étant donnée que les attributs sont définis comme des fonctions, dans un $F \in VS(C)$ il n'y aura pas deux faits $(e_1,v_1,\ldots,v_n)$ et $(e_2,w_1,\ldots,w_n)$ tels que $e_1=e_2$ même si $v_i=w_i$ pour $i=1,\ldots,n$.

3.2.3.2. Niveau Structures d'information

Nous nous intéressons maintenant à la représentation des concepts présentés ci-dessus. Nous considérons les types-entité et les types-association séparément.

On identifie de façon unique une entité par la valeur d'un ou plusieurs attributs qui constituent sa clé primaire. Autrement dit, une entité au niveau conceptuel est représentée par la valeur de sa clé primaire CP. En consequence, un fait $(e,v_1,\ldots,v_n)$ est représenté par un nuplet de valeurs

$$ef : (v_1,\ldots,v_n)$$

où CP : $(v_1,\ldots,v_k)$ $(1 \leq k \leq n)$ constitue la clé primaire de e (et de ef, à ce niveau). La sémantique de cette représentation est la suivante: l'entité identifiée par $v_1,\ldots,v_k$ est décrite par la valeur v_i de son attribut A_i pour $i = 1,\ldots,n$.

A ce niveau, l'ensemble de valeurs d'un type-entite E ayant comme attributs $A_1,\ldots,A_n$ est donné par

$$VS(E) = \left\{ F: \left\{ ef:(v_1,\ldots,v_n) \mid v_i \in VS(t_i) \text{ pour } i=1,\ldots,n \right\} \mid IC(F) \text{ et } \neg\exists\, ef_1, ef_2 \in F \ (ef_1 \neq ef_2 \text{ et } ef_1[A_1,\ldots,A_k] = ef_2[A_1,\ldots,A_k]) \right\}$$

où t_i est le type de l'attribut A_i. Les t_i doivent être des types de base ou des types construits; $ef_i[A_1,\ldots,A_k]$ sont les valeurs de la clé de ef_i.

On peut remarquer que les faits dans un ensemble F peuvent être considerés comme appartenant à l'ensemble de valeurs d'un <u>type fait</u> induit par les types t_i des attributs A_i. Nous le noterons ft(E). Cet ensemble de valeurs est donné par

$$VS(ft(E)) = \left\{ (v_1,\ldots,v_n) \mid v_i \in VS(t_i) \text{ pour } i=1,\ldots,n \right\}$$

$$= VS(t_1) \quad X \quad \ldots \quad X \quad VS(t_n)$$

Si nous notons $Attr(E) = \{A_1,\ldots,A_n\}$ les attributs de E

$Attr\text{-}cle(E) = \{A_1,\ldots,A_k\}$ les attributs appartenant à la clé de E

$Types(A_1,\ldots,A_n) = t_1,\ldots,t_n$ où t_i est le type de A_i

$VS(t_1,\ldots,t_n) = VS(t_1) \times \ldots \times VS(t_n)$

alors, $VS(ft(E)) = VS(Types(Attr(E)))$

et si le prédicat qui dit qu'il ne peut pas exister deux faits différents dans F ayant les mêmes valeurs de la clé primaire est noté CLE(F,Attr-cle(E)), alors nous pouvons écrire

$$VS(E) = \{F \in P(VS(ft(E))) \mid CLE(F,Attr\text{-}cle(E)) \text{ et } IC(F)\}$$

La représentation des types-association est obtenue de façon similaire. Une association r : $(e_1,\ldots,e_n)$ entre des entités $e_1,\ldots,e_n$ est représentée par un nuplet

$$(CP_1,\ldots,CP_n)$$

où CP_i est la valeur de la clé primaire de l'entité e_i. Un fait d'une association est donc représenté par un nuplet

$$rf : (CP_1,\ldots,CP_n, v_1,..,v_m)$$

où v_i est la valeur de l'attribut A_i propre à l'association. Ici, CP_1, ..., CP_n constituent la clé primaire de rf. Nous dirons que les faits $ef_1,\ldots,ef_n$ identifiés par $CP_1,\ldots,CP_n$ respectivement, participent au fait rf.

Pour un type-association A liant des types-entité $E_1,\ldots,E_n$ et ayant des attributs propres $A_1,\ldots,A_m$, nous avons

$$\text{Attr}(A) = \bigcup_{i=1}^{n} \text{Attr-clé}(E_i) \cup \{A_1,\ldots,A_m\}$$

$$\text{Attr-cle}(A) = \bigcup_{i=1}^{n} \text{Attr-clé}(E_i)$$

Nous pouvons supposer que les noms des attributs appartenant à Attr-clé(A) sont tous différents; en effet, s'il y a deux attributs homonymes on peut les préfixer avec le rôle que leurs types-entité jouent vis-à-vis du type-association (voir section suivante).
L'expression qui donne l'ensemble de valeurs de A est donc identique à celle de VS(E) en changeant E par A. Si nous appellons $CP(E_i)$ l'ensemble des clés primaires de E_i donné par

$$\{ v \in VS(Types(Attr\text{-}clé(E_i))) \mid \exists\ ef \in VS(ft(E_i))\ (\ ef[Attr\text{-}clé(E_i)] = v)\}$$

alors, $VS(ft(A)) = CP(E_1)\ X\ \ldots X\ CP(E_n)\ X\ Vs(Types(A_1,\ldots,A_m))$

Comme pour les types-entité, les types des attributs $A_1,\ldots,A_m$ sont des types de base ou des types construits.

3.2.3.3. Définition des types-classe avec LAMBDA

LAMBDA est un langage de définition et manipulation correspondant aux structures de données définies plus haut. Les énoncés du DDL de LAMBDA permettent la définition de types appartenant aux trois structures de données considerées dans le modèle TIGRE: types de Base, Construits et Classe. Le LMD fournit les opérations sur les réalisations de ces types. Nous avons déjà donné des examples de définition de types de base et construits. Nous présentons ensuite les énoncés de définition de types-classe.

Les attributs des types-classe prenent leurs valeurs dans des types de base ou construits. Comme nous l'avons déjà dit, les types construits fournissent un moyen d'abstraction et modularité lorsqu'ils constituent l'ensemble de valeurs d'un attribut d'un type-classe. Les types Array, par exemple, sont très utiles pour modéliser des attributs multi-valués.

Le schéma de la base de données est composé par une collection de types-classe Les définitions des types de base et construits appartiennent au schéma seulement à travers les types-classe. En effet, leurs valeurs n'ont pas une existence indépendante dans la base de données; elles n'existent que comme valeurs d'attributs des types-classe.

Seuls les attributs non-clés d'une entité ou d'une association peuvent prendre la valeur nulle (indéfinie) s'ils sont déclarés dans le schéma avec l'option "nulls allowed". Nous utilisons l'approche suggeré dans [Dat 82]: l'usager défini la valeur interne "par défaut" et ce sera cette valeur qui sera stockée et manipulée par le système.

Dans les exemples qui suivent, les types-classe sont écrits en majuscules pour mieux les distinguer des autres types.

Exemple : Définition du type-entité EMPLOYE

```
type  EMPLOYE: Entity

   key numéro : Integer end key;
       nom    : String (20);
       salaire : Argent ; nulls-allowed (0)
       adresse : Adresse;
       tâches  : List-travaux; nulls-allowed ("?")
       categorie: (ingénieur, secrétaire, programmeur)
   end.
```

Un fait de ce type est par exemple (253, Dupont, 5400, (9 Alsace-lorraine Grenoble), (programmeur, chef-projet), ingénieur). Il correspond donc à un élément du produit cartesien des ensembles de valeurs des types des attributs. Ce produit cartesien peut être consideré comme l'ensemble de valeurs du type-fait sousjacent au type-entité. Une instance de EMPLOYE est donc un ensemble de faits appartenant au type-fait sousjacent, tel qu'il n'y a pas deux faits ayant la même valeur de clé.

Les documents étant des instances d'un type construit, ils sont considérés dans TIGRE comme la valeur d'un attribut. On peut définir par exemple le type-entité suivant:

```
type CONTRAT : Entity

  key n°-ref : Integer;
      date-signature : Date;
      texte-contrat : Contrat-travail;
end;
```

Les types-association matérialisent un lien sémantique entre des types-entité. Chaque fait d'un type-association lie un fait de chaque type-entité qui "participe" au type-association. Par exemple, (259, 301, (decembre 83)) est un fait du type association défini ci-dessous. Il lie le fait de EMPLOYE identifié par le numéro 259 et le fait de DEPARTEMENT identifié par 301.

Exemple : Définition du type-association AFFECTATION entre EMPLOYE et DEPARTEMENT (on suppose que ce dernier a été déjà défini)

```
type AFFECTATION : Relationship

      between EMPLOYE : employés (1,1)
          and DEPARTEMENT : dept-affecté (1,*);

          date : Date;
      end;
```

Les types-entités liés jouent un rôle vis-a-vis du type-association, qui est spécifié dans la définition de ce dernier. On peut imposer aussi des restrictions sur le nombre minimal et maximal de fois qu'un fait d'un type-entité peut participer à chaque association.

Ici, chaque fait de EMPLOYE doit être lié avec exactement un fait de DEPARTEMENT et un fait de DEPARTEMENT doit être lié avec au moins un fait de EMPLOYE.

Une instance de AFFECTATION est un ensemble de faits de AFFECTATION qui satisfont les contraintes de cardinalité maximale et minimale et tel qu'il n'y a pas deux faits ayant la même valeur de clé (rappel: la clé d'un type-association est l'union des clés primaires des types-entité liés). En considerant la définition de l'ensemble de valeurs de AFFECTATION, VS(AFFECTATION), le prédicat correspondant aux contraintes d'intégrité appliquées à un ensemble de faits, IC(F), aura la valeur vrai si:

pour toute valeur e_i de la clé d'un fait de EMPLOYE
et pour toute valeur s_i de la clé d'un fait de DEPARTEMENT,

$$1 \leq \text{card}\left[Fe_i : \left\{ f \mid f \in F \text{ et il existent } s, v_1, \ldots, v_n \text{ tels que } f=(e_i, s, v_1, \ldots, v_n) \right\} \right] \leq 1$$

$$1 \leq \text{card}\left[Fs_i : \left\{ f \mid f \in F \text{ et il existent } e, v_1, \ldots, v_n \text{ tels que } f=(e, s_i, v_1, \ldots, v_n) \right\} \right] < \infty$$

3.2.3.4. Variables

Il y a deux types de variables dans LAMBDA: variables base de données, et variables temporaires

Conceptuellement, une variable base de données est un moyen de nommer l'ensemble de tous les faits d'un type-classe qui sont stockés à un moment donné dans la base. Nous verrons dans la section suivante que

les opérateurs de types offerts dans le modèle TIGRE ne font pas sentir le besoin d'offrir la possibilité de pouvoir déclarer plusieurs variables base de données par type-classe (contrairement aux approches décrits dans [Sch 77], [BRA 79], [Adi 81]). Pour cela, un type classe pour nous sera type et variable à la fois. La valeur de la variable sera l'ensemble de tous les faits du type-classe qui sont stockés dans la base.

Une variable temporaire est un moyen de nommer le résultat d'une requête. En LAMBDA, le résultat d'une requête est un ensemble de faits d'un type-classe (cf.section 5.1). Dans l'interface procédurale PASCAL-LAMBDA, les variables temporaires devront être explicitement déclarées sur le type-classe correspondant (bien que celui-ci corresponde ou non à un type-classe défini dans le schéma de la base). Les valeurs des variables temporaires disparaissent quand le programme qui les a crée termine.

3.3. OPÉRATEURS DE TYPES

Les opérateurs TIGRE fournissent une capacité d'abstraction sur les objets que l'on modélise. Comme dans [SS 77], [SNF 79], [Bro 80], [HM 81], [SSW 79], [Cod 79], [Elm 81], nous explorons deux formes d'abstraction: la généralisation et l'agrégation. Les opérateurs de généralisation et d'agrégation s'appliquent sur des types-classe et produisent des types-classe: nous dirons que ce sont des types-classe dérivés. Les types-classe dérivés héritent des attributs et ils peuvent avoir des attributs propres.

Nous avons défini trois opérateurs de généralisation: la spécialisation, l'union et l'intersection. Ils sont un outil pour modéliser le fait qu'une classe d'objets est plus spécifique ou plus générale qu'une autre. On exprime implicitement une dépendance existentielle et un héritage des attributs. Par exemple, si INGENIEUR est obtenu par application de l'opérateur de spécialisation sur EMPLOYE, alors INGENIEUR hérite implicitement les attributs de EMPLOYE et tout fait de INGENIEUR doit être un fait de EMPLOYE.

Les opérateurs d'agrégation sont un outil pour modéliser le fait que, dans le monde réel, certaines entités peuvent faire partie d'autres. Nous distinguons deux opérateurs d'agrégation: l'agrégation d'entités , une extension du concept de regroupement du modèle SDM [HM 81], de l'agrégation de recouvrement de RM/T [Cod 79] et du concept d'association dans SMH+ [Bro 81], et l'agrégation associative, comme dans [SNF 79] et [SSW 79].

3.3.1 Opérateurs de généralisation

Avec les opérateurs de généralisation on forme ce que l'on appelle une hiérarchie de généralisation dans laquelle le sommet est un type-classe non derivé par généralisation et les autres noeuds sont des derivations par généralisation de ses types parents. La clé primaire de chaque type-classe derivé par généralisation dans la hiérarchie sera la clé primaire du type-classe C au sommet. Alors, pour tout type-classe D derivé par généralisation les faits de D auront la forme $(k_1,\ldots,k_m, v_1,\ldots,v_n)$ où les k_i sont les valeurs des attributs dans Attr-cle(C) et les v_i sont des valeurs des attributs hérités non-clés ou des attributs propres.

Formellement, Attr-clé(D) = Attr-clé(C) et l'expression de VS(D) est la même que celle de VS(C) en changeant C par D (cf. section 3.2.3.2). Pour définir completement les opérateurs de généralisation il faut définir VS(ft(D)) pour chacun.

Dans le but de définir formellement les ensembles de valeurs des types derivés (par généralisation et par agrégation), nous introduisons la notation suivante:

Soit E un ensemble de faits ayant les mêmes attributs, notés Attr(E), soit C un type-classe,

(a) F(E,C) est un ensemble de fonctions qui font correspondre à chaque fait de E un fait de VS(ft(C)) en préservant les valeurs des attributs communs.

$$F(E, C) = \{ f: E \longrightarrow VS(ft(C)) \mid \forall e \in E \ (e[A] = f(e)[A]) \}$$

où $A = Attr(E) \cap Attr(C)$

On dira que chaque f F établit une correspondance entre E et C.

(b) SEL(E,C,P) est une SELection de faits de E tels qu'il existe une correspondance entre E et les faits de C qui satisfont P

$$SEL(E,C,P) = \{ e \in E \mid \exists f \in F(E,C) \ (P(f(e))) \}$$

L'opérateur de spécialisation produit un type-classe à partir d'un type-classe argument. On peut définir des attributs propres au type-classe spécialisé. Les faits d'un type-classe résultat sont ceux du type-classe parent qui satisfont un prédicat de restriction, augmentés avec les valeurs ses éventuels attributs propres. La spécialisation sur un type-entité produit un type-entité; sur un type-association, elle produit un type-association.

Avec ces définitions, si S est un type-entité specialisé à partir d'un type-entité E, avec attributs propres $A_1,\ldots,A_n$ et prédicat de restriction P, alors

$$Attr(S) = Attr(E) \cup \{A_1,\ldots,A_n\}$$

$$\text{et} \quad VS(ft(S)) = SEL(VS(Types(Attr(S))), E, P)$$

Un prédicat de restriction en LAMBDA peut être composé de combinaisons booléenes (avec les connecteurs logiques "et", "ou" et "non") de plusieurs prédicats appelés prédicats simples. Ceux-ci induisent deux types de raffinement sur les ensembles de valeurs soit des types des attributs des types-classe auxquels on applique les opérateurs de généralisation, soit des types-classe eux-mêmes.

(i) Raffinement par valeur des attributs. Par exemple,

salaire > 8000
catégorie : 'ingénieur'
sport contient 'tennis'
budget : (3500..4500)

(ii) Raffinement par schéma. Ce type de raffinement porte sur les liaisons que les faits d'un type-entité peuvent avoir avec des faits d'autres types-classe dans le schéma conceptuel. Deux types de liaisons peuvent exister: liaisons par association et liaisons par agrégation. Par exemple, si nous définissons

Type DIRECTION: Relationship

between EMPLOYE: chef (0,1)
and DEPARTEMENT: dept-dirigé (1,1)

end;

le prédicat "chef of DEPARTEMENT" appliqué à EMPLOYE est satisfait par des employés qui sont chefs d'un departement. Il s'agit ici d'une liaison par association.

Comme exemple de liaison par agrégation nous prenons l'exemple des dossiers et des lettres présenté dans la section 3.3.2. Le prédicat "part-of DOSSIER" est satisfait par des lettres (faits de l'entité "agrégée") qui appartiennent à un dossier quelconque (un fait de l'entité "agregat").

On peut ajouter une option appelée "manuelle" à un prédicat de restriction. Si cette option est prise, seuls les faits qui satisfont le prédicat et qui sont explicitement introduits dans le type dérivé appartiendront à celui-ci. Sinon, les faits qui satisfont le prédicat, appartiendront automatiquement à la classe dérivée. Dans le cas des spécialisations, s'il n'y a pas de prédicats de restriction dans la définition d'un type spécialisé, l'option manuelle devient obligatoire

parce que définir un type-classe dérivé qui aura exactement les mêmes faits que son type parent n'a pas de sens.

Par exemple, nous pouvons définir des secrétaires, des ingénieurs et des programmeurs comme des spécialisation des employés.

```
Type SECRETAIRE : Specialization-of EMPLOYE
                  where    categorie    =    'secretaire';
        Bilingue : Booléen;
      end;

   INGENIEUR : Specialization-of EMPLOYE
               where categorie = 'ingenieur';
     spécialité : (informatique, éléctronique,
                   automatique)
     école :  String(30)
   end;

PROGRAMMEUR : Specialization-of EMPLOYE
              where categorie = 'programmeur';
  qualification : (senior, junior, débutant)
end ;
```

Un type-association spécialisé matérialise une liaison sémantique entre des faits appartenant aux mêmes types-entités liés par le type-association père, ou appartenant à des types-entités dérivés de ceux-ci. Pour des raisons évidentes, les noms des rôles restent les mêmes, mais on peut modifier les restrictions de cardinalité dans le type-association spécialisé.

Formellement, soit A un type-association liant les types-entité $E_1,\ldots,E_n$. Si SA est un type-association spécialisé de A liant les types $DE_1,\ldots,DE_n$, alors le prédicat ANCETRE appliqué à E_i, DE_i pour i= 1, ... ,n doit être satisfait, c'est à dire, ANCETRE(E_i, DE_i) = vrai pour i = 1,...,n, où

$$\text{ANCETRE}(A, B) = (A = B)$$

$$\text{ou } (\exists C\ (B = \text{spécialisation}(C) \text{ et } \text{ANCETRE}(A, C)))$$

$$\text{ou } (\exists C_1,\ldots,C_n\ (B = \text{Union}\ (C_1,\ldots,C_n) \text{ et } \text{ANCETRE}(A, C_i) \text{ pour } i = 1,\ldots,n))$$

$$\text{ou } (\exists C_1,\ldots,C_n\ (B = \text{Intersection}(C_1,\ldots,C_n) \text{ et } \text{ANCETRE}(A, C_i) \text{ pour } i = 1,\ldots,n))$$

Si SA a les attributs propres $A_1,\ldots,A_n$ et s'il est spécialisé à partir de A à l'aide du prédicat P, alors

$$\text{Attr}(SA) = \bigcup_{i=1}^{n} \text{Attr-clé}(E_i) \cup \{B_1,\ldots,B_m\} \cup \{A_1,\ldots,A_n\}$$

$$= \text{Attr}(A) \cup \{A_1,\ldots,A_n\}$$

et $\quad VS(ft(SA)) = SEL(S, A, P)$

où $\quad S = CP(DE_1) \text{ X } \ldots \text{ X } CP(DE_n) \text{ X } VS(\text{Types}(B_1,\ldots,B_m, A_1,\ldots,A_n))$

Par exemple, supposons que nous avons défini le type-association LOCALISATION entre EMPLOYE et SITE; nous pouvons alors définir le type-association LOC-BUREAU spécialisé de LOCALISATION entre EMPLOYE-BUREAU (défini plus loin) et SITE. Les cardinalités et les noms des rôles restent les mêmes.

```
Type    LOC-BUREAU    :    Specialization-of    LOCALISATION
        between EMPLOYE-BUREAU
        and     SITE;
            no-de-bureau : Integer
        end;
```

Les opérateurs d'union et d'intersection produisent un type-entité comme résultat à partir de deux (ou plus) types-entité arguments ayant un ancêtre commun. Un fait appartenant à un type-entité derivé par union (intersection) doit appartenir à un (tous les) type(s)-entité argument(s) et éventuellement satisfaire des prédicats de restriction. Des options d'insertion manuelle pourront être appliquées aux types-entités arguments dans l'opérateur d'union et au type-entité résultant dans l'opérateur d'intersection.

Les attributs d'un type dérivé par intersection à partir des types-entité $E_1,\ldots,E_n$ doivent contenir les attributs de chaque type-entité. Par exemple, il semble naturel que si le type EMP-PROMETTEURS est défini comme l'intersection des types INGENIEUR et FEMME, un employé prometteur est à la fois un ingénieur et une femme, donc il doit contenir les attributs de tous les deux. Par contre, dans le cas des unions, à part les attributs propres on ne peut avoir des attributs qui n'appartienent pas à l'intersection des attributs des opérandes. Sinon, on aura des valeur nulles non applicables. On arrive donc à une dualité: les attributs d'une intersection sont les attributs propres plus l'union ensembliste des attributs des opérandes; les attributs d'une union sont les attributs propres plus l'intersection ensembliste des attributs des opérandes.

Formellement, si I est une intersection de $E_1,\ldots,E_m$ $(m \geq 2)$ ayant des attributs propres $A_1,\ldots,A_n$ $(n \geq 0)$ et des prédicats de restriction $P_1,\ldots,P_m$ (P_j étant le prédicat associé à E_j), alors

$$Attr(I) = \bigcup_{j=1}^{m} Attr(E_j) \cup \{A_1,\ldots,A_n\}$$

et $$VS(ft(I)) = \bigcap_{j=1}^{m} SEL(VS(Types(Attr(I))), E_j, P_j)$$

Symetriquement, si U est une union de $E_1,\ldots,E_m$ $(m \geq 2)$ ayant des attributs propres $A_1,\ldots,A_n$ $(n \geq 0)$ et des prédicats de restriction

$P_1,\ldots,P_m$, alors

$$Attr(U) = \bigcap_{j=1}^{m} Attr(E_j) \cup \{A_1,\ldots,A_n\}$$

et $$VS(ft(U)) = \bigcup_{j=1}^{m} SEL(VS(Types(Attr(U))), E_j, P_j)$$

Par exemple,

```
Type EMPLOYE-BUREAU : union of  SECRETAIRE
                      and  INGENIEUR manual
                      and  PROGRAMMEUR
                           where qualification = 'senior' manual
                           n°-telephone : Integer
                      end ;
```

3.3.2 Opérateurs d'agrégation

L'opérateur d'agrégation d'entités produit un type-entité à partir d'un ensemble de types-entité arguments. Si E est dérivé par agrégation d'entités à partir des types-entités E_1, ...,E_m, ces types-entités sont considerés implicitement comme des attributs de E. Des attributs propres $A_1,\ldots,A_n$ peuvent être définis pour E. Les propriétés sémantiques de E sont les suivantes:

1. Un fait de E est "moleculaire" : il contient des valeurs pour chaque attribut propre A_j et un ensemble de faits de E_i comme valeur de l'attribut de nom E_i.

2. Pour tout i, un fait de E_i peut être contenu dans au plus un fait de E (il peut ne pas être contenu dans aucun).

3. On peut imposer des contraintes de cardinalité sur le nombre maximal et minimal de faits d'un type-entité "agrégé" (un des E_j) pouvant appartenir à un fait de E.

4.On peut définir des attributs clé pour E: ils sont un sous-ensemble des attributs propres.

5.Les E_i sont des types-entité quelconques (peut-être obtenus aussi par application de l'opérateur d'agrégation d'entités).

La propriété 1. différence notre agrégation d'entités du concept de connexions de groupe ("grouping connections") de SDM dans la mesure où l'on peut définir des membres non homogènes dans un ensemble. La propriété 2. caractérise les objets "moleculaires" ou "complexes" rencontrés dans des application orientés vers l'ingénierie. Normalement, un composant appartient seulement à un objet. Par exemple, un dossier contient des lettres, des contrats, des curriculum-vitae; un circuit VLSI peut être vu come l'agrégation de circuits plus simples. Bien entendu, ce concept n'est pas approprié pour modéliser tous les ensembles. Par exemple, une personne peut appartenir à des comités différents. Dans ce cas, comme dans le modèle ER, l'ensemble COMITE peut être modélisé en utilisant des associations. Cette proriété rend notre agrégation d'entités différente à l'agrégation de recouvrement ("cover aggregation") de RM/T et au concept d'association de SMH+.

L'agrégation d'entités est le seul cas où, du point de vue formel, un attribut d'un type-classe est défini sur un type-clase. Comme le concept correspond à une agrégation, nous avons préferé de le rendre explicite en le considérant comme l'un des opérateurs possibles. Dans les énoncés de manipulation de LAMBDA l'accès à ces attributs (donc, aux faits "agrégés") est noté de façon syntaxiquement différente à l'accès aux attributs propres définis sur des types de base ou construits.

Formellement, si E est dérivé par agrégation d'entités à partir de $E_1,\ldots,E_m$ ayant des attributs propres $A_1,\ldots,A_n$ $(n \geq 1)$ et des attributs-clé $A_1,\ldots,A_k$ $(1 \leq k \leq n)$, alors

$$Attr(AT) = \{A_1,\ldots,A_n, \quad E_1,\ldots,E_m\}$$

Si m_j et M_j sot les bornes minimale et maximale pour E_j, l'ensemble de valeurs du type-fait de E est

$$VS(ft(E)) = \{(v_1,\ldots,v_n, S_1,\ldots,S_m) \mid v_i \in VS(Type(A_i)) \text{ pour } i=1,\ldots,n \text{ et } S_j \in VS(E_j) \text{ et } m_j \leq card(S_j) \leq M_j \text{ pour } j=1,\ldots,m\}$$

La propriété 2. peut être formalisée comme faisant partie des contraintes d'intégrité IC(F) définies sur les éléments F de l'ensemble de valeurs VS(E) :

$$\forall i, \forall f,f' \in F, \quad f[E_i] \cap f'[E_i] = \phi$$

exemple: Type DOSSIER : entity-aggregate of LETTRE (0,*)
and CONTRAT (1,1);
n°-de-dossier : Integer;
classeur: string (30)
end;

Dans cet exemple on suppose que le type-entité LETTRE a été déjà défini. (cf.schéma conceptuel, annexe 1). Un fait de DOSSIER a un n° de dossier, un classeur, 0 ou plusieurs faits de LETTRE et un fait de CONTRAT. On aurait pu ajouter à la définition de DOSSIER un attribut "texte-dossier" défini sur un type document contenant des lettres, le contrat et peut-être plus d'information structurée hiérarchiquement. Dans ce cas, le modèle offre une double facilité à l'usager: s'il veut retrouver des lettres qui se trouvent à un certain endroit dans la structure du document "texte-dossier", il pourra les accéder via la structure du document; s'il veut retrouver les lettres qui appartiennent au dossier et qui satisfont certaines conditions portant sur les attributs propres aux lettres, il pourra les accéder directement via l'attribut LETTRE de DOSSIER.

L'opérateur d'agrégation associative permet de voir un type-association et les types-entité liés comme un tout. Cet opérateur produit donc un type-entité E à partir d'un type-association A et

plusieurs types-entités $E_1,\ldots,E_m$ liés par A. On peut définir aussi des attributs propres $A_1,\ldots,A_n$ ($n \geq 0$). Les attributs qui forment la clé de E sont les attributs de la clé de chaque E_i . Un fait de E a la forme $(e_1, e_2,\ldots, e_m, r_1,\ldots,r_p,v_1,\ldots,v_n)$ où e_i est un fait de E_i, $r_1,\ldots,r_p$ sont les valeurs des attributs propres d'un fait de A qui lie les faits $e_1,\ldots,e_m$ et $v_1,\ldots,v_n$ sont les valeurs des attributs propres $A_1,\ldots,A_n$.

Alors,

$$\text{Attr}(E) = \bigcup_{j=1}^{m} \text{Attr}(E_j) \cup \text{Attr}(A) \cup \{A_1,\ldots,A_n\}$$

S'il y a deux attributs homonymes appartenant à deux types-entités différents, on les préfixe avec le nom du type-entité pour les distinguer. Les noms de $A_1,\ldots,A_n$ doivent être différents aux noms des autres attributs.

$$\text{Attr-cle}(E) = \bigcup_{j=1}^{m} \text{Attr-clé}(E_j)$$

L'ensemble de valeurs du type-fait de E est défini par

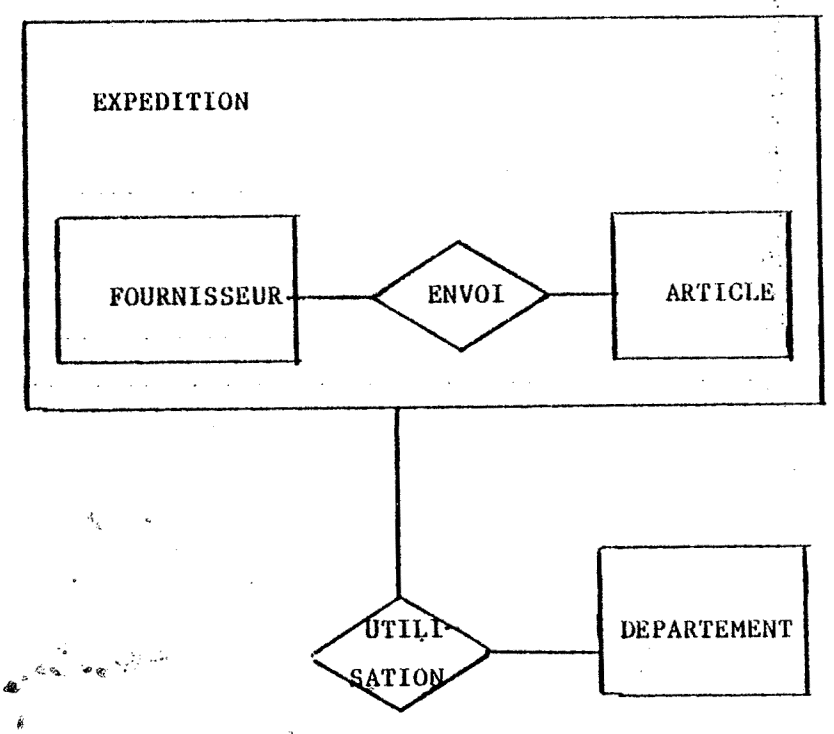
$$\text{VS}(\text{ft}(E)) = \bigcup_{j=1}^{m} \text{SEL}(\text{VS}(\text{Types}(\text{Attr}(E))), E_j, \text{vrai}) \cup \text{SEL}(\text{VS}(\text{Types}(\text{Attr}(E))), A, \text{vrai})$$

Par exemple :

```
Type ENVOI : Relationship
               between    FOURNISSEUR        and    ARTICLE
                 quantité : Integer ;
             end;

Type EXPEDITION : relationship-aggregation of ENVOI
          date : Date;
          quantité : Integer end key;
end.
```

Cet opérateur permet de voir la même information d'une manière différente. Nous pouvons, avec cette capacité d'abstraction, exprimer des associations entre des associations [SNF 79], [SSW 79], comme le montre le diagramme ci-dessous.



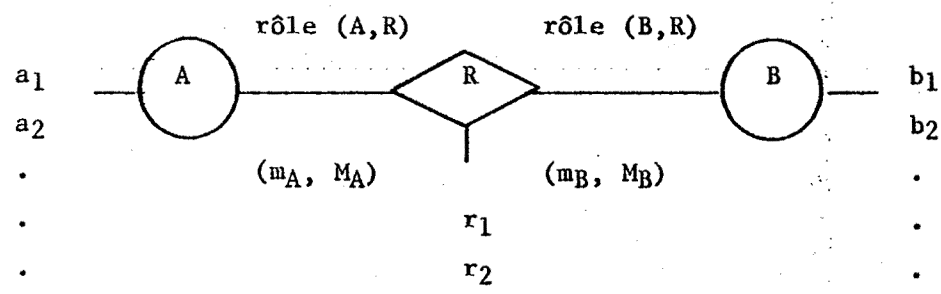
On aurait pu aussi modéliser la fourniture d'un article à un département par un fournisseur comme une association ternaire FOURNITURE entre les trois types entités concernés. Une différence importante avec l'approche proposé ci-dessus est que le fait d'avoir une seule association ne permet pas qu'un article et un fournisseur soient liés sans connaitre forcément le département. Autrement dit, on ne peut pas avoir des valeures nulles (comme dans toute association n-aire). Par contre, il suffit que la participation minimale de EXPEDITION à UTILISATION soit 0 pour que l'on puisse lier un fournisseur à un article sans avoir à lier tous les deux avec un département.

3.4. DIAGRAMMES CONCEPTUELS

Tout schéma conceptuel TIGRE pourra être mis sous forme de diagramme. Les conventions sont les suivantes :

- les entités seront représentées par des ronds,

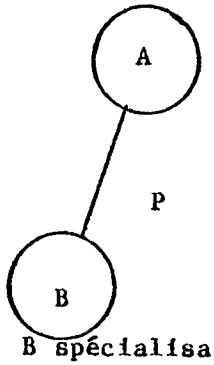
- les associations seront représentées par des losanges liant les ronds correspondants aux entités liées.



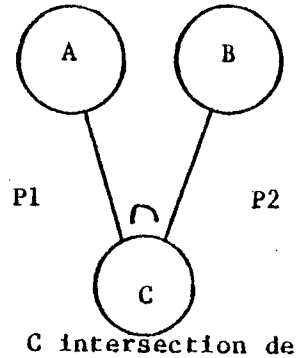
- a_1, a_2, ..., b_1, b_2, ..., r_1, r_2, sont respectivement les attributs de A, B et R.

- Rôle (A,R) est le rôle de l'entité A vis à vis de l'association R, et (m_A, M_A) sont les cardinalités minimales et maximales de fois où un fait de A peut participer à R.

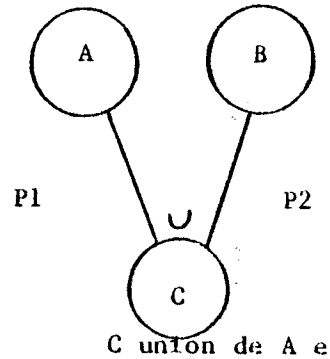
- Les entités dérivées sont représentées comme suit :



B spécialisation de A avec prédicat de restriction P

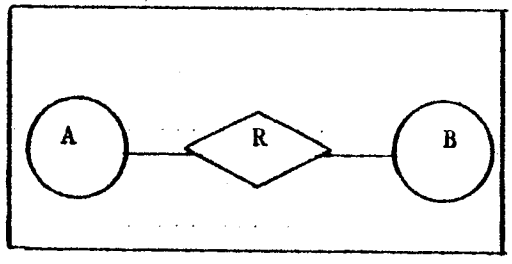


C intersection de A et B avec prédicats de restriction P1 et P2 respesctivement

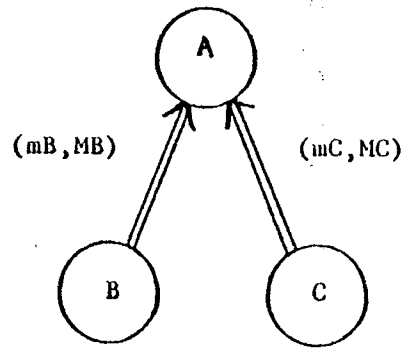


C union de A et B avec prédicats de restriction P1 et P2 respectivement

C



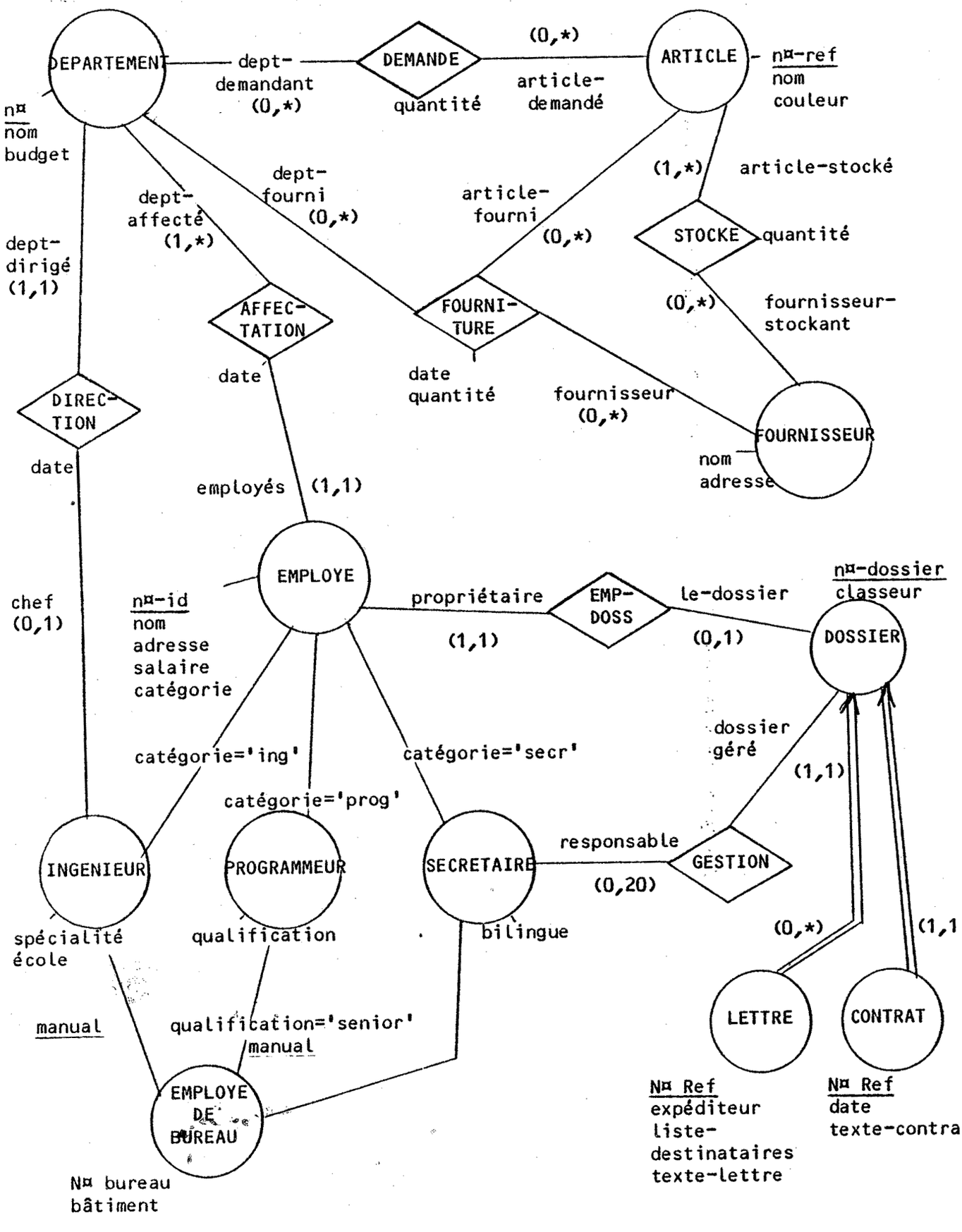
C : agrégation associative de A, B et R



A : agrégation d'entités de B et C

(mA,MA) : cardinalités minimale et maximale imposées au nombre de faits de B qui peuvent appartenir à un fait de A.

Dans les chapitres suivants nous utiliserons un schéma conceptuel de réference, une extension de celui que nous avons présenté jusqu'ici. Les énoncés de définition LAMBDA corespondants à ce schéma conceptuel se trouvent dans l'annexe 1. Le diagramme est presentée dans la page suivante.


DEPARTEMENT
dept-demandant (0,*)
DEMANDE
quantité
(0,*)
article-demandé
ARTICLE
n¤-ref
nom
couleur
n¤
nom
budget
dept-affecté (1,*)
dept-fourni (0,*)
article-fourni
(0,*)
(1,*)
article-stocké
STOCKE
quantité
dept-dirigé (1,1)
AFFEC-TATION
FOURNI-TURE
(0,*)
fournisseur-stockant
date
date
quantité
fournisseur (0,*)
DIREC-TION
FOURNISSEUR
nom
adresse
date
employés
(1,1)
EMPLOYE
chef (0,1)
n¤-id
nom
adresse
salaire
catégorie
propriétaire
(1,1)
EMP-DOSS
le-dossier
(0,1)
n¤-dossier
classeur
DOSSIER
dossier géré
(1,1)
catégorie='ing'
catégorie='prog'
catégorie='secr'
INGENIEUR
PROGRAMMEUR
SECRETAIRE
responsable
(0,20)
GESTION
spécialité
école
qualification
bilingue
(0,*)
(1,1
manual
qualification='senior'
manual
EMPLOYE DE BUREAU
LETTRE
CONTRAT
N¤ Ref
expéditeur
liste-destinataires
texte-lettre
N¤ Ref
date
texte-contra
N¤ bureau
bâtiment

Chapitre 4

LAMBDA : ENONCES D'INTERROGATION

SOMMAIRE

4.4. EXPRESSION D'ENSEMBLES

4.4.1. Famille S_1 : Ensembles non indexés

4.4.2. Famille S_2 : Ensembles indéxés

4.4.3. Famille S_3 : Ensembles construits par des opérateurs ensemblistes

4.4.4. Famille S_4 : Ensembles constantes

4.5. EXPRESSION DE CONDITIONS

4.5.1. Famille C_1 : Conditions de comparaisons de valeurs

4.5.2. Famille C_2 : Conditions d'appartenance d'une valeur à un ensemble

4.5.3. Famille C_3 : Conditions de comparaisons ensemblistes

4.5.4. Famille C4 : Combinaisons booléenes

4. LAMBDA : ENONCES D'INTERROGATION

LAMBDA (Langage de définition et Manipulation des Bases de Données généRAlisées) est le langage associé au modèle de données TIGRE. Il est un langage non procédural de haut niveau basé sur la notion de type et permettant la manipulation de variables.

Les caractéristiques les plus importantes de LAMBDA du point de vue interrogation sont :

1. La capacité de retrouver des documents ou des sous-ensembles des documents de par leurs liaisons sémantiques avec le reste d'informations de la base ou par leur contenu: par la présence de certaines chaines de caractères dans des unités texte, par la valeur des paramètres ou par le résultat de l'évaluation de fonctions.

2. L'utilisation de trajets et de variables permettant de supprimer les prédicats de jointure présents dans des langages comme SQL ou QUEL. Les trajets reflètent les liaisons sémantiques entre les données (associations, agrégations, généralisations).

3. La capacité de construire des ensembles dans les énoncés d'interrogation (et de mise à jour), ainsi que la possibilité de leur appliquer des fonctions d'agrégation.

4. Du fait que LAMBDA est basé sur la notion de type, il fournit une bonne base pour construire des interfaces à des langages de programmation.

La conception du langage a été influencée par le désir de faciliter son couplage avec les langages de programmation de haut niveau. De ce fait, nous considérons que LAMBDA ne constitue pas une interface conversationnelle adéquate pour les usagers "casuels", généralement des non-informaticiens (malgré la relativement bonne lisibilité du langage grace à la caractéristique 2.). LAMBDA est plutôt considéré comme une

couche au dessus de laquelle on pourra construire des interfaces conversationnelles de type "graphique". Dans une telle interface, le schéma des objets de la base sera visualisé graphiquement et un dispositif de pointage qui remplacera la notion de variable sera disponible (cf. section 7.5).

Ce chapitre est organisé comme suit : La section 4.1 introduit les éléments de base de tout énoncé d'interrogation de LAMBDA : les trajets, les fonctions et les conditions, et définit la sémantique d'une requête LAMBDA. La section 4.2 présente la structure des trajets et la section 4.3 celle des fonctions. Dans la section 4.4 on présente les ensembles que l'on peut former dans une requête LAMBDA et la section 4.5 sera dédiée aux conditions.

La présentation du langage s'appuiera sur des exemples qui seront construits sur le schéma conceptuel introduit dans le chapitre précedent.

4.1. INTRODUCTION

La forme générale d'un énoncé d'interrogation (aussi appelé expression de sélection) est la suivante :

```
Select  VALEURS
from  TRAJETS
[where CONDITION]
```

Comme dans la clause-from de SQL, les TRAJETS servent à désigner où se trouve l'information demandée, les VALEURS précisent l'information que l'on veut obtenir et la CONDITION spécifie des prédicats qui doivent satisfaire les données à retrouver.

Exemple 1 : Pour tous les départements, retrouver le nom des employés affectés

```
Select e.nom, d.nom
from  employés e of DEPARTEMENT d;
```

Ici, le trajet "employés e of DEPARTEMENT d" sert à designer où se trouve l'information demandée. L'entité DEPARTEMENT est la racine du trajet. Le trajet va de DEPARTEMENT à EMPLOYE par un pas à travers l'association AFFECTATION. L'entité EMPLOYE est la destination du pas. Elle est mise en évidence à travers le rôle ("employés") qu'elle joue en tant que participant dans l'association. Les variables e et d désignent respectivement un EMPLOYE et un DEPARTEMENT liés par l'intermédiaire de l'association AFFECTATION. Elles sont appelées variables de désignation, et le trajet est appelé trajet de désignation. Ainsi, ce trajet rend accessible l'information à propos des DEPARTEMENTs et des EMPLOYEs liés par AFFECTATION.

L'expression de valeurs "e.nom, d.nom" sert à préciser l'information que l'on veut obtenir : le nom du DEPARTEMENT (désigné par d) et le nom de l'EMPLOYE (désigné par e) qui y est affecté, et ceci pour tous les départements, étant donné qu'aucune condition n'est imposée sur d. L'expression de valeurs retrouve donc des valeurs des attributs des classes désignées par des variables de désignation. Pour les attributs définis sur un type construit (enregistrement, tableau ou document) on peut écrire des expressions visant à retrouver seulement une partie des valeurs des attributs. On peut également noter des fonctions d'agréga-tion, comme nous le verrons dans la section 4.3.

La condition de l'expression de sélection, appelée aussi la clause where indique les conditions que doivent satisfaire les données que l'on souhaite retrouver. Ces conditions s'établissent en spécifiant des prédicats pour les faits appartenant aux classes désignées par des variables.

Exemple 2 Retrouver le nom des secrétaires responsables des dossiers où se trouvent des lettres expédiées par Martin, ainsi que le n° de dossier et le classeur où se trouve le dossier.

```
Select  s.nom, d.n°-dossier, d.classeur
from  responsable s of dossier d containing LETTRE l
where  l.expéditeur= "Martin"
```

Dans cet exemple, le trajet de désignation est composé de deux pas. Le premier est un pas par agrégation qui va de l'entité "agrégée" LETTRE à l'entité "agrégat" DOSSIER. Le mot-clé "containing" reflète, au niveau du langage, qu'un dossier contient des lettres car le type DOSSIER est construit par l'opérateur d'agrégation d'entités à partir des types LETTRE et CONTRAT. Il est également possible de construire des pas par agrégation dans le sens inverse, de l'entité agrégat vers l'entité agrégée. Le mot-clé utilisé dans ce cas là est "part of". Par exemple, "lettre l part of DOSSIER d".

Le deuxième pas est un pas par association de l'entité DOSSIER à l'entité SECRETAIRE à travers l'association GESTION. C'est le même type de pas que nous avons rencontré dans l'exemple précédent. Quand on veut retrouver de l'information qui se trouve dans les attributs d'une association, il est nécessaire de faire des pas vers l'association. Nous y reviendrons dans la section 4.2.

Les trajets imposent des conditions implicites sur les variables de désignation. Dans l'exemple 1, pour que les valeurs des attributs "nom" des variables e et d soient retrouvées, il ne suffit pas que e $\in$ EMPLOYE et que d $\in$ DEPARTEMENT, encore faut-il que e et d soient liées à travers l'association AFFECTATION, c'est à dire, (e,d) $\in$ AFFECTATION. Chaque type de pas dans un trajet impose des contraintes implicites aux variables. Les conditions figurant dans la clause where peuvent être considérées comme des contraintes explicites.

Formellement, une expression de sélection peut s'écrire sous la forme générale suivante :

Select f_1 (X), ..., f_p (X)
From Trajets X
[Where c(X)]

Une telle expression produit un ensemble de faits. Les faits sont obtenus par l'évaluation des fonctions f_1, ..., f_p sur chaque élément X qui satisfait la condition c. Les éléments X sont des n-uplets de

faits (x_1, ..., x_n) où x_i est un fait de la classe C_i. On dit aussi que x_i désigne C_i. Les classes C_i sont des classes liées par des trajets TRAJETS X notés dans l'expression de sélection.

Chaque fonction f_i s'applique en réalité sur des faits appartenant à une seule classe C_i. La variable x_i qui désigne la classe C_i dans les trajets de désignation apparaît dans la notation syntaxique de la fonction f_i. Dans l'exemple 1, "e.nom" est considéré comme le nom d'une fonction qui s'applique sur des faits e appartenant à l'entité EMPLOYE. Le codomaine (c.a.d. l'ensemble d'arrivée) de cette fonction correspond à l'ensemble de valeurs du type sur lequel l'attribut "nom" est défini, c'est à dire, le type chaîne de caractères de longueur 30.

En général, le codomaine d'une fonction f_i, noté $Cod(f_i)$, est un ensemble de valeurs d'un type, noté "$type(f_i)$". Autrement dit,

$$Cod(f_i) = VS\ (type(f_i))$$

Une expression de sélection produit des faits qui appartiennent à l'ensemble

$$Cod(f_1)\ X\ \ldots\ X\ Cod(f_n) = VS\ (type(f_1))\ X\ \ldots\ X\ VS\ (type(f_n))$$

Dans l'ensemble des faits produits par une expression de sélection, les faits dupliqués ne sont pas éliminés, à moins que l'utilisateur le spécifie à l'aide de la clause unique en écrivant "Select unique $f_1(X)$... ".

4.2. EXPRESSION DE TRAJETS

Les trajets sont composés de plusieurs types de sous-trajets. A leur tour, les sous-trajets sont composés de plusieurs types de pas. Par la suite nous définissons formellement chaque type de pas et de sous-trajet ainsi que les expressions LAMBDA correspondantes.

4.2.1 Types de pas

Soient E_i et E_j deux entités. Un pas de E_i à E_j, noté PAS(E_i, E_j) est l'expression syntaxique en LAMBDA pour rapprocher dans un trajet deux entités ayant des liens communs dans un schéma conceptuel. Un pas peut être de trois types :

(i) Pas à travers une Association A : PA(E_i, E_j, A)

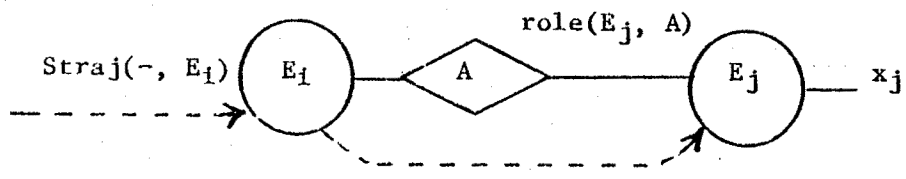
Ce type de pas utilise le rôle de E_j vis à vis de l'association A, noté rôle(E_j, A).

La forme générale d'une expression de pas à travers une association est donc:

PA(E_i, E_j, A) = role(E_j, A) [x_j] of STRAJ(-, E_i)

où STRAJ(-, E_i) est un sous-trajet qui arrive à E_i. Nous définirons cette dernière expression plus loin. La variable x_j qui désigne l'entité E_j est optionnelle si on ne l'utilise pas dans l'expression de valeurs ni dans la clause where.

La figure suivante explique graphiquement ce type de pas. La flèche en pointillés indique la direction du pas. L'entité E_j pointée par la flèche est la destination du pas ; E_i est le départ du pas.



Pour qu'un fait x_j de E_j soit retrouvé, il faut qu'il soit lié avec un fait x_i de E_i dans l'association A (bien que la variable x_i peut ne pas apparaître dans STRAJ(-, E_i)), qui à son tour satisfait des contraintes implicites et explicites. Autrement dit, $(x_i, x_j) \in A$ si A est binaire, ou $(x_i, x_j, -, \ldots, -) \in A$ si A est n-aire.

Exemple 3 Le trajet "employés e of dept-dirigé d of INGENIEUR i" a deux pas par association :

PA (INGENIEUR, DEPARTEMENT, DIRECTION)

et PA (DEPARTEMENT, EMPLOYE, AFFECTATION)

Ce trajet est équivalent au trajet "chef i of dept-affecté d of EMPLOYE e" dans la mesure où il impose les mêmes contraintes sur les faits des entités liées que le premier trajet (ces entités étant les mêmes dans les deux trajets). La seule différence concerne le sens des pas. L'usager a donc le choix d'exprimer sa requête comme il l'entend, ce qui à notre avis est très souhaitable.

Dans PA (E_i, E_j, R) le type de E_i peut être une spécialisation du type de l'entité directement associée à E_j par R.

Exemple 4 Le pas

PA (INGENIEUR, DOSSIER, EMP-DOSSIER) = le-dossier d of INGENIEUR i

est un pas valide, étant donné que tout ingénieur est un employé, alors que le pas

PA (EMPLOYE, DEPARTEMENT, DIRECTION) = dept-dirigé d of EMPLOYE e

n'est pas valide, parce qu'il y a des employés qui ne sont pas des ingénieurs, et donc le rôle de chef de département ne leur est pas applicable.

Les deux règles suivantes s'appliquent en ce qui concerne les participations des types-entités d'une hiérarchie de généralisation aux associations :

a. Si E est un type-entité, elle participe aux associations des entités E' dans la hiérarchie de généralisation tel que Attr(E') est contenu dans Attr(E).

b. Si E' est une spécialisation de E alors E ne participe pas aux associations où E' participe.

Il faut remarquer que a. n'implique pas b., étant donné que l'on peut définir une spécialisation E' de E qui n'a pas d'attributs propres.

Une variable peut apparaitre plusieurs fois dans un trajet. Le nombre de conditions implicites sur cette variable augmentent linéairement avec son nombre d'occurrences. Par exemple, si l'on veut retrouver des sécretaires qui manipulent leurs propres dossiers, on doit écrire le trajet "responsable s of le-dossier of EMPLOYE s". Les conditions implicites sur s sont: s ∈ SECRETAIRE, (s,d) ∈ DOSS-EMP, (s,d) ∈ GESTION.

(ii) Pas par agrégation de l'agrégat vers l'agrégée : PT (E_i, E_j)

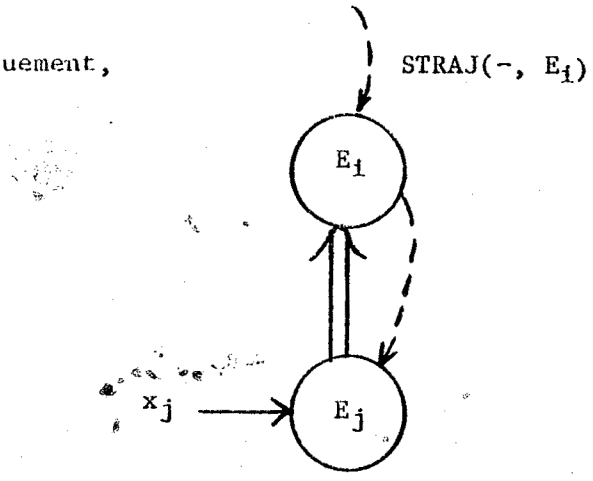
Ce type de pas utilise le nom de l'entité E_j et le mot-clé "part-of". L'entité E_i est l'agrégat, ce qui veut dire que son type est défini en utilisant l'opérateur d'agrégation d'entités à partir de types-entités dont le type de E_j. Les faits de E_j se trouvent donc agrégés dans un fait de E_i ; c'est pour cela que l'on dit que E_j est l'entité agrégée.

L'expression donnant la forme générale de ce type de pas est :

$$PT\ (E_i,\ E_j) = E_j\ [x_j]\ \underline{\text{part of}}\ STRAJ(-,\ E_i)$$

La contrainte implicite sur un fait x_j de E_j est qu'il doit faire partie d'un fait de E_i.

Graphiquement,

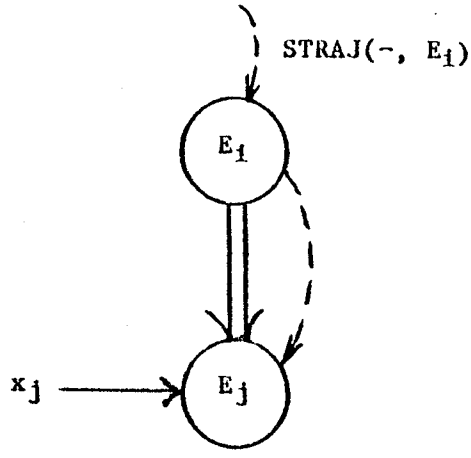


(iii) Pas par agrégation de l'agregée vers l'agrégat: PE(E_i, E_j)

Ici, on utilise le mot-clé "containing", ce qui reflète qu'un fait de l'agrégat E_j contient des faits de l'agrégée E_i (cf. exemple 2).

La forme générale est:

PE (E_i, E_j) = E_j [X_j] containing STRAJ(-, E_i)



La contrainte implicite sur les faits x_j de E_j est qu'il faut qu'ils contiennent des faits de E_i qui à leur tour satisfont des contraintes implicites et explicites. Rappel : un fait de E_i peut être contenu dans au plus un fait de E_j.

Un sous-trajet arrivant à une entité E_i, STRAJ (-,E_i), peut prendre l'une des trois formes suivantes :

a. E_i [x_i] {part of / containing} si l'on est arrivé à E_i par un pas par agrégation

b. rôle (E_i, A) [x_i] si l'on est arrivé à E_i à travers une association A.

c. E_i [x_i] si E_i est la racine du trajet

4.2.2 Sous-trajets

Un trajet est composé de sous-trajets. Ils peuvent être de deux types: linéaires ou arborescents.

4.2.2.1 Sous-trajets linéaires

Un sous-trajet linéaire entre deux entités E_i et E_j, noté STRAJLIN(E_i, E_j) est une suite finie de pas telle que la destination d'un pas est le départ du pas suivant. Le trajet de l'exemple 3 est un trajet linéaire dont tous les pas sont des pas par association.

On peut définir STRAJLIN (E_i, E_j) de façon récursive :

STRAJLIN (E_i, E_i) = STRAJ (-, E_i)

STRAJLIN (E_i, E_j) = PAS (E_{j-1} E_j) où STRAJ (-, E_{j-1}) est le dernier pas de STRAJLIN (E_i, E_{j-1})

4.2.2.2 Sous-trajets arborescents

Un sous-trajet arborescent, comme son nom l'indique, se divise en plusieurs branches à partir d'une entité ; c'est à dire, on peut effectuer deux pas ou plus, à partir d'une même entité. Ce type de sous-trajets est très utile dans le cas où l'on veut désigner des entités (et des associations) d'un schéma conceptuel telles qu'elles ne peuvent pas être liées par un trajet linéaire.

Exemple 5 Si l'on veut désigner les entités DOSSIER, CONTRAT, SECRETAIRE et EMPLOYE en utilisant les liaisons sémantiques offertes par des associations ou des agrégations dans le schéma conceptuel, il n'y a pas de trajet linéaire les liant. La solution consiste donc à construire un trajet contenant des sous-trajets arborescents à partir d'une certaine entité. L'expression

((propriétaire p, responsable r) of, contrat c part of) DOSSIER d

est un trajet arborescent à partir de l'entité DOSSIER.

Les pas à travers des associations sont regroupés dans le but de factoriser le mot-clé "of". Ils forment ce qu'on appelle un sous-trajet arborescent par association. De même, on regroupe les pas par agrégation de l'agrégat vers l'agrégée pour factoriser le mot-clé "part of", et les pas de l'agrégée vers l'agrégat pour factoriser le mot-clé "containing". Ce sont des sous-trajets arborescents par agrégation.

Bien entendu, les sous-trajets arborescents peuvent être utilisés aussi pour lier des entités qui pourraient être liées pour un trajet linéaire. Nous avons montré dans l'exemple 3 deux trajets linéaires qui lient les entités EMPLOYE, DEPARTEMENT et INGENIEUR par l'intermédiaire des associations AFFECTATION et DIRECTION. Nous pouvons les lier aussi en utilisant le trajet arborescent

(chef i, employés e) of DEPARTEMENT d

les contraintes implicites sur les variables étant les mêmes qu'avec les trajets de l'exemple 3, comme nous le verrons dans le cas général.

Si E_j est une entité, alors STRAJARB (E_j, -) dénote un sous-trajet arborescent à partir de l'entité E_j. Un sous-trajet arborescent par association à partir de E_j est noté STRAJARB-AS (E_j, -) ; de même, STRAJARB-AGT (E_j, -) et STRAJARB-AGE (E_j, -) sont des sous-trajets arborescents par agrégation à partir de E_j dont le type de pas ayant comme origine E_j sont des pas par agrégation de l'agrégat vers l'agrégée et de l'agrégée vers l'agrégat respectivement. Alors,

STRAJARB(E_j,-) = [STRAJARB-AS(E_j,-)] [STRAJARB-AGT(E_j,-)]

[STRAJARB-AGE(E_j,-)]

S'il y a plus d'un type de sous-trajet arborescent dans STRAJARB(E_j,-) on les sépare par des virgules et on met des parenthèses, comme on l'a fait dans le sous-trajet arborescent de l'exemple précédent.

4.2.2.2.1. Sous-trajets arborescents par agrégation

La forme générale de STRAJARB-AGT(E_j,-) et de STRAJARB-AGE(E_j,-) ne diffère que par le mot-clé que l'on factorise, "part-of" et "containing" respectivement. Ils sont composés d'une ou plusieurs "branches par agrégation".

En général, la forme d'une branche par agrégation de E_j à E_k, notée BRANCHE-AG(E_j, E_k) est donnée par

$$\text{BRANCHE-AG}(E_j, E_k) = \text{STRAJ}(E_k,-)\ E_k\ [x_k]$$

où STRAJ(E_k,-) est un sous-trajet linéaire ou arborescent à partir de E_k dont la forme générale sera donnée par la suite.

La forme générale de STRAJARB-AGT(E_j,-) est donnée par

STRAJARB-AGT(E_j,-) =

$$\left\{\begin{array}{l}\text{BRANCHE-AG}\ (E_j, E_k) \\ (\text{BRANCHE-AG}\ (E_j, E_{k1}), \ldots, \text{BRANCHE-AG}\ (E_j, E_{kn}))\end{array}\right\} \underline{\text{part of}}\ \text{STRAJ}(-,E_j)$$

Ici, le type de E_j est défini en utilisant l'opérateur d'agrégation d'entités à partir des entités dont $E_{k1},\ldots, E_{kn}$. Les conditions implicites sur des faits x_{ki} de E_{ki} sont celles correspondant au type de pas, autrement dit, chaque x_{ki} doit faire partie d'un fait x_j de E_j.

La forme générale de STRAJARB-AGE(E_j,-) est obtenue en changeant le mot-clé "part of" par "containing". Chaque fait x_{ki} de E_{ki} doit alors contenir un fait x_j de E_j qui satisfait à son tour des contraintes implicites et explicites.

4.2.2.2.2. Sous-trajets arborescents par association

Les sous-trajets arborescents sont également composés de branches. Cependant, ces branches, appelées branches par association, sont plus

délicates à définir, à cause de deux raisons :

(i) l'existence d'associations n-aires, où $n > 2$

(ii) les "pas vers une association" que nous définirons par la suite.

La raison (i) implique que l'on pourrait en théorie construire des trajets qui sont logiquement arborescents à partir d'une association.

Exemple 6 Si l'on veut désigner des faits des entités FOURNISSEUR et ARTICLE liés à un même fait de DEPARTEMENT dans l'association FOURNITURE, on écrira

(article-fourni a, fournisseur f) of DEPARTEMENT d

Ici, les faits a, f et d sont liés par l'association FOURNITURE, autrement dit, $(a,f,d) \in$ FOURNITURE. C'est la contrainte qui lie implicitement ces variables. Nous dirons qu'il y a un pas multiple à travers l'association FOURNITURE vers les entités FOURNISSEUR et ARTICLE qui forment la destination du pas multiple. Par contre, dans le sous-trajet linéaire

article-fourni a of fournisseur f of DEPARTEMENT d

un fait a de ARTICLE ne dépend que d'un fait f de FOURNISSEUR, c'est à dire, $(a,f,-) \in$ FOURNITURE. Pour compléter l'exemple, mettons sous forme tabulaire le contenu de l'association FOURNITURE à un moment donné:

FOURNITURE

quantite	DEPT.	FOURNISSEUR	ARTICLE
1	d_1	f_1	a_1
2	d_1	f_2	a_2
1	d_2	f_1	a_3
2	d_2	f_3	a_4

Si on s'intéresse aux fournisseurs et aux articles fournis au département d_1, avec le premier sous-trajet la réponse serait (f_1, a_1), (f_2, a_2), alors que, avec le deuxième, la réponse serait (f_1, a_1), (f_2, a_2), (f_1, a_3).

Regardons maintenant la raison (ii). Comme nous l'avons déjà vu, dans les pas par association, qu'ils soient linéaires ou arborescents, on ne désigne pas l'association elle-même. L'entité destination du pas est reconnue à travers le rôle qu'elle joue dans l'association, mais dans le pas "on ne s'arrête pas" à l'association, on passe seulement "à travers" pour s'arrêter à l'entité destination du pas. Les raisons de ce choix tiennent à la clarté et à la compacité d'écriture des trajets. Cependant, le problème arrive quand dans une requête on veut extraire de l'information qui se trouve dans une association. Par exemple, on aurait voulu savoir quelle est la quantité fournie de tel article à tel département par tel fournisseur.

Pour rester cohérent avec le principe de "faire des pas vers là d'où on veut extraire de l'information", nous avons créé les pas vers les associations. Dans un tel pas, on doit passer par au moins deux entités liées par une association avant de faire le pas vers l'association. La syntaxe des pas vers les associations est d'abord introduite à l'aide de l'exemple suivant :

Exemple 7 Désigner l'association FOURNITURE dans les deux sous-trajets introduits dans l'exemple précédent.

(a) FOURNITURE t:(a,f,d) of (article-fourni a, fournisseur f) of DEPARTEMENT d

(b) FOURNITURE t:(a,f) of article-fourni a of fournisseur f of DEPARTEMENT d

Ces deux sous-trajets finissent par des pas vers l'association FOURNITURE. Dans (a) la variable t désigne un fait de FOURNITURE qui lie les faits a, f et d. Il s'agit d'un pas des entités ARTICLE, FOURNISSEUR

et DEPARTEMENT vers l'association FOURNITURE. Par contre, dans (b), la variable t lie des faits a et f avec n'importe quel fait de DEPARTEMENT, c'est à dire, t:(a,f,-) $\in$ FOURNITURE. Il s'agit d'un pas des entités ARTICLE et FOURNISSEUR seulement vers FOURNITURE. Par rapport à l'exemple précédent si, en plus de s'intéresser aux fournisseurs at aux articles fournis au département d1, on voulait aussi connaître les quantités fournies, avec le sous-trajet (a) on obtiendrait comme réponse : (1, f_1, a_1), (2, f_2, a_2) alors qu'avec le sous-trajet (b) la réponse serait (1, f_1, a_1), (2, f_2, a_2), (1, f_1, a_3).

En général, si l'association A lie des entités E_1, ..., E_n, alors la forme syntaxique d'un pas des entités E_1, ..., E_k (où $2 \leqslant k \leqslant n$) vers l'association A notée PVA (E_1,...,E_k, A) est :

$$\text{PVA } (E_1, \ldots, E_k, A) = A\ x_a : (x_1, \ldots, x_k)$$

ce qui indique d'une part, que x_a désigne un fait de A et, d'autre part, que ce fait lie les faits x_1,...,x_k où x_i désigne un fait de E_i.

Il faut remarquer que les pas vers association compliquent la définition des sous-trajets linéaires, car le sous-trajet (b) de l'exemple précédent est un sous-trajet linéaire. A la définition des sous-trajets linéaires il faut donc ajouter une variante :

$$\text{STRAJLIN } (E_i, A) = \text{PVA } (E_j, E_{j-1}, A)\ \text{STRAJLIN } (E_i, E_j)$$

où le dernier pas du sous-trajet linéaire STRAJLIN (E_i, E_j) est, bien entendu, PA (E_{j-1}, E_j, A).

Un sous-trajet linéaire qui comporte un pas vers une association ne peut pas continuer au-delà de l'association. Ceci est dû au fait que le nom de l'association n'est pas le rôle d'aucune entité, et alors il est insensé d'écrire des trajets de la forme "...rôle of PVA(E_j,E_{j-1},A)". Si le trajet que l'on veut décrire doit logiquement continuer après l'entité E_j, il faut donc construire un sous-trajet arborescent à partir de E_j dont l'une des branches est un pas vers une association, comme le montre l'exemple suivant.

Exemple 8 Construire un trajet pour retrouver la quantité demandée des articles d'un département, ainsi que les fournisseurs stockant ces articles.

(fournisseur-stockant f, DEMANDE e:(d,a)) of article-demandé a of
DEPARTEMENT d

on voit que l'une des formes qu'une branche par association peut avoir est un pas vers une association. Ici, le sous-trajet est arborescent à partir de l'entité ARTICLE; on est déjà passé par l'entité DEPARTEMENT, et donc, une des branches peut être bien un pas vers l'association DEMANDE. L'autre branche est un pas vers FOURNISSEUR via STOCKE (elle avait pu être un sous-trajet arborescent aussi).

Une branche par association à partir de l'entité E_j passant par l'association A, notée BRANCHE-AS (E_j, A) peut donc correspondre à l'une des trois expressions suivantes :

(i) STRAJ (E_k,-) Role(E_k, A) [x_k]

Ici, la branche commence par un pas à travers A, suivie éventuellement par un sous-trajet. Pour qu'un fait x_k de E_k soit retrouvé, il doit être lié avec un fait de E_j via A comme dans tout pas à travers une association.

(ii) [PVA (E_j, E_{i1},..., E_{im}, A) of] (STRAJ (E_{i1}, -) Role(E_{i1},A) [x_{i1}],...
..., STRAJ (E_{im}, -) Role(E_{im},A) [x_{im}])

Ici la branche comporte un pas multiple de E_j vers $E_{i1},\ldots,E_{im}$ à travers A. Le sous-trajet (a) de l'exemple 7 a cette forme.

En général, les faits $x_{i1},\ldots,x_{im}$ appartenant aux entités destination du pas multiple doivent être liés par l'intermédiaire d'un même fait x_j de E_j, l'origine du pas. Autrement dit, si A est une association n-aire liant les entités E_{k1}, ..., E_{kn} alors $n \geq m + 1$ et

$$C = \{E_j, E_{i1}, \ldots, E_{im}\} \subseteq \{E_{k1} \ldots E_{kn}\} = B$$

et $(x_j, x_{i1}, \ldots, x_{im}, -, -, \ldots -) \in A$, les "-" correspondant aux entités de B - C.

(iii) PVA (E_{j-1}, E_j, A)

Dans ce cas, la branche est un pas vers l'association A. Elle est une branche terminale (elle ne se ramifie plus), comme pour les sous-trajets linéaires qui comportent ce type de pas. Le dernier pas avant d'arriver à E_j est un pas $PA(E_{j-1}, E_j, A)$.

Une fois que l'on a défini la forme générale d'une branche par association, nous sommes en mesure de définir des sous-trajets arborescents par association à partir d'une entité E_j.

STRAJARB-AS $(E_j,-)$ =

$$\left\{\begin{array}{l} \text{BRANCHE-AS}(E_j, A_1) \\ (\text{BRANCHE-AS}(E_j, A_1), \ldots, \text{BRANCHE-AS}(E_j, A_n)) \end{array}\right\} \underline{\text{of}}\ \text{STRAJ}(-,E_j)$$

Bien entendu, $A_1,\ldots,A_n$ sont des associations auxquelles l'entité E_j participe.

Il ne nous reste qu'à définir l'expression STRAJ $(E_i,-)$, qui donne la forme générale qui prend en compte tout sous-trajet à partir de E_i.

STRAJ $(E_i,-)$ = [STRAJARB $(E_j,-)$] STRAJLIN (E_i, E_j)

Ceci signifie qu'un sous-trajet est linéaire jusqu'à une certaine entité E_j (qui peut être égale à E_i) à partir de laquelle il peut devenir arborescent.

4.2.3 Trajets

Un trajet peut être regardé comme étant composé par une racine et un sous-trajet à partir de la racine. La racine est le nom d'une classe désignée éventuellement par une variable. Nous avons déjà donné beaucoup d'exemples où la racine est une entité. On peut construire aussi un trajet ayant une association comme racine pour aller ensuite aux entités liées.

Exemple 9 Dans le trajet

DEMANDE e:(article-demandé a, dept-demandeur d)

la racine est l'association DEMANDE. Les variables de désignation vérifient aussi la contrainte implicite e:(a,d)$\in$ DEMANDE. On doit regarder ce type de trajets comme une commodité syntaxique utile dans le cas où on veut désigner une association et ses entités liées. Le trajet suivant, qui a comme racine l'entite DEPARTEMENT, une des entités participant à DEMANDE, est équivalent au trajet ci-dessus.

DEMANDE e:(a,d) of article-demandé a of DEPARTEMENT d

Les trajets ayant comme racine une association sont les seuls cas où l'on fait un accès à l'association avant de passer par les entités liées. Par contre, on ne peut pas construire des sous-trajets à partir d'une de ces entités. Ceci est dû au fait que, d'une part, l'écriture syntaxique de ce type de trajets deviendrait peu claire, et d'autre part, cette possibilité a déjà été offerte dans les trajets ayant comme racine une entité. Donc, si l'on a besoin de désigner des entités qui ne sont pas toutes liées par la même association, on n'a qu'à construire un trajet qui commence par une entité. La forme générale d'un trajet à partir d'une classe (entité ou association) C, noté TRAJ (C), est la suivante :

$$\text{TRAJ (C)} = \begin{cases} \text{(i) STRAJ (C,-) C } [x_C], & \text{si C est une entité} \\ \text{(ii) C } [x_C] \text{ (Role } (E_1, C)\ x_1, \ldots, \text{Role } (E_n, C)\ x_n), & \text{si C est une association et } E_1,\ldots,E_n \text{ sont les entités liées par C} \end{cases}$$

Dans une expression générale de sélection LAMBDA, nous pouvons noter plusieurs trajets. Ceci est très important, car si nous n'en admettions qu'un, même arborescent, la puissance d'expression resterait limitée. En effet, il serait impossible d'écrire des requêtes où l'on rapproche deux entités par une liaison sémantique autre que celles découlant des associations dans le schéma (voir exemple 10). D'autre part, comme nous le verrons dans la section 4.4, les ensembles que nous pouvons former en LAMBDA font toujours référence à des trajets via des variables de désignation. Autrement dit, il n'y a pas de trajets internes dans l'expression d'ensembles.

Exemple 10 Supposons qu'on s'intéresse aux fourniseurs qui habitent la même ville que le chef du département de jouets. Il nous faut deux trajets pour répondre à la requête, car il n'y a pas d'association liant les employés et les fourniseurs. Pour des explications concernant la clause-where, voir les sections 4.3 et 4.5.

```
select f.nom
from chef c of DEPARTEMENT d, FOURNISSEUR f
where d.nom = "jouets" and c.adresse.ville = f.adresse.ville
```

L'expression TRAJETS X, qui apparaît dans la forme générale d'une expression de sélection, correspond aux trajets de désignation où l'on utilise des variables $X = (x_1,\ldots,x_k)$ pour désigner des classes notées dans les trajets. Cette expression est donnée par :

$$\text{TRAJETS } X = \text{TRAJ } (C_1), \ldots, \text{TRAJ } (C_k)$$

où $k \geqslant 1$, C_1, ... C_k sont des classes et X est un n-uplet de toutes les variables notées dans tous les trajets TRAJ (C_1).

4.3. EXPRESSION DE FONCTIONS

On appelle F_s l'ensemble de fonctions de l'expression générale de sélection, c'est à dire l'ensemble auquel doivent appartenir les fonctions f_1, ..., f_p qui apparaissent après le mot-clé select dans l'expression générale de sélection présentée dans la section 4.1. Dans cet ensemble F_s on distingue 4 sous-familles notées F_1 à F_4.

Dans la clause-where d'une expression LAMBDA on peut avoir des fonctions appartenant non seulement à F_s mais aussi à la famille F_5 de fonctions constantes que nous définissons plus loin. Par la suite, on supposera qu'il y a k classes $C_1,\ldots,C_k$ définies dans le schéma conceptuel et que la variable u_i désigne la classe C_i.

4.3.1. Famille F_1 : Choix des faits d'une classe

Les fonctions de cette famille permettent de retrouver des faits appartenant à une classe désignée dans les trajets d'une expression de sélection. Les faits retrouvés doivent satisfaire les contraintes implicites dans les trajets ainsi que les contraintes explicites notées dans la clause-where.

Exemple 11 Retrouver le dossier 520 et son propriétaire

```
Select e, d
from propriétaire e of DOSSIER d
where d.n°-ref.=520 ;
```

Ici, les deux fonctions sont représentées syntaxiquement par les variables e et d. La première retrouve les faits de EMPLOYE qui satisfont la contrainte implicite d'être associés aux dossiers retrouvés par la deuxième fonction; Cette dernière retrouve les dossiers qui satisfont la contrainte explicite de la clause-where. Tout le dossier est retrouvé, y compris le contrat et les lettres. Il n'est pas nécessaire de désigner explicitement les lettres et le contrat, car ils font partie du dossier.

Formellement, on peut peut considérer que la fonction

$$u_i : C_i \longrightarrow C_i$$

est l'identité et qu'elle s'applique aux faits de C_i qui satisfont les contraintes implicites et explicites.

Dans le but d'homogénéiser la définition de familles de fonctions, nous considérons que le codomaine de la fonction u_i, noté $Cod(u_i)$, est l'ensemble

$$VS(Types(Attr(C_i)) = VS(t_1) \times \ldots \times VS(t_n)$$

où t_j est le type de l'attribut a_j de la classe C_i.

4.3.2. Famille F_2 : Choix des valeurs des attributs d'une classe

Soit a_j un attribut de la classe C_i. Soit t_j le type de a_j. Les fonctions de F_2 sont la projection des faits de C_i sur l'attribut a_k. Syntaxiquement, elles ont la forme "$u_i.a_j$".

$$u_i.a_j : C_i \longrightarrow VS(t_j) = Cod(u_i.a_j)$$

$$(v_1,\ldots,v_j,\ldots,v_n) \longmapsto v_j$$

où v_i est la valeur de l'attribut a_i. Ces fonctions s'appliquent sur des faits $(v_1,\ldots,v_n)$ de C_i qui satisfont des contraintes implicites et explicites. Il faut rémarquer que le type t_j peut être un type enregistrement, tableau ou document. Les valeurs des types tableau étant considérés aussi comme un ensemble, c'est un des seuls cas où l'on pourra noter un ensemble dans l'expression de valeurs (pour l'autre cas, voir la section 4.3.4).

4.3.3. Famille F_3 : Choix d'un sous-ensemble des valeurs d'un attribut

Les fonctions de cette famille s'appliquent aux attributs définis sur un type enregistrement, tableau ou document. Elles servent à choisir un élément d'un enregistrement ou d'un tableau, ou un sous-ensemble d'un document, ou encore la valeur d'un paramètre ou d'une fonction d'un document.

Les fonctions applicables aux types enregistrement, à part celles appartenant à F_2, sont des fonctions d'accès aux champs des enregistrements. Elles ont la forme "$u_i.a_j.c_k$" où c_k est un champ du type enregistrement t_j dont relève l'attribut a_j. C'est une double projection des faits de C_i sur le champ c_k de l'attribut a_j.

Les fonctions appartenant aux types tableau, à part celles appartenant à F_2, sont des fonctions d'accès aux éléments des tableaux. Elles ont la forme "$u_i.a_j(k)$". Elles choissisent le k-ième élément du tableau a_j pour chaque fait de C_i qui satisfait des contraintes implicites et explicites notées dans l'expres-sion de sélection.

4.3.4. Fonctions applicables aux documents

Ces fonctions peuvent être divisées en deux catégories:

(1) les fonctions qui retrouvent un sous-ensemble du document, et
(2) les fonctions portant sur les paramètres et les fonctions.

2.2.4.1. Fonctions retrouvant un sous-ensemble des documents

Les fonctions de cette catégorie ont la forme d'un chemin dans la structure arborescente d'un document pour accéder à un sous-arbre.

Exemple 12 Retrouver le contenu de la clause 35 du contrat de Martin

```
Select contenu of clause 35 of corps of c.texte-contrat
From contrat c part-of le-dossier of EMPLOYE e
where e.nom = 'Martin' ;
```

Les fonctions de cette catégorie ont la forme générale

CHEMIN-DANS-STRUCTURE(a_j) $u_i.a_j$

où a_j est un attribut de la classe C_i designée par u_i défini sur un type document. Une valeur de a_j pour un fait u_i donnée est donc un document. CHEMIN-DANS-STRUCTURE(a_j) dénote un chemin dans la structure arborescente du document. Il doit suivre rigoureusement la structure du type document sur lequel a_j est défini. Ce chemin peut être "variable", c.a.d., on peut retrouver ou non des sous-arbres en fonction des conditions portant sur les noeuds de ce sous-arbre. Pour cela, on étend l'idée des variables de désignation aux documents en permettant qu'une variable désigne des noeuds d'un document. Les conditions sur cette variable se notent dans la clause-where de l'expression de sélection (voir exemple 14). Nous y reviendrons plus loin.

Pour mieux illustrer la syntaxe des chemins dans la structure d'un document, nous définissons les type-documents ci-dessous:

```
type Proceedings : Document

    structure
       Liste (1,*) of Session.structure;
    end.

type Session : Document

    structure
       begin
          titre : texte;
          président-de-session : texte;
          contenu : List (1,20) of Article.structure;
       end.
```

```
type Article : Document

    structure
      begin
        auteur : Liste (1,*) of texte;
        titre : texte;
        contenu : cas  type-article of
                    article-complet : Rapport.structure;
                    résumé: Liste (1,5) of paragraphe.structure;
                  end;
      end.
```

Nous supposons qu'il y a une entité CONGRES qui a un attribut appelé "texte-proc" défini sur le type Proceedings. Nous supposerons également que les types "Rapport" et "Paragraphe" ont été définis par ailleurs. Dans ce qui suit nous introduisons la syntaxe des chemins dans la structure du document à l'aide des exemples. Une définition plus générale sera présentée plus loin.

4.3.4.1.1 - Exemples

Exemple 13 Retrouver dans le proceedings du VLDB 80, le titre de chaque session et le titre des articles présentés à chaque session.

```
Select (titre, titre of article 1,* of contenu)
                        of session 1,* of c.texte-proc
from CONGRES c where c.nom = VLDB and c.année = 1980;
```

Cette expression de sélection retrouve un seul fait contenant un sous-ensemble de la valeur de l'attribut "texte-proc". Le chemin dans la structure est (triplement) arborescent. Il passe d'abord pour chaque noeud "session" ; ensuite, pour chaque session, il se ramifie pour retrouver le titre et le contenu de la session, et finalement, il se ramifie encore et passe pour chaque noeud "article" pour retrouver le titre. Dans les chemins arborescents, les sous-arbres correspondants

sont retrouvés en préordre. Dans cet exemple, tous les sous-arbres sont des unités texte.

Exemple 14 Retrouver les résumés des trois premières sessions du proceedings du VLBD 80

```
Select article a of contenu of session 1..3 of c.texte-proc
from CONGRES c
where c.nom = VLDB and c.année = 1980
  and type-article of contenu of a = 'résumé';
```

Cet exemple montre que les chemins dans la structure d'un document peuvent être conditionnés. la variable a désigne les articles du contenu des trois premières sessions du proceedings. Le prédicat associé à a dans la clause-where compare la valeur du sélecteur (type-article) du contenu de l'article selon laquelle le contenu a été construit avec 'résume'. Si pour un article a, ce prédicat est vrai, l'article (ou plus précisement, son contenu) est un résumé, et il sera donc retrouvé. On peut désigner avec des variables soit un noeud, soit un ensemble de noeuds construits suivant le même constructeur (autrement dit, les éléments d'une liste). Bien entendu, un chemin peut continuer "au-delà" du (des) noeud(s) désigné(s) par une variable.

Dans la structure du type 'proceedings', tous les éléments des noeuds construits avec le constructeur liste sont eux-mêmes construits suivant la structure d'un type prédéfini (ou sont des unités prédéfinies, par exemple, 'texte'). Ceci permet de nommer plus aisément ces éléments dans un chemin dans la structure d'un document : on les nomme avec le nom du type (par exemple : "session", "article"). Cependant, les énoncés de définition des types document permettent que les éléments d'un noeud liste soient construits avec les constructeurs cas-ou, agrégat ou liste. Dans ce cas, nous avons choisi de nommer ces éléments en donnant le nom du constructeur.

Exemple 15 Supposons que le noeud "contenu d'une session avait été construit comme suit :

```
contenu : list (1,10) of cas type-article of
                        |  article complet : ...
                        |  résumé : ...
                      end
```

Alors, si on veut retrouver le premier article du contenu de la session 2, il faut écrire "cas 1 of contenu of session 2 of c.texte-proc". Si on avait voulu retrouver le premier résumé, on aurait écrit "résumé 1 of contenu of...". Autrement dit on ne retrouve pas le premier élément de la liste (le premier article), mais le premier élément de la liste ayant été construit suivant la valeur "résumé" du sélecteur.

4.3.4.1.2. Formalisation de la syntaxe des chemins dans la structure d'un document

Pour définir formellement la syntaxe des chemins dans la structure d'un document, nous définirons la forme de tout chemin partant d'un noeud n construit à l'aide de n'importe quel constructeur K. Ce chemin sera noté CHEMIN-STRUCT(n, K). D'abord, nous introduisons la notation suivante :

- A dénote le constructeur agrégat
- L dénote le constructeur liste
- C dénote le constructeur cas-où

- CONSTRUCTEUR (n) dénote le constructeur à l'aide duquel le noeud n de la structure du document a été construit

. Si n est une unité de nature U alors CONSTRUCTEUR(n) = U

. Si n est construit avec la structure d'un type document T déjà défini, alors CONSTRUCTEUR(n) donne le constructeur le plus externe suivant lequel la structure de T est définie,

c'est à dire, CONSTRUCTEUR(n) $\in \{A,L,C\}$.

. Sinon, n est un élément composé et alors CONSTRUCTEUR(n) $\in \{A,L,C\}$

Selon cette notation, si a_j est un attribut défini sur un type document noté type(a_j), alors

CHEMIN-DANS-STRUCTURE(a_j) = CHEMIN-STRUC (a_j, CONSTRUCTEUR(type(a_j)))

Le reste de cette section est dédié à la définition de CHEMIN-STRUC(n, K) pour un noeud n construit avec le constructeur K.

Constructeur Agrégat

Soit d un document, soit n un noeud de d construit avec A et ayant comme constituants de l'agrégation les noeuds $n_1,\ldots,n_k$. Les chemins dans la structure de d qui partent de n ont la forme suivante :

$$\text{CHEMIN-STRUCT}(n,A) = \begin{cases} 1.\ [\text{CHEMIN-STRUCT}(n_i, \text{CONSTRUCTEUR}(n_i))\ \underline{of}]\ n_i\ \underline{of} \\ 2.\ ([\text{CHEMIN-STRUCT}(n_{i_1}, \text{CONSTRUCTEUR}(n_{i_1}))\ \underline{of}]\ n_{i_1},\ldots \\ \quad ..,[\text{CHEMIN-STRUCT}(n_{i_j}, \text{CONSTRUCTEUR}(n_{i_j}))\ \underline{of}]\ n_{i_j}\ )\ \underline{of} \end{cases}$$

Il y a donc deux possibilités : la première est un chemin linéaire dans la structure passant par un noeud quelconque n_i. La deuxième est un chemin arborescent à partir de n dans lequel chaque branche passe par le noeud n_{i_1}, ($n_{i_1} \in \{n_1,\ldots,n_k\}$) pour $1 = 1,\ldots,j$. De même que pour les trajets de désignation arborescents, on utilise des parenthèses et des virgules pour séparer les branches du chemin.

Constructeur Liste

Soit n un noeud de d construit avec L. Les constituants de n dans la liste sont tous des éléments construits avec le même constructeur. Ces

éléments seront notés dans la syntaxe des chemins en accord avec ce constructeur, comme nous l'avons vu dans les exemples. Nous représenterons par n() un quelconque des éléments de la liste, et par NOM-CONSTR(n()) le nom par l'intermédiaire duquel nous y ferons référence syntaxiquement dans les chemins. Alors,

NOM-CONSTR(n()) =

"Agrégat",	si CONSTRUCTEUR(n()) = A
"Liste",	si CONSTRUCTEUR(n()) = L
"cas" valeur-sélecteur	si CONSTRUCTEUR(n()) = C (cf. exemple 15)
T	si n() a été construit suivant la structure d'un type document T
U	si CONSTRUCTEUR(n()) = U, autrement dit, si n est une unité U.

Par "valeur-sélecteur" nous indiquons une valeur quelconque du sélecteur du cas-où. Par exemple, la valeur "résumé" du sélecteur "type-article" dans l'exemple 15.

Les chemins dans la structure de d qui partent de n ont la forme suivante :

CHEMIN-STRUCT(n, L) = [CHEMIN-STRUCT(n(), CONSTRUCTEUR(n())) of]
NOM-CONSTR(n()) ELEMS of

où l'expression ELEMS, correspondant aux ordinaux des éléments de la liste par lesquels passent les branches du chemin, peut prendre l'une des formes suivantes :

(a) un entier i

(b) un intervalle i..j, j pouvant être indéterminé (*)

(c) une liste d'entiers i_1, i_2, ... i_n (où n ≥ 2)

(d) une variable x

Dans (a) il s'agit d'un chemin linéaire : on ne s'intéresse qu'à un seul élément de la liste, indiqué par l'ordinal i. Dans les cas (b) et (c), il s'agit d'un chemin arborescent où chaque branche passe par un élément de la liste dont l'ordinal est noté dans l'intervalle i..j dans le cas (b), ou dans la liste i_1, ... i_n dans le cas (c). Dans (d), le chemin est conditionné : on ne s'intéresse qu'aux éléments de la liste qui satisfont une certaine condition. Cette possibilité a été illustrée dans l'exemple 14. Les conditions ont la forme

$$\text{valeur} \left\{ =,\ \neq,\ \underline{\text{in}} \right\} \text{ CHEMIN-STRUCT(n, CONSTRUCTEUR(n)) } \underline{\text{of}} \text{ x}$$

où "valeur" est soit une unité texte (une chaîne de caractères), soit une expression regulière de chaines de caractères (cf. section 4.3.6), soit une variable définie sur le type chaîne de caractères. Le chemin qui suit à partir de n doit aboutir à une unité texte, ou à un ensemble d'unités texte (cf. section 4.4.2), et dans ce cas on utilise le mot-clé in.

Constructeur cas-où

Soit n un noeud de d construit avec C. Il peut donc être construit d'autant de manières différentes qu'il y a de valeurs de sélecteur. Soient $v_1,\ldots,v_k$ les valeurs du sélecteur s du cas-où. Nous noterons CONSTRUCTEUR(n suivant v_i) le constructeur qui construit n suivant la valeur v_i (remarque : CONSTRUCTEUR(n) = C). Par exemple, si m est un noeud contenu d'un article, alore CONSTRUCTEUR(m suivant résumé) = L. Les chemins dans la structure de d qui partent de n ont la forme :

CHEMIN-STRUCT(n,C) = CHEMIN-STRUCT(n,CONSTRUCTEUR(n suivant v_i))

Le chemin continue à partir de n en supposant que n a été construit

suivant une certaine valeur de sélecteur. Il est évident qu'une erreur se produira non pas à la compilation mais à l'exécution, si n a été construit suivant une valeur de sélecteur différente de celle donnée dans le chemin.

Finalement, si n est une unité U,

CHEMIN-STRUCT(n, CONSTRUCTEUR(n)) = CHEMIN-STRUCT(n, U) = U

c'est à dire, le chemin (ou la branche du chemin) passant par n s'arrête dès qu'on donne le nom d'une unité.

4.3.4.1.3. Manipulation des documents ou des sous-ensembles retrouvés

En général, on peut considérer que les fonctions qui retrouvent un sous-ensemble du document ont la forme d'un chemin arrivant à un noeud de la structure du document. En effet, du point de vue formelle, on peut considérer qu'une fonction dont le chemin est arborescent (arrivant donc à plusieurs noeuds) est une composition de plusieurs fonctions où chacune arrive à un seul noeud.

Soit f une fonction de la forme

$$\text{CHEMIN-DANS-STRUCTURE}(a_j)\ u_i.a_j$$

où le chemin arrive au noeud n. Alors, f est une fonction

$$f : C_i \longrightarrow \text{VS (SOUS-ARBRE(n))} = \text{Cod(f)}$$

Le résultat d'une telle fonction est en réalité un identificateur interne d'une partie d d'un document (d $\in$ VS (SOUS-ARBRE (n)). La partie d elle-même sera manipulée dans des environnements spécifiques. Les seuls opérateurs définis en LAMBDA pour les types document (ou pour des sous-arbres des types document) sont les opérateurs edit, mail et print qui font rentrer les documents dans des environnements où ils seront traités avec des objectifs spécifiques : edit appelle l'éditeur

interactif, mail envoie les documents par le courrier électronique et print envoie les documents au serveur d'impression. Dans une optique "types abstraits" on peut dire que ces opérateurs peuvent être vus comme des opérateurs associés au "constructeur abstrait" prédéfini Document. Le terme "constructeur abstrait" se rapproche du terme "type abstrait générique" [Ber 81] : chaque type construit par le constructeur hérite des opérations associées au constructeur. Il est clair que nous ne fournirons ni le mécanisme d'abstraction (les usagers ne peuvent pas définir un nouveau type abstrait avec des opérations associées) ni le mécanisme de généricité (il y a un nombre fixe de constructeurs l'usager ne peut pas en créer d'autres).

Dans le chapitre 5.1.2. nous présentons les transactions d'édition de documents, c'est à dire des transactions où l'opérateur edit s'applique au résultat d'une expression de sélection délivrant des (identificateurs des) documents ou des parties de documents. Nous n'avons pas abordé en détail les autres opérateurs mail et print. Cependant, il semblerait que les problèmes posés par l'implantation de transactions utilisant ces opérateurs ne sont pas très différents de ceux posés par l'implantation des transactions d'édition de documents (cf.section 8).

Comme nous le verrons dans le chapitre 4.5.1, on pourra utiliser les fonctions retrouvant un sous-ensemble d'un document dans la clause-where d'une condition LAMBDA. Le sous-ensemble retrouvé doit être une unité de nature texte. On pourra noter également dans la clause-where des fonctions retrouvant des ensembles d'unités texte d'un document (cf. section 4.2.2.2).

Dans l'expression de valeurs d'un énoncé LAMBDA on peut trouver aussi des fonctions qui comptent le nombre d'occurrences d'un noeud de la structure d'un document. Ces fonctions peuvent être considérées comme une sous-classe de la classe de fonctions qui retrouvent un sous-ensemble du document.

Exemple 16 Retrouver le nombre de sessions qu'il y a dans les proceedings du VLDB80

```
Select COUNT (session of c.texte-proc)
from CONGRES c where c.nom = 'VLDB' and c.année = 1980;
```

Ces fontions ont donc la forme générale

COUNT (CHEMIN-DANS-STRUCTURE(a_j) of $u_i.a_j$)

Elles sont utiles dans le cas où le noeud auquel on arrive dans le chemin est construit à l'aide du constructeur liste, et aussi pour les documents en cours de création, car le nombre d'occurrences d'un noeud qui n'est pas encore créé est 0. Dans la syntaxe du chemin, quand on passe par une liste, l'expression ELEMS, qui correspond aux ordinaux des éléments de la liste, peut être omise, étant donné qu'on ne s'intéresse pas ici aux éléments proprement dits mais au nombre d'éléments de la liste.

2.2.4.2 Fonctions portant sur les éléments associées "paramètre" et "fonction"

Les fonctions appartenant à cette classe permettent

(a) de retrouver la valeur d'un ou plusieurs paramètres d'un document
(b) d'évaluer une ou plusieurs fonctions d'un document.

Exemple 17 Retrouver les valeurs des paramètres "date-embauche" et "niveau" de tous les contrats.

```
Select parameter date-embauche, niveau of c.texte-contrat
from CONTRAT c ;
```

En général, si $p_1,\ldots,p_n$ sont des paramètres définis dans le type t_j de l'attribut a_j de la classe C_i, cette dernière étant désignée par la variable u_i, les fonctions qui retrouvent la valeur des paramètres ont la

forme parameter $p_1,\ldots,p_n$ of $u_i.a_j$

Bien entendu, on pourra utiliser ces fonctions dans la clause-where d'un énoncé de sélection ou de mise à jour de LAMBDA, par exemple pour retrouver des documents suivant la valeur de leurs paramètres (voir exemple 51, section 5.4.3.).

En ce qui concerne les évaluations d'une fonction f d'un document, si elle est déclarée comme ayant des paramètres formels, il faut l'évaluer avec des paramètres actuels. Ces paramètres actuels sont des constantes (ou des variables) du même type du paramètre formel.

Le cas où un paramètre formel d'une fonction est un paramètre du document peut être assez fréquent, surtout dans certaines applications comme la bureautique. Par exemple, les paramètres formels de la fonction "salaire-embauche" declarée dans le type Contrat-travail sont les paramètres de document "niveau" et "établissement". Pour ces cas là, le langage offre une commodité syntaxique illustrée dans l'exemple suivant.

Exemple 18 Evaluer la fonction "salaire-embauche" du contrat ayant le n° de référence 423

```
Select eval salaire-embauche(parameter niveau, établissement)
       of c.texte-contrat
from  CONTRAT c where c.no-ref = 423 ;
```

Ici, les valeurs actuelles des paramètres "niveau" et "établissement" sont obtenues en mettant parameter niveau, établissement dans l'argument de la fonction "salaire-embauche".

En général, une fonction LAMBDA évaluant les fonctions $g_1,\ldots,g_k$ d'un document $u_i.a_j$ a la forme

eval ..., g_i [(parameter $p_1,\ldots,p_m$] [$var_1,\ldots,var_n$])] ,... of $u_i.a_j$

où $p_1,\ldots,p_m$ sont des paramètres du document et en même temps des

paramètres formels de la fonction g_i, et les variables $var_1,\ldots,var_n$ sont des paramètres actuels de g_i.

2.2.5. Famille F_4 : Fonctions d'agrégation

Les fonctions d'agrégation s'appliquent aux ensembles que l'on peut construire en LAMBDA. A moins que la différence soit explicitée, le mot "ensemble" regroupera les vrais ensembles, où il n'y a pas de dupliqués, et les listes d'éléments, où il peut y en avoir.

Bien que l'expression d'ensembles soit définie dans la section suivante, nous présentons cette famille ici afin de garder groupée l'expession de toutes les familles de fonctions.

Les fonctions d'agrégation sont les suivantes : COUNT, qui compte les éléments d'un ensemble ; SUM et AVG qui produisent respectivement la somme et la moyenne d'un ensemble de valeurs numériques ; MAX et MIN produisent respectivement le maximum et le minimum d'un ensemble de valeurs numériques.

Exemple 19 Donner le maximum de l'ensemble des salaires de tous les employés

```
Select MAX{e.salaire}
from Employe e ;
```

Cet ensemble appartient à la famille S_1 d'ensembles "non-indexés" (cf. section 4.4.1). La forme générale des fonctions de cette famille est la suivante :

$$\text{fn-agr}: S \longrightarrow \text{Cod(fn-agr)}$$
$$s \longmapsto \text{fn-agr}(s)$$

où S est l'ensemble de tous les ensembles non constantes que nous pouvons construire en LAMBDA, c'est à dire, c'est l'union des familles S_1, S_2, S_3 et S_4. L'ensemble argument d'une fonction d'agrégation est noté explicitement dans l'expression de valeurs (les parenthèses habituelles

peuvent être omises). Le codomaine de la fonction d'agrégation Cod(fn-agr) est l'ensemble des entiers, sauf pour la fonction AVG, où il est l'ensemble de réels.

4.3.6. Famille F_5 : expressions constantes

La famille F_5 regroupe les expressions constantes. Une expression constante peut être aussi une variable temporaire (cf.section 5.1.1.). Dans l'interface programmable PASCAL - LAMBDA, une expression constante sera aussi une variable déclarée dans un programme PASCAL. Ces variables doivent être définies sur un type de base ou construites (sauf document). Les constantes admises sont:

(i) des constantes de base simples,

(ii) des constantes enregistrement, c'est à dire, des n-uplets de constantes chacune d'un type de base simple, ou des constantes tableau, où tous les éléments du tableau ont le même type de base simple.

(iii) null, la constante nulle. On peut comparer des attributs non clés ayant été déclarés dans le schéma avec l'option "nulls allowed" avec la constante null (cf. section 3.2.3.2). Les opérateurs de comparaison sont = , ≠. Le système manipulera en realité la valeur interne "par défaut" définie par l'usager dans le schéma.

(iv) des expressions regulières de chaines de caractères de la forme

$$['*']\ \text{chaine}_1\ ('*'\ \text{chaine}_2\ )^n\ ['*'],$$

où $n \geqslant 0$. Ces expressions sont utilisées pour effectuer des recherches "par le contenu" à l'intérieur d'une unité texte d'un document. Le caractère '*' est un "dont care": il correspond à n'importe quelle chaine de caractères (voir ex. 31, section 4.5).

Si f est une fonction de cette famille (donc, une constante), Cod(f) est défini comme l'ensemble de valeurs du type T de la constante.

4.4. EXPRESSION D'ENSEMBLES

Les ensembles que l'on peut former en LAMBDA se répartissent en quatre familles S_1 à S_4. Ils peuvent apparaître dans la clause-where dans un prédicat qui compare des ensembles (voir section 4.5) et comme arguments d'une fonction d'agrégation dans la clause-where ou dans l'expression de valeurs. Par ailleurs, toute expression de séléction est considérée aussi comme un ensemble; ceci a l'avantage de permettre de combiner des expressions de sélection avec des opérateurs ensemblistes.

Les trois premières familles regroupent des ensembles non constantes. Ils dépendent des variables de désignation notées dans les trajets de désignation de l'expression de sélection. Il n'y a donc pas d'ensembles ayant des trajets de désignation "internes".

Tout ensemble est une collection de valeurs du même type. Il ne contiendra pas de dupliqués (donc il sera un vrai ensemble) s'il est noté entre accolades "{...}", et il contiendra éventuellement des dupliqués s'il est noté entre parenthèses angulaires "[...]".

4.4.1. Famille S_1 : Famille d'ensembles non indexés.

Les ensembles de cette famille rassemblent des valeurs de variables qui sont notés dans les trajets de désignation, c'est à dire, le résultat de l'évaluation des fonctions des familles $F_1,\ldots,F_4$ appliqués à des faits désignés par des variables.

Exemple 20 Retrouver les noms de départements demandant des articles stockés par des fournisseurs qui habitent Paris

```
Select d.nom
From article-demandé a of DEPARTEMENT d, article stocké s of
     FOURNISSEUR f
where  a  in  {s where f.adresse.ville = 'Paris'} ;
```

L'ensemble noté dans la clause-where rassemble les faits s de ARTICLE

tels que (s,f) ∈ STOCKE et f.adresse.ville = 'Paris'. Remarque : une requête équivalente résulterait à changer les deux trajets par le trajet suivant: "fournisseur-stockant f of article-demandé a of DEPARTEMENT d" et la clause-where par f.adresse.ville = 'Paris'. Par contre, si on avait voulu retrouver les départements tels que les articles qu'ils demandent sont bleus ou sont stockés par des fournisseurs qui habitent Paris, il faut spécifier la requête comme précedamment, avec deux trajets. Autrement dit, les conditions imposées aux variables de par leur apparition dans un trajet sont toujours combinées avec "and".

Exemple 21 Retrouver la moyenne des salaires de tous les chefs de département

```
Select AVG [e.salaire]
from chef e of DEPARTEMENT d;
```

L'ensemble "[e.salaire]", qui n'est pas un vrai ensemble parce qu'il contient des dupliqués, est noté comme argument de la fonction AVG. C'est un ensemble de salaires d'employés qui sont des chefs de n'importe quel DEPARTEMENT d. La variable e a une portée interne à cet ensemble. Si on veut imposer une condition sur les chefs il faut alors la noter dans la clause-where interne à l'ensemble et non pas dans la clause-where de l'expression de sélection. Une même variable ne peut pas avoir deux portées différentes.

En LAMBDA, une entité E peut être considérée comme un ensemble. Pour ce faire, il est nécessaire d'inclure un trajet de la forme "E e" formé uniquement de sa racine E, et de noter alors {e} dans la clause-where (où dans l'expresion de valeurs comme argument d'une fonction d'agrégation). Cependant, ce procédé étant peu naturel, nous avons prévu d'abréger la notation de l'ensemble, en notant directement le nom de l'entité sans avoir besoin d'avoir un trajet supplémentaire.

Exemple 22 Retrouver les noms des employés n'ayant pas de bureau

```
Select e.nom from EMPLOYE e
where not (e in EMPLOYE-DE-BUREAU)
```

La forme générale des ensembles de cette famille est la suivante :

$$\{g_1(u),\ldots,g_r(u) \ [\underline{\text{where}}\ c(u)]\}$$

ou $$[g_1(u),\ldots,g_r(u) \ [\underline{\text{where}}\ c(u)]]$$

où les g_i $(i = 1,\ldots,r)$ sont des fonctions appartenant aux familles de fonctions non constantes. Nous dirons qu'elles forment l'expression de valeurs de l'ensemble. Les variables $u = (u_1,\ldots,u_k)$ doivent être notées dans les trajets de désignation, et de ce fait elles sont contraintes aux liaisons explicites avec d'autres variables; elles doivent satisfaire les conditions c si ces dernières existent. Un ensemble de cette famille est donc composé par des valeurs appartenant à l'ensemble

$$Cod(g_1) \ X \ \ldots \ X \ Cod(g_n) = VS \ (type(g_1)) \ X \ \ldots \ X \ VS \ (type(g_n))$$

Les variables notées dans l'expresion de valeurs d'un ensemble de cette famille ont une portée locale à la fonction d'agrégation.

4.4.2 Famille S_2 : Famille des ensembles indexés

Les exemples de cette famille se construisent avec des valeurs des variables u_i liées à la valeur d'une ou plusieurs variables d'indexation. Le concept d'ensemble indexé est une extension du concept de correspondance dans le modèle ER présenté dans [SNF 79]. Ce dernier concept est le suivant : Soit A, une association définie sur les entités $E_1,\ldots,E_n$. Une correspondance entre les entités "indexantes" $E_1,\ldots,E_{n-1}$ et l'entité indexée E_n, notée $(E_1,\ldots,E_{n-1})\{E_n\}$, associe chaque n-uplet $(e_1,\ldots,e_{n-1})$ pour lequel il existe un $e_n \in E_n$ tel que $(e_1,\ldots,e_n) \in A$ avec l'ensemble $\{e \mid e \in E_n \text{ et } (e_1,\ldots,e_{n-1}, e) \in A\}$. Ce dernier ensemble est appelé "l'ensemble indexé" et le n-uplet $(e_1,\ldots,E_{n-1})$ est appelé le "composant indexant".

En LAMBDA, on construit des ensembles indexés à l'aide de variables d'indexation qui sont notées dans les trajets de désignation. Les extensions au concept de correspondance sont les suivantes :

(a) On peut construire des ensembles indexés sur des associations du schéma où certaines entités indexantes ne sont pas prises en compte. Ceci est utile dans des associations n-aires où n⩾3.

(b) Les entités indexantes et l'entité indexée ne doivent pas obligatoirement participer à une association dans le schéma : on peut construire des ensembles indexés sur une association dérivée [Kent 78] définie ici par une trajet de désignation.

(c) On peut écrire des ensembles indexés en utilisant des correspondances basées non seulement sur des associations mais aussi sur des agrégations, sur la structure hiérarchique des documents, et sur les attributs définis sur un type tableau.

4.4.2.1 - Ensembles indexés sur des associations

Exemple 23 Retrouver les articles fournis par tous les fournisseurs (sans tenir compte du département).

```
Select a
from fournisseur f of ARTICLE a
where {f} by a = FOURNISSEUR;
```

Les ensembles indexés sont des sous-ensembles de FOURNISSEUR, construits sur l'association FOURNITURE. Ici, la variable d'indexation est a : il n'y a qu'une seule entité indexante, l'entité ARTICLE (l'entité DEPARTEMENT n'étant pas prise en compte). Cette requête montre la possibilité de comparer des ensembles (cf. section 4.5.3.).

On aurait pu dans cette requête écrire le trajet suivant à la place "article-fourni a of FOURNISSEUR f", le résultat aurait été strictement équivalent. Cependant, il est conseillé, pour des raisons linguistiques évidentes, d'écrire des trajets où la variable d'indexation apparaisse avant (dans le sens du trajet) les variables indexées. Ceci n'est pas toujours possible en LAMBDA, par exemple, quand il y a plusieures variables d'indexation, comme il est montré dans l'exemple suivant, et dans l'exemple 26.

Exemple 24 Retrouver les articles fournis à un département donné par tous les fournisseurs.

```
Select a
from (fournisseur f, dept-fourni d) of ARTICLE a
where {f} by a, d = FOURNISSEUR ;
```

Ici, toutes les entités sont prises en compte dans la correspondance sur FOURNITURE.

Exemple 25 Pour chaque département, donner le nom et le nombre de fournisseurs (différents) qui stockent des articles demandés par le département.

```
Select d.nom, COUNT {f} by d
from fournisseur-stockant f of article demandé of DEPARTEMENT d;
```

Ici, le trajet établit une association dérivée entre les fournisseurs, les articles et les départements. Cette dérivation est la composition de deux associations, DEMANDE et STOCKE, qui utilise l'entité ARTICLE comme "pont" entre les deux. l'entité indexante est DEPARTEMENT, l'entité indexée est FOURNISSEUR, l'entité ARTICLE n'est pas prise en compte. Chaque ensemble indexé est passé comme paramètre de la fonction d'agrégation COUNT. La variable f a une portée locale à l'ensemble (et donc, à la fonction d'agrégation). la variable a est considérée comme ayant aussi une portée locale à l'ensemble car elle intervient dans la composition de l'association dérivée. Si on avait voulu retrouver pour chaque département le nombre de fournisseurs qui stockent des articles rouges demandés par le département, il aurait fallu introduire une variable a désignant les articles demandés dans le trajet et le prédicat a.couleur = 'rouge' dans la clause-where de l'ensemble indexé.

Cependant, si on avait voulu retrouver le nom des départements qui démandent des articles rouges ainsi que le nombre de fournisseurs qui stockent des articles (pas uniquement rouges) demandés par le département, il aurait fallu introduire une autre variable a' liée au département d et écrire la condition a'.couleur = 'rouge'.

Exemple 26 Retrouver, pour chaque département et chaque fournisseur, le nombre d'articles demandés par le département qui sont stockés par le fournisseur.

```
Select d.nom, f.nom, COUNT{a} by f, d
from fournisseur-stockant f of article-demande a of DEPARTEMENT d;
```

Dans cet exemple, la même association dérivée entre les fournisseurs, les articles et les départements est utilisée. L'entité indexée est ARTICLE et les entités indexantes sont FOURNISSEUR et DEPARTEMENT. LAMBDA offre aussi la possibilité de décomposer la requête en la formulant en des termes plus "relationnels" :

```
DEPTARTFOUR(D,A,F) : = Select d.nom, a.nom, f.nom
                       from fournisseur-stockant f of article-demande a
                            of DEPARTEMENT d;

Select unique x.C, x.D, COUNT{y where y.C = x.C and  y.D = x.D}
from DEPTARTFOUR x, DEPTARTFOUR y;
```

Cet exemple montre deux choses :

- d'abord la possibilité d'utiliser des variables temporaires en LAMBDA, ce qui sera discuté plus longuement dans la section 5.1.1.

- Ensuite la possibilité de faire l'équivalent d'une indexation par valeur d'un attribut, comme ceci est fait dans des langages de requêtes relationnels comme SQL et QUEL. Cette indexation par valeur passe par l'introduction d'un trajet supplémentaire.

4.2.2.2 - Ensembles indexés via des agrégations, des tableaux et des documents

L'écriture des ensembles indexés via des agrégations reste la même que celle que nous avons vu jusqu'ici (mot-clé by). Les ensembles indexés via des tableaux ou via la structure d'un document s'écrivent respectivement comme une fonction d'accès au tableau (cf. section 4.3.2) ou

comme un chemin dans la structure d'un document (cf. section 4.3.4).

Exemple 27 Retrouver les dossiers tels que tous les expéditeurs des letttres contenues dans le dossier soient inscrits dans la liste des destinataires de la lettre ayant le n° de référence 512.

Select d
from lettre l_1 part-of DOSSIER d, LETTRE l_2
where $\{l_1.$ expéditeur$\}$ by d $\subseteq l_2.$ liste-destinataires
and $l_2.$ n°ref = 512 ;

Dans cet exemple, on compare deux ensembles indexés (cf. section 4.5.3), l'un étant indexé via une agrégation (agrégation de lettres dans un dossier) et l'autre via un tableau (attribut liste-destinataires d'une lettre).

Dans la structure du type document "doc-lettre" (qui est le type de l'attribut "texte-lettre" de l'entité LETTRE), nous avons défini un noeud appelé "destinataires", défini sur l'unité texte. Il est impossible de comparer l'ensemble indexé "l_2 . liste-destinataires" avec "destinataires of l_2.texte lettre". En effet, ce dernier est une unité "texte" et, bien qu'un texte soit compatible avec une chaîne de caractères, on ne peut pas le comparer avec un ensemble de chaînes de caractères. Si le noeud Destinataires avait été défini comme une liste de textes, la comparaison serait possible. Dans ce cas, on aurait pu écrire une requête où l'on compare les expéditeurs des lettres de chaque dossier avec l'ensemble (indexé) "destinataires of l_2. texte lettre". Cette comparaison peut ne pas donner le même résultat que celle de la requête précédente (avec l_2.liste destinataires") à cause de l'incohérence potentielle entre les valeurs dans le tableau et les valeurs dans le texte de la lettre elle-même.

Les ensembles indexés dans un document que l'on manipulera dans la clause-where d'une requête LAMBDA seront uniquement des ensembles d'unités texte. Un ensemble d'unités textes est un chemin dans la

structure du document où tous les noeuds où le chemin arrive sont du type "texte" ou "liste de texte". S'il n'y a que des noeuds du type "texte", le chemin doit être arborescent, c'est à dire qu'il doit arriver à plusieurs noeuds. Les ensembles d'unités textes peuvent être indexés :

(i) soit par la racine du document, auquel cas il a la forme (cf. section 4.3.4.1.2)

CHEMIN-DANS-STRUCTURE (a_j) $u_i.a_j$

où u_i est une variable de désignation sur la classe c_i et a_j est un attribut de c_i défini sur un type document.

(ii) soit par une variable x qui désigne un noeud n du document, auquel cas il a la forme

CHEMIN-STRUC (n, CONSTRUCTEUR (n)) of x

la variable x conditionne un chemin dans la structure d'un document $u_i.a_j$; ce chemin doit être noté dans l'expression de valeurs de la requête (cf. section 4.3.4.1).

Exemple 28 Retrouver les articles écrits ou co-écrits par 'R. Martin' dans les proceedings des congrès ayant eu lieu pendant l'année 1981

```
Select article a of contenu of session 1,* of c.texte-proc
from CONGRES c
where c.année = 1981 and 'R.Martin' in auteurs of a ;
```

Ici, chaque ensemble "auteurs of a" est indexé par un noeud de type 'article'. A leur tour, ces noeuds sont indexés par la racine du document, c.text-proc.

4.4.3. Famille S_3 : Ensembles construits par des opérateurs ensemblistes

Cette famille regroupe les ensembles construits par union (+), intersection (*) ou différence (-) à partir d'autres ensembles. Il faut que le type de valeurs des ensembles soient compatibles. Pour une discussion sur la compatibilité de types en LAMBDA, voir l'annexe 3.

Exemple 29 Pour chaque article, retouver son nom ainsi que les noms des départements qui ne les demandent pas

```
Select  a.nom d2.nom
from  DEPARTEMENT d2, dept-demandant d1 of ARTICLE a
where d2 in  DEPARTEMENT - {d1} by a  ;
```

Le type des valeurs des deux ensembles est compatible, puisque c'est le même. La différence ensembliste sert donc à trover le "complement" d'une association. Dans ce cas, l'ensemble différence est aussi indéxé par a, et de ce fait, il appartiendrait à S_2. Cependant, nous avons différencié les familles S_2 et S_3 de façon purement syntaxique (selon qu'il y a des opérateurs ensemblistes ou non), et donc, pour les ensembles de S_3 nous dirons s'ils sont indexés ou non.

Exemple 30 Retrouver les articles présentés aux congrès ayant eu lieu la même année que le premier séminaire ADI de Base de données et ayant été écrits par au moins un auteur ayant présenté une communication à ce séminaire.

```
Select article a of contenu of session 1,* of  c2.texte-proc, c2.nom
 from CONGRES c1, CONGRES c2
 where c1.nom = 'premier séminaire ADI de bases de données'
     and c1.année = c2.année
     and (auteurs of a) * (auteurs of articles 1,* of contenu
         of session 1,* of c1.texte-proc) ≠ { }
```

Cet exemple montre la possibilité d'effectuer l'intersection de deux ensembles d'unités textuelles. L'ensemble { } dénote l'ensemble vide. Le

deuxième ensemble contient les auteurs des articles présentés au séminaire ADI ; le premier est indexé par un article d'un autre congrès ayant eu lieu la même année que le séminaire ADI.

En général, soit $S = S_1 \cup S_2 \cup S_3$, soient s et t deux ensembles de S tels que le type de leurs valeurs est compatible. Supposons que s est indexé par $x_1,\ldots,x_n$ $(n \geqslant 0)$ et t par $y_1,\ldots,y_m$ $(m \geqslant 0)$; alors l'ensemble

s op-ensembliste t

appartient à S_3 et il est indexé par $x_1,\ldots,x_n$, $y_1,\ldots,y_m$. Les opérateurs possibles sont l'union (+), l'intersection (*) et la différence (-).

4.4.4. Famille S_4 : Ensembles constantes

Cette famille regroupe des ensembles de la forme

$$\{c_1,\ldots,c_n\}$$

où $c_1,\ldots,c_n$ sont des constantes (ou des variables temporaires) du même type simple de base, ou des constantes enregistrement, ou encore des constantes tableau. Par exemple, 'Dupont', 'Martin' est un ensemble constante dont les éléments ont le type simple chaîne de caractères.

Le cas n = 0 (l'ensemble vide) est noté $\{\}$.

4.5 EXPRESSION DE CONDITIONS

Avec cette section nous complétons la présentation des expressions de sélection de LAMBDA. La condition c de l'expression de sélection peut appartenir aux familles suivantes :

4.5.1. Famille C_1 : Conditions de comparaisons de valeurs

Nous entendons par "valeur" l'application d'une fonction quelconque g

appartenant à une des familles $F_1,\ldots,F_5$ à un fait u d'une certaine classe désignée dans le trajet de désignation. Une valeur a donc la forme g(u). Elle appartient à l'ensemble Cod(g), qui est un ensemble de valeurs d'un certain type (cf. § 4.3.) noté type(g).

Une condition de comparaison de valeurs a la forme

$$g_1(u)\ \Theta\ g_2(v)$$

où Θ est un opérateur scalaire parmi $=,\neq,>,\not>,<,\not<$. Les comparaisons de valeurs doivent remplir les condition suivantes :

(i)Θ est un opérateur scalaire défini sur type(g_1) et type (g_2).

(ii) les types type(g_1) et type(g_2) sont compatibles.

Dans l'annexe 3 nous présentons des règles de compatibilité de types ainsi que la table des opérateurs scalaires qui s'appliquent à chaque type.

Exemple 31 Retrouver les articles ayant été présentés dans des sessions qui touchent à la modélisation de données dans le cadre des conférences VLBD.

```
Select contenu of session s of c.texte-proc.
from CONGRES c
where c.nom = VLBD
   and * model * = titre of s ;
```

Cette requête montre un exemple d'utilisation d'expression régulières de chaînes de caractères : on retrouve des articles d'une session suivant si le titre de la session contient la chaîne 'model'. Elle peut se trouver n'importe où dans le texte du titre, étant donné que le caractère '*' peut correspondre à n'importe quelle chaîne de caractères. les sessions ayant un titre comportant des mots du style "modelling", "models", "modélisation", etc, vérifieront la condition.

4.5.2. Famille C_2: Conditions d'appartenance d'une valeur à un ensemble

L'opérateur d'appartenance d'une valeur à un ensemble est l'opérateur in (cf. exemples 28, 29). Il retourne la valeur vrai si la valeur est dans l'ensemble et faux sinon. La forme générale de ces conditions est la suivante:

$$\left\{ \begin{array}{c} g_1(u) \\ \\ (g_1(u), \ \ldots \ , g_n(u)) \end{array} \right\} \quad \underline{\text{in}} \ \ s$$

où s est un ensemble quelconque et $g_1,\ldots,g_n$ des fonctions quelconques. Le type de valeurs de s doit être compatible avec le type

$$\text{type}(g_1) \ X \ \ldots \ X \ \text{type}(g_n).$$

4.5.3. Famille C_3 : Conditions de comparaisons ensemblistes

Les comparaisons ensemblistes supportées par LAMBDA sont:

= égalité (ensembliste)
= inégalité (ensembliste)
$\supset\!\!=$ contient ou égal
$\supset$ contient strictement

Ces comparaisons ensemblistes permettent de repondre à des requêtes où l'on utiliserait les quantificateurs existentielles. La forme générale de conditions de comparaisons d'ensembles est

$$s \ \Psi \ t$$

où s et t sont deux ensembles quelconques (tous les deux n'étant pas constantes eu même temps), et Ψ est un opérateur de comparaison d'ensembles appartenant à la liste notée plus haut.

4.5.4. Famille C_4 : Combinaisons booléennes

Si c_1 et c_2 sont deux condition quelconques appartenant aux familles $C_1,\ldots,C_4$, alors

(c_1)

c_1 and c_2

c_1 or c_2

not c_1

sont des conditions qui appartiennent à la famille C_4 de combinaisons booléennes. Leur sémantique est évidente.

Chapitre 5

LAMBDA : ENONCES DE MISE A JOUR

SOMMAIRE

5. LAMBDA : ENONCES DE MISE A JOUR

Les énoncés de mise à jour de LAMBDA peuvent être de trois types :

- l'insertion de faits dans les classes,
- la suppression de faits d'une classe,
- la modification de valeurs d'attributs d'une classe.

L'enchaînement des énoncés de mise à jour, logiquement reliés entre eux, doit faire passer la base d'un état cohérent à un autre, à travers des états intermédiaires qui peuvent être incohérents. De ce fait, pour garantir la cohérence de la base il faut que, soit tous les énoncés soient exécutés entièrement, soit qu'aucun énoncé ne soit exécuté ; cette propriété est appelée atomicité. Mais il faut aussi que les effets sur la base soient durables, c'est à dire qu'ils puissent survivre à une panne matérielle ou logicielle et que la transformation sur la base soit correcte, dans le sens que les contraintes d'intégrité du schéma conceptuel (implicites ou explicites) soient vérifiées.

La notion de transaction [EGLT 76], [Gra 81] est un regroupement d'énoncés de mise à jour (et d'interrogation) qui a les trois propriétés désirables : atomicité, durabilité, intégrité. Elle fait donc passer la base d'un état cohérent à un autre état cohérent. Cette notion de transaction sera offerte dans LAMBDA.

Dans cette section nous présentons d'abord les transactions en LAMBDA: l'utilisation des variables temporaires et la notion de transaction d'édition de documents. Ensuite, nous présentons les énoncés d'insertion de faits, puis les suppressions et les modifications.

5.1. TRANSACTIONS

Une transaction LAMBDA est une suite d'énoncés de sélection et de mise à jour comprise entre les instructions BEGIN-TRANSACTION et END-TRANSACTION. L'instruction RESTORE-TRANSACTION est aussi disponible, comme dans SQL, pour les transactions qui, à un moment donné, décident de faire marche arrière et d'annuler tout ce qu'elles venaient de faire. Tout énoncé de mise à jour est inclus dans une transaction LAMBDA qui ne sera pas validée tant que les contraintes d'intégrité du schéma ne seront pas vérifiées.

L'approche pris dans TIGRE pour préserver les contraintes d'intégrité est de permettre des mises à jour dans une classe et d'essayer de les propager aux classes adjacentes. Les propagations seront automatiques dans la mesure du possible; Autrement, elles devront être accomplies par l'usager avant la fin de la transaction. Les règles de propagation sont une extension de celles presentées dans [SSW 79], adaptées au modèle TIGRE. Ces règles seront présentées dans les sections concernant les insertions et les suppressions.

Le résultat des expressions de sélection peut être affecté à des variables temporaires. Ces variables peuvent être utilisées dans d'autres énoncés de sélection ou de mise à jour. Elles peuvent être utilisées aussi comme arguments des opérateurs de manipulation de documents edit, print et mail.

5.1.1 Variables temporaires

Nous avons vu au chapitre 4 qu'une expression de selection S définie par

```
Select  f1(x), ... , fn(x)
from    Trajets x
where   c(x) ;
```

produit un ensemble de faits où chacun appartient au type t défini par

$$t : type(f_1) \times \ldots \times type(f_n)$$

Dans une affectation d'une expression de sélection S à une variable V, notée V := S, V aura le type T où ce dernier est défini par

```
T = entity
    |
    | f1 : type(f1);
    |    .
    |    .
    |    .
    | fn : type(fn);
    end;
```

Autrement dit, T est le type "ensemble de faits du type t". Les fonctions $f_1,\ldots,f_n$ constituant l'expression de valeurs de S sont considérés comme les noms des attributs de V. Ils peuvent être renommés en notant

$V(a_1, \ldots, a_n)$:= Select $f_1(x) \ldots f_n(x)$ from ... ;

comme dans l'exemple 26.

Dans l'interface procédurale PASCAL- LAMBDA les variables temporaires devront être déclarées explicitement sur le type T précédent. Les noms des attributs sont données directement dans la déclaration de T.

Il est nécessaire parfois de spécifier que le résultat d'une expression de sélection n'est pas un ensemble de faits mais qu'il est un seul fait. En effet, du point de vue de LAMBDA, qui est un langage typé, les types T et t sont considérés comme des types différents.

Pour ce faire, on dispose en LAMBDA d'un opérateur de conversion de types. Cet opérateur, appelé "First-fact", s'applique à une expression de sélection S et retourne un seul fait : le premier délivré par S. Dans une affectation

```
V := First-fact(S)
```

le type de v est implicitement t au lieu de T. La variable v se comporte donc comme un enregistrement. On pourra se référer au constituant f_i de v en notant $v.f_i$ comme traditionnellement.

Exemple 32 : insérer un nouvel employé avec un salaire égal à la moyenne des salaires des programmeurs

```
begin-transaction ;

S := First-Fact (select AVG [p.salaire]
                 from PROGRAMMEUR p ;)

Insert (... salaire = s ...) to EMPLOYE ;

end transaction ;
```

Dans cet exemple, même si l'on sait que le résultat de cette requête se compose d'un fait, si on n'utilise pas l'opérateur First-Fact la variable s sera considérée comme ayant le type

```
Entity
|
|  AVG : integer ;
|
end;
```

c'est à dire comme étant un ensemble d'entiers. Pour qu'elle ait le type entier il faut utiliser l'opérateur first-fact.

5.1.2 Transaction d'édition de documents

L'édition des documents consiste à réaliser les fonctions suivantes :

- créer des documents (choisir un type document, identifier un nouveau document , le ranger dans la base de données)

- accéder à un ou plusieurs documents (les désigner via une requête, les extraire en totalité ou en partie de la base de données)

- modifier et parcourir des documents.

Pendant le création ou la modification interactive d'un document dans l'éditeur, on pourra utiliser des parties d'autres documents présents dans l'éditeur. La création ou la mise à jour d'un document, appelé objet, à partir d'autres, appelés documents sources, peut être accomplie de deux façons différentes :

- par copie, auqel cas la partie du document source sera dupliquée et incorporée au document objet,

- par partage, auquel cas dans le document objet, il y aura un pointeur vers la partie du document source à partager. Les modifications sur un document partage seront reflètées dans tous les documents qui y font réference.

Remarque : Les documents pourront aussi être mis à jour par programme, en utilisant la commande replace de LAMBDA (cf. section 5.4.3).

Le scénario le plus courant de création ou de modification d'un ou plusieurs document est alors : l'usager lance une session d'édition. Il fait accès à la base pour désigner des documents (ou demander la liste de types-document disponibles) en utilisant une requête LAMBDA et il les extrait de la base. Après modification ou création, il les range dans la base en utilisant les commandes insert ou replace de LAMBDA, toujours depuis l'éditeur. Après des éventuelles repétitions de ce cycle, l'usager quitte l'éditeur.

Il existe cependant des applications où l'on a besoin de programmer des appels à l'éditeur. La commande edit est sensée être utile dans ce genre d'applications. Une transaction d'édition de documents est une transaction LAMBDA où l'on trouve cette commande. Elle a comme paramètre la valeur d'une variable temporaire auquel on a affecté le résultat d'une expression de sélection LAMBDA qui désigne les documents que

l'on veut manipuler. Elle effectue l'appel à l'éditeur et l'extraction des documents de la base. L'appel est un appel "bloquant", c'est à dire que tant que l'on n'a pas quitté l'éditeur, la transaction reste bloquée. Ceci pose des problèmes non négligeables concernant la mise en oeuvre, plus particulièrement en ce qui touche aux accès concurrents et la reprise après pannes. Ces problèmes seront détaillés dans la section 8.

Trois types d'édition sont possibles : édition de lecture d'un (de) document (s) existant (s), édition de modification d'un (de) document (s) et création d'un document non existant. Dans ce dernier type d'édition, un nom T de type document est passé en paramètre de la commande pour créer un document du type T (cf. section 5.2.1.2., exemple 38).

Les éditions de lecture ne donnent pas de résultat, les éditions de modification délivrent un ou plusieurs documents du même type que ceux passés en paramètre. S'il n'y a qu'un seul document à éditer, un seul document sera délivré ; s'il y en a plusieurs, plusieurs seront délivrés. Autrement dit, le type de la variable d'entrée est égal au type de la variable de sortie de la commande edit.

Les documents affectés à la variable temporaire de sortie pourront être introduits par la suite dans la base via les commandes insert ou replace (cf. sections 5.2. et 5.4.).

Exemple 33 : modifier la clause 4 du contrat de l'employé de numéro 1500 avec l'éditeur

```
begin-transaction ;

 le-contrat(clause,nref) :=

   First-Fact (select clause 4 of corps of c.texte-contrat,  c.n°-ref
               from contrat c part-of le-dossier of EMPLOYE e
               where e.n°-id = 1500 ; )
```

```
nouv-clause := edit (le-contrat.clause) for update ;

replace clause 4 of corps of c.texte-contrat := nouv-clause
        from CONTRAT c where c.n°-ref = le-contrat.nref;

end-transaction ;
```

Dans cet exemple, on affecte la variable "le-contrat" avec le premier (et seul) fait retrouvé dans l'expression de sélection, tout en renommant ses constituants. "le-contrat.nref" aura le type entier, et "le-contrat.clause" aura le type clause (c'est à dire un agrégat d'un titre et d'un paragraphe). Ce dernier sera passé comme paramètre de l'opérateur edit qui retournera une clause modifiée. Cette clause sera désignée par la variable "nouv-clause" qui a le même type que "le-contrat.clause". Puis, on introduit la clause modifiée dans la base à l'aide de la commande replace: à la place de l'ancienne clause on copie la nouvelle.

Exemple 34 : Modifier le contrat de Dupont à partir des contrats du dossier du classeur "nouveaux"

```
begin-transaction ;

CONTRATS : = select c.texte-contrat, c.n°-ref
             from   contrat c part-of le-dossier d of EMPLOYE e
             where d.classeur = "nouveaux"
                   or e.nom = "dupont" ;

NOUV-CONTRATS (doc, id) : = edit (CONTRATS) for update ;

replace c.texte-contrat := nc.doc
        from CONTRAT c , NOUV-CONTRATS nc
        where c.n°-ref = nc.id;

end-transaction;
```

Dans cet exemple, plusieurs documents sont envoyés à l'éditeur pour être modifiés. La variable temporaire CONTRATS est une "entité temporaire" contenant les contrats du classeur "nouveaux" ainsi que le contrat de Dupont. Chaque contrat est accompagné de son numéro de référence de façon à pouvoir identifier ces documents au moment de la re-insertion dans la base. Le résultat est aussi un ensemble de contrats (qui peut ne contenir qu'un seul contrat, comme dans ce cas) que l'on stocke dans la variable NOUV-CONTRATS. On a renommé les attributs de cette variable: "doc" a le même type que "texte-contrat" et "id" a le même type que "n°-ref".

Dans une édition de modification, l'usager peut créer un nouveau document à partie d'autres. Dans ce cas, c'est sa responsabilité de donner la valeur du n° de référence (ainsi que la valeur des autres attributs) pour ensuite l'insérer dans la base via la commande insert.

Nous avons considéré la possibilité de programmer des appels à l'éditeur avec des documents de type différent. La solution qui nous semble la meilleure est de considérer edit comme une commande avec un nombre indéfini de paramètres, un pour chaque type de document. Une commande d'édition de modification à n paramètres donnerait comme résultat n variables, la i-ème variable de sortie ayant le même type que la i-éme variable d'entrée. Dans l'état actuel de nos recherches, on peur considérer qu'il s'agit d'une voie à explorer.

5.2. INSERTIONS

En LAMBDA, on peut insérer des faits dans des entités ou des associations via la commande INSERT. L'insertion de faits est une opération ensembliste. Cela signifie que l'on peut insérer dans une classe des faits provenant d'autres classes de la base. Bien entendu, les commandes s'avèrent inadéquates lorsqu'il s'agit de charger la base avec une quantité importante de faits provenant de l'extérieur de la base. Il faut disposer dans le système d'un utilitaire de chargement.

5.2.1. Insertion dans les entités

Pour insérer un fait dans une entité, on spécifie le nom de l'entité et les valeurs pour chaque attribut de l'entité.

Exemple 35 : Insertion d'un nouvel employé

```
Insert (no-id = 4334, nom = 'Durand', adresse = (27, Bd Lyautey,
        Nice), salaire = 9800, catégorie = 'ing') to EMPLOYE;
```

Le système vérifie qu'il n'y a pas d'autre employé ayant la même valeur de clé. L'insertion de ce fait dans EMPLOYE amène à insérer d'autres faits dans d'autres classes du schéma conceptuel. En effet, on peut exprimer de nombreuses contraintes implicitement dans un schéma conceptuel TIGRE. Par exemple, du fait que la participation minimale de EMPLOYE dans l'association EMP-DOSS est 1, tout employé doit être propriétaire d'un dossier. Ceci implique qu'il faut propager l'insertion à travers l'association EMP-DOSS dans la même transaction. On peut être amenés éventuellement à insérer un fait dans DOSSIER, si le dossier correspondant au nouvel employé Durand ne s'y trouve pas, ce qui à son tour fait propager l'insertion à travers l'association GESTION.

La propagation d'insertions peut avoir lieu, non seulement à travers des associations, mais aussi au travers de la hiérarchie de généralisation. Dans cet exemple, le fait inséré correspond à celui d'un ingénieur. Il satisfait donc le prédicat associé à la spécialisation de l'entité INGENIEUR à partir de EMPLOYE, et du fait que l'option "manuelle" n'a pas été prise, il doit être inséré dans INGENIEUR avant la fin de la transaction. Cette insertion doit être explicite de la part de l'usager, parce que l'entité INGENIEUR a des attributs propres. Autrement, elle serait accomplie automatiquement par le système. Cette propagation s'arrête là étant donné que l'option "manuelle" a été prise pour l'insertion de faits dans l'entité EMPLOYE-DE-BUREAU à partir de INGENIEUR. Autrement dit, il est déclaré dans le schéma qu'il n'est pas obligatoire que tous les INGENIEURS soient des EMPLOYES-DE-BUREAU.

Nous introduisons maintenant, toujours avec des exemples, les outils offerts par LAMBDA pour l'insertion des faits dans des entités dérivées par. généralisation et par agrégation. A la fin de cette section, nous ferons un résumé de tous les cas de propagation d'insertions.

5.2.1.1. Insertions dans les entités dérivées par généralisation

On peut insérer des faits dans une entité dérivée par généralisation soit directement, soit à partir d'une entité père, dans la hiérarchie de généralisation. L'exemple suivant illustre ce dernier cas. Il s'agit de l'insertion du nouvel employé dans INGENIEUR, qui, comme nous l'avons déjà mentionné, doit être impérativement effectuée avant la fin de la transaction.

Exemple 36 : insertion du nouvel employé dans INGENIEUR

```
Insert e with (spécialité : 'électronique', école : 'ENSIEG')
to INGENIEUR
from EMPLOYE e where e.no-id = 4334 ;
```

Les ingénieurs héritent des attributs des employés et ils ont deux attributs propres : la spécialité et l'école. Le mot-clé "with" ajoute des valeurs aux attributs propres à INGENIEUR à un fait de EMPLOYE en le rendant un fait de INGENIEUR. Ici la clause where qualifie un seul fait de EMPLOYE. Dans le cas général, plusieurs faits peuvent vérifier la clause-where. Par exemple s'il y avait une entité EMPLOYES-PROMETTEURS spécialisée de EMPLOYE avec l'option "manuelle" ayant l'attribut propre "mois-de-promotion" on pourrait insérer plusieurs faits à la fois, comme le montre l'énoncé suivant :

Exemple 37 : insertion dans EMPLOYES-PROMETTEURS pendant le mois de Mars aux ingénieurs qui viennent de l'ENSIMAG et qui sont affectés au département d'informatique.

```
Insert i with (mois-promotion = 'Mars') to EMPLOYES-PROMETTEURS
from dept-affecté d of INGENIEUR i
where d.nom = 'informatique' and i.école = 'Ensimag';
```

Ici tous les faits insérés auront 'Mars' comme valeur de l'attribut "mois-promotion".

Pour illustrer l'insertion directe de faits dans une entité dérivée par généralisation, nous reprenons l'exemple 36.

Une telle insertion a la même forme que celle de l'exemple 35, avec les valeurs des attributs "spécialité" et "école". Elle sera propagée automatiquement par le système aux ancêtres, c'est à dire à l'entité EMPLOYE. Toutefois, il faut que le fait inséré dans l'entité dérivée par généralisation (et ceci s'applique aussi aux associations spécialisées) vérifie les prédicats de restriction associés à la définition de l'entité, et ceci pour toutes les entités de la chaîne des ancêtres pour arriver au sommet (l'entité non-dérivée) de la hiérarchie. Si un seul prédicat n'est pas vérifié, le fait ne sera pas inséré dans aucune des entités de la hiérarchie où il devrait l'être, la transaction sera abandonnée, et un message approprié sera donné à l'usager.

La propagation des insertions aux ancêtres ne peut pas être automatique dans le cas des entités dérivées par union. En effet, il faudrait spécifier quelle est l'entité père ainsi que fournir les valeurs propres des attributs du père pour pouvoir l'insérer dans ce dernier. Par exemple, si on veut insérer directement un fait dans EMPLOYE-DE-BUREAU, il faudrait savoir s'il s'agit d'un INGENIEUR, d'une SECRETAIRE ou d'un PROGRAMMEUR. Ce procédé étant peu naturel et pas clairement exprimable dans le cadre d'un langage comme LAMBDA, nous avons préféré ne pas propager l'insertion des faits aux ancêtres dans le cas des unions. Pour insérer un fait dans une union, il faut le faire à partir d'une entité père (dans l'esprit de l'insertion de l'exemple 37).

5.2.1.2. Insertions dans les entités dérivées par agrégation

Considérons d'abord le cas des entités définies par agrégation d'entités. Prenons l'exemple des dossiers. Quand on désire insérer un dossier dans une transaction, il faut qu'à la fin de la transaction celui-ci contienne un CONTRAT et zéro ou plusieurs LETTRES. On doit

pouvoir insérer le contrat et les lettres dans leurs entités avant, pendant ou après l'insertion du dossier dans la transaction. On peut les lier au dossier pendant ou après l'insertion de ce dernier. En plus, on doit pouvoir continuer à agréger des lettres dans un dossier pendant la vie de la base, c'est à dire dans d'autres transactions.

Exemple 38 : Insertion d'un dossier

```
t := edit (contrat-travail) for création ;

(i)  Insert (n°-dossier = 542, classeur = 'nouveaux') to DOSSIER ;
       and (n°-ref = 728, date = (3,11,1982), texte-contrat = t)
        to contrat part-of DOSSIER ;
```

Ici on insère un fait dans CONTRAT tout en l'agrégeant avec un dossier dans le même énoncé d'insertion du dossier. Autrement dit, le contrat est inséré et lié avec le dossier pendant l'insertion de ce dernier. Le fait inséré dans CONTRAT contient un document identifié par la variable t. Cette variable est celle de sortie d'une édition de création de documents (cf. section 5.1.2.). Comme dans les énoncés d'interrogation, on utilise le mot-clé "part-of" pour désigner des faits d'une entité agrégée. Dans le cadre des insertions, ce mot-clé indique que les faits de l'entité "agrégée" doivent faire partie d'un fait de l'entité "agrégat". Dans cet exemple, les faits de l'entité "agrégée" (le contrat) n'existaient pas dans la base.

Dans le cas où le contrat que l'on veut agréger dans le dossier existe déjà dans la base, on utilise une clause-where pour le qualifier.

```
(ii) Insert (n°-dossier = 542, classeur = 'nouveaux') to DOSSIER
       and c to contrat c part-of DOSSIER
     where c.n°-ref = 728 ;
```

Ici les contrats c qui satisfont la clause-where (il n'y en a qu'un seul) seront agrégés dans le dossier que l'on est en train d'insérer. Si à la fin de la transaction, la cardinalité maximale de faits agrégés pouvant être contenus dans un fait agrégat est dépassée (ou si la

cardinalité minimale n'est pas atteinte), les contraintes d'intégrité du schéma ne seront pas satisfaites, l'usager sera averti et la transaction sera avortée.

Dans l'exemple précédent, on insère le dossier sans lettres. On peut ne pas agréger des lettres dans un dossier dans la même transaction car la cardinalité minimale est 0. Cependant, si nous voulons agréger des lettres à un dossier qui a déjà été inséré, on pourra toujours le faire, mais à l'aide d'un énoncé de modification (commande "replace" - cf. section 5.4.2., exemple 49); les lettres devront déjà être insérées dans l'entité LETTRE.

L'insertion de ce dossier se propage obligatoirement à l'association GESTION (cf. exemple 41). Les insertions (et les modifications) dans une entité E dérivée par agrégation d'entités peuvent se propager aux entités dérivées par généralisation des entités agrégées de E. En effet, les prédicats de restriction associés à une entité dérivée par généralisation peuvent induire ce que nous appelons "raffinement par agrégation" (cf. section 3.3). Par exemple, on peut définir une entité spécialisée de LETTRE, appelée LETTRE-DE-DOSSIER, ayant comme prédicat de restriction "part-of DOSSIER". Chaque fois qu'une lettre est incluse dans un dossier, on doit insérer la lettre dans LETTRE-DE-DOSSIER. La propagation sera conditionnelle ou obligatoire selon que l'option "manuelle" est prise ou non dans la définition de LETTRE-DE-DOSSIER.

Nous considérons maintenant les insertions de faits aux entités dérivées par agrégation associative. Prenons l'exemple cité dans la section 3.3.2.: sur une association ENVOI entre FOURNISSEUR et ARTICLE, on définit une entité EXPEDITION par agrégation associative. On supposera que l'attribut "quantité a été défini sur EXPEDITION. Rappel: La clé de EXPEDITION est formée par les clés des entités liées, nom (de fournisseur) et n°-réf (de article).

Dans le but de préserver l'orthogonalité du langage, on pourra insérer des faits dans une entité dérivée par agrégation associative soit directement, soit à partir des classes sous-jacentes.

Exemple 39 : Insertion d'une expédition

(i) Insertion directe

```
Insert (nom = 'Fourcault',..., n°-ref = 32, ..., date =
        (1,1,80), quantité = 20) to EXPEDITION ;
```

Ici l'insertion est propagée automatiquement aux classes sous-jacentes FOURNISSEUR, ARTICLE et ENVOI. L'unicité de la clé est vérifiée dans FOURNISSEUR et ARTICLE (et a fortiori dans EXPEDITION).

(ii) Insertion à partir de FOURNISSEUR

```
Insert f with (n°-ref = 32, ..., date = (1,1,80),
               quantité = 20) to EXPEDITION ;
  from FOURNISSEUR f
 Where f.nom = 'Fourcault';
```

L'insertion est propagée automatiquement aux classes ARTICLE et ENVOI. On peut parler ici (ainsi que dans (iii)) d'insertion "semi-directe". Dans ce cas, seul un fournisseur est qualifié ce qui implique qu'il y aura seulement un fait inséré dans ENVOI et EXPEDITION correspondant à l'article inséré et au fournisseur qualifié. Dans le cas général, plusieurs fournisseurs peuvent être qualifiés et donc il y aura un fait inséré dans ENVOI et dans EXPEDITION par fournisseur qualifié.

On peut également faire des insertions à EXPEDITION à partir de ARTICLE et FOURNISSEUR (auquel cas on propage l'(es) insertion(s) à ENVOI automatiquement), et aussi à partir de FOURNISSEUR, ARTICLE et ENVOI:

(iii) Insertion à partir de ENVOI, FOURNISSEUR et ARTICLE

```
Insert f,a,e with (quantité = 20) to EXPEDITION
  From ENVOI e : (fournisseur f, article a)
 Where f.nom = 'Fourcault' and a.n°-ref = 32 ;
```

Dans ce cas, comme il est indiqué par le trajet ayant comme racine l'association ENVOI, l'article, le fournisseur et la liaison entre les deux se trouvent déjà dans la base. Il n'y a pas de propagation aux classes sous-jacentes. Si EXPEDITION fait partie d'une hiérarchie de généralisation ou si elle est liée avec d'autres entités par l'intermédiaire d'associations, il faudra également propager cette insertion. Ceci s'applique aussi aux exemples précédents (i), (ii).

5.2.2. Insertion dans les associations

Quand on insère un fait dans une association, on lie deux, ou plusieurs, faits qui appartiennent chacun aux entités qui participent dans l'association. Il faut donc que ces faits existent au préalable.

Exemple 40 : affecter l'employé Bonnet au département 520.

```
Insert (e,d) : (date =(1,1,84) ) to INSCRIPTION
  From DEPARTEMENT d, EMPLOYE e
 where e.nom = 'Bonnet' and d.n°-dept = 520 ;
```

Ici, il y a autant de trajets qu'il y a de classes liées. Les faits de celles-ci sont qualifiées à l'aide d'une clause-where. L'insertion d'un fait (e,d) avec la valeur de l'attribut "date" dans AFFECTATION lie les faits e et d dans cette association. Les contraintes de cardinalité pour chaque entité sont testées: si à la fin de la transaction elles ne sont pas verifiées, la transaction ne sera pas validée.

On peut insérer plusieurs faits dans une association dans une commande insert, comme il est montré dans l'exemple suivant.

Exemple 41 : responsabiliser la secrétaire de Monsieur DUPONT des dossiers récemment insérés qui se trouvent dans le classeur "Nouveaux" (ici on suppose qu'il y a une association SECRETARIAT entre INGENIEUR et SECRETAIRE, et que chaque ingénieur a au plus une secrétaire).

```
Insert (s,d) to GESTION
From secrétaire s of INGENIEUR i, DOSSIER d
where d.classeur = 'nouveaux' and i.nom = 'Dupont';
```

Dans cet exemple, il faut qu'il y ait au plus 20 dossiers qui se trouvent dans le classeur "nouveaux" autrement les contraintes de cardinalité de l'entité SECRETAIRE vis-à-vis de l'association GESTION ne seront pas respectées. Les contraintes de cardinalité relatives à DOSSIER seront satisfaites si les dossiers qui se trouvent dans le classeur "nouveaux" ont été insérés dans la même transaction et si M. DUPONT a une secrétaire (et une seule). Il ne peut pas avoir plus d'une secrétaire car on a supposé que la participation de INGENIEUR à SECRETARIAT est minimum 0, maximum 1. S'il n'a pas de secrétaire, aucun fait s ne sera qualifié et donc aucun fait (s,d) ne sera inséré dans GESTION.

Dans une transaction on est très souvent amenés à propager l'insertion d'une entité à travers une association. On serait obligé d'écrire des suites d'énoncés d'insertion dont l'insertion dans l'association nécessiterait une qualification des faits des entités qui auraient été insérées au préalable dans la suite. Pour éviter cela, LAMBDA offre une syntaxe plus commode où l'on évite la qualification de faits que l'on vient d'insérer.

Exemple 42 : Insérer l'article 'tournevis', le fournisseur 'dubois' et lier tous les deux au département de quincallerie par l'intermediaire de l'association FOURNITURE à la date 1-12-82.

```
Insert a : (nom = 'tournevis', ...) to ARTICLE
  and f:  (nom = 'dubois', ...) to FOURNISSEUR
  and (a,f,d) : (date = (1,12,82)) to FOURNITURE
 From DEPARTEMENT d
Where d.nom = 'Quincallerie' ;
```

La désignation des faits, insérés dans les entités par des variables, évite la qualification dans l'insertion à l'association FOURNITURE.

L'insertion d'un fait à une association A se propage aux entités E dérivées par agrégation associative telle que A est sous-jacent à E ; cette propagation est obligatoire. Toute insertion dans ENVOI doit être propagée à EXPEDITION. Comme nous l'avons vu dans la section précedente, l'inverse est aussi vrai. En effet, le concept d'agrégation associative étant très proche du concept de vue couramment utilisé dans les bases de données relationnelles, il y a une correspondance bijective entre les faits d'une entité dérivée par agrégation associative et les faits de l'association sous-jacente.

Quand une entité E liée à une association A a une entité D dérivée par généralisation dont le prédicat de restriction inclut un "raffinement par association" (cf. section 3.3), l'insertion d'un fait dans A se propage (conditionnellement ou obligatoirement) à D. Par exemple si CHEF est une entité spécialisée de INGENIEUR dont le prédicat est "chef of DEPARTEMENT", l'insertion d'un fait dans DIRECTION qui lie un INGENIEUR, qui ne participait pas au préalable à DIRECTION, doit se propager à CHEF. Si l'option "manuelle" est prise, cette propagation est conditionnelle.

5.2.3. Propagations des insertions dans un schéma TIGRE

Cette sous-section présente un sommaire concernant la propagation des insertions dans un schéma TIGRE. Ce concept constitue le moyen que nous avons adopté pour satisfaire des contraintes d'intégrité implicites (appelées aussi contraintes structurelles) dans un schéma conceptuel. Nous ne traiterons pas ici le cas des contraintes explicites qui doivent être vérifiées lors des insertions.

Les propagations obligatoires seront automatiquement effectuées par le système quand toutes les valeurs des attributs (qu'on ne permet pas qu'ils prennent la valeur nulle) seront spécifiées.

1. Soit E une entité. L'insertion d'un fait dans E a les conséquences suivantes :

a) Propagation aux associations A auxquelles E participe

Les propagations aux associations ne sont pas automatiques; l'usager doit lier explicitement les faits et donner des valeurs aux attributs ds associations.

Cette propagation est obligatoire si la cardinalité minimale de participation de faits de E dans A, notée min (E,A), est supérieure ou égale à 1.

Si E_2, ..., E_n sont les autres entités qui participent à A, il faut choisir min (E,A) faits de E_2 x ... x E_n ; chacun donne lieu à l'insertion d'un fait dans A qui lie les faits correspondants dans E_1, E_2, ... E_n.

Il y a ici une propagation conditionnelle à chaque E_i, car le choix des min (E,R) faits de E_2 x ... x E_n peut impliquer des insertions dans les entités E_i.

b) Propagation dans la hiérarchie de généralisation dans laquelle E se trouve

La propagation aux ancêtres de E est obligatoire (E n'est pas dérivée par l'opération d'union). En plus, les prédicats de restriction dans la définition de E et dans les ancêtres de E doivent être satisfaits. La propagation aux entités dérivées D_i de E, tel que le prédicat de restriction associé à la définition de D_i est satisfait par le fait inséré dans E, est conditionnelle si l'option "manual" n'est pas prise. Elle est obligatoire dans le cas contraire.

c) Propagation si E est dérivée par agrégation

- Si E est dérivée par agrégation associative, et si l'insertion est directe ou semi-directe, la propagation est obligatoire aux classes sous-jacentes. Si l'insertion est accomplie à partir de toutes les classes sous-jacentes (comme dans l'exemple 39 (iii)) l'insertion ne se propage pas à ces classes.

- Si E est dérivée par agrégation d'entités, et si les faits des classes agrégées sont insérés directement (exemple 38 (1)), la propagation est obligatoire aux classes agrégées. Les contraintes de cardinalité minimale et maximale de faits agrégés pouvant être contenues dans un fait agrégat doivent être vérifiées.

- Propagation conditionnelle ou obligatoire aux entités dérivées de E par généralisation ayant comme prédicat de restriction "part-of E".

2. Soit A une association. L'insertion d'un fait dans R a comme conséquences :

   a) Vérification des cardinalités minimale et maximale pour chaque entité qui participe à A.

   b) Propagation dans la hiérarchie de généralisation dans laquelle A se trouve. Elle est obligatoire aux ancêtres de A, et les prédicats de restriction de R et de ses ancêtres doivent être vérifiés. La propagation aux associations spécialisées de A tel que le fait inséré dans A satisfait le prédicat de définition de ces associations, est conditionnelle si l'option "manuelle" est prise. Autrement elle est obligatoire.

   c) Propagation obligatoire aux entités dérivées E par agrégation associative tel que A est sous-jacente à E.

   d) Propagation (conditionnelle ou obligatoire) aux entités D dérivées par généralisation d'une entité E liée à A tel que le prédicat de restriction associé à D est Role(E,A) of E', où E' est une autre entité liée à A.

En général, les propagations de mises à jour forment des "trajets" dans le graphe du schéma. Ces trajets peuvent être arborescents et même circulaires. Il a été montré dans [SSW 79] que le résultat global de la propagation est indépendant de l'ordre d'execution de la propa-

gation des sous-trajets. En ce qui concerne les propatations circulaires, on a montré qu'elles doivent forcément terminer. En plus, on peut "optimiser" les propagations si l'on connait les cardinalités des entités participant aux associations qui se trovent dans un trajet circulaire.

5.3. SUPPRESSIONS

Les suppression en LAMBDA sont aussi des opérations ensemblistes, elles s'expriment à l'aide de la commande delete. Pour supprimer un ensemble de faits on le qualifie à l'aide d'une clause-where. S'il n'y a pas de clause-where tous les faits sont qualifiés et la suppression portera sur tous. La suppression d'un fait dans une classe C peut impliquer la propagation de la suppression à d'autres classes liées avec C dans le schéma conceptuel. Nous considérons ces propagations par la suite, comme ceci a été fait pour les insertions. Par opposition aux propagations par insertion, les propagations par suppression sont toutes automatiques.

5.3.1. Suppressions dans les entités

5.3.1.1. Entités non dérivées et entités dérivées par généralisation

Pour supprimer les faits dans une entité E on utilise un trajet pour désigner E avec une variable. Si la qualification des faits à supprimer ne dépend que des valeurs de faits dans E, le trajet se réduit à sa racine E. Par contre, si la qualification dépend des liaisons (par association ou par agrégation) avec des faits d'autres classes, celles-ci seront notées et désignées par des variables dans le trajet.

Exemple 43 : supprimer l'ingénieur Dupont

```
Delete i from INGENIEUR i
where i.nom = Dupont ;
```

La suppression d'un INGENIEUR se propage aux classes dérivées (EMPLOYE-DE-BUREAU). Elle ne se propage pas aux ancêtres. Dupont con-

tinue à exister dans la base en tant qu'employé. Il conserve les valeurs des attributs de EMPLOYE, en particulier la valeur ("ing") de l'attribut "catégorie". La suppression de Dupont en tant qu'ingénieur devrait s'accompagner d'un changement de la valeur de cet attribut car l'option "manual" n'a pas été prise dans la définition de INGENIEUR : tout fait de EMPLOYE qui satisfait le prédicat "catégorie = 'ing' " doit être inséré dans INGENIEUR.

Exemple 44 : supprimer les fournisseurs du département de jouets

```
Delete f from fournisseur f of DEPARTEMENT d
 where d.nom = 'jouets' ;
```

Cet exemple montre que l'on peut supprimer des faits d'une entité (FOURNISSEUR) qui sont liés par association avec d'autres faits (de DEPARTEMENT) que l'on peut qualifier dans une clause-where. La suppression d'un fournisseur se propage aux associations auxquelles l'entité FOURNISSEUR participe, c'est à dire, STOCKE et FOURNITURE. Il faut supprimer les faits de ces deux associations qui lient les fournisseurs suprimés avec des articles dans l'association STOCKE et avec des articles et des départements dans l'association FOURNITURE. En particulier, à cause de la suppression dans FOURNITURE, le département de jouets se trouve désormais non seulement sans fournisseurs, mais aussi sans articles fournis. C'est un effet de bord qui résulte du fait que l'association est ternaire. La suppression des faits d'une association peut à son tour se propager aux entités liées, comme nous le verrons plus tard. Dans ce cas, il n'y a pas de propagation, car la participation minimale de ARTICLE à DEMANDE et à STOCKE est zéro.

5.3.1.2. Suppression de documents partagés

Nous avons mentionné dans le chapitre 5.1.2. que l'on peut mettre à jour un document à partir d'autres documents ou de parties de documents (que nous appelons sous-documents) auxquels on a le droit d'accéder. Ceci peut être accompli dans l'éditeur ou par langage de deux façons différentes : par copie ou par partage. Nous nous intéressons dans cette section au problème de la suppression de documents partagés.

Il y a plusieurs politiques de suppression de documents partagés. Une première politique est celle utilisée dans le système d'autorisation SAGE [AA 84]. Dans cette politique, on part du principe que tout objet de la base a un propriétaire qui est le créateur de l'objet. Le propriétaire de l'objet peut transmettre des droits sur l'objet à d'autres usagers. En particulier, un document ou un sous-document est un objet. Si un (sous-)document est partagé par plusieurs usagers, c'est parce que le propriétaire leur a donné le droit de le lire ou de le modifier, mais ils n'ont pas le droit de le détruire : il n'y a que le propriétaire qui a ce droit.

Une deuxième politique consiste à dire qu'à partir du moment où un (sous-) document est partagé, personne ne peut le détruire, même pas le propriétaire (s'il y en a un). Par rapport à la première politique, ceci revient à dire que le propriétaire d'un objet perd le droit de destruction d'un objet à partir du moment où il est partagé.

Les deux politiques diffèrent donc en la gestion de droit sur les (sous-) documents. Mais quelle que soit la politique adoptée, quand un usager supprime un fait d'une classe qui contient un document, on supprime physiquement de la base :

- les valeurs du fait qui ne sont pas des documents,
- les sous-ensembles du document qu'on a le droit de supprimer.

5.3.1.3. Entités dérivées par agrégation

Quand on veut détruire un fait d'une entité dérivée par agrégation, par exemple un dossier, la question que l'on se pose est de savoir si les lettres et le contrat du dossier seront supprimés aussi de leurs entités respectives. Le principe le plus important dans un langage de manipulation est que l'on puisse faire des opérations sur les objets que l'on modélise qui ressemblent le plus possible aux opérations que l'on fait sur les objets dans la réalité. Ceci est particulièrement vrai dans le cas des agrégations d'entités. Quand on détruit un dossier, on détruit tout ce qu'il y a à l'intérieur. Quand on détruit

une FLOTTE (agrégation de NAVIRES, AVIONS et SOUS-MARINS), on l'anéantit complètement.

Exemple 45 : Suppression du dossier 425

```
Delete d from DOSSIER d
 where d.n°-dossier = 425 ;
```

Il est donc naturel de dire que la suppression d'un fait d'une entité "agrégat" (DOSSIER) implique la suppression de tous les faits agrégés (CONTRATs et LETTREs) dans le fait agrégat.

Si on veut supprimer un dossier sans détruire le contrat ni les lettres, on peut soit les transférer dans un autre dossier, soit "vider" le dossier avant de les supprimer. Ceci peut se faire en utilisant la commande replace (cf. section 5.4.2.).

Les suppressions dans les entités dérivées par agrégation associative sont propagées à l'association sous-jacente. L'inverse comme dans les insertions est aussi vrai. Ceci est indispensable pour maintenir la correspondance bijective entre les faits d'une entité dérivée par agrégation associative et les faits de l'association sous-jacente.

5.3.2. Suppression dans les associations

La suppression des faits dans les associations utilise un trajet ayant comme racine l'association et qui se ramifie ensuite aux entités liées (cf. section 4.2.3). La clause-where qualifie les faits des entités liées identifiant ainsi les faits de l'association à supprimer.

Exemple 46 : la secrétaire de numéro 514 ne s'occupe plus des dossiers du classeur "nouveaux"

```
Delete g from GESTION g = (responsable r, dossier-géré d)
 where r.n°-id = 514
   and d.classeur = 'nouveaux' ;
```

Dans cet exemple, on supprime autant de faits de l'association GESTION qu'il y a de dossiers appartenant au classeur 'nouveaux' gérés par la secrétaire de numéro 512. Pour chaque fait f de l'association que l'on supprime, on doit regarder si les faits liés par f participent encore un nombre de fois compris entre les cardinalités minimale et maximale associées à leurs entités respectives. Etant donné que les cardinalités de DOSSIER vis-à-vis de GESTION sont (1,1), chaque dossier lié à la secrétaire 512 par l'intermédiaire des faits de GESTION (que l'on supprime ici) ne participera plus à GESTION. Il est donc nécessaire que chacun de ces dossiers soit lié de nouveau à une (et une seule) secrétaire avant la fin de la transaction. Autrement, les contraintes de cardinalité ne seront pas satisfaites, et l'on est obligé de supprimer ces dossiers, car ils n'ont pas le droit d'exister sans participer à l'association GESTION. Autrement dit, la propagation de la suppression dans l'association se propage conditionnellement aux entités liées.

Si une entité E liée à une association A a une entité D dérivée par généralisation tel que son prédicat de restriction associé induit un "raffinement par association", la suppression d'un fait de A se propage conditionnellement à D. Dans l'exemple cité dans le chapitre 5.2.2., CHEF est une entité spécialisée de INGENIEUR dont le prédicat est "chef of DEPARTEMENT". La suppression d'un fait de DIRECTION sera propagée à CHEF si l'ingénieur lié ne participe plus à DIRECTION.

5.3.3 Propagation des suppressions dans un schéma TIGRE

1 - La suppression d'un fait f d'une entité E a comme conséquence :

(a) Propagation aux associations A auxquelles E participe.

Cette propagation est obligatoire (et automatique, comme toutes les propagations des suppressions). Il faut supprimer les faits de A qui lient le fait supprimé de E avec d'autres faits des entités $E_2, \ldots, E_n$ qui participent à A.

La propagation est conditionnelle à chaque E_i. Si $\min(E_i,A) \geqslant 1$, (c'est à dire si la participation de E_i à A est totale), la propagation aura lieu à E_i.

(b) Propagation dans la hiérarchie de généralisation dans laquelle E se trouve.

La propagation est obligatoire aux entités dérivées de E. Elle ne se propage pas aux ancêtres. Si A est un ancêtre direct de E et si E a été définie sans utiliser l'option "manual" et avec un prédicat de restriction P, la suppression de f dans E doit s'accompagner d'un changement des valeurs des attributs de f dans A (ou d'un changement structurel dans les liaisons de f avec d'autres faits dans la base) de façon à ce que f ne satisfasse plus P.

(c) Propagation si E est dérivée par agrégation.

- Si E est dérivée par agrégation associative, la propagation est obligatoire à l'association sous-jacente.
- Si E est dérivée par agrégation d'entités, la propagation est obligatoire aux entités "agrégées".

2 - La suppression d'un fait f d'une association A a comme conséquences :

(a) vérification des cardinalités minimale et maximale pour chaque entité qui participe à A.
(b) Propagation dans la hiérarchie de généralisation dans laquelle A se trouve - idem que pour les entités.
(c) Propagation obligatoire aux entités dérivées E par agrégation associative telle que A est sous-jacente à E.
(d) Propagation conditionnelle aux entités D dérivées d'une entité E qui participe à A. Les entités D sont telles que le prédicat de restriction de D induit un "raffinement par association" de la forme "rôle (E,A) of E'", où E' est une autre entité participant à A.

5.4. MODIFICATIONS

Les modifications en LAMBDA sont exprimées par la commande replace, qui permet de changer la valeur d'un ou plusieurs attributs d'un ensemble de faits d'une entité ou d'une association. Dans sa forme la plus simple, cette commande a un style assez standard :

Exemple 47 : mettre les dossiers des ingénieurs de l'ENSIMAG dans le classeur 'Grenoble'

```
Replace d.classeur : = 'Grenoble'
   from le-dossier d of INGENIEUR i
  where i.école = 'ENSIMAG' ;
```

Les attributs à modifier doivent appartenir à une seule entité ou association. Les nouvelles valeurs sont des constantes ou des variables temporaires auxquelles on a affecté le résultat d'un énoncé de sélection (cf. section 5.1) ayant un type compatible au type de l'attribut à modifier. Il n'y aura pas d'opérations arithmétiques où les (anciennes) valeurs des attributs interviennent ; cette possibilité sera offerte au moment de l'intégration de LAMBDA à un langage hôte.

La modification d'un attribut appartenant à la clé ne sera admise que si un seul fait est qualifié. Il faut en plus assurer l'unicité des clés à la fin de la transaction. Les modifications des clés ne seront pas admises dans les entités dérivées par agrégation associative (cf. section 5.4.2).

Les modifications des valeurs des attributs définies sur des types document seront discutées dans la section 5.4.3. En ce qui concerne des modifications sur des enregistrements on pourra modifier soit tout l'enregistrement soit des constituants particuliers. Pour les tableaux, on pourra également modifier tout le tableau ou seulement quelques éléments du tableau.

5.4.1. Modifications dans les entités dérivées par généralisation

Un énoncé de modification peut entraîner la suppression d'un ou plusieurs faits d'une entité dérivée par généralisation : c'est le cas quand l'on modifie la valeur d'un attribut intervenant dans le prédicat de restriction associé à l'entité dérivée.

Exemple 48: le programmeur 'Dupont' dévient ingénieur

```
Replace p.catégorie := 'ing'
   from  PROGRAMMEUR p
        where p.nom = 'Dupont' ;
```

Dans cet exemple, le fait correspondant à Dupont ne satisfait plus le prédicat "catégorie = prog". Il sera automatiquement supprimé de l'entité PROGRAMMEUR. Cette suppression peut se propager encore (cf. section 5.3), p.ex., si Dupont est un employé de bureau il sera aussi supprimé de cette entité. Avant la fin de la transaction, il faut insérer le fait de Dupont dans INGENIEUR, car il satisfait le prédicat associé et l'option "manuelle" n'est pas prise dans la définition de INGENIEUR. Cette insertion doit être explicite, car il faut spécifier les valeurs propres à INGENIEUR pour le fait de Dupont.

5.4.2. Modifications dans les entités dérivées par agrégation

Dans le cas des entités définies par agrégation d'entités (entités "agrégats"), les entités "agrégées" sont considérées comme attribut de l'entité agrégat. On les désigne comme toujours, en utilisant le mot-clé "part-of".

Exemple 49 : Agréger les lettres envoyées à Bertholet au dossier 542 (qui avait été inséré sans lettres dans l'exemple 49-(ii))

```
L : = Select l from LETTRE l
      where 'Bertholet' in l.liste-destinataires ;
```

```
Replace lettre part-of d := add L
   from DOSSIER d
  where d.n°-dossier = 542 ;
```

Ici les lettres envoyées à Bertholet sont ajoutées à celles qui existaient avant dans le dossier (mot-clé "add"). On peut aussi enlever des lettres d'un dossier (mot-clé "remove") sans pour autant les supprimer de l'entité LETTRE (ce qui s'accomplit avec la commande "delete", cf. section 5.3). On peut également tout simplement modifier les lettres d'un dossier par d'autres (auquel cas on n'utilise pas "add" ni "remove"). A la fin de la transaction il faut que les lettres appartiennent au plus à un seul dossier.

Les modifications sur une entité E définie par agrégation associative ne portent que sur les attributs propres et les attributs de l'association A sous-jacente. Elles ne peuvent pas porter sur les attributs correspondants aux entités E_i liées par A car les faits de E ne sont pas en correspondance bi-univoque avec les faits de E_i. Le problème est le même que celui de la propagation de mises à jours au travers d'une vue [BS 81]. Bien entendu, les modifications sur E portant sur les attributs de A sont propagées à A. L'inverse est aussi vrai. En plus, les modifications sur un fait e_i de E_i sont propagées aux faits e de E qui coïncident avec e_i dans les attributs de E correspondant à E_i.

Exemple 50 : la quantité des articles de luxe expédiés par le fournisseur 'Fourcault' est fixée à 40 unités à partir de Mars 83 (cet exemple utilise le schéma défini dans le chapitre 5.1.1.2., on suppose que l'attribut "type" est défini dans le type de ARTICLE)

```
Replace x.quantité := 40, x.date := (1,3,83)
   from EXPEDITION x
   where x.nom = 'Fourcault' and x.type = "luxe" ;
```

Dans cet exemple, la mise à jour des faits qualifiés sur l'attribut "depuis" est propagée automatiquement à ENVOI. Cette propagation ne pose aucun problème car il existe une correspondance bi-univoque entre les faits de EXPEDITION et les faits de ENVOI.

5.4.3. Modifications dans les documents

LAMBDA permet de modifier des documents ou des parties de documents (sous-documents) de deux manières différentes :

- soit en utilisant l'éditeur,
- soit directement, sans passer par l'éditeur.

Nous avons présenté dans la section 5.1.2. des exemples d'utilisation de la commande replace dans un scénario de mise à jour d'un (sous-) document en utilisant l'éditeur, le(s) (sous-) document(s) objet(s) est (sont) remplacé(s) par le (sous-) document source, du type compatible, en provenance de l'éditeur. Ceci correspond à l'option par copie décrite la section 5.1.2.

Dans le cas d'une mise à jour d'un (sous-) document sans passer par l'éditeur, le (sous-) document source se trouve dans la base. Il pourra être incorporé au(x) (sous-) document(s) objet(s) soit par copie -c'est l'option par défaut-, soit par partage. Le (sous-) document source et le (les) (sous-) document (s) objet (s) doivent être de types compatibles.

Exemple 51 Remplacer (par partage) la 4ème clause du contrat de P. FREMONT par la 3ème clause du contrat de "R. DUBOIS"

```
X := First-fact (Select clause 3 of corps of c.texte-contrat
                 from CONTRAT c where parameter Titulaire of
                 c.texte-contrat = "R. DUBOIS" ;)

replace clause 4 of corps of c.texte-contrat := share X
   from CONTRAT c
  where parameter titulaire of c.texte-contrat = "P. FREMONT" ;
```

Ici, l'identificateur interne de la 3ème clause du contrat de "R. DUBOIS" est affecté à la variable X. C'est cet identificateur qui remplace celui de la 4ème clause du contrat de "P. FREMONT" car le remplacement est par partage (mot-clé "share"). Toutes les modifications de la 3ème clause du contrat de "R. DUBOIS" seront visibles dans la 4ème clause du contrat de "P. FREMONT" et vice-versa.

LAMBDA permet également d'affecter des valeurs à des paramètres d'un document:

Exemple 52 Le nouvel employé dont le contrat a le n° 528 est affecté à Paris

```
Replace parameter établissement of c.texte-contrat : = "Paris"
   from CONTRAT c
   where c.n°-référence = 528 ;
```

Toutes les occurrences du paramètre "établissement" dans le contrat 528 auront la valeur 'Paris'. Ce type de mise à jour d'un document ne passe pas non plus par l'éditeur.

CHAPITRE 6

COMPLETUDE ET DISCUSSION

SOMMAIRE

6 - COMPLETUDE ET DISCUSSION

Dans ce chapitre, nous évaluerons le pouvoir d'expression de LAMBDA et nous comparerons LAMBDA à d'autres langages proposés dans la littérature. Nous n'avons pas défini des notions de complétude pour le modèle TIGRE. En particulier, la partie manipulation de documents n'a pas un support formel, et à notre connaissance, aucune notion de complétude n'a été définie concernant les recherches par contenu dans un texte ou la navigation dans la structure hiérarchique d'un document.

En conséquence, nous évaluerons le pouvoir d'expression d'une partie des énoncés d'interrogation de LAMBDA : celle associée à un modèle de données (le plus "proche" possible de TIGRE) où l'on ait défini une notion de complétude. Deux candidats possibles sont le modèle Relationnel et le modèle Entité-Association (ER) ; nous choisirons ce dernier, car TIGRE est une extension directe de ce modèle.

Deux mesures de complétude de langages associés au modèle ER ont été définies récemment : un calcul de prédicats (calcul ER) [AC 81] et une algèbre (Algèbre ER) [MR 83a]. Cependant, ces deux langages ne sont pas équivalents. Dans le calcul ER, la restriction suivante est imposée : on ne peut pas écrire des requêtes dont la liaison sémantique entre deux entités ne découle pas des associations existant dans le schéma entre ces deux entités. L'argument qui soutient cette restriction est : "Si vous voulez rapprocher deux entités dans une requête, faites-le à priori dans le schéma par l'intermédiaire d'une association". Cette restriction n'est pas imposée dans l'algèbre ER (ni dans LAMBDA, comme nous l'avons vu dans la section 4.2.3). C'est pour cela que nous allons démontrer la complétude de la partie Entité-Association de LAMBDA par rapport à l'algèbre ER. Nous présenterons d'abord l'algèbre ER et ensuite, nous démontrerons la complétude. Dans la section 6.3, nous ferons une comparaison de LAMBDA avec d'autres langages proposés dans la littérature.

6.1 L'algèbre ER

La correspondance entre les modèles ER et relationnel peut être établie en choisissant une relation comme l'unité structurelle du niveau

"structures d'information" du modèle ER [Che 76]. L'idée de base est donc de faire en sorte que le modèle ER hérite de l'algèbre relationnelle. Mais, de même que le modèle ER offre des concepts proches de la manière dont les personnes perçoivent les données, l'algèbre ER offre des analogies avec la façon dont les personnes communiquent (la langue naturelle).

A chaque classe C, on associe une relation rC, dont le schéma est un ensemble de couples de la forme R : E ou A : V, où R est un rôle, E un ensemble d'entités, A un attribut et V un ensemble de valeurs.

Par exemple, àl'association AFFECTATION définie dans le chapitre 3, on associe la relation rAFFECTATION(employés: EMPLOYE, dept-affecté : DEPARTEMENT, date : DATE). Pour les entrées, le rôle est l'entité elle-même.

L'originalité de cette Algèbre ER est basée sur le fait que l'on peut dans une expression, référencer des occurrences des attributs par l'intermédiaire de variables (ou symboles) de corrélation. Chaque variable de corrélation a un compteur associé ; à l'aide de ce compteur, chaque fois que l'on évalue un opérateur algébrique, on se rend compte si les attributs obtenus seront nécéssaires par la suite ou non (sinon, il y a une projection implicite). On détermine également si les relations qui sont arguments des opérateurs binaires ont des attributs corrélés par la même variable (auquel cas, il y a une jointure implicite). Les variables de corrélation ont donc un rôle global dans une expression.

Dans l'algèbre ER, on spécifie ce qu'on appelle un schéma opérationnel à chaque occurrence de relation dans une expression P, ainsi que le schéma opérationnel de l'expression P. Un schéma opérationnel pour une occurrence r^J d'une relation r dans P, dénoté par $S(r^J)$ est un ensemble de couples (variable-de-corrélation : attribut). Le schéma opérationnel pour P a la même structure et donne l'ensemble de réferences globales. Une variable de corrélation ne référence que des attributs du même nom ; Inversement, un même attribut peut être référencé par des variables différentes. Outre que par des variables de corrélation,

on peut référencer des attributs via certains opérateurs ER, comme la sélection , le Θjoin et la division. On dit qu'un attribut apparaissant comme argument d'un de ces trois opérateurs est opérationnellement référencé. Ce concept permet de "garder" des attributs qui seront utilisés dans un de ces trois opérateurs jusqu'à ce qu'on ait évalué l'opérateur (voir plus loin).

Nous introduisons le principe par un exemple, puis nous donnerons les définitions correspondantes. Le schéma ER utilisé dans l'exemple est le sous-ensemble du schéma TIGRE qui concerne les départements, les articles, les fournisseurs, les stocks et les fournitures.

6.1.1 Exemple

Retrouver le département et l'article sachant que l'article est demandé par ce département en quantité $\geqslant 10$ et qu'il est fourni (à n'importe quel département) par un fournisseur qui stocke tous les articles fournis par lui à ce département (cet exemple est pris dans [MR 83a]).

L'expression algébrique est :

$$P : rDEMANDE^1 \textcircled{:} x_3 \geqslant 10 \textcircled{\times} (rFOURNITURE^1 \textcircled{\times} (rSTOCKE^1 \textcircled{\div}_{D=\{x_5:ARTICLE\}} rFOURNITURE^2))$$

Les schémas opérationnels correspondants sont :

$S(P) = \{x_1 : DEPARTEMENT, x_2 : ARTICLE\}$
$S(rDEMANDE^1) = \{x_1 : DEPARTEMENT, x_2 : ARTICLE, x_3 : QUANTITE\}$
$S(rFOURNITURE^1) = \{x_2 : ARTICLE, x_4 : FOURNISSEUR\}$
$S(rSTOCKE^1) = \{x_5 : ARTICLE, x_4 : FOURNISSEUR\}$
$S(rFOURNITURE^2) = \{x_5 : ARTICLE, x_4 : FOURNISSEUR, x_1 : DEPARTEMENT\}$

Les compteurs de variables sont :

$compteur(x_1) = 3$, $compteur(x_2) = 3$, $compteur(x_3) = 1$,
$compteur(x_4) = 3$, $compteur(x_5) = 2$.

L'évaluation de l'expression suit les pas suivants :

(1) Les valeurs des occurrences des relations sont déterminées par leurs schémas correspondants. Cette opération, appelée "établissement des corrélations" ("correlation assesment") est l'équivalent de l'opérateur de "renommage" d'attributs de l'algèbre relationnelle. Cependant, des projections implicites peuvent avoir lieu. Par exemple, dans rFOURNITURE[1], on ne s'intéresse pas au département.

(2) La division, $\oplus_D$, est basée sur la division généralisée de l'algèbre relationnelle dont la définition est : étant donné r(A) et s(B) avec $x = A \cap B$, la division généralisée de r par s est donnée par l'expression :

$$r \div s = \{t \mid t = t_1t_2,\ t_1 \in r[A-x],\ t_2 \in s[B-x] \text{ et } \forall t_3(t_2t_3\ s \Rightarrow t_1t_3 \in r)\}$$

Autrement dit, la division généralisée agit sur l'intersection des attributs des opérandes. Dans la division ER, elle agit sur un ensemble D appelé l'ensemble de division qui n'est pas contraint d'être égal à l'intersection des schémas opérationnels des opérandes, mais il peut y être contenu strictement.

Dans ce dernier cas, une jointure implicite est effectuée sur les attributs $(s(r) \cap s(s)) - D$.

L'ensemble D est un paramètre de l'opérateur de division. Le schéma opérationnel d'une relation argument de la division est partitionné en deux : l'ensemble de Division et le composant de Division. Tous les élements de ce dernier sont opérationnellement référencés.

Dans notre cas particulier, $D = \{x_5 : \text{ARTICLE}\}$, donc une jointure est effectuée implicitement sur $\{x_4 : \text{FOURNISSEUR}\}$. Les éléments $\{x_4 : \text{FOURNISSEUR}\}$ et $\{x_1 : \text{DEPARTEMENT}\}$ sont opérationnellement référencés.

Pour chaque variable qui apparaît dans l'intersection des schémas des opérandes, on diminue son compteur de 1 ; le schéma opérationnel de la relation résultat d'un opérateur algébrique ER est déterminé par les

variables appartenant à l'union des schémas opérationnels des opérandes dont le compteur est supérieur à 1 ou qui sont encore opérationnellement référencés. Autrement dit, si un attribut n'est plus référencé dans l'expression, une projection implicite est effectuée. Dans notre cas,

$$R_1 = rSTOCKE^1 \textcircled{\div} rFOURNITURE^2$$
$$D = \{x_5 : ARTICLE\}$$

compteur(x_4) = 2, compteur(x_5) = 1, x_1 et x_4 ne sont plus opérationnellement référencés.

Alors, $s(R_1) = \{x_4 : FOURNISSEUR, x_1 : DEPARTEMENT\}$

(3) L'opérateur $\textcircled{x}$, le produit cartésien ER, est identique au correspondant produit cartésien relationnel. Toutefois, si dans $r \textcircled{x} s$, on a $S(r) \cap S(s) \neq \emptyset$, alors on a une jointure implicite, suivie éventuellement d'une projection implicite. Dans notre cas :

$R_2 = rFOURNITURE^1 \textcircled{x} R_1$ avec $s(rFOURNITURE) \cap s(R_1) = \{x_4 : FOURNITURE\}$

alors compteur(x_4) = 1

et $s(R_2) = \{x_2 : ARTICLE, x_1 : DEPARTEMENT\}$

(4) L'opérateur de sélection ER : correspond à l'opérateur de sélection de l'algèbre relationnelle ; la seule différence est la projection implicite éventuelle. Tous les attributs apparaissant dans le prédicat de sélection sont opérationnellement référencés. Dans notre cas $\{x_3 : QUANTITE\}$ est opérationnellement référencé. Il ne le sera plus après l'exécution de la sélection.

$$R_3 = rDEMANDE^1 \textcircled{:} x_3 \geq 10$$

alors compteur(x_3) = 1, x_3 n'est plus opérationnellement référencé, donc :

$$s(R_3) = \{x_2 : ARTICLE, x_1 : DEPARTEMENT\}$$

(5) $$R_4 = R_3 \textcircled{x} R_2$$
avec $s(R_3) \cap s(R_2) = \{x_2 : ARTICLE, x_1 : DEPARTEMENT\}$

alors $s(R_4) = \{x_2 : \text{ARTICLE}, x_1 : \text{DEPARTEMENT}\}$

et $\text{compteur}(x_2) = \text{compteur}(x_1) = 2$

6.1.2 Les opérateurs ER

La projection et la jointure (naturelle) exceptées, tous les opérateurs algébriques relationnels ont des correspondants directs dans l'algèbre ER. Cependant, même si le sens des opérateurs est maintenu, les opérateurs ER ont une dimension "sémantique", car ils peuvent donner lieu à des projections implicites ou à des jointures implicites ou aux deux.

Soit Ψ un opérateur algébrique ER, soient R_1 et R_2 les arguments de L'ensemble des références mutuelles de R_1 et R_2 est noté :

$$M(R_1,R_2) = s(R_1) \cap s(R_2).$$

L'évaluation de $R_3 = R_1 \Psi R_2$ dans une expression P est accomplie en deux temps :

(a) L'application de l'opérateur relationnel homonyme (voir tableau ci dessous), combiné éventuellement avec une jointure implicite. Celà donne une relation R_3' avec un schéma $s(R_1) \cup s(R_2)$.

(b) La projection de R_3' sur $s(R_3)$ donnant R_3. Le schéma $s(R_3)$ est obtenu comme suit :

(i) pour chaque variable de corrélation x qui apparaît dans $M(R_1,R_2)$, faire compteur(x) : = compteur(x) - 1

(ii) $s(R_1)$ et $s(R_2)$ sont remplacés par

$$s(R_3) = \{(x:D) \mid (x:D) \in s(R_1) \cup s(R_2) \text{ et } \text{compteur}(x) > 1 \text{ ou } x \text{ est encore opérationnellement référencé dans P.}\}$$

Le tableau suivant définit schématiquement les opérateurs ER :

Opérateurs	Opérateur relationnel	Observations
$\circledast$	x	Jointure implicite si $M(R_1,R_2) \neq \emptyset$
$\circledcup$, $\circledcap$, $\circledminus$	$\cup, \cap, -$	Applicables si $S(R_1) = S(R_2)$
θ - join	Θ - join	Jointure implicite si $M(R_1,R_2) \neq \emptyset$ Les opérants de la Θ-comparaison sont opérationnellement référencés
$\circledcolon$	: (sélection)	Les attributs du prédicat sont opérationnellement référencés
$\circledast_D$	division généralisée	Division généralisée sur $D \subseteq M(R_1,R_2)$ Jointure implicite si $M(R_1,R_2) - D$ Les éléments de $S(R_1) - D$ sont opérationnellement référencés.

6.2 - COMPLÉTUDE DE LAMBDA

Dans ce qui suit, nous démontrerons qu'il est possible d'exprimer en LAMBDA n'importe quelle expression P de l'algèbre ER qui contient les opérateurs de produit cartésien, union, intersection, différence, sélection et division. Le θ-join ne sera pas considéré car on peut l'exprimer en termes du produit cartésien et de la sélection. Enfin, nous utiliserons dans 6.2.6 l'exemple présenté dans la section 6.1 pour construire une requête LAMBDA équivalente. D'abord, un peu de

notation. Soit E une entité (ensemble d'entités) du modèle ER avec attributs $A_1 \ldots A_n$, chaque attibut A_i étant une fonction de E à un ensemble de valeurs D_i ; Soit A une association (ensemble d'associations) entre les entités $E_i,\ldots,E_n$ avec attributs $A'_1,\ldots,A'_m$, où chaque A'_i est une fonction de A à un ensemble de valeurs D'_i. Nous dénoterons par rôle (E_i,A) le rôle que E_i joue vis-à-vis de A.

Soient rE et rA, les deux relations respectivement sous-jacentes à E et à A. Ces deux relations ont le schéma suivant :

rE (E : E, A_i : $D_1,\ldots,$ A_n : D_n)

rA (rôle (E_1,A) : $E_1,\ldots,$ rôle (E_1,A) : E_n, A_i : $D'_i,\ldots,A'_m$: D'_m)

Soit P, une expression de l'algèbre ER où l'on a défini le schéma opérationnel de chaque occurrence de relation dans P, ainsi que le schéma opérationnel de P lui même. Nous associerons d'abord une expression LAMBDA à chaque valeur d'occurrence de relation dans P obtenue par l'opération d'établissement des corrélations.

6.2.1 - Etablissement des corrélations

(a) Soit rE^i, une occurrence de rE dans P, avec un schéma opérationnel donné par :

$$S(rE^i) = \{x_0 : E,\ x_1 : A_i,\ \ldots,\ x_k : A_{i_k}\}\ ,\ k \geq 1$$

La variable de corrélation x_0 sera la variable qui désignera l'entité E dans l'expression LAMBDA (rE^i) correspondante à l'établissement de la corrélation rE^i. La "classe de rattachement " (CR) des variables $x_1, \ldots,x_k$ sera la variable de désignation x_0. Autrement dit :

$$CR(x_1) = CR(x_2) = \ldots = CR(x_k) = x_0$$

L'expression LAMBDA (rE^i) est donné par :

$$\text{LAMBDA}(rE^i) = \underline{\text{select}}\ x_0,\ x_0.A_{i_1},\ \ldots,\ x_0.A_{i_k}$$
$$\underline{\text{from}}\ E\ x_0$$

(b) Soit rA^i, une occurrence de rA dans P avec un schéma opérationnel :

$$s(rA^i) = \{x_1 : E_{i_1}, \ldots, x_m : E_{i_m}, x_{m+1} : A'_{i_{m+1}}, \ldots, x_{m+k} : A'_{i_{m+k}}\},\quad k \geqslant 0$$

L'expression LAMBDA (rA^i) contiendra un trajet formé d'un pas à travers l'association A, à partir d'une racine E_{i_1}. (Nous avons vu dans la section 4.2.3 que ce trajet est équivalent à celui qui résulterait du choix d'une autre entité comme racine). Le pas à travers A est multiple si $m \geqslant 3$ ou simple si $m = 2$.

Selon si $k = 0$ ou $k>0$, deux cas se présentent. Si $k = 0$, il n'y a pas d'attributs de A à retrouver ; il n'est donc pas nécessaire d'effectuer un pas vers l'association A. Si $k>0$, il faut par contre, l'effectuer. Nous avons donc :

$$\text{LAMBDA}(rA^i) = \underline{\text{Select}}\ x_1,\ldots,x_m,\ x'_0.A'_{i_{m+1}},\ldots,\ x'_0.A'_{i_{m+k}}$$
$$\underline{\text{from}}\ T$$

où :

$$T = \begin{cases} \cdot\ (\text{rôle}(E_{i_2},A)\ x_2\ ,\ldots,\ \text{rôle}(E_{i_m},A)\ x_m)\ \underline{\text{of}}\ E_{i_1}\ x_1,\ \text{si } m \geqslant 3,\ k = 0 \\ \cdot\ Ax'_0 : (x_1,\ldots,x_m)\ \underline{\text{of}}\ (\text{rôle}(E_{i_2},A)\ x_2,\ldots,\text{rôle}(E_{i_m},A)\ x_m \text{ of } E_{i_1}\ x_1 \\ \qquad\qquad \text{si } m \geqslant 3,\ k > 0 \\ \cdot\ \text{rôle}(E_{i_2},A)\ x_2\ \underline{\text{of}}\ E_{i_1}x_1,\ \text{si } m = 2,\ k = 0 \\ \cdot\ Ax'_0 : (x_1,x_2)\ \underline{\text{of}}\ \text{rôle}(E_{i_2},A)\ x_2\ \underline{\text{of}}\ E_{i_1}\ x_1,\ \text{si } m = 2,\ k > 0 \end{cases}$$

La variable x'_0 qui désigne l'association A est différente de toutes les variables de corrélation qui apparaîssent dans P. Nous avons en plus :

$$CR(x_{m+1}) = \ldots\ldots = CR(x_{m+k}) = x'_0$$

6.2.2 Produit cartésien

Soit R_1 et R_2, les deux relations (correspondant à des sous-expressions de P) arguments du produit cartésien ;

Soit $R_3 = R_1 \otimes R_2$. Nous construisons LAMBDA(R_3) à partir de LAMBDA(R_2) et LAMBDA(R_1).

L'ensemble $M(R_1,R_2) = S(R_1) \cap S(R_2)$ est partitionné en deux : l'ensemble des références mutuelles à des attributs d'une classe que l'on dénotera par $A(R_1,R_2)$ et l'ensemble des références mutuelles à des classes (entités) elles-mêmes, que l'on dénotera par $C(R_1,R_2)$. D'après la définition du produit cartésien, si $M(R_1,R_2) \neq \emptyset$, il y aura des jointure implicites dans P. Ces jointures implicites seront traduites en LAMBDA de la façon suivante :

(a) On produira un nouvel ensemble de trajets à partir des trajets notés dans LAMBDA(R_1) et LAMBDA(R_2) avec la propriété suivante : Si un trajet t_1 de LAMBDA(R_1) et un trajet t_2 de LAMBDA(R_2) contiennent une (ou plusieurs) variables en commun, appartenant à $C(R_1,R_2)$, ces deux trajets seront "fondus" en un seul trajet dans LAMBDA(R_3).

(b) On générera les prédicats de jointure explicite pour les attributs apparaîssant dans $A(R_1,R_2)$.

Finalement, on tiendra compte des projections implicites en construisant une expression de valeurs (clause select) pour LAMBDA(R_3) correspondant aux éléments de $S(R_3)$.

Pour accomplir (a), on appliquera l'algorithme suivant :

- Soit T_i, l'ensemble de trajets de LAMBDA(R_i), i=1,2

```
· pour chaque t2 de T2 faire
    pour chaque t1 de T1 faire
        si t2 et t1 ont une variable en commun qui se trouve dans
           C(R1,R2)
           alors "combiner" t1 et t2 en un seul trajet tc (voir
                                                          ci-dessous)
                t2 ← tc
                T1 ← T1 - {t1}
           fin
    fin
    T1 ← T1 ∪ {t2}
fin
```

· T_1 contient l'ensemble de nouveaux trajets de LAMBDA (R_3)

Pour "combiner" deux trajets t_1 et t_2 on applique les règles suivantes :

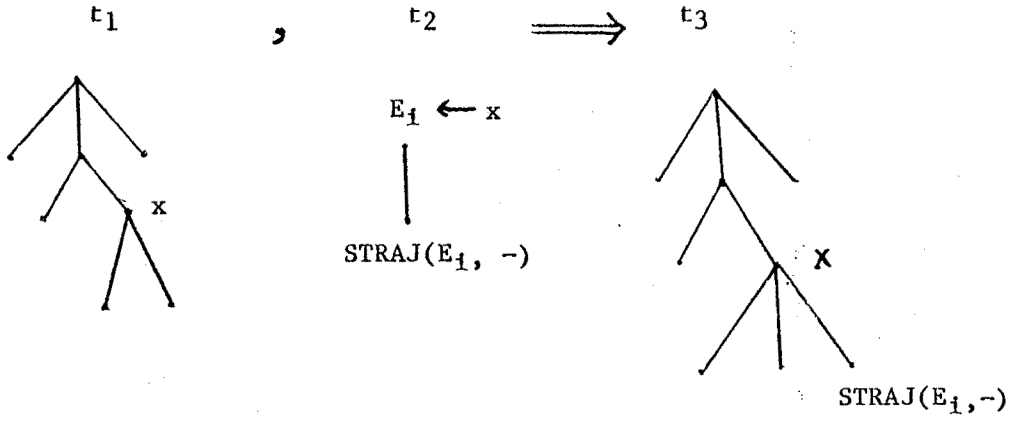
· soit X l'ensemble de variables en commun entre t_1 et t_2 ; choisir une variable x de X.

· si x n'apparaît comme racine d'aucun des trajets t_1 et t_2, alors on modifie un des deux trajets, p. ex. t_2, pour que x apparaisse comme racine. Pour celà, on considère t_2 comme une suite de pas linéaires ou/et arborescents (cf section 4). On change le sens des pas de façon à ce que x apparaisse comme racine, ce qui implique un changement de noms des rôles dans le trajet.

· A cette étape au moins un trajet a x comme racine ; supposons qu'il s'agit de t_2. Ce trajet a donc la forme STRAJ(E_i,-) of E_i x (cf section 4) où E_i est l'entité désignée par x.

· On étend le trajet t_1 à partir de l'endroit où x apparaît (il peut s'agir de la racine ou non), en faisant disparaître le nom de

l'entité E désignée par x en t_2. Le trajet résultant est le trajet t_c. Graphiquement :



Nous construisons maintenant la clause-where de LAMBDA(R_3) pour accomplir (b).

Pour chaque élement (x_1 : A_{j1}) de A(R_1,R_2), il doit exister (par construction) dans l'expression de valeurs de chaque LAMBDA (R_i), i = 1, 2, des valeurs (fonctions) de la forme $x_{ki}.A_{j1}$; les variables x_{ki} (i = 1,2) correspondent à CR(x_1) dans chaque LAMBDA (R_i). Elles désignent des classes C_i, telles que l'attribut A_{ji} appartient à C_i. Le prédicat de jointure associé à chaque élément (x_1 : A_{j1}) est P_{j1} : $x_{k1}.A_{j1} = x_{k2}.A_{j1}$.

La clause-where de LAMBDA(R_3) est alors :

$$P_1 \text{ and } P_2 \text{ and } P_{j1} \text{ and } \ldots\ldots \text{ and } P_{jm}$$

où chaque P_i est le prédicat de la clause-where de LAMBDA(R_i) et les P_{j1} sont les prédicats de jointure des éléments de A(R_1,R_2).

Règle PI (Projections implicites) : Les expressions de valeurs de LAMBDA(R_i) (i = 1,2) contiennent une fonction par élément de S(R_i). L'expression de valeurs de LAMBDA (R_3) sera composée de fonctions qui correspondent à :

$$s(R_3) = s(R_1) \cup s(R_2) - \{(x : D) \mid \text{compteur } (x) = 1 \text{ et } x \text{ n'est plus opérationnellement référencé.}\}$$

Cette règle sera appliquée pour tous les autres opérateurs de l'algèbre ER.

6.2.3 - Sélection

Soit R_1 la relation argument de l'opérateur de sélection. Soit $R_2 = R_1 (:) P$ où P est une combinaison boolléene de prédicats de la forme $x_i \Theta c$ ou $x_i \Theta x_j$. Les variables de corrélation x_i et x_j référencent respectivement des attributs A_i et A_j.

Les prédicats LAMBDA correspondants à $x_i \Theta c$ et $x_i \Theta x_j$ sont $x_{k_i}.A_i \Theta c$ et $x_{k_i}.A_i \Theta x_{k_j}.A_j$, où $x_{k_i} = CR(x_i)$ et $x_{k_j} = CR(x_j)$. Si P' est le prédicat de LAMBDA correspondant à P, l'expression LAMBDA (R_2) est obtenue en faisant une conjonction de la clause-where de LAMBDA(R_1) et P'.

Les projections implicites éventuelles seront considérées en appliquant la règle PI.

6.2.4 - Union, intersection, différence.

Soit (Ψ) un opérateur parmi l'union, l'intersection et la différence et soient R_1 et R_2 deux arguments de (Ψ) dans P. Soit $R_3 = R_1 (\Psi) R_2$.

Une façon évidente de construire LAMBDA(R_3) est :

$$\text{LAMBDA}(R_3) = \text{LAMBDA}(R_1) \Psi \text{ LAMBDA}(R_2)$$

Le problème de cette approche est que si R_3 est argument d'un autre opérateur ER dans l'expression P, il n'est pas aisé de construire d'autres expressions LAMBDA à partir de LAMBDA(R_3). Nous procéderons donc autrement : nous construirons deux ensembles LAMBDA correspondants à LAMBDA(R_1) et LAMBDA(R_2) pour ensuite leur appliquer l'opérateur Ψ dans la clause-where de LAMBDA(R_3).

On effectue d'abord un changement de variables dans LAMBDA(R_1) et LAMBDA(R_2). Chaque variable x_i apparaissant dans LAMBDA(R_1) sera changée par une variable x'_i différentes de celles qui se trouvent dans l'expression algébrique P ; les trajets T_i deviennent T'_i, le prédicat P_1 de la clause-where devient P'_1 et les fonctions $f_1 \ldots f_n$ de l'expression valeurs deviennent $f'_1, \ldots, f'_n$. On opère de la même manière pour LAMBDA(R_2) ; on trouve des variables $x"_i$ différentes de celles utilisées dans LAMBDA(R_1) et dans l'expression P, les trajets deviennent $T"_2$, le prédicat P_2 devient $P"_2$ et les fonctions $f_1 \ldots f_n$ (qui étaient par construction les mêmes que celles de LAMBDA(R_1)) deviendront $f"_1 \ldots f"_n$.

Soient $(x_1 : E_1)$, ..., $(x_k : E_k)$ les références mutuelles à des entités dans $M(R_1,R_2) = S(R_1) = S(R_2)$; l'ensemble de ces références mutuelles a été dénoté par $C(R_1,R_2)$ dans la section 6.2.2.

L'expression LAMBDA(R_3) est donnée par :

$$\begin{aligned} \text{LAMBDA}(R_3) = \ & \underline{\text{select}}\ F \\ & \underline{\text{from}}\ T'_1,\ T"_2,\ E_1\ x_1,\ \ldots,\ E_k\ x_k \\ & \underline{\text{where}}\ (f_1, \ldots, f_n)\ \underline{\text{in}}\ \{(f'_1, \ldots, f'_n)\ \underline{\text{where}}\ P'_1\} \\ & \qquad \Psi\ \{(f"_1, \ldots, f"_n)\ \underline{\text{where}}\ P"_2\} \end{aligned}$$

où F est l'ensemble de fonctions qui reste après avoir appliqué la règle PI aux fonctions $f_1, \ldots, f_n$. Si n = 1, on élimine les parenthèses de la clause-where.

6.2.5 - Division

Soient R_1 et R_2, deux arguments de l'opérateur de division $\oplus$, soit D l'ensemble division associé à $\oplus$ et soit R_3 le résultat. Comme pour les autres opérateurs, on construira LAMBDA(R_3) à partir de LAMBDA(R_1) et LAMBDA(R_2).

Soient $f_1, \ldots, f_k$ les fonctions correspondantes à l'ensemble D. Puisque $D \subseteq M(R_1,R2)$, ces fonctions sont les mêmes dans LAMBDA(R1) et

LAMBDA(R_2). Dans ce cas, nous changeons seulement les variables qui correspondent à l'ensemble D. Les trajets T_2 de LAMBDA(R_2) deviennent T'_2, le prédicat P_2 de la clause-where devient P'_2 et les fonctions $f_1, \ldots, f_k$ deviennent $f'_1, \ldots, f'_k$ (les fonctions correspondant à $S(R_2) - D$ ne changent pas).

Supposons que les fonctions $g_1, \ldots, g_n$ de LAMBDA(R_1) correspondant aus éléments de $S(R_1) - D$ et les fonctions $h_1, \ldots, h_p$ de LAMBDA(R_2) correspondants à $S(R_2) - D$ sont toutes des fonctions appartenant à la famille F_1 (cf. section 4.3.1), c'est à dire de la forme "x_1".

L'expression LAMBDA(R_3) est donnée par :

$$\begin{aligned}\text{LAMBDA}(R_3) = \ & \underline{\text{Select}}\ F \\ & \underline{\text{from}}\ T_3 \\ & \underline{\text{where}}\ \{(f_1, \ldots, f_k)\ \underline{\text{where}}\ P_1\}\ \underline{\text{by}}\ g_1 \ldots g_n \ \geqslant \\ & \qquad \{(f'_1, \ldots, f'_k)\ \underline{\text{where}}\ P'_1\}\ \underline{\text{by}}\ h_1 \ldots h_p\end{aligned}$$

où T_3 est l'ensemble de trajets obtenus après avoir appliqué l'algorithme de combinaison de trajets pour le produit cartésien (décrit dans 6.2.2) sur les trajets T_1 de LAMBDA(R_1) et T'_2 de LAMBDA(R_2) ; p_1 est le prédicat de la clause-where de LAMBDA(R_1) et F est l'ensemble de fonctions parmi $f_1, \ldots, f_k$, $g_1, \ldots, g_n$, $h_1, \ldots, h_p$ qui restent après leur avoir appliqué la règle PI.

S'il y a des fonctions parmi $g_1, \ldots, g_m$, $h_1, \ldots, h_p$ qui n'appartiennent pas à la famille F_1, l'expression LAMBDA(R_3) devient plus compliquée car l'indexation d'ensembles en LAMBDA est accomplie uniquement par des variables de désignation et non pas par des valeurs d'attributs. Il faut passer par l'introduction de trajets supplémentaires, comme celà a été illustré dans la section 4.4.2, exemple 26.

Les étapes à suivre sont les suivants :

Supposons, pour simplifier, qu'il n'y a que g_1 qui a la forme $x_1.A_j$ et que $k = 1$, autrement dit que f_1 est la seule fonction correspondante à l'ensemble D. Supposons que f_1 a la forme x_1 (ou $x_1.A_m$).

Si x_i et x_l appartiennent au même trajet t_1 dans T_1, soit t_{11} le sous trajet minimal de t_1 qui lie x_i à x_l. On l'écrit sous la forme d'un sous-trajet LAMBDA allant de x_i à x_l et on effectue un changement de variable sur t_{11} pour obtenir t'_{11} ; g_1 devient g'_1 et f_1 devient f'_1. L'expression LAMBDA (R_3) devient alors :

$$\text{LAMBDA}(R_3) = \underline{\text{Select}}\ F \quad \underline{\text{from}}\ T_3,\ t'_{11} \quad \underline{\text{where}} \left\{ f'_1\ \underline{\text{where}}\ g'_1 = g_1\ \underline{\text{and}}\ f'_1 = f_1 \right\} \underline{\text{by}}\ g_2,\ldots,g_m \geqslant \left\{ f_1\ \underline{\text{where}}\ P'_2 \right\} \underline{\text{by}}\ h_1,\ldots,h_p$$

Dans le cas (pathologique) ou x_i et x_l appartiennent à des trajets différents t_{12} et t_{13} de T_1, on détermine la partie P_{11} du prédicat P_1 liant les variables de t_{12} à celles de t_{13}. Un nouveau changement de variables est effectué sur t_{12} et t_{13} donnant t'_{12} et t'_{13} ; P_{11} devient P'_{11}, f_2 devient f'_2 et g_1 devient g'_1. L'expression LAMBDA(R_3) est :

$$\text{LAMBDA}(R_3) = \underline{\text{Select}}\ F \quad \underline{\text{from}}\ T_3,\ t'_{12},\ t'_{13} \quad \underline{\text{where}} \left\{ \begin{array}{l} f'_1\ \underline{\text{where}}\ g_1 = g'_1\ \underline{\text{and}}\ f_1 = f'_1 \\ \quad \underline{\text{and}}\ P_1\ \underline{\text{and}}\ P'_{11} \end{array} \right\} \underline{\text{by}}\ g_2,\ldots,g_m \geqslant \left\{ f_1\ \underline{\text{where}}\ P'_2 \right\} \underline{\text{by}}\ h_1,\ldots,h_p$$

6.2.6. Exemple de construction d'une expression LAMBDA à partir d'une expression algébrique.

Dans cette section, on construira un énoncé LAMBDA correspondant à l'expression algébrique de la section 6.1.1. Nous utiliserons les noms des rôles définis dans le schéma conceptuel de la section.

On applique d'abord l'opérateur d'établissement de corrélation aux occurrences de l'expression algébrique de 6.1. Celà donne les expressions suivantes :

LAMBDA(rDEMANDE^1) = select x_1, x_2, x'_3.quantité
from DEMANDE $x'_3 = (x_1, x_2)$ of article-demande x_2
of DEPARTEMENT x_1

LAMBDA(rFOURNITURE^1) = select x_2, x_4
from fournisseur x_4 of ARTICLE x_2

LAMBDA(rSTOCKE^1) = select x_5, x_4
from fournisseur-stockant x_4 of ARTICLE x_5

LAMBDA(rFOURNITURE^2) = select x_5, x_4, x_1
from (article-fourni x_5, fournisseur x_4) of
DEPARTEMENT x_1

Nous appliquons le procédé décrit dans 6.2.5 pour déterminer LAMBDA(R_1) corresondant à :

$$R_1 = rSTOCKE^1 \textcircled{\div} rFOURNITURE^2$$

$$D = \{x_5 : ARTICLE\}$$

$$S(R_1) = \{x_4 : FOURNISSEUR,\ x_1 : DEPARTEMENT\}$$

La variable x_5 est changée dans LAMBDA(rFOURNISSEUR^2) ; étant donné que les variables de S(rSTOCKE^1) et S(rFOURNITURE^2) sont toutes des variables de S(rSTOCKE^1) et que S(rFOURNITURE^2) sont toutes des variables désignant des entités, l'expression LAMBDA(R_1) a la forme :

LAMBDA(R_1) = select x_4, x_1
from T_1
where $\{x_5\}$ by $x_4 \geqslant \{x'_5\}$ by x_4, x_1

Le trajet T_1 est le résultat de la combinaison des trajets :

fournisseur-stockant x_4 of ARTICLE x_5
et (article-fourni x'_5, fournisseur x_4) of DEPARTEMENT x_1

La variable commune est x_4 et elle n'est racine d'aucun trajet. On

réécrit, par exemple, le deuxième trajet sous la forme (équivalente) suivante :

(article-fourni x'_5, dept-fourni x_1) of FOURNISSEUR x_4

Le trajet combiné T_1 est alors :

(article-fourni x'_5, dept-fourni x_1) of fournisseur-stockant x_4 of ARTICLE x_5

Il faut maintenant déterminer LAMBDA(R_2) correspondant à :

$R_2 = rFOURNITURE^1 \otimes R_1$
et $S(R_2) = \{x_2 : ARTICLE, x_1 : DEPARTEMENT\}$

Il s'agit ici de composer le trajet T_1 de LAMBDA(R_1) et le trajet "fournisseur x_4 of ARTICLE x_2". Si on change ce dernier de sens, on aura x_4 comme racine et on pourra combiner, ce qui donne :

LAMBDA(R_2) = select x_2, x_1
from ((article fourni x'_5, dept-fourni x_1), article-fourni x_2) of fournisseur-stockant x_4 of ARTICLE x_5
where $\{x_5\}$ by $x_4 \geqslant \{x'_5\}$ by x_4, x_1

L'expression LAMBDA(R_3) correspondante à :

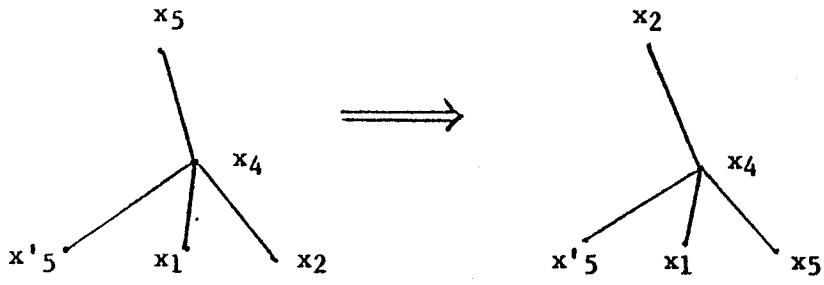
R_3 : $rDEMANDE^1 \oplus x_3 \geqslant 10$
$S(R_3) = \{x_2 : ARTICLE, x_1 : DEPARTEMENT\}$

est LAMBDA(R_3) = select x_2, x_1
from DEMANDE x'_3 : (x_1, x_2) of article-demande x_2 of DEPARTEMENT x_1
where x'_3. quantité $\geqslant 10$

Finalement, dans l'expression LAMBDA(R_4) correspondante à :

$R_4 = R_3 \otimes R_2$
$S(R_4) = \{x_2 : ARTICLE, x_1 : DEPARTEMENT\}$

on doit composer deux trajets qui contiennent deux variables communes x_2 et x_1. Il faut choisir une de ces variables, disons x_2 et la rendre racine d'un trajet, disons celui de LAMBDA(R_2). Le changement peut être représenté graphiquement comme suit :



L'expression LAMBDA finale est alors :

LAMBDA(R_4) = select x_1, x_2
from ((article-stocke x_5, (article-fourni x'_5, dept-fourni x_1))of fournisseur x_4, DEMANDE x'_3 : (x_1, x_2))
of article-demandé x_2 of DEPARTEMENT x_1
where x'_3 . quantité $\geqslant$ 10
and $\{x_5\}$ by $x_4 \geqslant \{x'_5\}$ by x_4, x_1

6.3 - COMPARAISON DE LAMBDA AVEC D'AUTRES LANGAGES

Nous allons comparer LAMBDA à d'autres langages sous deux points de vue différents:

1 - Pouvoir d'expression et lisibilité par rapport aux langages associés aux modèles sémantiques.
2 - Capacité de manipulation (recherches, mise à jour) des données généralisées.

6.3.1 - Comparaison avec d'autres langages associés à des modèles sémantiques.

Beaucoup de modèles sémantiques ont été proposés dans la littérature (cf. Chapitre 2) mais peu d'entre eux ont proposé des langages de re-

quêtes associés. Le modèle ER est peut être celui qui a reçu le plus d'attention de la part des chercheurs. Cinq langages de requêtes ont été proposés dans la littérature : GORDAS [Elm 81], Executable-langage [AC 81], ERROL [MR 83b], CLEAR [Poo 79] et CABLE [Sho 79], ainsi que deux langages formels : un calcul ER et une algèbre ER (cf. section 6.1). Deux langages associés au modèle fonctionnel ont été proposés : DAPLEX [Shi 81] et FQL [BF 79]. Le premier fait en effet partie d'une interface programmable alors que le deuxième est présenté comme un langage de requête. D'autres efforts sont RM/T [Cod 79] et GEM [Zan 83], [TZ 84]. Le premier propose un jeu d'opérateurs algébriques (une extension de l'algèbre relationnelle) pour interroger une base de données RM/T. Le second est un langage de requête et de mise à jour associé au modèle DSIS [LST 81] qui est un modèle sémantique (extension directe du modèle relationnel) offrant des notions d'entités avec "surrogates", agrégation, généralisation, valeurs nulles et attributs multivalués.

Comparons d'abord LAMBDA aux langages de requête ER. Du point de vue du pouvoir d'expression, LAMBDA, GORDAS et ERROL sont complets par rapport à l'algèbre ER (bien qu'à notre connaissance, celà n'a pas été démontré pour GORDAS). Executable-langage est complet par rapport au calcul ER ; il est donc moins puissant que les autres langages car il suppose que les données d'intérêt dans une requête ne sont pas dispersées, autrement dit, elles sont toujours liées par l'intermédiaire d'associations.

CLEAR et CABLE ne sont pas complets par rapport au calcul ER ; en effet, on ne peut pas construire des ensembles ni les comparer, comme dans les autres langages. Ils ne peuvent pas répondre aux questions pour lesquelles on utiliserait des quantificateurs universels dans le calcul ER.

La différence essentielle entre les langages de requête relationnels et les langages de requête ER est l'utilisation de l'information à propos des associations dans le schéma (noms d'associations, rôles) pour spécifier des opérations de jointure entre des entités. La formu-

lation de requêtes est rendue plus aisée du fait de la ressemblance avec la formulation de la requête en langue naturelle.

LAMBDA et tous les langages de requêtes ER (sauf ERROL) utilisent soit les noms des associations, soit des rôles des entités vis à vis des associations. La formulation de requêtes en GORDAS est hiérarchique, alors que dans CLEAR, CABLE, Ex-L et LAMBDA elle est linéaire (même s'il y a des trajets arborescents). Autrement dit, les concepts d'expression de valeurs et de trajets de désignation de LAMBDA se trouvent mélangés en GORDAS. La séparation artificielle entre les conditions s'appliquant aux entités et associations du trajet principal (appelés expressions de restriction logique) et les autres conditions est due à ce mélange.

Exemple Retrouver le nom du département dirigé par Durand et les noms des fournisseurs de ce département qui stockent des tournevis.

```
GORDAS : Get ⟨nom, ⟨nom of fournisseur : nom of article-stocké = 'tournevis'⟩⟩
              of DEPARTEMENT WHERE nom of chef = 'Durand'

LAMBDA : Select d.nom, f.nom
           from    article-stocké a of fournisseur f of département-
                   dirigé of EMPLOYE e
          where  e.nom = 'Durand'
            and    a.nom = 'tournevis'
```

Dans la requête GORDAS, la condition appliquée aux fournisseurs est un exemple d'expression de restriction logique. La formulation hiérarchique des requêtes peut être considérée comme une caractéristique désirable, car on "factorise" des valeurs qui se répètent (dans ce cas, les départements). Cet objectif pris dans GORDAS s'est heurté au problème d'extraire de l'information qui se trouve dans les attributs d'une association. Contrairement aux pas vers les associations de LAMBDA, un "abus de langage" est fait en GORDAS pour maintenir leur formulation hiérarchique : les attributs d'une association sont considérées à travers leurs trajets comme appartenant à l'entité destination dans un pas du trajet.

A notre avis, la formulation hiérarchique a une importance moindre quand on veut intégrer un langage de requêtes à un langage de programmation. Tel est notre cas. Il est important en outre, que le résultat d'une requête ait la même structure qu'une entité. L'utilisation de variables rend plus naturelle cette intégration. Par exemple, des langages tels que RIGEL [RS 79] et PASCAL-R [Sch 77] exploitent à fond l'utilisation de variables.

La construction de requêtes en ERROL est analogue à la construction de phrases en langue naturelle. Il est donc structurellement différent des autres langages de requêtes ER. Une requête ERROL a deux parties : la clause GET où l'on indique ce que l'on souhaite obtenir et une clause TIS (TIS est une abréviation de THAT-IS) dans laquelle on spécifie les associations qui lient les éléments de la clause GET. La dernière entité de la clause GET est le sujet de la clause TIS. Les associations, ainsi que les attributs d'une entité ou d'une association sont dénotés en ERROL par des squelettes de paraphrases en langue naturelle décrivant les associations et les propriétés (attributs) des entités ou des associations.

Nous écrivons la requête de l'exemple précédent en ERROL pour montrer ceci (nous mettons l'information du schéma en Anglais pour des raisons de lisibilité) :

```
ERROL :  GET NAME OF DEPARTMENT, NAME OF SUPPLIER TIS 'STOCKING ITEM
         'HAVING NAME = 'Tournevis' AND 'SUPPLYING DEPARTMENT! 'DIRECTED
         "BY EMPLOYEE 'HAVING NAME = 'Dupont'
```

Cette requête illustre plusieurs points importants de ERROL. D'abord on peut, dans la clause-TIS, corréler des référence à un même ensemble d'entités. Ceci est fait à l'aide du symbole "!". Ensuite, on peut connecter des paraphrases d'associations par deux moyens : enchaînement ("chaining") et branchement ("branching"). Ici, les associations SUPPLY (FOURNITURE) et DIRECTION sont connectées en chaînant leurs paraphrases correspondantes : SUPPLIER 'SUPPLYING DEPARTMENT! 'DIRECTED "By EMPLOYEE. Les mots préfixés par le caractère ' sont considérés comme des rôles aidant à identifier l'association; les mots préfixés par le caractère " sont des commentaires pour formuler la requête de

façon plus naturelle. Les associations STOCK et SUPPLY sont connectés par branchement en utilisant le connecteur AND 'SUPPLYING DEPARTMENT! Finalement, les selections sur des attributs apparaîssent toujours enchaînées aux paraphrases des associations.

On voit que la philosophie du langage est complètement différente de LAMBDA. Il est parfois difficile de connecter des paraphrases quand la requête est compliquée. ERROL a cependant des points communs avec LAMBDA : la spécification d'ensembles est basée sur le même concept, le concept de correspondance sur une association du schéma ou sur une association dérivée.

Du point de vue énoncé de mise à jour, nous ne connaissons que ceux de GORDAS. Le concept de transaction paramètrée y est offert. L'approche prise dans GORDAS pour maintenir l'intégrité sémantique de la base est d'interdire certaines mises à jour. On n'essaie donc pas de propager la mise à jour aux entités et associations adjacentes comme en LAMBDA.

Les trajets de LAMBDA ressemblent un peu à la composition de fonctions effectuées dans des langages associés aux modèles de données fonctionnels comme DAPLEX. Cependant, il y a des différences significatives. D'abord, les requêtes de DAPLEX sont basées sur la notion d'itération sur les ensembles d'entités et elles utilisent des variantes des quantificateurs existentiels et universels (énoncés for some, for all). Il n'y a donc pas d'opérateurs de comparaison d'ensembles, ce qui contraste avec la plupart des langages de requête ER. Ensuite, le mécanisme de fonctions dérivées utilisé en DAPLEX permet à l'usager de représenter des liaisons arbitraires entre les entités. On peut écrire des requêtes en utilisant des fonctions dérivées. En particulier, on peut spécifier des spécialisations, des unions et des intersections. La dépendance existentielle et l'héritage de fonctions applicables aux ancêtres est commune en LAMBDA (où l'on hérite des attributs et on participe aux associations des ancêtres) sauf que le concept de prédicat de restriction n'est pas offert. DAPLEX offre d'autres opérateurs pour définir des fonctions : INVERSE OF, TRANSITIVE OF (pour la clôture transitive), COMPOUND OF (agrégation "cartésienne"). La mise à jour de fonctions dérivées doit

être, par contre, spécifiée explicitement par l'usager.

GEM est une généralisation de QUEL. Les extensions sont simples et donnent une facilité d'utilisation et un pouvoir d'expression accrus. Elles adressent les concepts nouveaux du modèle DSIS par rapport au modèle relationnel. En ce qui concerne l'agrégation (la possibilité d'avoir des attributs de type "référence", c'est à dire des attributs dont la valeur est un n-uplet d'une autre relation), on propose une notation appelée jointure fonctionnelle qui évite la spécification explicite de prédicats de jointure. C'est l'équivalent de la notion de pas par association en LAMBDA. La différence est que l'on ne peut pas "enchaîner" des jointures fonctionnelles pour former des trajets comme en LAMBDA. Pour ce faire, GEM propose les jointures explicites d'entités dans lesquelles on peut comparer deux attributs référence. Ce sont des expressions de la forme "a is b". En ce qui concerne la généralisation (l'équivalent du concept de spécialisation en TIGRE), GEM permet à une variable de désigner une entité dérivée (par exemple, "range of s is SECRETAIRE" où SECRETAIRE est une entité derivée de EMPLOYE) et que l'on puisse utiliser les entités dérivées dans une clause-where (par exemple, "S is MARRIE", où MARRIE est une autre entité dérivée de EMPLOYE").

De ce point de vue GEM propose des constructions équivalentes à celles de LAMBDA. Une autre ressemblance est le traitement des attributs multivalués. La valeur d'un attribut multivalué est considérée comme un ensemble GEM (QUEL), de la même façon que les attributs tableau sont manipulés comme des ensembles LAMBDA. Le traitement des valeurs nulles est plus général en GEM : d'une part, il utilise une logique à trois valeurs booléennes (vrai, faux, nul) et d'autre part il permet à un attribut réference de pouvoir prendre la valeur nulle (cela équivaudrait à ce qu'on puisse lier un fait non nul avec un fait nul dans une association en TIGRE). La jointure fonctionnelle de GEM est en fait une jointure externe ("outer join") et la jointure explicite est une jointure naturelle. Finalement, à la différence de LAMBDA, GEM ne propage pas une mise à jour quand l'intégrité sémantique est compromise : il l'interdit.

6.3.2 Comparaison avec d'autres langages manipulant des données généralisées.

Actuellement, il existe très peu de langages permettant de manipuler des données généralisées. Regardons d'abord ceux qui manipulent des données généralisées non multi-média, autrement dit, des données alphanumériques à structure interne complexe et manipulées comme une unité.

6.3.2.1 - Manipulation de données alphanumériques généralisées

Dans SMH$^+$ [Bro 81] ce type de données correspondent aux éléments des classes S construites par application de l'opérateur d'association sur des classes M. Un élément de S est un ensemble d'éléments de M. Les primitives de manipulation définies sur S sont :

- insérer, supprimer un $m \in M$ à un $s \in S$

- union, intersection, différence sur deux éléments s_1, $s_2 \in S$.

- itération pour appliquer des actions sur les éléments m d'un ensemble $s \in S$

Ces primitives sont en fait des concepts de modélisation du comportement des objets décrits dans SMH$^+$; à notre connaissance, aucun langage de requête et de mise à jour proprement dit ayant ces concepts de base n'a été défini.

Dans les extensions récentes proposées pour System-R, le concept d'objet complexe (cf. section 2.1.1) est offert pour manipuler les données généralisées alphanumériques. Deux extensions au langage SQL ont été proposées pour retrouver des objets complexes : les curseurs complexes et les jointures explicites. La première permet à une application de pouvoir retrouver tous les n-uplets de l'objet complexe (qui généralement se retrouvent dans plusieurs relations) en une seule requête. Les liaisons entre les n-uplets composants de l'objet sont representées dans le programme d'application par des pointeurs

(ceux-ci sont en fait des entrées dans un "directory" qui contient les adresses des n-uplets dans les tampons-mémoires fournis par l'énoncé FETCH). Les jointures implicites permettent de supprimer les prédicats de jointure des relations contenant des n-uplets d'un même objet complexe dans une requête SQL. Une jointure implicite entre deux relations est une jointure naturelle entre l'attribut du type IDENTIFICATEUR de la relation mère et l'attribut du type COMPONENT-OF de la relation fille.

En ce qui concerne les mises à jour, le système implante des contraintes d'intégrité sémantique sur les objets complexes. Par exemple, on ne peut pas insérer un composant d'un objet complexe sans père ; on supprime tous les composants de l'objet en une seule commande de suppression.

Les données alphanumériques généralisées sont prise en compte en TIGRE par le concept d'agrégation d'entités. On peut retrouver en LAMBDA tous les composants d'un fait "agrégé" en une seule requête sans avoir besoin de désigner explicitement les composants de l'agrégation (cf. exemple 11, section 4.3.1). Cependant, si on veut retrouver certains composants uniquement, on peut désigner leurs entités respectives par l'intermédiaire des pas par agrégation (cf. 4.2.1). Ces pas sont donc l'équivalent des jointures implicites sur des objets complexes en system-R et des itérations sur les éléments d'un ensemble en SMH$^+$. Concernant les mises à jour sur les agrégations d'entités, LAMBDA offre une syntaxe qui facilite l'écriture des insertions de composants à un fait agrégé (cf. section 5.2.1.2). La suppression d'un fait agrégé est propagée automatiquement aux composants comme en system-R. Finalement les modifications à un fait "agrégé" peuvent être facilement exprimées en utilisant les options ADD ou REMOVE de la commande replace (cf. section 5.4.2).

6.3.2.2 - Manipulation de données multi-média

Considérons maintenant les langages manipulant des données multi-média, qu'ils soient stucturés ou non. Regardons d'abord le type de manipulation sur chaque média séparément.

Recherche

Un média des plus courants est le texte. Le type de recherche auquel on s'intéresse généralement est la recherche des mots dans le texte ou recherche par contenu. Depuis longtemps, les chercheurs en Recherche Documentaire (RD) se sont intéressés à ce genre de recherche. Les systèmes de RD [Sal 83], [Rij 79] offrent généralement des fonctionnalités sophistiquées pour retrouver des documents par leurs contenus qui vont au-delà des requêtes "booléenes" (dans lesquelles chaque prédicat spécifie un mot à retrouver dans le texte). Par exemple, des requêtes de voisinnage entre deux mots ou plus ayant des conditions portant sur la distance entre des occurences des mots peuvent être posées ; on peut spécifier aussi des requêtes comportant n mots et où les documents seront retrouvés s'ils contiennent m de ces mots ou plus (où $1 \leq m \leq n$). On peut également spécifier partiellement un mot de façon à pouvoir retrouver des documents contenant des occurences du mot ayant éventuellement des suffixes ou des préfixes différents. Certains systèmes de RD essayent même de "comprendre" le texte [Mc_G 81], [Haf 81], soit conceptuellement, soit "statistiquement", de façon à construire un thésaurus dans lequel des relations de synonimie entre des mots sont exprimées [Bru 81]. Les documents sont généralement retournés en ordre de relevance.

Dans les systèmes de gestion de données qui stockent des textes, des requêtes "booléenes" peuvent être posées. Par exemple, dans le système multi-média développé à Toronto (cf. Section 2.1.3) et dans TQL (cf. section 2.1.5). Dans les extensions au SGBD INGRES orientées vers le traitement de documents, (cf section 2.1.2), on va plus loin : on peut spécifier partiellement des mots ou des chaînes de caractères à l'aide de caractères spéciaux qui peuvent correspondre à n'importe quel caractère ou n'importe quelle chaîne de caractères ("don't care characters").

Dans la mesure où ces sytèmes permettent de stocker aussi des données alphanumériques (BIG, INGRES, System-R), on peut évidemment associer des attributs alphanumériques (Entier, réel, chaîne de caractère) aux textes et les retrouver par la valeur de leurs attributs alphanumé -

riques. Bien entendu, ceci est applicable aussi aux autres médias : images, graphiques, voix digitalisée.

LAMBDA permet de retrouver des textes aussi bien par leur contenu que par attributs. En ce qui concerne la recherche par contenu, on peut spécifier complètement ou partiellement des chaînes de caractères (cf. section 4.3.6) ; comme dans les extensions à INGRES, le caractère '*' correspond à n'importe quelle chaîne de caractères. Les paramètres (et les fonctions) d'un document font partie du contenu d'un document ; cependant, on leur donne un nom et un type et donc la recherche d'un document par la valeur d'un paramètre est plus une recherche par attributs qu'une recherche par contenu. Par ailleurs, les documents étant la valeur d'un attribut d'un fait, on peut les retrouver en retrouvant le fait auquel ils appartiennent. Celui-ci peut être retrouvé par la valeur des autres attributs alphanumériques ou par des liaisons sémantiques (associations, agrégations) avec d'autres faits de la base.

A OLIVETTI, on a développé un système de recherche documentaire adapté à la bureautique, appelé OTTER [Sac 84]. Ce système se situe en quelque sorte à mi-chemin entre les systèmes classiques de RD et les systèmes de gestion de données manipulant des textes. En effet, ce système permet de combiner la recherche par contenu et la recherche par attribut et il offre un mécanisme de définition (manuelle) des "abstractions conceptuelles" (par exemple, matériel électronique est une abstraction conceptuelle de "diode" et "transistor"). Les documents sont retournés en ordre de relevance. La recherche par attributs ressemble fortement à notre concept de paramètre d'un document : on peut déclarer une portion de texte comme ayant une sémantique spéciale donnée par le nom de l'attribut et une structure spéciale donnée par son type.

En ce qui concerne les autres médias, la recherche par contenu est bien plus difficile. En effet, il n'est pas évident de définir le critère de recherche ("pattern") et les techniques de reconnaissances sont généralement compliquées et se heurtent à des problèmes de performances. A cause de celà, on essaie, dans la mesure du possible,

de transformer les recherches par contenu sur les images et graphiques en un problème de recherche par attributs ou en un problème de recherche par contenu sur le texte. En effet, les objets intervenant dans une image ou un graphique ainsi que leur position sont représentés avec des structures de données alphanumériques classiques ; la description d'une image est une description textuelle. Ce sont cette structure et ces textes qui sont utilisés dans les recherches. Cette approche a été prise dans le système multi-média développé à Toronto (cf section 2.1.3).

Création - mise à jour

Pour créer et mettre à jour des données multi-média, il est nécessaire de disposer d'éditeurs spécialisés. Il est donc nécéssaire que l'éditeur puisse communiquer avec le système de gestion de données multi-média pour poser des requêtes, extraire les résultats et les ranger à nouveau dans la base après modification. Cette approche "interactive" est prise dans le système multi-média développé à Toronto et dans BIG.

Dans TIGRE, il est envisagé d'offrir cette possibilité ainsi que la possibilité de pouvoir programmer des appels à l'éditeur depuis LAMBDA (cf. section 5.1.2) dans le but de pouvoir écrire des application manipulant des données multi-média. Ceci ne va pas sans problèmes de mise en oeuvre concernant la gestion de transactions de longue durée (cf. section 8).

Structure

Les données multi-média peuvent être structurées logiquement. Cette structure est généralement arborescente [Ecma], [Hor 84], [BRV 83]. En effet, les données structurées multi-média sont généralement conçues comme des documents multi-média ou généralisés. TQL est un langage de requêtes basé sur un modèle hiérarchique de structure de textes (cf section 2.1.5). La structure d'un texte est décrite par une grammaire. La notion de base de TQL est le curseur, une variable qui désigne un noeud de la structure d'un texte (un élément non terminal de la

grammaire). On peut retrouver des noeuds désignés par des curseurs ou des fils de ceux-ci, et on peut spécifier des conditions sur des composants ou la comparaison de deux composants ("jointure" entre deux structures de texte).

LAMBDA permet de désigner des noeuds par l'intermédiaire du concept de chemin dans la structure d'un document. Si l'on veut retrouver un noeud n en fonction des conditions qu'on impose sur les noeuds qui appartiennent au sous-arbre S dont n est racine, on peut désigner n par une variable de désignation (l'équivalent du concept de curseur de TQL). Si aucune condition n'est imposée sur S, il n'est pas nécessaire d'introduire une variable, contrairement à TQL. Le pouvoir d'expression de TQL et de LAMBDA nous semble équivalent bien qu'il n'existe pas à notre connaissance une notion de complétude permettant de comparer les deux langages.

CHAPITRE 7

LE SERVEUR TIGRE

SOMMAIRE

7. LE SERVEUR TIGRE

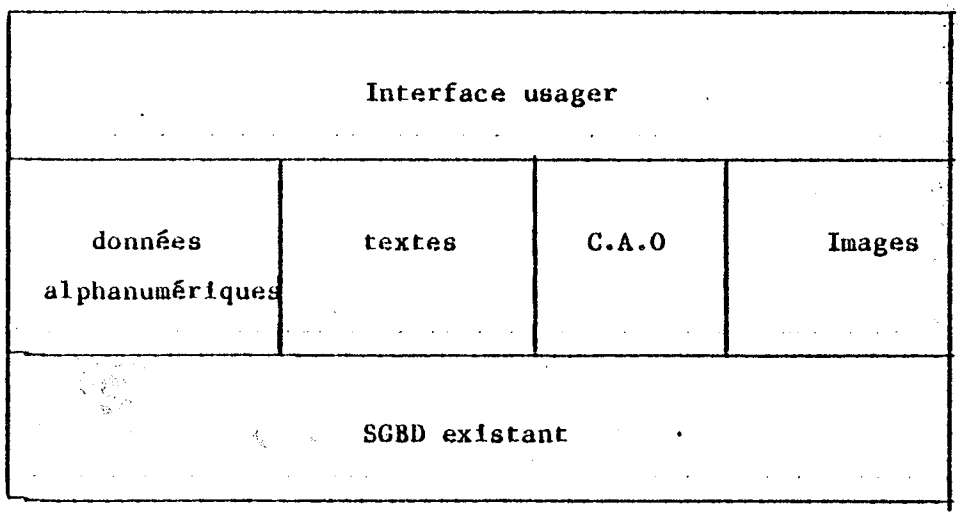
Ce chapitre sera consacré à la description de l'architecture du serveur de bases de données TIGRE. Nous nous concentrerons plus particulièrement sur le cadre de réalisation des énoncés d'interrogation de LAMBDA.

7.1 - ARCHITECTURE GÉNÉRALE

Les données généralisées, contrairement aux données alphanumériques, ne sont pas atomiques et possèdent une structure interne qui leur est propre. Elles ont, en général, une taille importante, ce qui pose de nouveaux problèmes de stockage et de manipulation.

Depuis quelques temps, l'intégration de ce type de données à un SGBD suscite beaucoup d'intérêt de la part des chercheurs. Dans [LS 83] quatre solutions possibles ont été proposées :

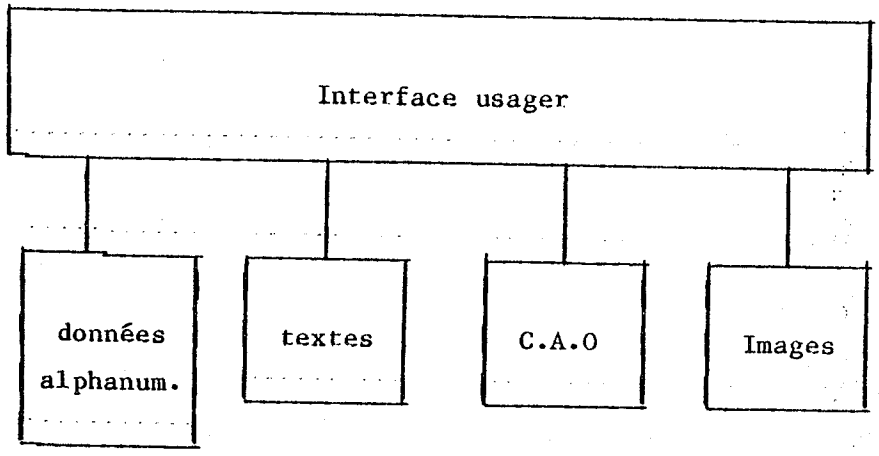
1 - Extension d'un SGBD existant au moyen de niveaux supplémentaires construits sur les interfaces déjà existantes pour supporter différentes applications



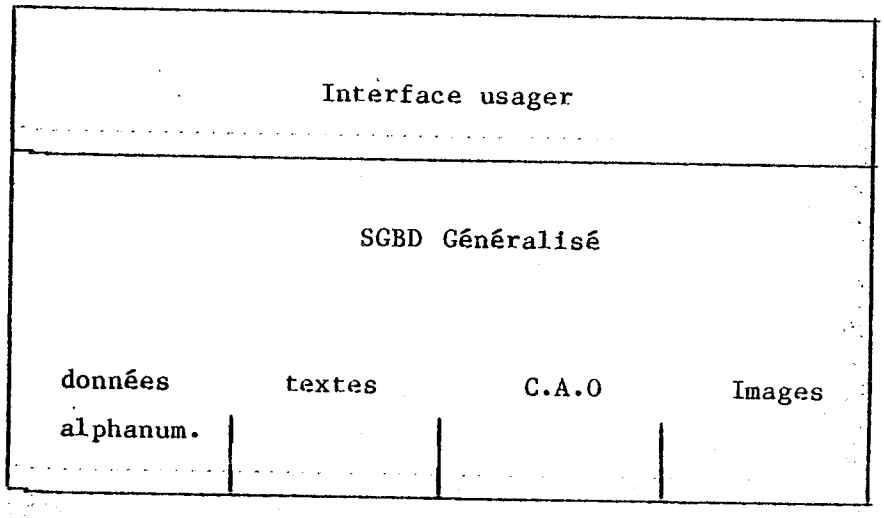
Les extensions à System-R visant à manipuler des données CAO ainsi que

celles proposées pour INGRES pour manipuler des textes (cf section 2.1.1 et 2.1.2) rentrent dans cette catégorie. Le système construit dans le projet BIG aussi, car il est construit sur le SGBD DMS II.

2 - Combinaison d'un SGBD avec des systèmes indépendants manipulant chaque type de données généralisées.



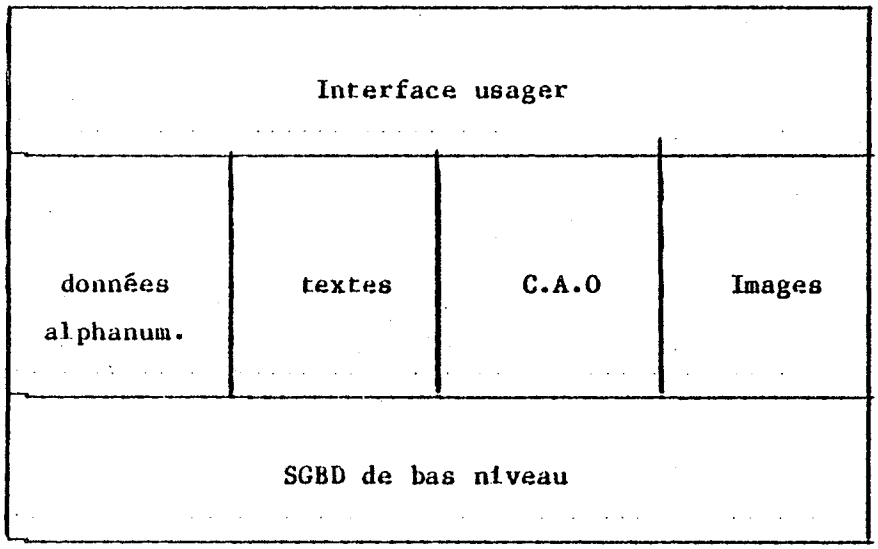
3 - Développement d'un nouveau SGBD étendu, capable de manipuler toutes sortes de données généralisées.



Cette solution est la plus intégrée mais elle semble la plus difficilement réalisable. Le système multi-média développé à Toronto suit dans une certaine mesure cette approche (cf 2.1.3). Il manipule des textes, des images et de la voix et il offre une interface usager pour

retrouver ce type de données. Cependant, l'intégration de ces données avec des données alphanumériques n'est pas adresée, ni d'autres fonctionnalités comme la reprise et la concurrence.

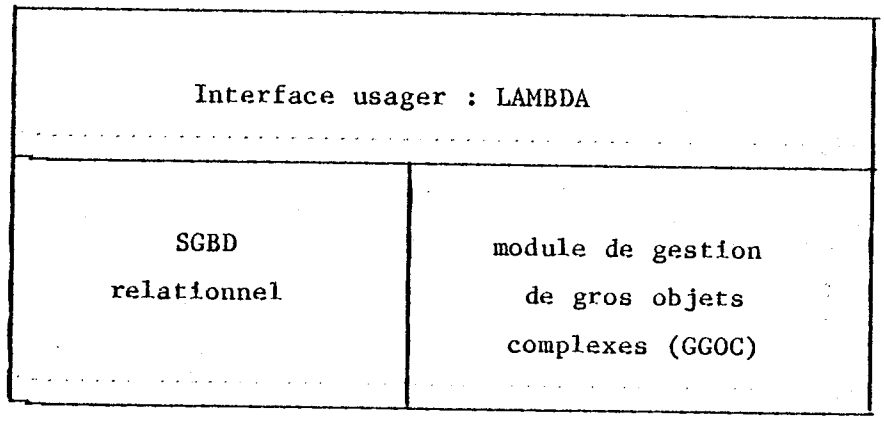
4 - Développement d'un SGBD de bas niveau avec des primitives génrérales et des couches spécifiques pour chaque application.



Le niveau des primitives et les fonctions du SGBD de bas niveau n'est pas très clair à l'heure actuelle.

Il semblerait donc que les solutions les plus viables à l'heure actuelle sont les solutions 1. et 2.

Pour la réalisation du serveur TIGRE nous avons pris l'approche 2. En effet, il est construit sur un SGBD relationnel et sur des modules indépendants pour la gestion des documents généralisés (appelés modules de gros objets complexes). Nous pourrons ainsi limiter l'effort de développement à l'extension sémantique du modèle relationnel qui est le modèle TIGRE et à l'introduction de nouveaux types de données sans avoir besoin de modifier le SGBD déjà existant. La gestion indépendante des données multi-média apparaissant dans un document généralisé permet de ne pas préjuger des capacités particulières des SGBD pour stocker des données volumineuses. la structure du prototype est la suivante :



LAMBDA est donc développé sur deux interfaces inférieures. Une interface de haut niveau pour les données alphanumériques et une interface au niveau méthode d'accès et stockage pour les documents généralisés. L'interface relationnelle de haut niveau se compose de primitives de l'algèbre relationnelle. Les données alphanumériques à structure complexe seront donc manipulées dans le SGBD relationnel sousjacent.

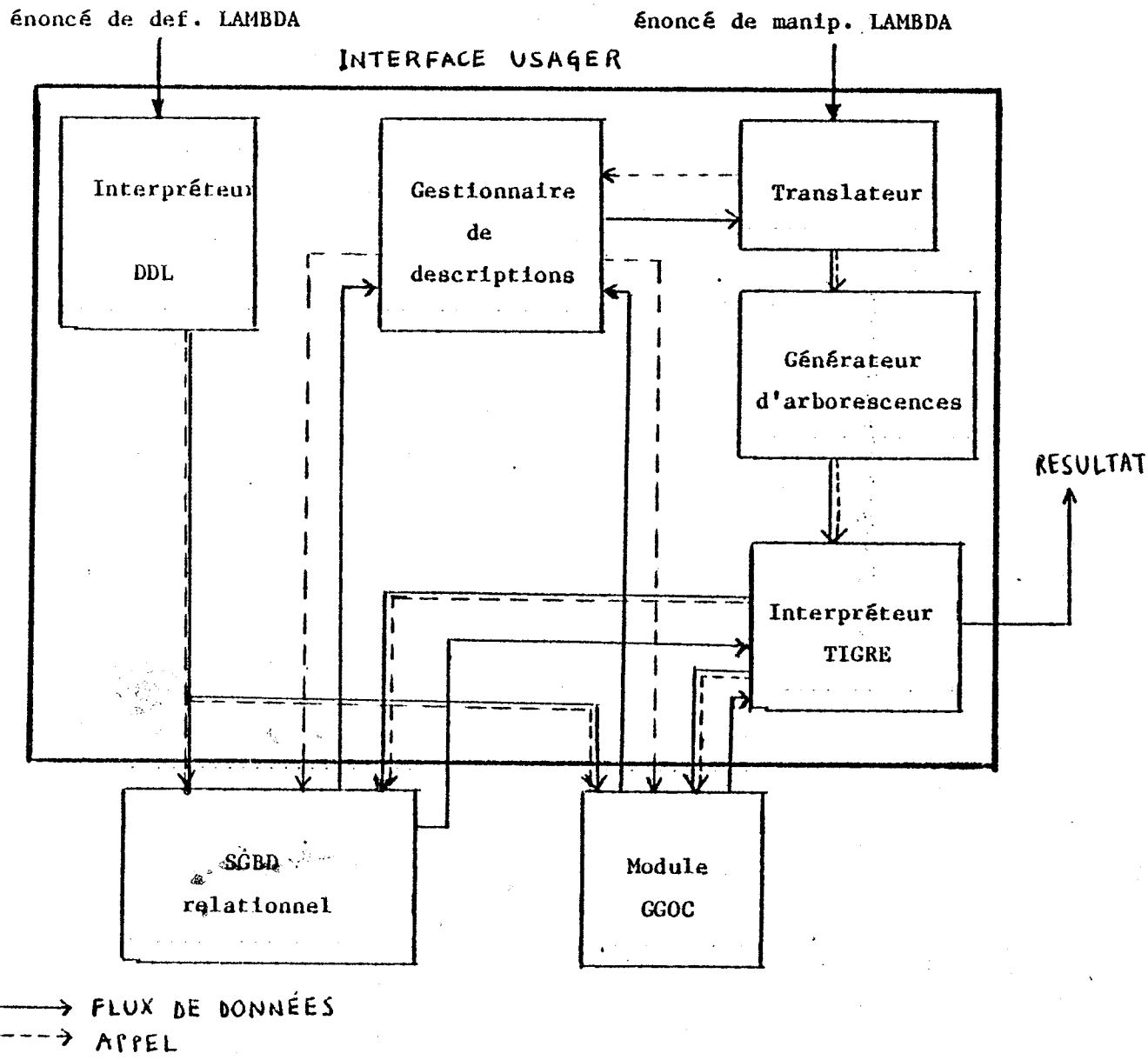
Dans la réalisation actuelle du prototype TIGRE, nous utilisons le système relationnel de base MICROBE [Mic 82] développé au laboratoire IMAG. Nous essayons cependant d'être le plus indépendants possible du système relationnel de base.

La correspondance de schémas entre les modèles TIGRE et relationnel a été établie [PV 83], [Pal 84]. Un schéma conceptuel TIGRE et sa correspondance avec un schéma conceptuel sont stockés dans un ensemble de relations (la métabase) appelé le catalogue TIGRE. L'interprétation d'une définition de schéma TIGRE insère des n-uplets dans le catalogue TIGRE et génère des relations vides correspondantes au schéma [Pal 84]. Les définitions des types documents sont stockées comme des listes parenthèsées dans le module GGOC.

Les énoncés de manipulation de LAMBDA sont traduits en termes d'opérateurs de l'algèbre relationnelle et d'autres opérateurs concernant les documents généralisés. Le traducteur est divisé en deux modules principaux, le "translateur" qui accomplit l'analyse lexicale, syntaxique et sémantique et le générateur d'arborescences qui effectue la traduction proprement dite à partir d'une représentation intermédiaire de l'énoncé construite par le translateur. L'analyseur sémanti-

que appelle un module spécial, le gestionnaire de descriptions, qui se charge d'interroger le catalogue TIGRE pour obtenir la description d'un sous-ensemble du schéma TIGRE. Cette description est mise sous une forme adéquate en mémoire centrale.

Le résultat de la traduction d'une requête se compose d'un ou plusieurs arbres constitués d'opérateurs algébriques et non algébriques. Ces arbres seront interprétés à l'execution par un module, appelé l'interpréteur TIGRE. Les sous-arborescences relationnelles seront envoyées au SGBD relationnel et celles concernant les documents au module GGOC. Les résultats de ces deux systèmes sont recombinés dans l'interpréteur et l'interprétation de l'arbre continue. L'architecture actuelle du serveur est la suivante :



L'opérateur edit est interprété en appelant l'éditeur par programme et en envoyant les paramètres actuels (les documents).

Une présentation de l'architecture du serveur basée sur la description des interfaces de la proposition d'architecture ANSI/X3/SPARC [TK 78] est présentée dans [Pal 84].

Dans la section 7.2, nous détaillerons la correspondance des schémas entre les modèles TIGRE et relationnel, ainsi que la structure du catalogue TIGRE. La section 7.3 présente les règles de génération d'arborescences pour les requêtes LAMBDA à partir d'une structure intermédiaire construite par le translateur. On introduit également les opérateurs concernant les documents. La section 7.4 décrit brièvement les fonctionnalités offertes par le module GGOC ; finalement la section 7.5 présente les deux types d'interfaces usager au serveur TIGRE actuellement en cours de réalisation : l'interface procédurale programmable PASCAL-LAMBDA et l'interface graphique conversationnelle orientée usager final construite au dessus de LAMBDA.

7.2. LA CORRESPONDANCE DE SCHÉMAS ENTRE LES MODÈLES TIGRE ET RELATIONNEL

Dans cette section nous décrivons très brièvement le processus de compilation du schéma conceptuel pour sa transformation en format objet. La définition du format objet du schéma conceptuel est basée sur des concepts proposés par Codd dans RM/T [Cod 79], [Dat 83]. Cette proposition est une extension du modèle relationnel originel [Cod 70]. L'utilisation de ces concepts nous est apparue comme un outil commode pour effectuer la correspondance entre schémas. En effet, RM/T se situe "à mi-chemin" entre TIGRE et le Relationnel puisqu'il est une extension du modèle relationnel qui va dans le sens de TIGRE. La structure du catalogue que nous définissons est une extension de la proposition initiale de RM/T. Nous y avons inclus plusieurs nouvelles relations pour pouvoir décrire sous forme relationnelle les types définis dans un schéma conceptuel TIGRE. Il est décrit dans l'annexe 4. La correspondance précise des concepts de RM/T et ceux de TIGRE est décrite dans [PV 83].

7.2.1 - Surrogate, Relations-E et Relation-P de RM/T.

RM/T remplace la notion de clé primaire définie par l'usager par la notion de "surrogate". Un surrogate est un identificateur unique généré par le système. Toute référence à une entité est faite par un surrogate ; les clés des usagers sont considérées comme des propriétés des entités. Le domaine-E (dom-E) est le domaine de toutes les valeurs possibles de surrogates. Par convention, un attribut défini sur dom-E a pour suffixe le caractère "c".

Une base RM/T contient une relation-E par type d'entité. Cette relation est une relation unaire qui contient tous les surrogates des entités appartenant à ce type et existant actuellement dans la base. Le but des relations-E est d'indiquer l'existance des entités et de servir comme point central de référence pour toutes les entrées dans la base de données concernant des entités appartenant au type. Par convention, le nom de la relation-E est le même que le nom du type d'entités qu'elle représente et le seul attribut de la relation-E a pour nom, le nom de la relation suivi par le caractère "c".

Les propriétés monovaluées d'une entité sont représentées par des attributs d'une ou plusieurs relations-P. Chaque relation-P a comme clé primaire un attribut défini sur le domaine-E qui lie les propriétés de chaque entité avec l'assertion de son existence définie par la relation-E correspondante. La manière précise dont les propriétés d'un type d'entités sont groupées en relations-P est sous la responsabilité du concepteur du schéma interne. Une solution extrême consiste à grouper toutes les propriétés dans une seule relation-P et une autre solution consiste à avoir une relation-P binaire pour chaque propriété.

7.2.2. Représentation relationnelle d'un schéma TIGRE

La représentation relationnelle d'une base de données TIGRE contient une relation-E par classe. Il y a aussi une relation-E par type construit Enregistrement, Tableau et Document. La relation-E sert à indiquer l'existence des occurrences. Toutes les valeurs des attributs des occurrences sont maintenues par une relation-P dans le prototype

actuel (il serait souhaitable que l'on puisse, dans une version ultérieure, disposer d'un langage de définition de schéma interne par l'intermédiaire duquel l'administrateur de la base pourra "segmenter" une classe en plusieurs relations-P s'il le désire, en fonction de critères de performance dans une application particulière). Pour les attributs définis sur des types construits, la relation-P contient un surrogate qui fait référence à la relation-E qui correspond au type construit. La convention de nommination des relations et des attributs sont les mêmes que dans RM/T. Dans le serveur, toutes les références internes sont faites via des surrogates. Les objets du catalogue identifiés par des surrogates sont: base de données, domaines, attributs, listes parenthèsées pour la description des types document, relations et prédicats. Les derniers correspondents aux prédicats de restriction que l'on peut spécifier dans la définition des entités dérivées par généralisation en TIGRE).

D'un point de vue conceptuel, les types, classes et faits des classes ainsi que les types construits et leurs occurrences sont identifiés dans la base par des surrogates.

7.2.2.1. Types simples et construits.

Un type simple de base est complètement décrit par un n-uplet dans la relation du catalogue CAT-D qui représente les domaines. Pour les types restreints (les scalaires et les intervalles) il faut des relations supplémentaires pour maintenir des valeurs possibles des domaines. Ces relations sont CAT-INTD pour les intervalles CAT-SCAD pour les scalaires. Si l'on veut ajouter des restrictions sur la longueur d'un type chaîne de caractères, on utilise la relation du catalogue CAT-STRING.

Un type renommé est également décrit par un seul n-uplet dans CAT-D. La représentation d'un type Enregistrement et d'un type Tableau correspondent à la création d'une relation-E et une relation-P par type dans la base, comme le montre le tableau suivant :

REPRESENTATION RELATIONNELLE

Type construit	Relation-E	Relation-P
Type R = Record | c_1 : T_1 | . | . | . | c_n = T_n end;	R (R-c : dom-E)	R-p (R-c :dom-E, c_1 : T_1,...,c_n : T_n)
Type T : Array (n) of T';	T (T-c : dom-E)	T-p (T-c : dom-E, ordre : entier, valeur T')

Dans le catalogue, les relations sont représentées par des n-uplets de CAT-R et les attributs par des n-uplets de CAT-A. Pour lier les relations-E et P, nous avons défini la relation CAT-COMP qui maintient un n-uplet par relation-P. Pour lier le n-uplets de CAT-D correspondant à un enregistrement ou un tableau à la relation le décrivant, il existe la relation du catalogue CAT-STRUC (cette relation est équivalente à la relation "CG-relation" du catalogue RM/T [cod 79]. La relation CAT-LIST maintient la cardinalité maximale du type tableau T.

La représentation des types document est assurée par une liste parenthèsée qui correspond à la représentation employée pour l'échange de documents entre l'éditeur et la base de données. Cette liste parenthèsée est stockée dans le module GGOC. Son surrogate est lié au nom du type document dans la relation CAT-DOC. Par exemple, la liste parenthèsée correspondant au type "contrat-travail" défini dans la section 3.2.2 est :

```
(Contrat-Travail DEFTYPE
(STRUC (AGR ((Contractants AGR ((Employeur Texte VAL (TO))
                                (Employé ref-par VAL (Titulaire)))
             (Corps LISTE 1 * STRUC (clause))
             (Signature   image))))

(CONST((texte TO  'compagnie ABC')
       (paramètre  titulaire texte)
       (paramètre  date embauche texte)
       (paramètre  établissement texte)
       (paramètre  niveau texte)
       (fonction  salaire embauche SALAIRE (niveau,établissement)))))
```

7.2.2.2. Types classe

Comme nous l'avons déjà mentionné, les entités (types-entités) seront représentées par une relation-E et une relation-P. Les attributs de cette dernière sont les mêmes que ceux de l'entité. En ce qui concerne les associations, on utilisera aussi une relation-E et une relation-P pour représenter ses propriétés ainsi qu'une relation, dite de type A, qui sert à stocker les surrogates des faits liés dans l'association.

La représentation relationnelle de l'association est donc explicite. Il faut remarquer que dans le cas où la participation maximale d'une entité E à une association A est 1, on peut représenter l'association implicitement en incluant des clés externes ("foreign key") dans la relation-P: celles des autres entités participant à l'association. Nous avons cependant voulu simplifier la correspondance de schémas en représentant explicitement les associations.

Dans la relation-A, les noms des constituants correspondant aux entités n'ont pas toujours le même nom que les attributs-E des entités. En effet, étant donné que nous admettons des associations entre des faits de la même entité (par exemple, l'association MARIAGE entre des PERSONNES), nous devons dans ce cas distinguer les entités via leurs rôles.

Types classe	Représentation relationnelle
Type E : Entity key a_1 : T_1; . . a_k : T_k end key; a_{k+1} : T_{k+1}; . . a_n : T_n; end;	E (E-c : dom-E) Relation-E E-p(E-c : dom-E, a_1:T_1,...,a_n:T_n) Relation-P
Type A : Relationship between E_1 : rôle(E_1,A)(min,max) . . and E_n : rôle(E_n,A)(min,max); a'_1 : T'_1; . . a'_m : T'_m end ;	A(A-c : dom-E) Relation-E A-p(A-d : dom-E,a'_1:T'_1,..,a'_m:T'_1) Relation-P A-desig(A-c:dom-E, E_1-c:dom-E,..., E_n-c:dom-E) Relation-A

Dans la relation CAT-A qui décrit les attributs, il y a un champ qui permet de savoir si l'attribut fait partie de la clé spécifiée par l'usager. D'autre part, pour maintenir l'information des entités participant à l'association, ses rôles et les cardinalités, nous employons la relation du catalogue appelée CAT-DESIG.

7.2.2.3. Entités dérivées par agrégation

Pour représenter une entité E dérivée par agrégation d'entités, il existe dans la base, en plus des relations E et P, une relation dite de type G, qui a pour but de maintenir les valeurs des surrogates des faits des entités "agrégées" qui appartiennent à un fait de E. La relation du catalogue CAT-EAGG est utilisée pour stocker la composition de E : une réference aux entités agrégées et les contraintes de cardinalité.

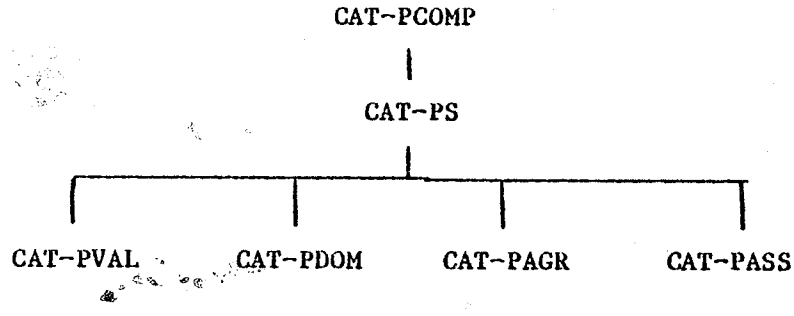
En ce qui concerne les entités E' dérivées par agrégation associative, étant donné qu'il y a une correspondance 1 : 1 entre les faits de l'association et les faits de E', la représentation de E' est accomplie en redéfinissant la relation-E correspondante à l'association sous-jacente par une relation-E "virtuelle". Tout se passe comme si la relation-E correspondante à l'association avait un deuxième nom -"alias"-, celui de E'. Pour stocker la valeur des attributs propres à E', on utilise une relation-P. La définition de E' est représentée par un n-uplet dans CAT-AAGG.

Types dérivés par agrégation	Représentation relationnelle
Type E : Entity-agrégation of E1(min,max) and ... and En(min,max) a1 : T1; . . an : Tn; end ;	E(E-c : dom-E) Relation-E E-p(E-c : dom-E,a1:T1,..,an:Tn) Relation-P E-agg(E-c:dom-E,E1-c:dom-E,... ..,En-c:dom-E) Relation-G
Type E': Relationship-agregation of A a'1 : T'1; . a'm : T'm; end ;	E'-p(E'-c:dom-E,a'1:T'1,..,a'n:T'n) Relation-P Pas de création explicite de Relation-E : elle est virtuelle et définie sur la relation-E de A.

7.2.2.4. Types dérivés par généralisation

Nous avons décidé de représenter la famille d'entités (ou d'associations) correspondant à une hiérarchie de généralisation par des relations qui correspondent à un partitionnement vertical de la famille. Autrement dit, les attributs propres de chaque entité dérivée sont stockés dans une relation-P. Ceci permet de réduire l'espace mémoire utilisé pour représenter une hiérarchie de généralisation. L'inconvénient est qu'il faut effectuer des jointures pour executer des requêtes adressées aux classes dérivées quant on veut retrouver des attributs hérités par ces classes (les attributs propres aux ancêtres). Cependant, si la fréquence d'utilisation de l'entité non dérivée (la racine de la hiérarchie) est elevée, la décomposition verticale est avantageuse.D'autres approches sont proposées dans [CAFL 82] où il existe plusieurs options incluant le partitionnement vertical et horizontal ainsi que le regroupement physique de fragments. Le contrôle direct de pointeurs est utilisé pour améliorer les performances - ce que nous ne pouvons pas faire, car TIGRE est construit sur un système relationnel. Une autre approche est prise dans l'implémentation de GEM [TZ 84] où l'on utilise une seule relation interne pour représenter une hiérarchie de généralisation.

Pour représenter la généralisation, nous employons la relation du catalogue CAT-GEN. Cette relation spécifie la classe résultat, la classe opérande, l'opérateur utilisé (spécialisation, union ou intersection) et le prédicat de restriction, s'il existe. Ces derniers sont décrits sous forme normale disjonctive. Dans le catalogue, la représentation est accomplie en utilisant la famille de relations suivante (voir Annexe 4) :



La relation CAT-PCOMP stocke les prédicats composés décrits sous forme normale disjonctive. Ils sont composés de prédicats simples, eux mêmes stockés dans CAT-PS. Les relations feuilles CAT-PVAL, ..., CAT-PASS contiennent les informations particulières à chaque type de raffinement (par valeur, par domaine, par agrégation, par association) induit par un prédicat simple.

7.2.2.5. Génération de la base

Après la génération de la représentation du schéma conceptuel d'une base dans le catalogue TIGRE, il faut initialiser la base. Une procédure spécifique effectue la lecture des relations du catalogue TIGRE et produit les appels nécessaires pour la création des relations. A la fin de ce processus, une base relationnelle vide correspondante au schéma TIGRE est disponible [Pal 84].

7.3 - LA TRADUCTION DE REQUETES

Dans cette section, nous donnons un aperçu des règles de traduction qui produisent des arborescences TIGRE à partir d'une représentation intermédiaire d'un énoncé d'interrogation LAMBDA. Cette représentation intermédiaire, produite par le translateur sera d'abord introduite à l'aide des exemples, puis définie plus formellement. Enfin, nous présenterons les opérateurs agissant sur les documents généralisés, appelés opérateurs document et nous démontrerons leur utilité à l'aide des exemples.

7.3.1 - L'analyse sémantique

Le translateur se compose des trois phases classiques dans le traitement des langages : analyse lexique, syntaxique et sémantique. Cette dernière phase vérifie d'abord que la requête est sémantiquement cohérente. la cohérence est vérifiée au niveau trajets de désignation (pour voir si l'enchaînement de pas traduit vraiment un enchaînement de classes sémantiquement associées dans le schéma conceptuel TIGRE), au niveau expression de valeurs (pour voir si les attributs notés appartiennent chacun à la classe où ils sont sensés appartenir et pour

vérifier la cohérence des chemins dans la structure d'un document) et au niveau clause-where (pour vérifier que les valeurs et les ensembles comparés ont un type compatible). Pour ce faire, il effectue des appels au gestionnaire de descriptions qui se charge d'interroger le catalogue TIGRE. Il appele aussi le module GGOC pour extraire la définition des types document utilisés dans la requête. Les primitives du catalogue de descriptions sont complètement spécifiées.

Après les vérifications sémantiques, la requête est mise sous une forme normalisée. Premièrement, le connecteur booléen not disparaît, c'est à dire les conditions not C apparaîssant dans la clause-where seront remplacées par une condition C' équivalente ne contenant pas de connecteur booléen not. Au niveau des conditions élémentaires, le remplacement est donné par :

a. not (⟨valeur⟩ Θ ⟨valeur⟩) $\longrightarrow$ ⟨valeur⟩ Θ' ⟨valeur⟩

b. not (⟨valeur⟩ in ⟨ensemble⟩) $\longrightarrow$ ⟨valeur⟩ in (⟨ens-domaine-de-valeur⟩ -⟨ensemble⟩)

c. not (⟨ensemble⟩ Ψ ⟨ensemble⟩) $\longrightarrow$ ⟨ensemble⟩ Ψ' ⟨ensemble⟩

Le seul cas où not ne disparait pas est dans les conditions b. quand ⟨ensemble⟩ est un ensemble de textes d'un document. Ici, Θ' est l'opérateur contraire à Θ , Ψ' est l'opérateur contraire à Ψ et ⟨ensemble-domaine-de-valeur⟩ est un nouvel ensemble contenant toutes les valeurs possibles que ⟨valeur⟩ peut prendre, si ⟨valeur⟩ n'est pas une constante. L'introduction de cet ensemble passe par l'introduction d'un (ou plusieurs) nouveau(x) trajet(s) de la forme E_i x_i où x_i sera générée par le système (et différente à toutes les variables apparaîssant dans les autres trajets de la requête). Si ⟨valeur⟩ est une constante c, ⟨ens-domaine-de-valeur⟩ est $\{c\}$.

Deuxièmement, les conditions de la requête seront mises sous une forme très proche de la forme normale disjonctive. En effet, les conditions de la forme c_1 or c_2 où :

$$c_i = \begin{cases} e_i.a_j \ \Theta \ c \quad \text{ou} \\ \\ e_i.a_j \ \Theta \ e_i.a_k \end{cases} \qquad \text{pour } i = 1,2$$

qui portent sur la même entité désignée par e_i, sont considérées comme atomiques, c'est à dire, elles ne seront pas décomposées dans la formation de la forme normale disjonctive.

Troisièmement, les entités E notées comme des ensembles dans les conditions de la clause-where (par exemple, cf. section 4.4.1, exemple 22) seront remplacées par l'ensemble $\{x_i\}$ et le (nouveau) trajet E x_i sera introduit. La variable x_i est une nouvelle variable générée par le système.

7.3.2 - La représentation intermédiaire

Après ces trois transformations, la requête est normalisée et on procède à la génération de la structure intermédiaire. Pour chaque variable de désignation, on détermine la portée (globale, locale à un ensemble, locale à un ensemble d'un ensemble ...) et on associe les variables de même portée d'un trajet dans des arbres où les noeuds sont des entités ou des associations. La structure de l'arbre reflète la composition du trajet et donc, les conditions implicites sur les variables. A chaque variable, dans chaque groupe de conditions de la forme normale disjonctive, on associe les attributs intervenant dans l'expression de valeurs (attributs à projeter), les conditions de sélection (de la forme $f(\ell_i)\Theta C$), les conditions de jointure explicite (de la forme $e_i.a_k \ \Theta \ e_j.a_l$), les conditions d'appartenance à un ensemble et les conditions sur les ensembles indexés par la variable. Pour la variable e_i, nous dénoterons ces ensembles respectivement par proj(e_i), cond-sel(e_i), cond-sel-join(e_i), cond-appart(e_i) et cond-ensindx(e_i).

Nous introduisons d'abord la représentation intermédiaire des requêtes ne manipulant pas des documents.

Exemple 1 : retrouver les départements d tels que tous les fournisseurs qui habitent Londres stockent des articles demandés par d ou tels qu'ils n'appartiennent pas à l'ensemble de départements demandant des articles rouges.

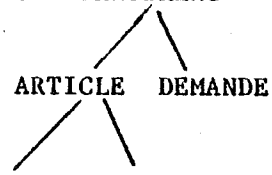
Select d_1
from fournisseur-stockant f_1 of article-demandé of DEPARTEMENTd,
FOURNISSEURS f_2, dep-demandant d_2 of ARTICLE a_1
where $\{f_1\}$ by $d_1 \geq \{f_2$ where f2.adresse.ville = Londres$\}$
or (not (d_1 in $\{d_2\}$ by a_1) and a_1.couleur = rouge

La condition "not(d_1 in $\{d_2\}$ by g_2)" est changée par "d_1 in $\{d_3\}$-$\{d_2\}$ by a_2)" et le nouveau trajet "DEPARTEMENT d_3" est crée. la requête est maintenant normalisée. La représentation intermédiaire est la suivante :

OR_1

T_1 : d_1 : DEPARTEMENT
proj(d_1) = {*} (tous les attributs)
cond-ensindx(d_1) = {ens1(d_1) $\geq$ ens2}

.ens1(d_1) : d_1 : DEPARTEMENT



proj(f_1) = {FOURNISSEUR-c}
proj(d_1) = {DEPARTEMENT-c}

.ens2 : f_2 : FOURNISSEUR

ad_1 : ADRESSE-p

cond-sel(ad_1) = {ville='Londres'}
proj(f_2) = {FOURNISSEUR-c}

OR_2

T_1 : d_1 : DEPARTEMENT
proj(d_1) = {*}
cond-appart(d_1) = {d_1 in ens3(a_1)}

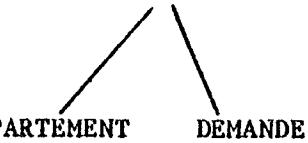
T_2 : a_1 : ARTICLE
cond-sel(a_1) = {couleur = 'rouge'}
cond-ensindx(a_1) = {d_1 in ens3(a_1)}

.ens3(a_1) : ens4 - ens5(a_1)

.ens4 : d_3 : DEPARTEMENT

proj(d_3) = {DEPARTEMENT-c}

.ens5(a_1) : a_1 : ARTICLE



proj(d_2) = {DEPARTEMENT-c}
proj(a_1) = {ARTICLE-c}

Dans le premier groupe de conditions de la forme normale disjonctive, la seule variable globale est d_1. La variable f_1 est locale à l'ensemble ensl(d_1) (ainsi que la variable sur l'article demandé, s'il y en avait eu une). L'arbre définissant l'ensemble reflète en termes d'entités et d'associations la structure du premier trajet. L'ensemble se compose de faits de FOURNISSEUR ; il suffit de retrouver seulement les surrogates. La variable f_2 est locale à ens2. L'accès au constituant "ville" qui se trouve dans la relation ADRESSE-p (représentant l'attribut-enregistrement "adresse") est représentée par une branche supplémentaire dans l'arbre représentant le deuxième trajet (qui se compose seulement d'un noeud. Cette branche se traduira par une jointure. Une nouvelle variable, ad_2, est générée et la condition de sélection est exprimée sur cette variable.

En général, l'accès à des objets du schéma qui sont modélisés par les concepts du modèle TIGRE qui étendent le modèle ER (généralisation, agrégation, enregistrement, tableaux) se traduisent par l'accès à des relations représentant ces objets du schéma. Ces accès sont déjà explicités dans la représentation intermédiaire, comme nous venons de voir avec l'exemple de l'enregistrement "adresse".

Dans le deuxième groupe de conditions, il y a deux arbres T_1 et T_2 correspondants aux variables de portée globale des trajets 1 et 3 respectivement. La variable globale a_1 indexe l'ensemble ens3, construit à partir de la différence d'un ensemble non indexé (ens4) et d'un ensemble indexé par a_1 (ens3).

Jusqu'ici, tous les ensembles indexés sont des ensembles indexés par association ce qui implique qu'il faudra faire des jointures avec les associations utilisées pour définir l'ensemble. C'est pour celà que dans la définition de l'ensemble on retient les surrogates des entités qui sont désignées par les variables indexantes. Ceci est aussi le cas des ensembles définis par agrégation et des ensembles tableau.

Exemple 2 Retrouver les expéditeurs des lettres où Dupont figure comme destinataire.

```
Select l.expéditeur
from LETTRE l where 'Dupont' in l.Liste-destinataires
```

Représentation intermédiaire :

T_1 : l : LETTRE
proj(l) = {expéditeur}
cond-ensindx(l) = {'Dupont' in ensl(l)}

ensl(l) : l : LETTRE
|
dest : DESTINATAIRE

proj(l) = {LETTRE — c}
proj(dest) = {val}

Ici, l'attribut "liste-destinataires" (un attribut tableau défini sur le type "destinataires") est manipulé comme un ensemble. Nous avons vu dans 7.2 la représentation relationnelle des tableaux : ils sont stockés dans une relation du même nom que le type tableau et liés aux classes par l'intermédiaire des surrogates. L'attribut "val" contient les valeurs des éléments du tableau. On ne s'intéresse pas ici à l'attribut "ordre" qui maintient l'ordre des éléments, car le tableau est manipulé ici comme un ensemble.

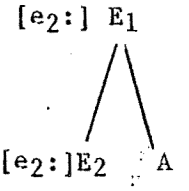
Nous avons vu dans la section 4.2 la structure générale des trajets. Il peut y avoir des pas par association, des pas par agrégation (dans les deux sens, de l'agrégat vers l'agrégé et vice versa), des pas multiples par association, des enchaînements linéaires de pas et des enchaînements arborescents à partir d'une entité. Les arbres représentant ces trajets ont donc la forme générale suivante :

- chaque noeud a un ou plusieurs groupes de fils

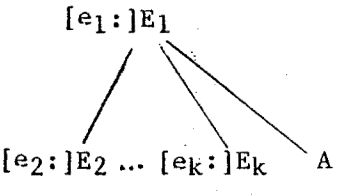
- chaque groupe de fils représente un pas du trajet (simple, multiple, par association ou par agrégation).

- les groupes de fils peuvent être des types suivants :

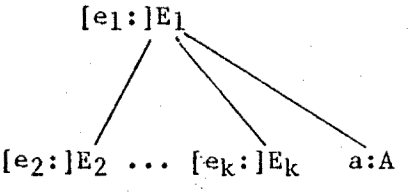
I Pas simple par association
PA(E_1, E_2, A)



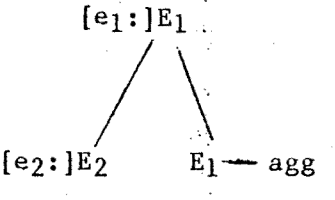
II Pas multiple par association
de E_1 à E_2,...,E_k à travers A



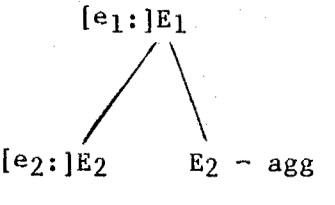
II(a) Pas vers l'association A
PVA(E_1,...,E_k, A)



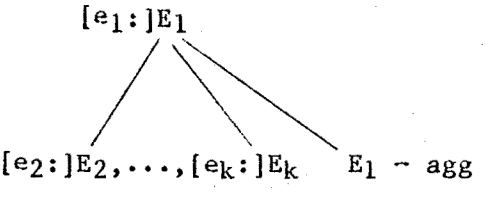
III Pas simple par agrégation
(a) de l'agrégat vers l'agrégé
PT(E_1, E_2)



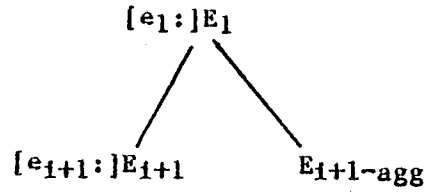
(b) de l'agrégé vers l'agrégat



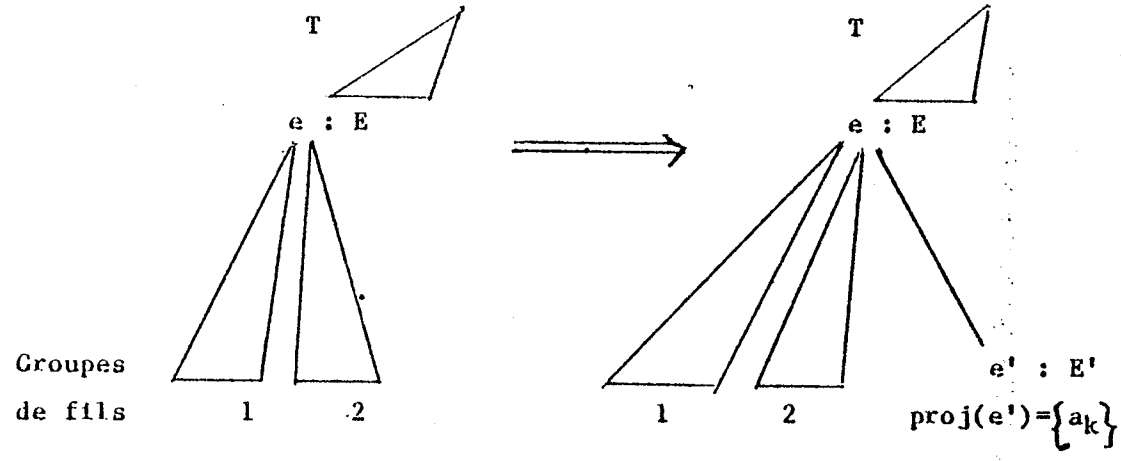
IV (a) Pas multiple de l'agrégat
E_1 vers les agrégés
E_2, ..., E_k



IV(b) Le pas multiple de l'agrégé vers les agrégats E_2, ..., E_k est représenté par k-1 groupes de fils de E_1 ; le i-ème groupe de fils est :



Nous avons vu dans les deux exemples précédents que les arbres qui représentent les variables de même portée dans un trajet peuvent être "agrandis" pour accéder à des relations représentant des enregistrements et des tableaux. Egalement, si dans un trajet Tr, on désigne avec une variable e une entité dérivée E par généralisation et si l'on note dans l'expression de valeurs ou dans la clause-where une fonction de la forme $e.a_k$ où a_k est un attribut hérité d'un ancêtre E' de E, l'arbre qui représente les variables de même portée que e dans Tr sera agrandi comme suit :



Si E est une entité dérivée par agrégation associative, l'accès aux attributs hérités induit aussi un agrandissement de l'arbre respectif. La nouvelle branche à partir de E aura en fait la forme d'un groupe de fils de type I ou II (selon si les attributs auxquels on veut accéder appartiennent à une ou plusieurs entités qui participent à l'association sous-jacente à E). Finalement, si E est une entité dérivée par agrégation d'entités désignée par la variable e et si dans l'expression de valeurs, on note la fonction "e" (par exemple, cf.

section 4.3.1, exemple 11), il faut expliciter l'accès aux entités agrégées. L'arbre respectif sera donc agrandi à partir de E avec un groupe de fils de type IV. Des nouvelles variables e_i seront générées (et $proj(e_i) = \{*\}$ pour chacune).

Nous introduisons maintenant la représentation intermédiaire des requêtes qui manipulent des documents.

Exemple 3 Nous prendrons comme exemple celui qui apparaît dans la section 4.4.3 (exemple 30). Nous rappelons la requête :

Select article a of session 1,* of c_2.texte-proc, c_2.nom
from CONGRES c_1, CONGRES c_2
where c_1.nom = 'ADI' and c_1.année = c_2.année
and (auteurs of a) * (auteurs of articles 1,* of contenu
of session 1,* of c_1.texte-proc) $\neq \{\}$

Représentation intermédiaire :

T_1 : c_1 : CONGRES

Cond-sel(c_1) = {nom = 'ADI'}
cond-sel-join(c_1) = {c_1.année = c_2.année}
cond-ensindx(c_1) = {ens1(c_1,a) = nil}

T_2 : c_2 : CONGRES
proj(c_2) = {chemin(a) of texte-proc, nom}
cond-sel-join(c_2) = {c_1.année = c_2.année}
chemin(a) = session1,* → contenu → a : article
cond-ensindx(a) = {ens1(c_1,a) = nil}

.ens1(c_1,a) : ens2(c_1) * ens3(a)
.ens2(c_1) : c_1 : CONGRES
proj-texte(c_1) = texte-lettre → session1,* → contenu → article 1,* → auteurs
.ens3(a) : a : article
proj-texte(a) = a → auteurs

Ici, les trois variables c_1, c_2 et a ont une portée globale. Cependant, étant donné que les documents ne sont pas stockés dans des relations, aucun arbre ne relie a à c_1. Ces deux variables sont reliées par le chemin dans la structure du document c_1.texte-proc. la notation "chemin(a)" veut dire que le chemin est conditionné par la variable a. Les ensembles ens2 et ens3 sont des ensembles d'unités texte à l'intérieur d'un document. Le premier est indexé par la racine c_1.texte-proc et le deuxième par un noeud de type "article" désigné par la variable a. Les chemins à partir de la variable indexante v sont spécifiés dans proj-texte(v). L'intersection de ces deux ensembles est donc indexée par c_1 et a.

7.3.3 - La génération d'arborescences

7.3.3.1 - Opérateurs algébriques et opérateurs documents

A partir de la représentation intermédiaire que nous venons de présenter, le générateur d'arborescences produit une ou plusieurs arborescences qui seront ensuite interprétées par l'interpreteur TIGRE. Nous utiliserons dans nos arborescences tous les opérateurs de l'algèbre relationnelle sauf la somme, l'antiprojection et le complément. Pour exprimer la comparaison ensembliste, nous utiliserons l'opérateur de division généralisé. La définition de cet opérateur a été rappelée dans la section 6.1.1. On peut l'exprimer à l'aide des autres opérateurs relationnels comme suit :

Soient R(AB), S(BC) deux relations où les groupes d'attributs A et C peuvent être vides.

$$R \div S(AC) \triangleq R*S\ [AC] - (R[A]xS - R*S)[AC]$$

Nous dénoterons cet opérateur par " $\div(\supset)$", ce qui indique que l'ensemble de valeurs du constituant B pour une valeur a de A de la relation R doit contenir ou être égal à l'ensemble de valeurs du constituant B pour une valeur c de C, pour que le couple (a,c) soit retrouvé. Quand $C = \phi$, cet opérateur devient l'opérateur de division habituel. On peut définir, de la même manière des opérateurs "$\div(=)$",

$\dot{\div}(\neq)$ et $\dot{\div}(>)$ ayant la signification évidente :

$$R \dot{\div} (=)S\ (AC) \triangleq (R*S)[AC] - ((R[A]xS) \cup (RxS[C]) - R*S)[AC]$$
$$R \dot{\div} (\neq)S\ (AC) \triangleq (RxS)[AC] - R \dot{\div} (=)S$$
$$R \dot{\div} (>)S\ (AC) \triangleq R \dot{\div} (\geqslant)S - R \dot{\div} (=)S$$

Pour la manipulation de documents, nous avons défini les opérateurs suivants :

1 - Projection à l'intérieur d'un attribut document :
project-doc(R, a, chemin-struct [,n])

Cet opérateur prend un attribut a d'une relation R contenant un identificateur interne d'un document et produit comme valeur de a une liste ordonnée de couples (nom-noeud, identificateur — interne) correspondant aux noeuds retrouvés par un chemin dans la structure du document. Le résultat est une relation ayant les mêmes attributs, mais seulement la valeur de a est modifiée. Un attribut identifiant un document est donc en général non-atomique. Ceci n'est pas gênant dans la mesure où seulement des opérateurs document manipulent ces valeurs.

Cet opérateur a un quatrième argument facultatif : un nom de noeud n à partir duquel le chemin dans la structure s'applique. Dans ce cas, la valeur de a contient déjà une liste de couples ; cet opérateur prend donc chaque couple (n, identificateur-interne) et le remplace par une liste ordonnée de couples correspondant aux noeuds retrouvés par le chemin qui démarre à partir de n.

Ce quatrième argument est important pour pouvoir résoudre les requêtes où le chemin dans la structure est conditionné par une variable qui désigne le(s) noeud(s) de nom n. Nous illustrerons ceci plus loin.

2 - Sélection à l'intérieur d'un attribut document :
select-doc (R, a, cond[,n])

Cet opérateur est utilisé pour effectuer de la recherche par contenu dans les unités texte d'un document. R est une relation, a est un at-

tribut identifiant le document ou des noeuds du document et "cond" est une condition (ou une combinaison boléene de conditions) de la forme "K ⟨op⟩ chemin-struc". K est une chaîne de caractères éventuellement partiellement spécifiée (à l'aide du "don't care character" '*'), ⟨op⟩ est un des opérateurs =, ≠, in et "chemin-struc" est un chemin dans la structure du document à partir de la racine a ou d'un noeud interne de nom n donné comme argument facultatif. Le chemin doit arriver à une unité texte ou à un ensemble d'unités texte (auquel cas l'opérateur est in). A chaque couple (n, ident-interne) (ou (a, ident-interne)) d'un n-uplet, on applique la condition ; si elle la satisfait, on laisse le couple dans la liste, autrement, on le supprime. Si à la fin de ce processus, la liste devient vide, on supprime le n-uplet.

3 - Extraction d'éléments d'un document :
extraire-elem (R, a, chemin-struc [,n])

On utilisera cet opérateur essentiellement pour extraire des unités textes d'un document dans le but d'effectuer des comparaisons ensemblistes avec d'autres ensembles. Chaque unité texte sera mise dans un n-uplet de la relation résultat. On peut cependant extraire aussi des identificateurs internes de noeuds pour, par exemple, les compter et répondre ainsi à des requêtes comme celle de la section 4.3.4.1.3 exemple 16. Cet opérateur prend donc comme arguments une relation R, un attribut a identifiant le document ou des noeuds du document, un chemin dans la structure à partir de la racine a d'un noeud interne de nom n donné comme argment facultatif.

Supposons que pour un noeud (n, ident-interne) qui apparaît dans la liste de l'attribut a pour un n-uplet t, on retrouve par chemin-struc les éléments $elem_1,\ldots,elem_k$ (des textes ou des identificateurs internes d'autres noeuds). Alors la relation résultat R' contient les k n-uplets $(t_1$ (n, ident-interne), $elem_1),\ldots,(t$(n, ident-interne), $elem_k)$. Elle a donc deux attributs de plus que R (ou un seul de plus, si n n'est pas spécifié) ; leurs noms sont n et "texte" (ou le nom du noeud des identificateurs internes extraits).

4 - Extraction de paramètres d'un document :

Extract-param (R, a, P_1, ..., P_n)

Cet opérateur retrouve la valeur des paramètres P_1, ..., P_n d'un document identifié par la valeur de l'attribut a de la relation R. La relation résultat a n-attributs de plus que R, nommés P_1, ..., P_n.

5 - Extraction de l'évaluation de fonctions d'un document :

Extract-evalfonct (R, A, $f_1[(a_1)]$, ..., $f_n[(a_n)]$)

Cet opérateur est analogue au précédent. Chaque fonction f_i est éventuellement munie d'un paramètre actuel (ou de plusieurs), donné par la valeur de l'attribut a_i. La relation résultat a n attributs de plus que R, nommés f_1, ..., f_n.

6 - Regroupement d'identificateurs de noeuds dans un attribut document :

Group (R, a, a_1, ..., a_n)

Cet opérateur est en quelque sorte l'inverse de l'extraction d'éléments. Les valeurs des attributs a_1, ..., a_n qui sont des identificateurs de noeuds d'un même document, sont regroupés dans une liste qui constituera la valeur de l'attribut a. Ce regroupement respecte l'ordre des attributs : a_i est ancêtre de a_{i+1}, comme il est montré ci-dessous.

Group(R,texte-proc,session,article)

R ⟶ R'

année	nom	texteproc	session	article
80	VLBD	id_1	s_1	a_1
80	VLBD	id_1	s_1	a_2
80	VLBD	id_1	s_2	a_4
81	VLDB	id_2	s_4	a_3
83	SIGMOD	id_4	s_1	a_1

année	nom	texte-proc
80	VLBD	(session s_1)(article a_1) (article a_2)(session s_2) (article a_4)
81	VLDB	(session s_4)(article a_3)
83	SIGMOD	(session s_1)(article a_1)

7.3.3.2 - Les règles de génération

Une requête LAMBDA est une expression de selection ou une construction d'expressions de sélection via des opérateurs ensemblistes. Ces opérateurs (+, *, -) se traduisent directement en termes des opérateurs relationnels union, intersection et différence sur les arborescences TIGRE obtenues des expressions de selection. Chaque expression de sélection donne lieu à une union de sous-arborescences, chacune correspondant à un groupe de conditions dans la forme normale disjonctive de la représentation intermédiaire de l'expression de sélection. Nous fixerons donc notre attention sur un groupe donné de conditions dans la forme normale disjonctive. Elle se compose de n arbres T_1, ..., T_n qui représentent explicitement les conditions implicites pour les variables de portée globale de n des m trajets de l'expression de sélection ($n \leqslant m$).

Les variables qui apparaissent dans des arbres différents sont forcément différentes. Par contre, une variable peut apparaître plusieurs fois dans un arbre (car elle peut apparaître plusieurs fois dans un trajet). Pour chaque variable e_i, nous associerons une fonction "compteur(e_i)" qui donne le nombre d'occurences d'une variable dans un arbre. Nous supposerons, par commodité que chaque noeud d'un arbre est désigné par une variable.

7.3.3.2.1 - Parcours des arbres

Chaque arbre T sera parcouru du bas vers le haut : le traitement pour chaque noeud désigné par e_i représentant une entité sera le même. Il donnera lieu à un arbre que nous appelerons ⟨arbre-noeud(e_i)⟩. Nous illustrerons ce traitement par la suite. Une association apparaît dans les groupes de fils de type I, II ou II(a) (cf section 7.3.2). Sans perte de généralité, soit A une association désignée par a qui apparaît dans un groupe de fils de E_1 de type II(a). A est une association n-aire entre E_1, ..., E_n. Nous supposerons que les attributs de la relation A-desig sont A-c, E_1-c .., E_n-c (et non pas A-c, rôle(E_1,A)-c, ..., rôle(E_n,A)-c). Nous allons donner les règles de génération de ⟨arbre-noeud(a)⟩.

Soit ent-proj(a) = $\{E_1\text{-}c, \ldots, E_k\text{-}c\}$ l'ensemble correspondant aux entités liées par A dans le groupe de fils de E_1. Bien entendu, $k \leq n$.

⟨arbre-noeud(a)⟩ :

(i) Project
- ent-proj(a), proj(a) , si K < n
- ⟨arbre-cond(a)⟩

(ii) project
- proj(a) , si K=n et proj(a)≠∅
- ⟨arbre-cond(a)⟩

(iii) arbre-cond(a) , si K=n et proj(a)=∅

⟨arbre-cond(a)⟩ :

(i) join , si cond-sel(a)≠∅
- A-desig
- select
  - A-p
  - ⟨cond-sel(a)⟩

(ii) join , si cond-sel(a)=∅ et proj(a)≠∅
- A-desig
- rA-p

(iii) A-desig , si cond-sel(a)=∅ et proj(a)=∅

Nous montrerons maintenant les règles générales d'un parcours d'un arbre T :

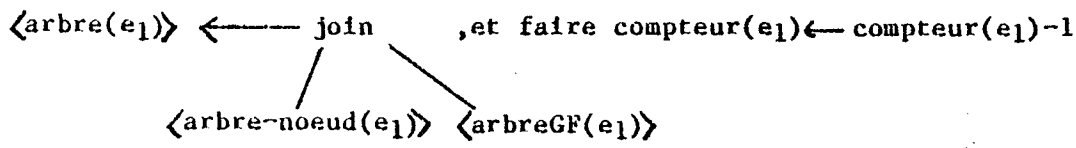
Soit proj-arbre(e_i) l'ensemble d'attributs de arbre-noeud(e_i) ; soit e_i la racine de T, soit proj(T) = $\bigcup_{v \in VAR(T)}$ proj(v), VAR(T) étant l'ensemble de variables de T.

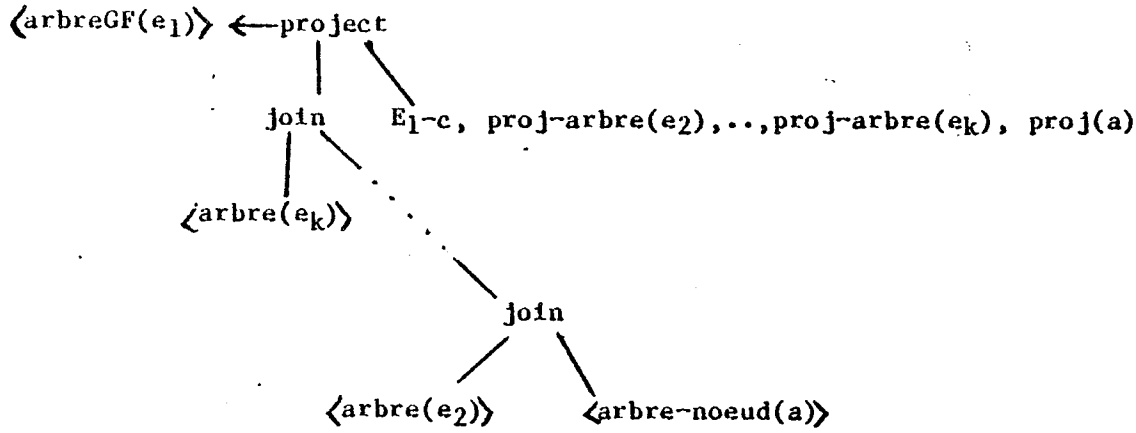
1 - ⟨arbre(T)⟩ ⟵ project
- ⟨arbre(e_i)⟩
- proj(T)

Soit maintenant e_1 un noeud quelconque de T.

2 - Si e_1 n'a pas de fils, ⟨arbre(e_1)⟩←⟨arbre-noeud(e_1)⟩, et compteur(e_1) ⟵ compteur(e_1) - 1

3 - Si e_1 a un groupe de fils GF de type II(a) (ou type I ou II comme cas particulier)

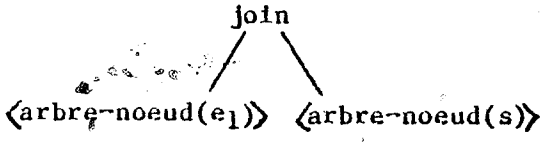




Ceci est dans le cas où compteur(e_2) = ... = compteur(e_k) = 0)

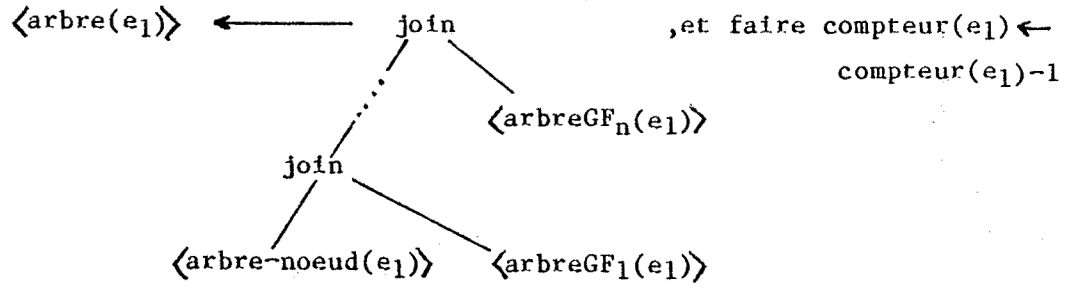
Si par exemple, compteur(e_j) ≠ 1, il faut inclure E_j-c dans les attributs à projeter. Bien entendu si un ⟨arbre(e_j)⟩ ou si ⟨arbre-noeud(e_1)⟩ est vide, on supprime la jointure respective.

Le cas du groupe de fils de type III ou IV est similaire : il suffit de changer ⟨arbre-noeud(a)⟩ par E_1-agg (ou E_2-agg selon le cas). les fils "seuls" désignés par s (représentant des tableaux, des enregistrements ou de ancêtres dans une hiérarchie de généralisation) génèrent une jointure



Le résultat d'une requête manipulant des agrégations d'entités ou des tableaux est une relation "plate", c'est à dire dont les n-uplets sont normalisés, contrairement à la sémantique "moléculaire" des agrégations d'entités et des tableaux. Des procédures de dénormalisation s'appliquant aux résultats d'une requête devront être offertes de façon à ce que l'usager manipule les tableaux et les agrégations en accord avec leur sémantique.

4 - Si e_1 a m groupes de fils GF_1, ..., GF_n, alors :



La définition de chaque $arbreGF_i(e_1)$ dépend, bien entendu du type de GF_i.

7.3.3.2.2 - Traitement pour chaque noeud d'un arbre

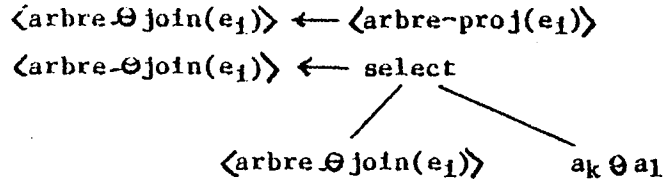
Pour chaque noeud e_i représentant une entité E_i (ou un tableau ou un enregistrement) apparaissant dans un arbre T, on appliquera les pas suivants, au bout desquels on obtiendra ⟨arbre-noeud(e_i)⟩

1. Les conditions de cond-sel(e_i) qui ne s'appliquent pas sur les documents. Ceci donne lieu éventuellement à une sélection sur E_i-p (la relation P de E_i).

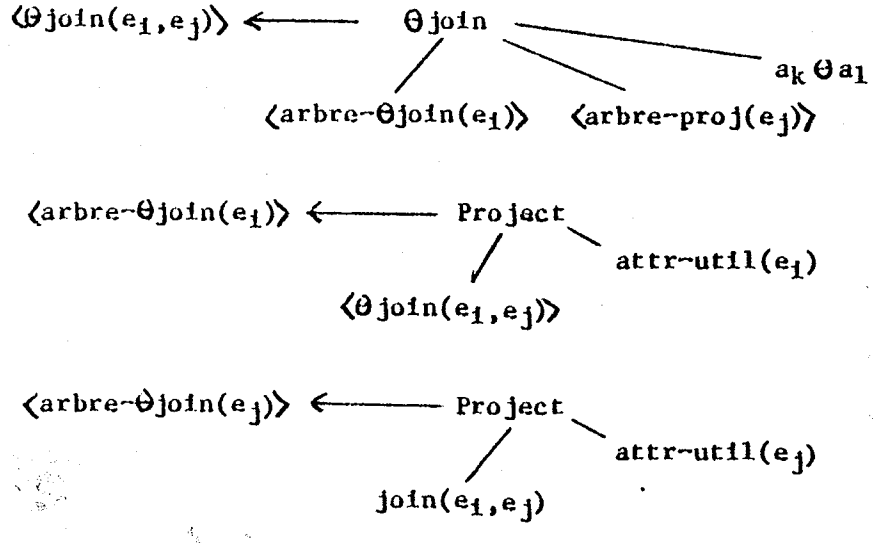
2. Les attributs de E_i qui seront utilisés par la suite. Pour celà on regarde proj(e_i), cond-sel-join(e_i), les conditions de la forme "e_i.a in ensemble" dans cond-appart(e_i), et les attributs tableau et document utilisés dans cond-ensindx(e_i). On projette E_i sur tous ces attributs que l'on dénote par attr-util(e_i). Si aucun attribut n'est utilisé et si cond-ensindx(e_i) = cond-appart(e_i) = ∅, alors

$\langle$arbre-noeud(e_i)$\rangle$ = $\emptyset$. Aussi, le seul attribut à utiliser est E_i-c mais E_i a un groupe de fils alors $\langle$arbre-noeud(e_i)$\rangle$ = $\emptyset$ (cèci dans le but d'éviter des jointures inutiles). Sinon, soit $\langle$arbre-proj(e_i)$\rangle$ l'arbre construit jusqu'ici.

3. Les conditions de jointure c = $e_i.a_k \Theta e_j.a_l$. On suppose que a_k et a_l sont des constituants de nom différents ; autrement on procède à une renomination de a_k. Si e_i et e_j se trouvent dans le même arbre et si e_j apparaît dans le sousarbre dont e_i est la racine, alors :



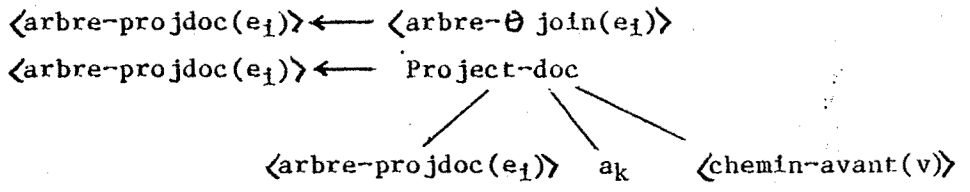
Sinon, e_i et e_j se trouvent dans des arbres différents T et T'. On traite T' du bas vers le haut jusqu'à ce qu'on trouve le noeud e_j, puis on effectue les pas 1 et 2 sur e_j, ce qui donne $\langle$arbre-proj(e_j)$\rangle$. Alors :



Un exemple d'application des règles exposées jusqu'ici sera montré plus loin (en bas de la règle 5.1). Il faut s'assurer que tous les attributs du résultat ont des noms différents. Autrement on renomme différemment au préalable. Quand on retraitera T', on ne tiendra plus compte de cette condition c et on saura qu'il faut redémarrer le traitemement à partir de $\langle$arbre-Θ join(e_j)$\rangle$ (qui constitue donc une arbo-

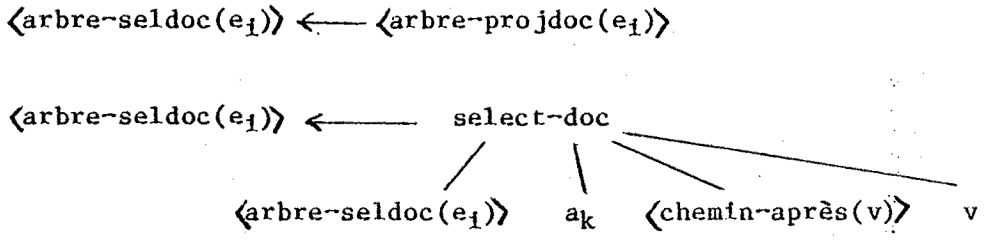
rescence temporaire). Le traitement de T' aboutira à l'agrandissement de l'arborescence TIGRE au cours de construction pour T.

4. On examine maintenant les chemins dans un document et les sélections à l'intérieur d'un document. S'il existe $p \in proj(e_i)$ tel que p = chemin(v) of $e_i.a_k$ (v peut ne pas exister), alors :



où ⟨chemin-avant(v)⟩ est la partie du chemin(v) jusqu'au noeud désigné par v (si v n'existe pas, il représente tout le chemin).

Les conditions de sélection "k $\{=, \neq\}$ chemin-après(v)" de cond-sel(v) ou "k in chemin-après(v)" de cond-ensindx(v) sont traitées comme suit :

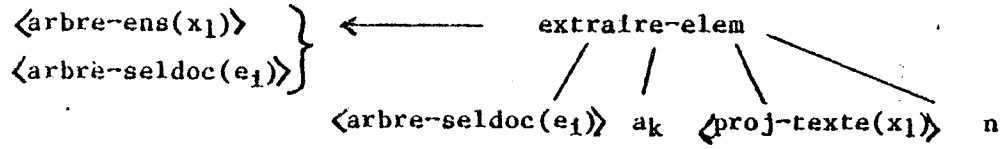


Les conditions de sélection sur a_k quand v n'existe pas sont un cas particulier de ceci; le quatrième fils de select-doc disparait.

5. Pour tenir compte des conditions dans cond-appart(e_i) et dans cond-ensindx(e_i), il faut construire des arbres correspondants à des ensembles. Soit ens(x_1, ..., x_n) un ensemble indexé par les n variables x_1, ..., x_n, $n \geq 0$

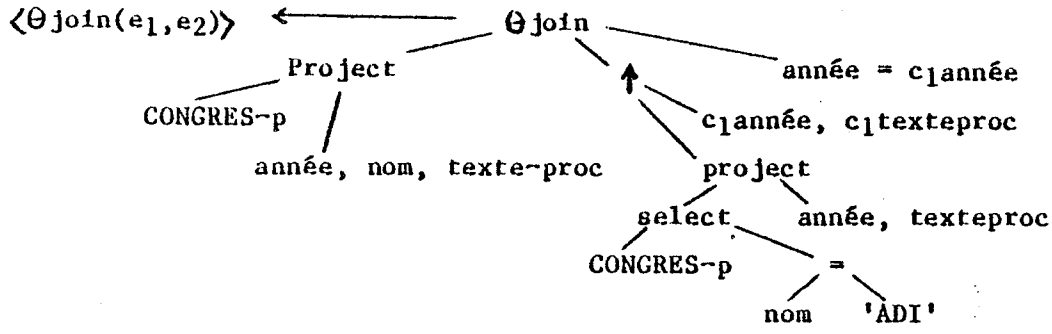
5.1 Si l'ensemble est un ensemble de textes à l'intérieur du document, alors, n = 1 et l'ensemble est défini par le chemin qui apparait dans proj-text(x_1) (cf section 7.3.2). Soit a_k l'attribut document d'une entité désignée par e_i. Si x_1 est une variable désignant un

noeud interne n du document, l'arborescence TIGRE générée pour cet ensemble est :

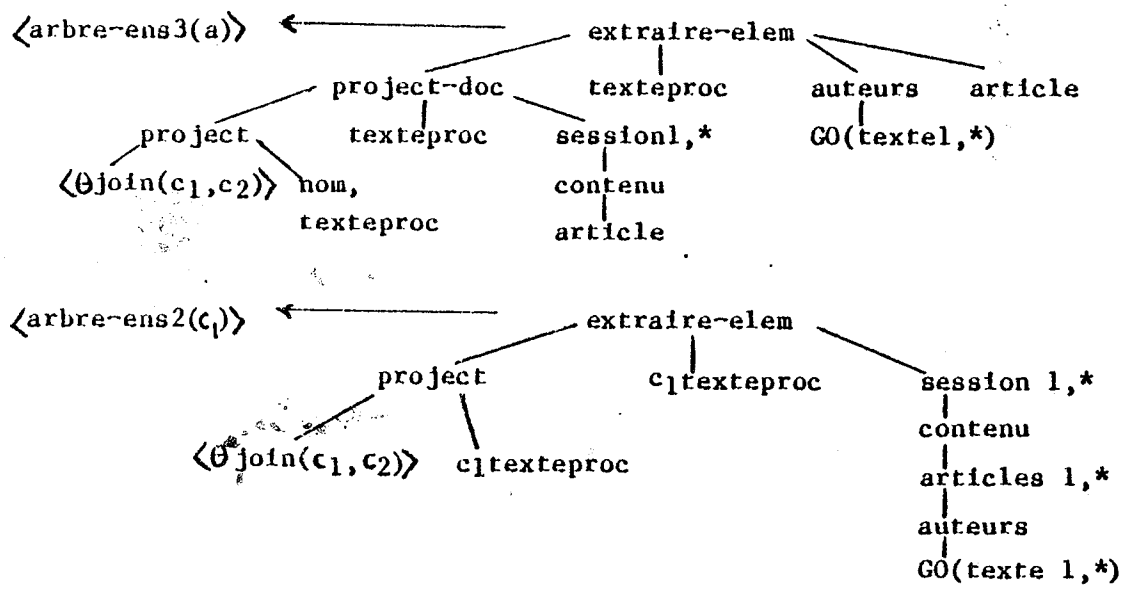


Sinon x_1 est la même e_i et on supprime le quatrième argument de l'opérateur extraire-elem. Avant de continuer nous montrerons la génération des arborescences pour l'exemple 3 de la section précédente 7.3.2.

. Traitement de conditions de sélection de projection et de jointure :



. Génération arborescences correspondantes aux ensembles ens3(a) et ens(c_1) :

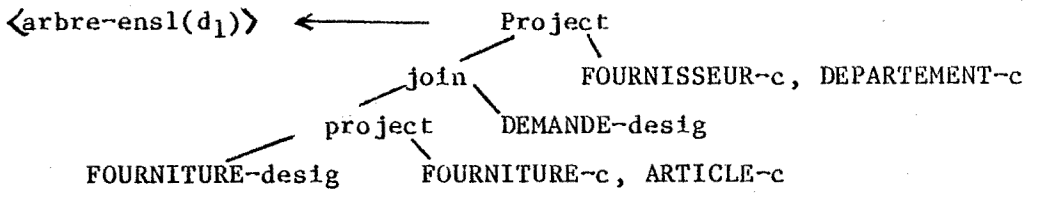


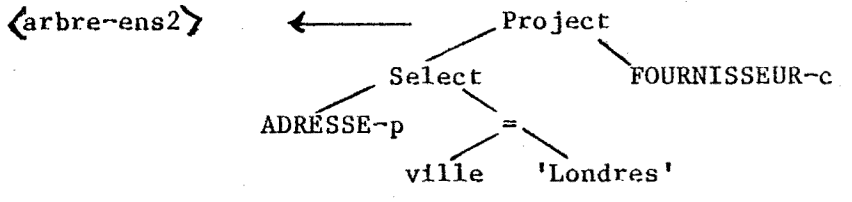
Ici, comme il s'agit de retrouver les textes eux-même et non pas leurs identificateurs, on l'indique par "GO" (Gros objet).

5.2 Si l'ensemble est défini par association, agrégation ou par un tableau, il y a un arbre T_{ens} le définissant. L'arborescence de l'ensemble est alors :

$$\langle \text{arbre-ens}(x_1,\ldots,x_n)\rangle \longleftarrow \langle \text{arbre}(T_{ens})\rangle$$

La construction de $\langle \text{arbre}(T_{ens})\rangle$ a été présentée dans la section précedante. Nous montrerons la génération des arborescences correspondants aux ensembles ens1(d1) et ens2 qui apparaissent dans la représentation intermédiaire de l'exemple 1, section 7.3.2.

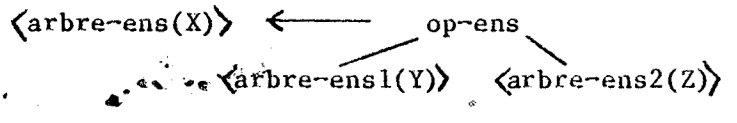




5.3 Si l'ensemble est construit par opérateurs ensemblistes à partir d'autres, alors

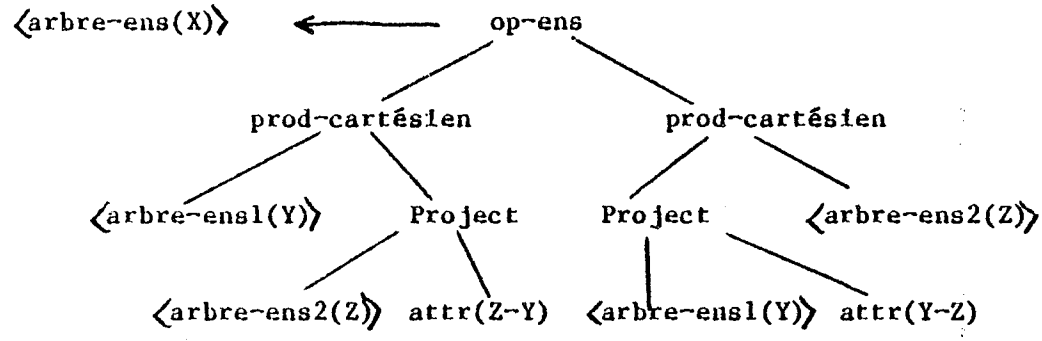
$$\text{ens}(x_1,\ldots,x_n) = \text{ens1}(y_1,\ldots,y_m)\ \text{op-ens}\ \text{ens2}(z_1,\ldots,z_k)$$

où n, m, k $\geqslant$ 0. Soient $Y = \{y_1,\ldots,y_m\}$, $Z = \{z_1,\ldots,z_k\}$. Il est toujours vrai que $X = \{x_1,\ldots,x_n\} = Y \cup Z$. Si Y = Z, alors :



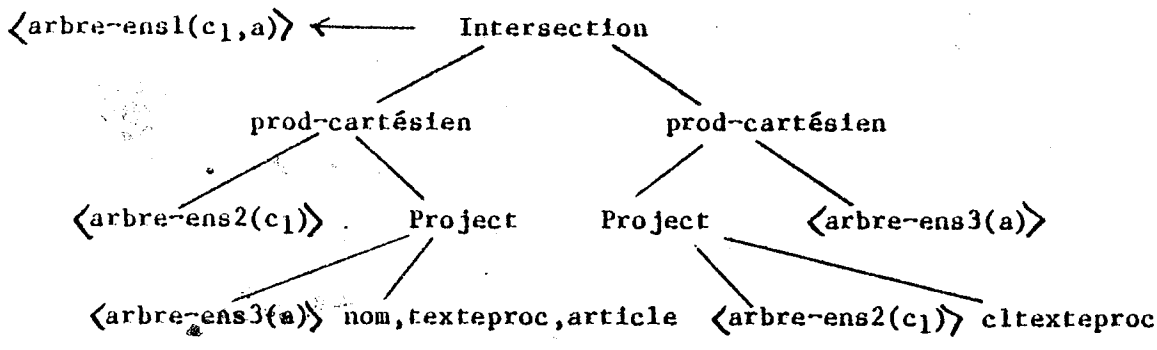
Si Y $\neq$ Z, alors, il faut rendre les deux arborescences ⟨arbre-ens1(Y)⟩ et ⟨arbre-ens2(Z)⟩ "union-compatibles". Pour cela, on

utilise le produit cartésien :



L'expression attr(V) où V est un ensemble de variables donne les attributs correspondants à chaque variable v de V. Si v désigne une entité E_i, attr(v) = E_i-c . Si v désigne un noeud n d'un document, attr(v) = n.

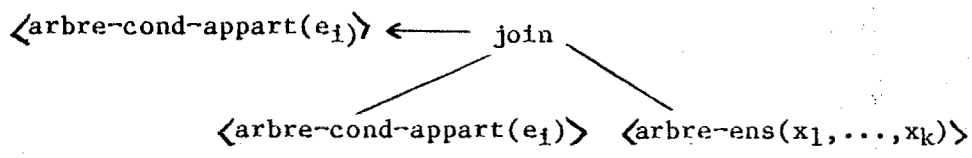
En réalité, si au moins un de ces ensembles est un ensemble de textes d'un document, cette règle ne s'applique pas tout à fait. En effet, nous avons vu dans la règle 5.1 que l'arborescence construite à la sortie de la règle 4 fait partie de la définition de la construction de l'ensemble. Ceci implique qu'il peut y avoir des attributs que l'on utilisera plus tard dans la relation représentant l'ensemble qui ne correspondent à aucune des variables de Y ni de Z. Les attributs à projeter avant les produits cartésiens ne sont pas ceux de attr(Z-Y) ni ceux de attr(Y-Z). Pour illustrer ceci, nous reprenons la génération des arborescences de la requête de l'exemple 3, pour montrer comment traiter ens1(c_1,a) = ens2(c_1) * ens3(a).



6. On examine maintenant les conditions dans cond-appart(e_i). Elles ont la forme "$(f_1(e_1),\ldots,f_n(e_n))$ in $ens(x_1,\ldots,x_k)$", où $e_i \in \{e_1,\ldots,e_n\}$, $n \geq 1$, $k \geq 0$. Elles génèrent un arbre ⟨arbre-condapart(e_i)⟩ D'abord :

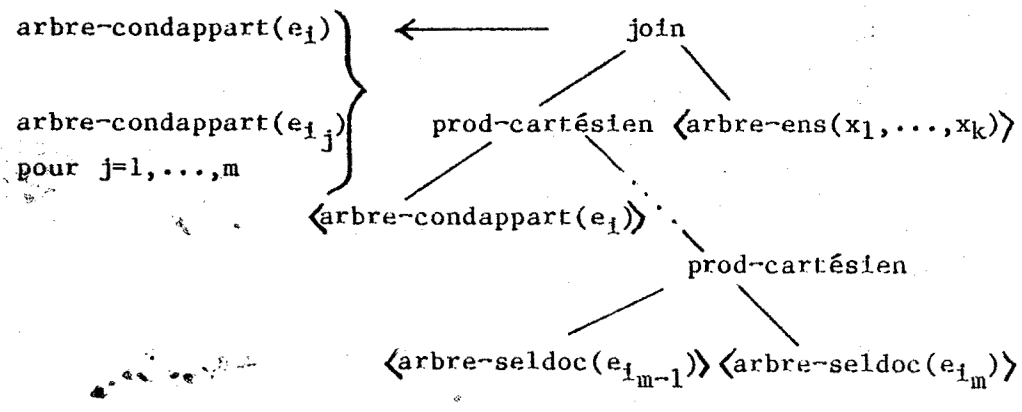
⟨arbre-condappart(e_i)⟩ ← ⟨arbre-seldoc(e_i)⟩ (l'arbre construit à la sortie du pas 5.1)

6.1 Si $e_1 = \ldots = e_n = e_i$, alors :



6.2 Sinon, si toutes les variables appartiennent au même arbre T et celle qui aparaît plus haut dans T est e_i, alors appliquer la règle précédente. S'il existe une variable e_j telle que e_i apparaît dans le sous-arbre de T dont e_j est la racine, cette condition sera traitée dans ⟨arbre-noeud(e_j)⟩.

6.3 Autrement, il existe des variables qui apparaissent dans d'autres arbres $T_1,\ldots,T_m$. De même que dans 6.2, la règle qui suit s'applique si e_i est la variable qui apparaît plus haut dans T de toutes celles parmi $e_1,\ldots,e_n$ qui apparaissent dans T. Soit e_{i_j} la variable apparaissant le plus haut dans T_j (pour j=1,..,m). Alors:



Quand on re-traitera les arbres $T_1,\ldots,T_m$, on ne tiendra plus compte de cette condition et on démarrera le traitement à partir de chaque

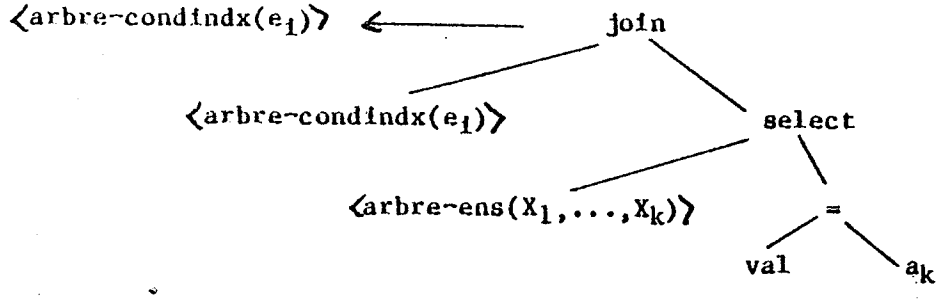
$\langle$arbre-condappart(e_{ij})$\rangle$. Egalement, pour chaque $x_i \in \{x_1,\ldots,x_k\}$, on considérera que cette condition, qui appartient à cond-ensindx(x_i), a déjà été traitée.

7. Ensuite, on examine les conditions dans cond-ensindx(e_i). Elles génèrent un arbre $\langle$arbre-condindx(e_i)$\rangle$. D'abord :

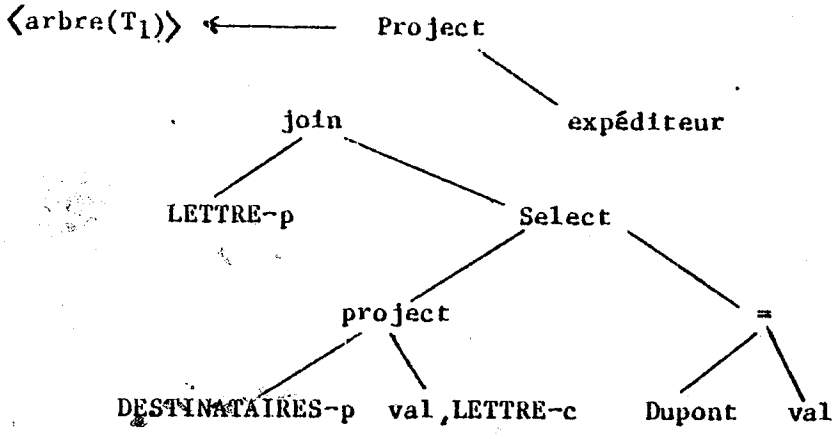
$\langle$arbre-condindx(e_i)$\rangle$ ⟵ $\langle$arbre-condappart(e_i)$\rangle$

7.1 Les conditions d'appartenance de la forme "val in ens($x_1,\ldots,x_k$)" où $e_i \in \{x_1,..,x_k\}$ sont traitées d'abord.

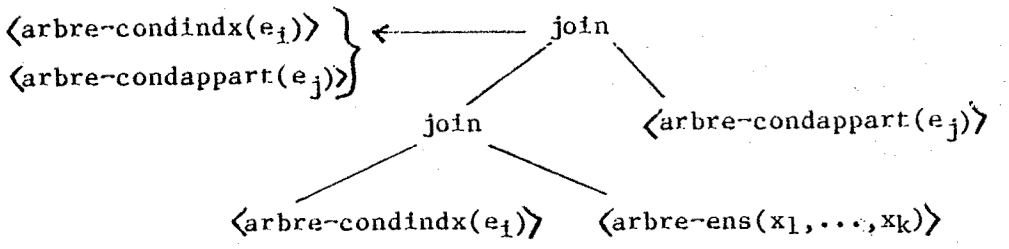
(a) Si val est une constante, celà donne :



où a_k est l'attribut correspondant à la variable de portée intérieure à ens($x_1,\ldots,x_k$). Cette règle s'applique pour générer l'arborescence de la requête de l'exemple 2, section 7.3.2 :



(b) Si val = ($f_1(e_j),\ldots,f_n(e_j)$ où $n \geq 1$,



(c) Si "val" a la forme $(f_1(e_1),\ldots,f_n(e_n))$ alors, il faut remplacer ⟨arbre-condappart(e_j)⟩ par des produits cartésiens comme dans la règle 6.3. Nous omettrons cette règle ici.

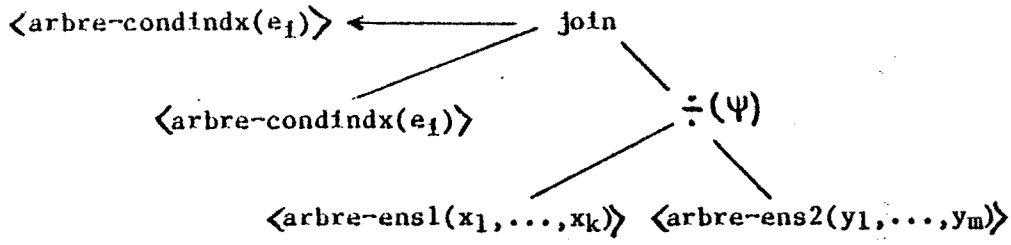
7.2 On considère maintenant des conditions de comparaison entre ensembles : elles ont la forme ens1($x_1,\ldots,x_k$) Ψ ens2($y_1,\ldots,y_m$) où k, m ≥ 0. Bien entendu, $e_i \in \{x_1,\ldots,x_k, y_1,\ldots,y_m\}$.

Soit proj(ens1($x_1,\ldots,x_k$)) (resp-proj(ens2($y_1,\ldots,y_m$))), l'union des attributs dans proj(v) pour chaque v apparaissant dans la représentation intermédiaire différente de $x_1,\ldots,x_k$. L'emploi de l'opérateur de division exige de prendre des précautions avec les noms des attributs:

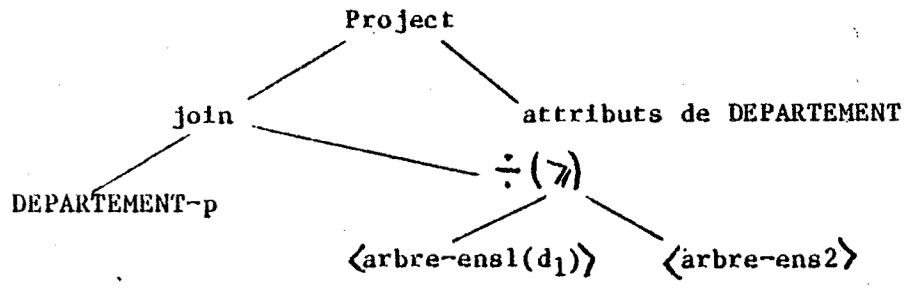
- Si proj(ens1($x_1,\ldots,x_k$) = proj(ens2($y_1,\ldots,y_m$)), on renomme les attributs de façon à ce qu'il y ait égalité (on sait qu'il existe un bijection entre les deux ensembles d'attributs telle que les attributs mis en correspondance ont un type compatible).

- Si $\{x_1,\ldots,x_k\} \cap \{y_1,\ldots,y_n\} \neq \emptyset$, alors, il y aura une jointure implicite (comme dans l'opérateur de division ER - cf section 6.1.1). Cette jointure doit être explicitée après avoir appliqué l'opérateur de division ; mais pour celà, il faut renommer les attributs d'une relation, disons ⟨arbre-ens2($Y_1,\ldots,Y_m$)⟩ de façon à ce que l'intersection soit vide. Supposons sans perte de généralité que seulement $x_1 = y_1$ et que l'on a renommé y_1 par y'_1. Après la division, on explicite la jointure : on fait une sélection du résultat sur la condition $x_1 = y'_1$ puis on projette le résultat sur sur $x_1,\ldots,x_k, y_2,\ldots,y_m$.

(a) Si aucun des ensembles n'est un ensemble de textes d'un document, alors :

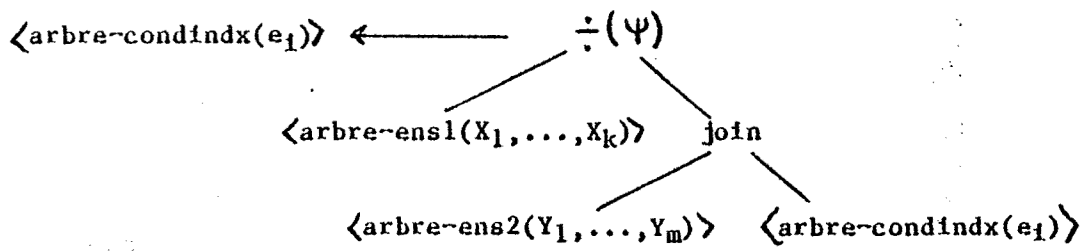


Par exemple, dans l'exemple 1, cond-indx(d_1) = {ens1(d_1) ⋑ ens2}, les arbres correspondants ont été présentés comme exemple d'application de la règle 5.2. L'arborescence TIGRE correspondante au premier groupe de conditions OR_1 est alors :



Dans ce cas particulier, comme l'ensemble ens2 n'est pas indexé, l'opérateur ÷(⋑) correspond à l'opérateur de division habituel.

(b) Si un des deux ensembles est un ensemble de textes d'un document, supposons ens1(X_1,...,X_k) et si $e_1 \in \{Y_1,...,Y_k\}$, alors :

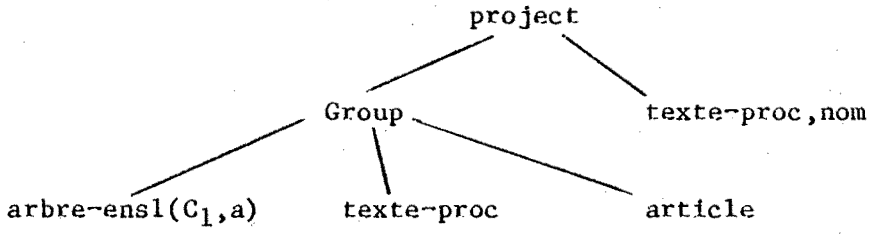


(c) Sinon, si $e_1 \notin \{y_1,...y_k\}$, ou si les deux ensembles sont un ensemble de textes d'un document, on supprime la jointure et on a la division de deux ensembles. La variable e_1 intervient dans la définition d'un de ces deux ensembles (suivant si $e_1 \in \{x_1,...,x_k\}$ ou $e_1 \in \{y_1,...,y_m\}$), à cause de la règle 5.1.

(d) Si un des deux ensembles, disons ens2 est $\{\}$, l'ensemble vide, et si $\Psi \in \{>, \geq, \neq\}$, on supprime l'opérateur de division des règles précédentes. Si Ψ est l'égalité, il faut effectuer une différence ensembliste entre l'ensemble domaine des élements $(x_1,\ldots,x_n)$ -un produit cartésien- et la projection de $\langle$arbre-ensl$(x_1,..,x_k)\rangle$ sur les attributs correspondants à $x_1,\ldots,x_k$.

8. Finalement, on effectue un regroupement des attributs document s'il est nécessaire. Les attributs que l'on a besoin de regrouper sont ceux générés par l'opérateur d'extraction d'éléments et que l'on veut retrouver comme résultat (ou dont le résultat dépend des valeurs de ces attributs ; dans ce cas, de nouvelles projections-document sur le résultat du regroupement doivent s'effectuer).

Nous présentons ci-desous l'arborescence résultat pour la requête 3. L'arborescence $\langle$arbre-ensl(c,a)$\rangle$ avait été présentée comme exemple de génération de la règle 5.3. La condition ensl$(c_1,a) \neq \{\}$ n'ajoute rien à $\langle$arbre-ensl$(c_1,a)\rangle$, d'après 7.2(d).



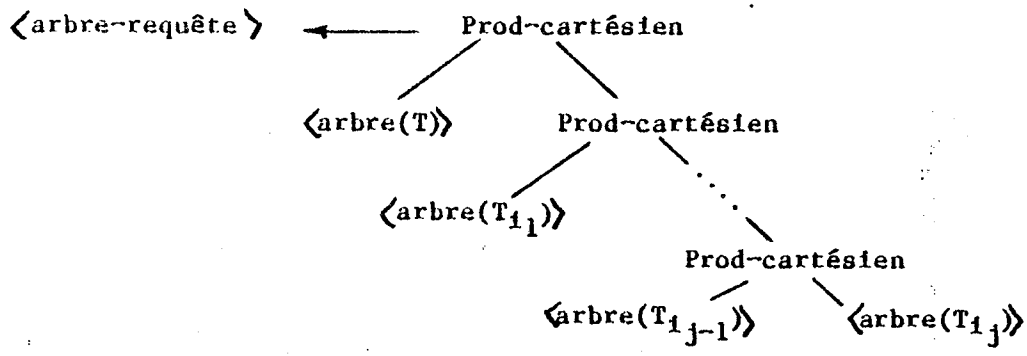
L'application des 8 pas précédents au noeud e_i donne lieu à l'arbre $\langle$arbre-noeud$(e_i)\rangle$.

7.3.3.2.3 - Résultat d'une requête

Après avoir appliqué ces 8 pas à tous les noeuds d'un arbre T, les autres arbres $T_1,\ldots,T_n$ peuvent être complètement traités, partiellement traités ou pas traités du tout. Pour les arbres partiellement traités, on continue leur traitement à partir du point où ils avaient été laissés, et les arborescences générées sont intégrées à l'arborescence générée pour T. Cette intégration a la forme d'une jointure ; elle s'effectue avant de projeter l'arbre $\langle$arbre$(e_1)\rangle$résultant après trai-

tement de la racine e_1 de T sur proj(T) - cf. section 7.3.3.2.1.

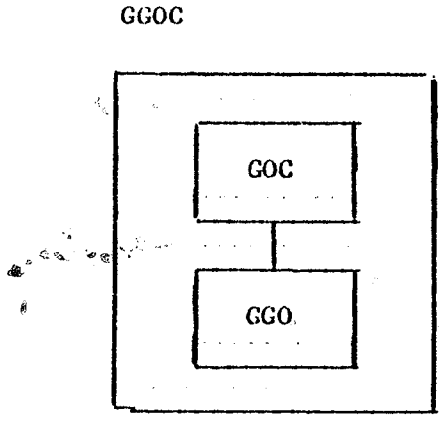
Finalement, on traite les arbres non traités (encore, deux arbres non traités peuvent donner lieu à une seule arborescence). Soient $\langle arbre(T_{i_1})\rangle, \ldots, \langle arbre(T_{i_j})\rangle$ $(j \geqslant 1)$, les arborescences générées. Le résultat de la requête est alors :



Quelques unes de ces règles ont été implantées [PV 83a] Celles-ci correspondent à un sous-ensemble du langage dans lequel les trajets sont linéaires, les valeurs à retrouver sont ceux des attributs définis sur un type de base et les conditions de la clause-where sont des combinaisons booléenes de prédicats de la forme attribut θ constante ou attribut θ attribut.

7.4 STOCKAGE ET ACCES AUX DOCUMENTS GENERALISES

Le stockage et l'accès aux documents généralisés est assuré par le module de gros objets complexes (GGOC). Ce module est lui-même décomposé en deux : le module de Gestion d'objets complexes (GOC) et le module de Gros objets (GGO). Graphiquement :



Le module GOC se charge de l'accès aux noeuds de la structure d'un document. La structure d'un document est représentée linéairement par une liste (cf. section 7.2.2) elle-même stockée comme un gros objet ; elle est stockée par le module GGO. Cette structure est parcourue par les primitives du module GOC: fils-suivant, frère-suivant, etc. Il y a aussi des primitives générales de mise à jour d'une sous arborescence par une autre.

Le module GGO est responsable du stockage de gros objets. Deux classes de gros objets ont été considérées : taille fixe et taille variable. Les premiers correspondent à des objets dont on connait la taille maximale à leur création. La gestion de l'espace physique est très simple dans le but d'obtenir des performances acceptables. Les objets de taille variable sont plus compliqués à gérer car il faut se préoccuper d'allouer efficacement l'espace physique de façon à avoir un taux d'occupation satisfaisant. Lorsque, par le jeu des mises à jour, l'espace physique devient trop émietté, il est possible d'exécuter des procédures de compactage. Chaque gros objet a un identificateur qui est unique du point de vue du module GGO. Les opérations sur les gros objets sont :

- créer, ce qui délivre un identificateur unique.

- supprimer

- lire : on lit une portion d'un gros objet à partir d'un certain déplacement ; le résultat est placé dans une zone fournie par l'usager.

- écrire : symétrique à lire.

- remplacer : remplace une portion d'un objet par un autre objet fourni par l'usager. Les deux portions peuvent être de taille différente. Cette opération s'applique aux objets de taille variable.

7.5. INTERFACES USAGER AU SERVEUR TIGRE

Deux type d'interfaces usager au serveur TIGRE sont actuellement en cours d'étude : une interface programmable dans laquelle on pourra écrire des énoncés LAMBDA à partir de PASCAL et une interface conversationnelle graphique. Nous détaillerons par la suite les caractéristiques principales de ces deux types d'interfaces.

7.5.1 - Interface programmable PASCAL-LAMBDA

Cette interface est adressée aux concepteurs d'application manipulant une base de données généralisées. Dans ce type d'interface, les types définissant un schéma conceptuel TIGRE sont invoqués dans un programme PASCAL. L'interface est de type pre-processeur :les énoncés LAMBDA sont interprétés avant de compiler le programme PASCAL. (Cette approche s'apparente donc à EQUEL, l'interface C-QUEL du SGBD relationnel INGRES [ASH 76]).

Les variables temporaires (résultats des requêtes) ainsi que les variables paramétrant les requêtes sont déclarées en LAMBDA et PASCAL sont assez voisins ; de ce fait, l'incorporation de structures de données PASCAL correspondantes aux variables temporaires et aux variables paramétrant des requêtes est une tâche aisée qui sera effectuée à la précompilation.

Pour ce faire, le pré-processeur effectuant ce traitement fait appel au gestionnaire de description du serveur. On vérifie aussi que l'utilisation de ces variables dans le corps du programme respecte le typage.

Le préprocesseur fait appel au traducteur LAMBDA pour obtenir les arborescences TIGRE correspondantes aux énoncés. Ce sont des arborescences qui seront passées en paramètre à l'interpréteur TIGRE au moment de l'éxécution du programme. Les valeurs actuelles des paramètres des requêtes seront incorporées à l'arborescence de l'exécution. Pour interfacer avec le résultat d'une requête, contenu dans une variable temporaire, on a prévu une construction qui permet le balayage de la

variable temporaire : "for each v in V do .... end". Cette construction est équivalente au concept de curseur de system-R, mais, du fait du rapprochement du système de types entre LAMBDA et PASCAL, la syntaxe est plus agréable. Un point délicat est la manipulation de n-uplets "moléculaires" correspondantes aux agrégations d'entités. Pour celà, on a prévu d'effectuer des "for each" imbriqués. Par exemple, si V est une agrégation des entités U et W, alors, on écrira :

```
for each v in V do...

        for each u in U part-of v do..

        for each w in W part-of V do..
end;
```

Nous avons vu dans la section 7.3.3 que le résultat d'une requête ne manipulant pas les documents est une relation "plate", c'est à dire, dont les n-uplets sont normalisés. Les "for each" imbriqués doivent agir sur le résultat des procédures de normalisation. La construction "for each" est, elle aussi changée lors de la précompilation, par des énoncés PASCAL équivalents. On a prévu également de pouvoir faire des requêtes sur des variables temporaires. Pour plus de détails, voir [Gar 84].

7.5.2 - Interface conversationnelle graphique

Les utilisateurs finaux potentiels du système TIGRE seront en grande partie totalement nouveaux, sans aucune ou très peu de connaissances en matière d'informatique. Cette interface est adressée à cette classe d'utilisateurs.

Les utilisateurs disposent d'une visualisation graphique du schéma affiché sur l'écran. A partir de ce schéma, ils construisent progressivement la question. Le schéma peut être affiné à des niveaux de détail variables ; par exemple, affichage des attributs, des entités d'une hiérarchie de généralisation, de la structure d'un document. Les notions de trajets et variables LAMBDA disparaissent ; les usagers poin-

tent sur des classes du schéma qui seront reliées entre elles s'ils le souhaitent.

On pourra construire des sous-questions imbriquées, correspondantes aux ensembles LAMBDA. La formulation d'une question ne préjuge d'aucun modèle de formulation de questions. A n'importe quel moment, on pourra spécifier des conditions sur une classe, désigner une classe sur le schéma ou construire une sous-question.

Le logiciel réalisant cette interface fait accès au gestionnaire de descriptions pour demander des renseignements sur le schéma ; quand l'usager aura fini de poser sa question, il construira la requête LAMBDA, correspondante et il l'enverra au traducteur. Le résultat sera passé à l'interpréteur TIGRE. Enfin, celui-ci enverra les résultats de la requête au logiciel d'interface graphique qui se chargera de les afficher. Pour plus de détails, voir [Ant 84].

CHAPITRE 8

CONCLUSIONS

8. CONCLUSIONS

Il est souhaitable, en terminant une thèse, de revenir sur le thème de celle-ci pour faire le point. Notre but était d'étudier l'aspect modélisation et manipulation de données généralisées intervenant dans des nouvelles applications informatiques ayant des besoins de gestion de données. Nous avons choisi la bureautique comme application représentative en essayant cependant de tenir compte des exigences d'autres applications. Pour celà, plusieurs études étaient nécéssaires :

1. La définition d'un modèle de données capable de permettre la définition de ces données généralisées. Ceci a donné lieu à un travail collectif auquel l'auteur a activement participé. Le résultat de cette étude, le modèle TIGRE, est présenté au chapitre 3. Sa caractéristique principale est l'inclusion de la sémantique explicite nécéssaire au développement d'applications nouvelles. Dans TIGRE, on peut classer cette sémantique en deux groupes : celle associée aux nouvelles données généralisées et celle liée à l'environnement où ces données évoluent. Nous avons essayé de prendre en compte deux types de données généralisées : les documents multi-média qui circulent entre les correspondants d'une organisation et les données formatées à structure complexe qui apparaissent dans des applications orientées vers l'ingénierie. La sémantique nécéssaire à la représentation de l'environnement est assurée par l'intégration au modèle Entités-Association des concepts proposés dans le domaine de la modélisation sémantique de données (à savoir, généralisation et agrégation).

   Le modèle est extrémement riche en structures. Au delà de débats dogmatiques sur les vertus et les défauts de celles-ci, il semble indispensable d'observer expérimentalement les besoins des usagers dans des cas réels. Seule la pratique peut indiquer quelles combinaisons de structures de données sont les plus adaptées aux besoins des applications.

2. La conception d'un langage de manipulation de données généralisées associé à ce modèle. Les énoncés d'interrogation de ce langage, appelé LAMBDA ont été présentés au chapitre 4. La notion de trajet introduite permet d'exprimer des requêtes complexes de façon aisée en évitant des prédicats de jointure et en reflètant les liaisons sémantiques entre les données. La notation des trajets est homogène avec celle des chemins dans la structure hiérarchique d'un document. Ces chemins permettent de retrouver des sous-ensembles d'un document en fonction de son contenu ou de ses liaisons sémantiques avec le reste de la base.

Les énoncés de mise à jour ont été présentés au chapitre 5. La notion de transaction a été introduite ; le problème de la cohérence sémantique des données a été abordé et l'approche choisie est de permettre les mises à jour dans une classe et de les propager aux classes adjacentes. Les rapports avec le poste de travail pour la définition de transactions d'édition de documents ont été également étudiés.

La complétude du langage par rapport à une algèbre Entité-Association a été démontrée. Même si une telle mesure n'est pas entièrement satisfaisante (car le modèle TIGRE inclut d'autres concepts étendant le modèle Entité-Association), elle établit de façon concrète la puissance du langage. Une comparaison avec d'autres langages a été effectuée, tant du point de vue "modèle sémantique" que du point de vue manipulation de données généralisées.

De même que le modèle, il semble indispensable d'effectuer des expérimentations avec le langage. Elles permettraient de mettre en évidence les problèmes existants (bien qu'eles n'indiquent rien sur les modifications nécéssaires pour les faire disparaître). Cependant, il faut souligner que le langage n'est pas destiné à l'usager final d'une base de données généralisées mais qu'il est plutôt destiné à être intégré dans un langage de programmation possédant la notion de type.

3. La spécification du cadre de réalisation des énoncés de manipulation de LAMBDA. L'architecture générale du serveur TIGRE a été présentée au chapitre 7. Il est construit sur un SGBD relationnel et sur un module de stockage et d'accès aux documents multi-média. Nous avons insisté sur la traduction de requêtes LAMBDA et plus particulièrement sur les règles de génération d'arborescences TIGRE. Ces dernières sont composées d'opérateurs relationnels algébriques et d'opérateurs document que nous avons défini.

Il serait souhaitable de prolonger ce travail dans les directions suivantes :

1. Continuer la réalisation du langage de manipulation. Une première version du langage de requêtes ne comportant pas de documents ni d'ensembles a été implantée [PV 83a] ; une version assez complète du langage de définition a été réalisée [Pal 84].

2. Fournir des méthodes daccès pour la recherche par contenu dans les textes. En effet, la détection de termes dans un texte est une opération très coûteuse ; si l'on veut obtenir des performances acceptables, il est nécéssaire d'accélérer cette opération, soit en diminuant l'espace de recherche (par exemple, par la méthode des signatures), soit en accélérant la recherche par l'utilisation de matériel spécialisé, soit en combinant les deux.

3. Implanter le concept de transaction en LAMBDA. Dans les transactions qui ne sont pas des transactions d'édition de documents, les propriétés d'atomicité et de durabilité, ainsi que la synchronisation des accès concurrents à la base, seront garanties par les mécanismes du système relationnel sous-jacent. L'intégrité sémantique doit, par contre, être implantée dans le serveur TIGRE.

   Les transactions d'édition de documents posent un certain nombre de problèmes du point de vue de la concurrence et de la reprise en cas de panne.

D'abord, elles risquent de suspendre d'autres transactions pendant de longues périodes de temps. En effet, une transaction d'édition de documents peut avoir déjà verrouillé des données autres que le document qu'elle édite. Si le mécanisme du SGBD sous-jacent est tel que les transactions sont suspendues jusqu'à ce que le verrou soit libéré, on peut faire attendre longtemps d'autres transactions. Dans le but d'éviter ce problème sur les documents eux-mêmes, le mécanisme de contrôle de concurrence sur les documents (indépendant de celui du SGBD sous-jacent) doit éviter les suspensions de transactions qui demandent un document déjà verrouillé (en mode conflictuel) mais plutôt les informer.

D'autre part, comme ce sont des transactions qui peuvent durer longtemps, il est inconcevable que la reprise en cas de panne redémarre la transaction au début, car trop d'efforts et de temps seraient perdus. Des sauvegardes partielles devraient être possibles (l'équivalent d'une commande SAVE-TRANSACTION) à partir de l'éditeur. Il semblerait évident de sauvegarder les changements de manière incrémentable. Reste à savoir si cette technique demeure valide dans le cas de changement d'une grande partie d'un document (par exemple, si un document contient beaucoup d'images et que l'on rentre des nouvelles versions de ces images). En cas de panne, la transaction reprend à partir du dernier point de sauvegarde. Ceci impliquerait que les verrous sur les documents ou sur des parties de documents doivent subsister après une panne; ceci dans le but d'empêcher aux autres transactions de voir des documents non cohérents dans le cas où la transaction d'édition de documents a pu sauvegarder les documents en cours d'édition.

Pour conclure, nous considérons que notre travail a cherché à résoudre de nombreux problèmes d'ordre conceptuel qui se posent quand on veut décrire et manipuler des données généralisées dans une base de données. En ce sens, le modèle de données TIGRE et la conception du langage LAMBDA constituent une réponse à ces problèmes. Certes, nous ne prétendons pas les avoir résolus ni même abordés tous, mais nous pensons avoir défini un outil assez général pour permettre le développement de fonctionnalités particulières correspondantes à des applications diverses.

ANNEXES

ANNEXE 1

DEFINITION DU SCHEMA CONCEPTUEL

Définition des types construits

```
Type    Ident-employé : Integer;
        Ident-papier : Integer;
        Argent : (3000..50000);

        Date : Record
                  .   .
               | Jour : (1...31);
               | Mois : (1...12);
               | Année : (1958...1990);

               end;

        Adresse : Record
                  | Numéro : Integer;
                  | Rue, ville : string (20);

                  end;

        Destinataires : Array (1,4) of string (20)

        Paragraphe : document
                     structure
                     | Liste (1,*) of unité;

                     end;
                   end.
```

```
Doc-lettre : document
             structure

              begin
              |
              |  En-tête : paragraphe-structure;
              |  Destinataires : texte;
              |  Corps : paragraphe-structure;
              |  Signature : image;
              |  Adresse-société-expéditeur : texte;
              |
              end;

              constantes
              |
              |  Paramètre date : texte;
              |                   Société-expéditeur : texte;
              |                   Expéditeur : texte;
              |
              end;
             end;

Clause : document
         structure

          begin
          |
          |  Titre : texte;
          |  Contenu : paragraphe-structure ;
          |
          end;
         end;
```

```
Type  Contrat-Travail: document
                       structure

                         begin
                           |
                           | Contractants : begin
                           |                  |
                           |                  | Employeur : Text:=TO ;
                           |                  | Employé : référence:=Titulaire;
                           |                  |
                           |                end;
                           |
                           | Corps : List (1,*) of clause-structure;
                           | Signature : image ;
                           |
                         end;

                       constantes
                         |
                         | Unité TO := 'Compagnie ABC' ;
                         | Paramètre Titulaire : texte  ;
                         |           date-embauche : texte;
                         |           établissement : texte;
                         |           Niveau : texte;
                         | Fonction salaire-embauche : SALAIRE(niveau,
                         |                                     établissement);
                       end;
                       end;
```

Definition des types classe

```
Type FOURNISSEUR : Entity
  |
  |          nom : String(20);
  |       adresse : Adresse;
end;
```

```
Type  EMPLOYE : Entity
|
|       Key n°-id : Ident-employé : end key;
|           nom : string (20);
|           adresse : adresse;
|           salarié : argent;
|           catégorie : (Ing, Progr, Sec);
end;

Type  INGENIEUR : Specialization of EMPLOYE
|       Where catégorie : 'Ing';
|       Spécialité : (informatique, mécanique, automatique);
|       Ecole : (ENSERG, ENSIMAG, ENSIEG);
end;

Type  SECRETAIRE : Specialization of EMPLOYE
|       Where catégorie : 'Secr';
|       Bilingue : boolean;
end;

Type  PROGRAMMEUR : Specialization of EMPLOYE
|       Where catégorie : 'Progr';
|       Qualification : (senior, junior, débutant);
end;

Type EMPLOYE-DE-BUREAU : Union of SECRETAIRE and INGENIEUR manual
|                              and PROGRAMMEUR where qualification='senior
|                              manual
|     N° bureau : (100...500);
|     Bâtiment : (A,B,C,D,E);
end;
```

```
Type LETTRE : Entity
|
|  key N°-ref. : Ident-papier end key;
|      Expéditeur : string (2);
|      Liste-destinataires : destinataires;
|      Texte-lettre : Doc-lettre;
end;

Type CONTRAT : Entity
|
|  key N°-ref. : Ident-papier end key;
|      Date-signature : date;
|      Texte-contrat : contrat-travail;
end;

Type DOSSIER : Entity-agregation of LETTRE (0,*)
|                                  and CONTRAT (1,1);
|  key N°-dossier : Intent-papier end key;
|      Classeur :String(30);
end;

Type DEPARTEMENT : Entity
|
|     key n° : Integer end key;
|         nom : String(10);
|         budget : Argent;
end;

Type ARTICLE : Entity
|
|        key n°-ref : Integer end key;
|         nom : String(20);
|         couleur : (bleu, blanc, rouge, vert);
end;
```

```
Type EMP-DOSS : Relationship between EMPLOYE : propriétaire (1,1)
|                                   and DOSSIER : le-dossier (0,1);
end;

Type GESTION : Relationship between DOSSIER : dossier-géré (1,1)
|                                  and SECRETAIRE : responsable (0,20);
end;

Type AFFECTATION : Relationship between EMPLOYE : employés (1,1)
|                                  and DEPARTEMENT : dept-affecté (1,*);
|    date : Date;
end;

Type DIRECTION : Relationship  between INGENIEUR : chef (0,1)
|                                     and DEPARTEMENT : dept-dirigé (1,1);
|    date : Date;
end;

Type DEMANDE : Relationship between DEPARTEMENT : dept-demandant (0,*)
|                                  and ARTICLE : article-demandé (0,*);
|    quantité : Integer;
end;

Type FOURNITURE : Relationship between DEPARTEMENT : dept-fourni (1,*)
|                                     and ARTICLE :article-fourni (0,*)
|                                     and FOURNISSEUR: fournisseur (0,*);
|    date : Date;
|    quantite : Integer;
end;

Type STOCKE : Relationship between ARTICLE : article-stocké (1,*)
|                              and FOURNISSEUR : fournisseur-stockant(0,*);
|    quantité : Integer;
end;
```

ANNEXE 2

SYNTAXE DES ENONCES DE DEFINITION ET DE MANIPULATION DE LAMBDA

Nous utilisons une grammaire BNF étendue dont les nouveaux symboles sont :

[x] : 0 ou 1 instance de x	| x | y : x ou y
x* : 0 ou plus instance de x	(x | | y) : groupement des alternatives mutuellement exclusives
x^+ : 1 ou plus instance de x	
	'x' : x est un meta-symbole litéral

<Def-schéma-LAMBDA>::= 'Define'<nom-bd> <enonce-def-type>$^+$ 'end.'
<énoncé-def-type>::= ('type' <def-type>';')$^+$
<def-type>::= <ident-type>':' (<type-de-base> | <type-construit> | <type-classe>)
<ident-type>::= <ident>
<type de base>::= <type-de-base-simple> | <type-de-base-restreint>

<type-de-base-simple>::= 'integer' | 'real' | 'boolean' | 'string' '(' <numero-sans-signe>')'
<type-de-base-restreint>::= <type-salaire> | <type-intervalle>
<type-salaire>::= '(' <ident> (',' <ident>)$^+$ ')'
<type-intervalle>::= '(' <constante-numérique> '..' <constante-numérique>')'

<type-construit>::= <type-enregistrement> | <type-tableau> | <type-document>
<type-enregistrement> ::= 'record' (<def-constituant> ';')$^+$ 'end'
<def-constituant>::= <ident> ':' <type-de-base>
<type-tableau> ::= 'Array' '(' <constante-numérique> ')' 'of' <type-de-base>
<type-document> ::= 'Document' <def-document> 'end'
<def-document>::= <def-structure> [<def-constantes>]

```
<def-structure>::= 'structure' <constructeur-structure>
<constructeur-structure>::= 'begin' <noeud-struc>+ 'end ;'
                          | 'list' <cardinalité> 'of' <constructeur-struc>
                          | 'case' <selecteur> 'of' <choix>+ 'end;'
                          | <ident-type-doc> '.structure' ';'
                          | <unité> [':=' <ident-constante>] ';'

<noeud-struct>::= <nom-noeud> [(attr-doc>] ':' <constante-doc>
<nom-noeud>::= <ident>
<attr-doc>::= 'attr' <nom-attr> (',' <nom-attr>)*
<nom-attr>::= 'highligting' | 'key-word' | 'justification' | ...
<cardinalité>::= '(' <min> ',' <max> ')'
<min>::= <numéro-sans-signe>
<max>::= <numéro-sans-signe> | '*'
<sélecteur>::= <ident>
<choix>::= (<nom-cas> ':' <constructeur-struct>)+
<nom-cas>::= <ident> | <numéro-sans-signe>
<ident-type-doc>::= <ident>
<unité>::= 'text' | 'image' | 'graphics' | 'formula' | 'table' | 'voice' |
           'référence' | 'unit'

<def-constantes>::= 'constants' <decl-const> 'end' ';'
<decl-const>::= [<const-unité>] [<const-param>] [<const-fonct>]
              [<const-elem-assoc>]
<const-unité>::= 'unité' (<ident-const> <unité> ':=' <constante> ';')+
<constante>::= <chaîne-constante> | <expression-de-selection>
<const-param>::= 'paramètre'(<ident-const> ':' (<unité> | <ident-type> ';')+
<const-fonction>::= 'function'(<ident-const> ':' <nom-fonct>
                    [<param-form>]';')+
<param-form>::= '(' <ident> (','<ident>)* ')'
<const-elem-assoc>::= <nom-elem-assoc> <ident-const> ':=' <constante>
<nom-elem-assoc>::= 'footnote' | 'bibliographic-reference' | 'page-header'
                  | 'illustration'
<ident-const> } ::= <ident>
<nom-fonct>   }
```

```
<type-classe>::= <type-non-derivé> | <type-dérivé>

<type-non-dérivé>::= <type-entité-non-dérivé> | <type-assoc-non-dérivé>
<type-entité-non-dérivé>::= 'Entity' <attributs> 'end'
<attributs>::= [<attr-clé>] <def-attributs>+
<attr-clé>::= 'key' <def-attribut>+ 'end key' ';'
<def-attribut>::= <ident-attr> ':' <type-attr> [<specif-null>] ';'
<ident-attr>::= <ident>
<type-attr>::= <type-de-base> | <ident-type>
<specif-null>::= 'nulls-allowed' '(' (<chaîne-constante> |
                 <constante-numérique>) ')'

<type-assoc-non-derivé>::= 'Relationship' <entités-liées> <def-attribut>+
                           'end'
<entités-liées>::= 'between' <entité> ('and' <entité>)+ ';'
<entité>::= <ident-entité> [ ':' <ident-rôle> <cardinalité>
<ident-entité>} ::= <ident>
<ident-rôle>    }

<type-dérivé>::= <type-dérivé-par-gen> | <type-dérivé-par-agr>
<type-dérivé-par-gen>::= <type-spécialisé> | <type-union> |
                         <type-intersection>
<type-spécialisé>::= <type-entité-spécialisé> | <type-assoc-spécialisé>
<type-entité-spécialisé>::= 'spécialisation of' <ident-identité>
              [<prédicat-de-restriction>] ['manual'] <def-attribut>* 'end'
<prédicat-de-restriction>::= 'where' <prédicat> ('or' <prédicat>)*
<prédicat>::= <préd-simple> ('and' <pred-simple>*
<pred-simple>::= <ident-attr> (<op-scalaire> | <op-ensembliste>)<expression>
               | <attr-ident> ':' | <type-scalaire> | <type-intervalle>)
               | <ident-rôle> 'of' <ident-entité>
               | 'part-of' <ident-entité-agrégée>
<type-assoc-spécialisé>::= 'spécialisation of' <ident-assoc>
              [<prédicat-restriction>] ['manual'] <entités liées>
              <def-attribut>* 'end'
<ident-entité-agrégée>::=  <ident>
<ident-assoc>           }
```

<type-union>::= 'union of' <type-entité-restreint>
('and' <type-entité-restreint>)+ ['manual] <def-attribut>* 'end'
<type-entité-restreint>::= <ident-entité> [<prédicat-de-restriction>]
['manual']
<type-intersection>::= 'intersection of' <type-entité-restreint> ('and'
<type-entité-restreint>)+ ['manual'] <def-attribut>* 'end'
<type-dérivé-par-agr>::= <type-der-agreg-entités> | <type-der-agreg-assoc>
<type-der-agreg-entité>::= 'entity aggregation of' <ident-entité>
<cardinalité> ('and' <ident-entité>
<cardinalité>)* <def-attribut>+ 'end'
<type-der-agreg-assoc>::= 'relationship aggregation of' <ident-assoc>
<def-attribut>+ 'end'

ENONCES DE MANIPULATION :

<transaction-LAMBDA>::= 'begin-transaction' ';' <énoncé>+ 'end-transaction'
';' | <requête>+
<énoncé>::= <requête> | <édition-doc> | <énoncé-de-maj>
<édition-doc>::= [<affect-var>] 'edit' '(' <variable> ')' 'for'
('consulting' | 'update' | 'creation')
<affect-var>::= <variable> ['(' <ident> (',' <ident>)*')'] ':='

<requête>::= [<affect-var>] (<requête-un-fait> | <requête-ens-faits>)
<requête-un-fait>::= 'first-fact' '(' <requête-ens-faits> ')'
<requête-ens-faits>::= [<requête-ens-faits> ('*' | '-' | '+')]
<expression-de-selection> | '(' <requête-ens-faits>')'

<expression-de-sélection> ::= 'select' ['unique'] <expression-de-valeurs>
from <trajets-de-désignation> [<clause-where>]
<expression-de-valeurs> ::= <valeur> (',' <valeur>)*
<valeur> ::= <valeur-de-fait> | <valeur-inter-fait>
<valeur-de-fait> ::= <variable-design>
<valeur-inter-fait> ::= <valeur-non-derivé> | <valeur-fonct-agreg>
<valeur-non-derivé> ::= <valeur-attribut> | <valeur-attr-tableau> | <valeur-attr-
attr-enregistrement> | <valeur-attr-document>

<variable-design> ::= <identificateur>
<valeur-attribut> ::= <variable-design> '.' <ident-attribut>
<ident-attribut> ::= <identificateur>
<valeur-fonct-agreg> ::= <fonct-agreg> <ensemble>
<fonct-agreg> ::= 'COUNT' | 'MAX' | 'SUM' | 'MIN' | 'AVG'
<valeur-attr-enregistrement> ::= <valeur-attribut> '.' <ident-champ>
<ident-champ> ::= <identificateur>

<valeur-attr-document> ::= <chemin-dans-structure> <valeur-attribut>
| 'parameter' <ident-param> 'of' <valeur-attribut>
| 'eval' <ident-fonction> ['('<param-actuel>
(',' <param-actuel>)*')'] 'of' <valeur-attribut>
<chemin-dans-structure> ::= <chemin-arborescent> [<chemin-linéaire>]
| [<chemin-arborescent>] <chemin-linéaire>
<chemin-arborescent> ::= '(' <branche-chemin> (',' <branche-chemin>)+ ')' 'of'
<branche-chemin> ::= [<chemin-dans-structure> <noeud-agregat>
<chemin-linéaire> ::= (<nom-noeud> [<elem>] 'of')* <nom-noeud> [<elem>] 'of'
<nom-noeud> ::= <noeud-agregat> | <type-doc> | <unité> | <valeur-de-sélecteur>
| <nom-constructeur>

<noeud-agrégat>
<type-doc>
<valeur-de-sélecteur>
<nom-de-sélecteur>
::= <identificateur>

<unité> ::= 'texte' | 'image' | 'graphique' | 'formule' | 'voix' | 'tableau'
<nom-constructeur> ::= 'agrégat' | 'liste' | 'cas'
<elem> ::= <entier> | <entier> (',' <entier>)+ | <entier> '..'(<entier> | '*')
| <variable-doc>
<variable-doc> ::= <identificateur>
<ident-param> ::= <identificateur>
<ident-fonction> ::= <identificateur>
<param-actuel> ::= <variable> | <constante-de-base>

<trajets-de-désignation> ::= <trajet> (',' <trajet>)*
<trajet> ::= <trajet-racine-entité> | <trajet-racine-association>

<trajet-racine-entité> ::= [<sous-trajet>] <racine-entité>
<racine-entité> ::= <ident-entité> <variable-desig>
<sous-trajet> ::= [<pas-vers-assoc> 'of'] <sous-trajet-linéaire>
| <sous-trajet-arborescent> [<sous-trajet-linéaire>]
<sous-trajet-linéaire> ::= <pas>$^{+}$
<pas> ::= <via-assoc> 'of' | <via-agreg> ('containing ' | ' part-of')
<via-assoc> ::= <ident-role> [<variable-desig>]
<via-agreg> ::= <ident-entité> [<variable-desig>]
<ident-role> ::= <identificateur>
<ident-entité> ::= <identificateur>
<sous-trajet-arborescent> ::= <pas-arborescent>
| '('<pas-arborescent> (',' <pas-arborescent>)$^{+}$')'
<pas-arborescent> ::= <pas-arborescent-assoc> | <pas-arborescent-agregat>
| <pas-arborescent-agrégé>
<pas-arborescent-assoc> ::= (<arbre-assoc> | <branche-assoc>) 'of'
<arbre-assoc> ::= [<pas-vers-assoc>'of'] '('<branche-assoc>
(','<branche-assoc>)$^{+}$')'
<branche-assoc> ::= [<sous-trajet>] <via-assoc> | <pas-vers-assoc>
| <arbre-assoc>
<pas-vers-assoc> ::= <ident-assoc> <variable-desig> ':('<variable-desig>
(',' <variable-desig>)$^{+}$ ')'
<ident-assoc> ::= <identicateur>
<pas-arborescent-agrégat> ::= (<arbre-agreg> | <branche-agreg>) 'part-of'
<pas-arborescent-agrégé> ::= (<arbre-agreg> | <branche-agreg>) 'containing'
<arbre-agreg> ::= '(' <branche-agreg> (',' <branche-agreg>$^{+}$')'
<branche-agreg> ::= [<sous-trajet>] <via-agreg>
<trajet-racine-association> ::= <ident-assoc> [<variable-design> ':']
['(' <ident-role> <variable-desig>
(',' <ident-role> <variable-desig>)$^{+}$ ')']

<clause-where> ::= 'where' <conditions>.
<conditions> ::= <conditions> ('and' | 'or') <conditions>
| 'not' <conditions> | '(' <conditions>')'
| (<valeur> <op-scalaire> | <chemin-linéaire> <variable-doc>)
(<valeur> | <expression>)

```
                | (<valeur> | <expression> | <expression-de-valeurs>)
                  'in' <ensemble>
                | <ensemble> <op-ensembliste> <ensemble>

<op-scalaire> ::= '=' | '^=' | '>' | '> =' | '<' | '< ='
<op-ensembliste> ::= '=' | '^=' | '<' | '< ='

<ensemble> ::= <ens-non-indexé> | <ens-indexé> | <ens-tableau> |
               <ens-unité-texte> | <ens-construit> | <ens-constant>
               | <ident-entité> | '(' <ensemble> ')'

<ens-non-indexé> ::= '[' <def-ens-non-ind> ']' | ' ' <def-ens-non-indexé> ' '
<def-ens-non-indexé> ::= <expression-de-valeurs> [<clause-where>]
<ens-indexé> ::= <ens-non-indexé> 'by' <variable-design>
<ens-tableau> ::= <valeur-attr>
<ens-unités-texte> ::= <chemin-dans-struc> ( <valeur-attr> | <variable-doc> )
<ens-construit> ::= <ensemble> ('+' | '*' | '-') <ensemble>
<ens-constant> ::= ' ' <constante-numérique> (',' <constante-numérique>)+ ' '
               | ' ' <chaîne-constante> (',' <chaîne-constante>)+ ' '
               | ' ' <constante-enreg> (',' <constante-enreg>)+ ' '
               | ' ' <constante-tableau> (',' <constante-tableau>)+ ' '
               | ' ' <variable> (',' <variable>)+ ' ' | ' '

<énoncé-de-mise-à-jour> ::= <insertion> | <suppression> | <modification>

<insertion> ::= 'insert' <fait-designé> 'to' <ident-classe>
              ('and' <fait-désigné> to (<ident-classe>
              | <via-agreg>,'part-of' <ident-classe>))*
               ['from' <trajets-de-désignation> [<clause-where>]]
<fait-désigné> ::= <variable-classe> | [<variable-classe> ':'] <fait>
                 | <variable-design> (',' <variable-design>)* 'with' <fait>
<variable-classe> ::= <variable-design> | '('<variable-design>
                                          (','<variable-desig>)+ ')'
<fait> ::= '('<ident-attribut> ':'<expression>
         (','<ident-attribut> ':'<expression>)* ')'
```

```
<suppression> ::= 'delete' <variable-desig> 'from' <trajets-de-désignation>
                  [<clause-where>]

<modification> ::= 'replace' <corps-modif> (','<corps-modif>)*
                  ['and' 'insert'<fait-désigné> 'to' <ident-classe>]
                   'from' <trajets-de-désignation> [<clause-where>]

<corps-modif> ::= (<valeur-non-dérivé> | <via-agreg> 'part-of'
                   <variable-design>) ':=' [('add' | 'remove' | 'share'
                                                | 'copy')] <expression>

<expression> ::= <constante-de-base> | <constante-tableau> | <constante-enreg>
                | <variable> | <expr-reg-chaine> | 'null'

<constante-de-base> ::= <constante-numérique> | <constante-bool>
                        | <chaîne-constante>
<constante-numérique> ::= [('+' | '-')] <numéro-sans-signe>
<numéro-sans-signe> :: = <entier> ['.' <entier>]
<entier> ::= <chiffre>+
<chaîne-constante> ::= ' " ' <caractère>+ ' " '
<constante-bool> ::= 'vrai' | 'faux'
<constante-tableau> ::= '(' <chaîne-constante> (',' <chaîne-constante>)+ ')'
                        | '(' <constante-numérique> (',' <constante-numérique>)+')'
                        | '(' <constante-bool> (',' <constante-bool>)+ ')'
<constante-enreg> ::= '(' <constante-de-base> (',' <constante-de-base>)+')'
<expr-reg-chaine> ::= ['*'] <chaine-constante> ('*' <chaine-constante> )* ['*']
<variable> ::= <identificateur>
```

ANNEXE 3

COMPATIBILITE DE TYPES

Dans ce qui suit, nous présentons les règles qui déterminent la compatibilité de deux types. Nous présentons également les opérateurs de comparaison définis sur les types et sur les ensembles.

A. Règles de compatibilité

(0) Un type est compatible avec lui-même si l'opérateur d'égalité est défini entre les instances du type.

(1) Types de base simples

Les quatre types de base simples (Entier, réel, Booléen et chaîne de caractères) sont incompatibles entre eux.

(2) Types de base Restreints

- les types intervalles d'entiers (réels) sont compatibles avec le type entier (réel),
- deux types intervalles d'entiers (réels) sont compatibles si leur intersection n'est pas vide,
- les types scalaires sont compatibles avec le type chaîne de caractères,
- deux types scalaires sont compatibles si leur intersection n'est pas vide.

Types construits

(3) Types renommés

un type renommé n'est compatible qu'avec lui-même et avec le type

de base sous-jacent. Par exemple, le type Argent et le type Entier sont compatibles ; les types renommés Ident-papier et Argent sont incompatibles.

(4) Types tableau

Un type tableau T est compatible avec des ensemnbles LAMBDA dont le type de valeurs est compatible avec le type de base sur lequel le tableau est construit. Il n'est pas compatible avec aucun type tableau T' différent de T.

Par exemple, on pourra écrire en LAMBDA la condition

$$1.\text{liste-destinataires} \gg \{e.\text{nom}\}$$

où 1 désigne l'entité LETTRE et e désigne l'entité EMPLOYE. En effet, l'attribut "nom" de EMPLOYE est défini sur le type String(20) qui coïncide avec le type de base sur lequel le type tableau Destinataires est construit, ce dernier étant le type de l'attribut "liste-destinataires".

(5) Types enregistrement

Soit R un type enregistrement défini sur des constituants ayant des types de base $t_1,\ldots,t_n$. Soit T le type de valeurs d'un ensemble LAMBDA. Ce dernier a la forme (cf. section 2.3)

$$T = \text{type}(g_1) \text{ X } \ldots \text{ X } \text{type}(g_r)$$

où les fonctions $g_1,\ldots,g_r$ forment l'expression de valeurs de l'ensemble. Alors, R et T sont compatibles si $n = r$ et si t_i est compatible avec $\text{type}(g_i)$ pour $i = 1,\ldots,n$.

Par ailleurs R n'est pas compatible avec aucun type enregistrement R' différent de R.

Par exemple, on pourra former en LAMBDA la condition

$$e.\text{adresse } \underline{\text{in}} \left\{ \begin{array}{l} (15, \text{Gal. Ferrié, Grenoble}) \\ (23, \text{Bd. Foch, Lille}) \\ (43, \text{Bd. Raspail, Paris}) \\ (4, \text{Rue de Vaugirard, Toulouse}) \end{array} \right\}$$

où e désigne l'entité EMPLOYE. Le type Adresse est défini comme un enregistrement composé de Numéro : Integer, et Rue, Ville : String(20). D'autre part, le type de valeurs de l'ensemble est un type T tel que

$$VS(T) = VS(\text{Integer}) \times VS(\text{String}(20)) \times VS(\text{String}(20))$$

On voit bien qu'il y a compatibilité entre T et Adresse.

(6) Types Document

Les seuls opérateurs définis sur les types document sont edit, qui appelle l'éditeur de documents, mail, qui envoie les documents par le courrier électronique et print qui imprime les documents dans le serveur d'impression. Il n'y a donc pas d'opérateurs de comparaison définis sur ces types, même pas l'égalité. L'affectation est définie entre (sous-)documents de type compatible. Les unités de nature textuelle sont les seules où l'on a défini les opérateurs de comparaison =, ≠. La nature texte est compatible avec le type chaîne de caractères. On pourra donc comparer, par exemple, le titulaire d'un contrat (un paramètre texte du contrat) avec un nom d'employé.

(7) Types de valeurs des ensembles LAMBDA

Soit t_1 le type de valeurs d'un ensemble s_1 et t_2 le type de valeurs d'un ensemble s_2. Les types t_1 et t_2 ont la forme

$$t_1 = type(g_1) \ X \ ... \ X \ type(g_m)$$

$$et \ t_2 = type(h_1) \ X \ ... \ X \ type(h_k)$$

Les types t_1 et t_2 seront compatibles si m=k et si $type(g_i)$ est compatible avec $type(h_i)$ pour i = 1,...,m.

Si ces types sont compatibles on pourra comparer s_1 et s_2 avec les opérateurs de comparaison d'ensembles =, $\neq$, $\supset$, $\not\supset$, et on pourra construire des nouveaux ensembles à partir de s_1 et s_2 à l'aide des opérateurs ensemblistes union, intersection et différence.

Nous avons vu dans 4.4.1. qu'une entité E peut être considérée comme un ensemble dans les énoncés de manipulation de LAMBDA. Dans ce cas, le type de valeurs de l'ensemble E coïncide avec le type t_1 X ... X t_n où t_i est le type de l'attribut a_i de E. Une variable de désignation qui désigne l'entité E a donc implicitement ce type.

B. Opérateurs de comparaison

Types	Opérateurs	type d'opérateurs
Entier, réel, intervalles	=, ≠, >, ≥, <, ≤	scalaires
Booléens Chaînes de caractères types scalaires types enregistrement unités de nature texte type d'une variable de désignation	=, ≠	scalaires
types renommés	opérateurs du type sous-jacent	scalaires
types tableau ensembles LAMBDA	=, ≠, >, ≥	ensemblistes

ANNEXE 4

DESCRIPTION DES RELATIONS DU CATALOGUE

1. Domaines

CAT-D(db-c, D-c, dom-name, of-type, data-type)

CAT-D est la relation centrale pour la représentation des domaines. Ceux-ci seront référencés en interne par leur surrogate. Le premier accès au catalogue concernant un domaine est cette relation . Avec les informations obtenues à partir de CAT-D, on peut décider si des nouveaux accès au catalogue auront lieu pour compléter la description du domaine.

Constituants

db-c :surrogate de la base de données

D-c : Surrogate du domaine ; clé primaire de la relation

dom-name : nom du domaine. Valeur : "not-named" si le domaine n'est pas défini comme un type LAMBDA.

of-type : donne le type du domaine. Celui-ci peut être un type de base simple, un constructeur, ou un type-classe. "Interval" et "scalar" sont considerés en interne comme des constructeurs. Les valeurs possibles sont donc: integer, real, boolean, string, scalar, interval, array, record, document, entity, relationship.

data-type : donne le type de base simple sur lequel le domaine est défini du point de vue du SGBD relationnel. Donc, pour les types construits, la valeur du constituant data-type sera dom-E , étant donné que pour le SGBD

relationnel, ce sont des surrogates ce que l'on stocke dans les champs d'un attribut défini sur un type construit.

2. Domaine intervalle

CAT-INTD (db-c, D-c, min, max).

Cette relation sert à maintenir les limites des domaines du type intervalle.

D-c : surrogate du domaine ; clef primaire de la relation.
min : entier qui donne la borne inférieure de l'intervalle.
max : entier qui donne la borne supérieure de l'intervalle.

3. Domaine scalaire

CAT-SCAD (db-c, D-c, element).

Cette relation est similaire à la précédente. Il y aura un n-uplet par élément scalaire du domaine.

D-c : surrogate du domaine ; clef primaire de la relation.
element: valeur d'un élément du domaine scalaire.

4. Relations

CAT-R (db-c, R-c, rel-name, rel-type)

Le rôle des nuplets que l'on stocke dans CAT-R est double. D'une part, ils contribuent à la représentation dans le catalogue des définitions des types construits tableau et enregistrement et les types-clase, dérivés ou non. D'autre part, ils indiquent le schéma relationnel des relations qui stockeront, dans la base, des instances des types con-

struits et des faits des types-classe. Donc, à chaque relation de la base correspond un n-uplet dans la relation CAT-R. Les relations de la base seront toujours référencées en interne par leur surrogate.

R-c : surrogate de la relation de la base ; clé primaire de CAT-R.
rel-name : nom de la relation.
rel-type : décrit le type de la relation.
valeurs: E pour une relation-E :

ER pour un type-enregistrement
ED pour un type-document
EL pour un type-tableau
EK pour un type-entité non-derivé (type-entité "inner kernel" en RM/T.)
ES pour un type-entité dérivé par spécialisation ("kernel subtype" en RM/T).
EU pour un type-entité derivé par union
EI pour un type-entité derivé par intersection
EE pour un type-entité derivé par agrégation d'entités
AA pour un type-entité derivé par agrégation associative
EA pour un type-association

A pour une relation associative, c'est à dire qui lie des faits des entités liées sémantiquement par une association

P pour une relation qui a pour fonction la représentation des atributs d'un fait d'une classe ou d'une instance d'un type tableau ou enregistrement.

G pour une relation qui a comme fonction de stocker les faits agregés dans chaque fait d'une entité derivée par agrégation d'entités.

5. Attributs

CAT-A(db-c, A-c, R-c, D-c, att-name, E-ref, user-key, nulls)

A chaque attribut d'une relation de la base correspond un n-uplet dans CAT-A. Les attributs seront référencés en interne par leur surrogate.

A-c : surrogate de l'attribut ; clé primaire de CAT-A
R-c : surrogate de la relation à laquelle l'attribut appartient
D-c : surrogate du domaine sur lequel l'attribut est défini
att-name: nom de l'attribut
E-ref : indique le surrogate de la relation-E qui doit contenir les valeurs de l'attribut, celui-ci étant défini sur dom-E valeurs : R-c, null
user-key : valeurs : T, F.
nulls : indique la possibilité que l'attribut prenne la valeur nulle. valeurs : T, F.

6. Valeurs par défaut

CAT-VDEFAUT(db-c, A-c, val)

Cette relation stocke la valeur par défaut prise par un attribut que l'on permet de prendre la valeur nulle.

7. Attribut défini sur le type string

CAT-STRING (db-c, A-c, length)

La relation CAT-STRING donne la longueur maximale admissible pour chaque chaîne de caractères de l'attribut identifie par le surrogate.

A-c : surrogate de l'attribut
Length: longueur maximale admissible

8. Domaines construits et classes

CAT-STRUC (db-c, D-c, R-c)

Cette relation lie la représentation dans CAT-D d'un type-classe ou construit, identifié par D-c, avec sa représentation dans CAT-R identifiée par R-c. Ce dernier identifie une relation-E. La relation-P correspondante complète la définition du type identifié par D-c.

D-c : surrogate du domaine
R-c : surrogate de la relation-E.

9. Liaison entre la relation-E et les relations-P ou A

CAT-COMP(db-c, R-prop-c, Re-c)

Elle associe une relation-E avec une (dans le prototype) ou plusieurs relations-P. Elle associe également une relation de type EA (une association) avec sa relation-A, et une relation de type EE (une agrégation d'entités) avec sa relation-G.

R-prop-c :surrogate de la relation P ou A, clé primaire.
Re-c : surrogate de la relation-E.

10. type tableau

CAT-LIST (db-c, D-c, n°-of-elements)

Cette relation maintient la cardinalité du tableau.

D-c : surrogate du domaine
n°-of-elements : cardinalité du tableau.

11. Types document

CAT-DOC (db-c, D-c, liste)

Cette relation stocke la définition d'un type document identifié par D-c

D-c : surrogate du type document
liste : référence à la liste parenthesée définissant le type document

12. Associations

CAT-DESIG (db-c, Dr-c, De-c, rôle, min, max)

Pour la représentation des types-associations nous avons besoin de stocker de l'information sur les rôles les types-entités associés, ainsi que la cardinalité des associations.

Dr-c : surrogate du type-association
De-c : surrogate d'un des types-entités associés
rôle : nom du rôle du type-entité identifié par De-c vis-a-vis du type-association référencé par Dr-c.
min : borne inférieure de la cardinalité
max : borne supérieure de la cardinalité

13. Prédicats simples

CAT-PS (db-c, PS-c, group-n°, refinement, P-c)

Les prédicats simples sont référencés par un surrogate et représentés par cette relation. L'attribut "refinement" indique le type de raffinement induit par ce prédicat simple. Cette information indique la relation du catalogue à laquelle il faut accéder pour compléter l'information sur le prédicat. Les types de raffinement sont : raffi-

nements par valeur, par domaine, par association et par agrégation.

PS-c : surrogate du prédicat simple - clé de la relation

P-c : surrogate du prédicat composé auquel le prédicat simple appartient

refinement: type de raffinement induit par le prédicat simple

group-n° : numéro du groupe dans la forme normale disjonctive du prédicat P-c auquel le prédicat simple PS-c apparaît.

14. Raffinement par valeur

CAT-P-VAL (db-c, PS-c, Ac, opérateur, valeur)

A-c : surrogate de l'attribut sur lequel porte le prédicat référencé par PS-c

opérateur : un des opérateurs admisibles pour ce type d'attribut

valeur : chaîne de caractères contenant une valeur compatible avec le type de l'attribut.

15. Raffinement par domaine

CAT-P-DOM (db-c, PS-c, A-c, D-c)

A-c : surrogate de l'attribut sur lequel porte le prédicat simple référencé par PS-c

D-c : surrogate du domaine intervalle ou scalaire.

16. Raffinement par association

CAT-P-ASS (db-c, PS-c, D-c, rôle)

rôle : nom du rôle du type-entité sur lequel porte le prédicat simple référencé par PS-c vis à vis de la liaison avec le type-entité référencé par D-c.

17. Raffinement par agrégation

CAT-P-AGR (db-c, PS-c, D-c)

D-c : surrogate du type-entité agregé.

18. Prédicats de restriction composés

CAT-P-COMP(db-c, P-c, D-c, manual)

Les prédicats ont la forme générale

C = $(P_{11}$ et P_{12} et P_{13} ... et $P_{1m})$
ou $(P_{21}$ et P_{22} et P_{23} ... et $P_{2n})$ ...
ou $(P_{j1}$ et P_{j2} et P_{j2} ... et $P_{jo})$

P-c : surogate du prédicat composé -clé de la relation
D-c : surrogate du type-classe sur lequel porte le prédicat
manual : Si la valeur est T, ceci indique qu'en plus de satisfaire le prédicat indiquée par la combinaison de prédicats simples, seules les faits explicitement insérées dans la classe dérivée appartiendront à celle-ci. valeurs: T, F.

19. Dérivation par généralisation

CAT-GEN (db-c, Dresult-c, Doper-c, Operator, P-c)

Dresult-c : surrogate identifiant le type-classe dérivée
Doper-c : surrogate identifiant le type-classe opérand. Il y a un opérand pour une spécialisation et deux ou plus opérands (qui correspondent à deux ou plusieurs n-

uplets dans CAT-GEN) pour une union ou une intersection

Operator : l'opérateur employé dans la dérivation
valeurs : spécialisation, union, intersection

P-c : surrogate du prédicat de restriction composé du type dérivé (s'il existe)

20. Graphe de dérivation par généralisation

CAT-ANCETRES (db-c, Dd-c, Da-c, att-her)

Cette relation facilite la recherche des ancetres d'un type-classe dérivé donnée (Dd-c) lors de la traduction d'une requête LAMBDA. Il représente le graphe de dérivation.

Dd-c : surrogate du domaine identifiant la classe derivé
Da-c : surrogate du domaine identifiant la classe ancêtre
att-her : indique si les attributs de cet ancêtre (Da-c) doivent être herités par la classe dérivée (Dd-c). Les règles d'héritage sont spécifiées dans la section 3.3.1.

21. dérivation par agrégation d'entités

CAT-EAGG(db-c, Dagg-c, Dcomp-c, min, max)

Cette relation a pour but de stocker les entités qui peuvent appartenir à l'agrégation et ses contraintes de cardinalité.

Dagg-c : surrogate qui identifie le type-entité agrégé
Dcomp-c : surrogate qui identifie un type-entité qui appartient à l'aggrégation
min : la borne inférieure de la cardinalité
max : la borne supérieure de la cardinalité.

22. dérivation par agrégation associative

CAT-AAGG(db-c, Dagg-c, Dassoc-c)

Cette relation associe une entité derivée par agrégation associative et l'association sousjacente.

Dagg-c : surrogate de l'entité derivée par agrégation associative.
Dassoc-c : surrogate de l'association sousjacente

REFERENCES BIBLIOGRAPHIQUES GENERALES

[AA 84]

Adiba M., Azrou F.
An authorization mechanism for generalized DBMS
3rd. Seminar on distributed data sharing systems, Parma, Italy, March 1984

[Abr 74]

Abrial J.R.
Data Semantics
In Data Base Management, Eds. Klimbie & Koffman, North Holland, 1974

[ABCE 76]

Astrahan M et al.
System R: A relational approach to data base management
ACM TODS Vol. 1, N°. 2, Juin 1976

[ABDR 82]

Adiba M., Burnier M., Delobel C., Rialhe D.
Bases de données et nouvelles applications
Rapport de Recherche N° 349
Laboratoire IMAG, Grenoble, Fev. 1983.

[AC 81]

Azteni P., Chen P.P-S
Completeness of query languages for the Entity-relationship model, in [Che 81]

[Adi 82]

Adiba M.
Derived Relations : An Unified mechanism for views, snapshots and distributed data
Proc. of the 7th. Int. Conf. on VLDB, Cannes, France, 1981

[Ant 84]

Antunes F.
Une interface graphique pour une base de données généralisées
Rapport de DEA, INPG, Grenoble, en cours de préparation

[ASH 76]

Allman E., Stonebraker M., Held G.

Embedding a relational data sublanguage in a general purpose programming language

Proc. on a conf. on data: abstraction, definition and structure , SIGPLAN notices, 11, special issue, March 1976

[Azr 84]

Azrou F.

Un mécanisme d'autorisation pour les Bases de Données Généralisées, Thèse de 3ème cycle, INP de Grenoble, Juin 1984

[Bal 84]

Baltas S.

Accès concurrents aux documents

Rapport de Recherche TIGRE N° 10

Laboratoire IMAG et CII Honeywell Bull, Grenoble, Janvier 1984

[BD3 83]

Groupe BD3

Bases de données: nouvelles perspectives

INRIA - ADI, Jan. 1983

[BF 79]

Buneman P., Fraenkel R.E.

FQL - A functional query language

Proc. ACM SIGMOD Conf., Boston, Mass., Juin 1979

[Ber 81]

Bert D.

Software components construction

Tools and notions for program construction, Cambridge univ. Press, ed. D. Neel, 1982

[BR 82]

Bancilhon F., Richard P.

TQL : A textual query language

Rapport Interne INRIA, 1982

[BS 81]

Bancilhon F., Spyratos N.

Update semantics of relational views

Rapport de Recherche INRIA N° 329, Oct. 1978

[BS 82]

Bancilhon F., Scholl M.

Design of a Back End Processor for a Data Base Machine

Actes ACM SIGMOD, Los Angeles, 1980.

[BRA 79]

Bratbergsengen K., Risnes O., Amble T.

ASTRAL: A structured and unified approach to data base design and manipulation.

IFIP TC-2 working conference on data base architecture

Venise, Juin 1979

[Bro 80]

Brodie M.

The application of data types to data base semantic integrity, Information systems, vol. 5, pag 287, 1980

[Bro 81]

Brodie M.

Association, a data base abstraction for semantic modeling in [Che 81]

[Bro 82]

Brodie M.

Axiomatic definitions for data model semantics

Information systems, vol. 7, N° 2, pag 183, 1982

[Bru 81]

Bruandet M. F.

Notion de concept pour la construction automatique d'un thesaurus évolutif

Congrès AFCET, Nov. 1981.

[BRV 83]

Bogo G., Richy H., Vatton I.

Proposition de modèle pour la manipulation de documents.

Rapport de Recherche TIGRE N° 3

Cii Honeywell Bull, Grenoble, Novembre 1982.

[CCZ 83]

Crampes J.B., Chrisment C.Y., Zurfluh G.

The BIG project

Proc. of the 2nd. Int. Conf. on Databases (ICOD-2)

Cambridge, Sept. 1983

[CF 84]
Christodoulakis S., Faloutsos C.
Performance considerations for a message file server
IEEE Trans. softw. eng., Mars 1984

[CFDL 82]
Chan A., Danberg S., Fox S., Lin W., Nori A., Ries D.
Storage and acces structures to support a semantic data model
Proc. of the 8th. Int. Conf. on VLDB, Mexico City, 1982

[Che 76]
Chen P.P-S
The Entity Relationship Model - Toward a unified view of data
ACM TODS, Vol. 1, Number 1, 1976

[Che 79]
Chen P.P-S (ed.)
Proc. of the 1st. Int. Conf. on the Entity-Relationship Approach to systems analysis and design
Los Angeles, California, December 1979 (North Holland)

[Che 81]
Chen P.P-S (ed.)
Proc. of the 2nd. Int. Conference on Entity-Relationship Approach to Information modeling and analysis
Washington, D.C., 1981 (North Holland)

[Cod 79]
Codd E.F
Extending the data base relational model to capture more meaning.
ACM TODS, Vol. 4, N° 4, 1979

[Col 83]
Collet C.
Caracterization des formulaires pour une base de données généralisées, Rapport de Recherche TIGRE N° 9
Laboratoire IMAG ET CII Honeywell Bull, Grenoble, Dec. 1983

[CVLL 84]
Christodoulakis S., Vanderbroek J. et al.
Development of a Multimedia Information system for an office environment
Proc. of the 10th. Int. Conf. on VLDB, Singapour, Aug.1984

[Dar 84]
Dardailler D.
Un éditeur interactif de tableaux
Rapport de DEA, INPG, Grenoble, Juin 1984

[Dat 82]
Date C.J.
Null values in Data Base Management
2nd. British Nat. Conf. on Data Bases
University of Bristol, July 1982

[DJNY 83]
Davis C., Jajodia S., Ng P., Yeh R.
Proc. of the 3rd. Int. Conf. on Entity-Relationship approach to software engineering
Anaheim, California, Oct. 1983

[Ecma]
ECMA / TC29 / 83 / 56
Office Document Architecture
Fourh Working draft

[EN 79]
Ellis C. and Nutt G.
Computer Science and Office Information Systems.
Report SSL-79-6
XEROX PARC, Palo Alto, California, June 1979.

[EGLT 76]
Eswaran M., Gray J., Lorie R., Traiger I.
The notions of consistency and predicate locks in a data base system, CACM, Nov. 1976

[Elm 81]
Elmasri R.
GORDAS : a data definition, query and update language for the entity-category relationship model of data
Honeywell Research Report, Bloomington, Minnesota, 1981

[Gar 84]
Garcia C.
Intégration d'un langage de manipulation de bases de données à un langage de programmation.
Rapport de DEA, IMAG, Grenoble, en cours de préparation

[Gra 81]
Gray J.
The Transaction concept: virtues and limitations
Proc. of the 7th. Int. Conf. on VLDB, Cannes, France, 1981

[GLPT 76]
Gray J., Lorie R., Putzolu G., Traiger I.
Granularity of locks and degrees of consistency in a shared data base, in Information Processing, North Holland, 1976

[Haf 81]
Hafner C.D.
Representation of knowledge in a legal information retrieval system, in [Odd 81]

[HL 81]
Haskin L., Lorie R.
On extending the functions of a relational database system
Research report RJ3182(38988)
IBM Research laboratory, San Jose, California, 1981

[HM 81]
Hammer M., MacLeod D.
Data Base description with SDM: A Semantic Data Model
ACM TODS, Vol 6, Number 3, 1981

[Hor 84]
Horak W., Kronert G.
An object oriented office document architecture model for processing and interchange of documents
Proc. of the 2nd. ACM-SIGOA conference on Office Information Systems, Toronto, June 1984.

[JLV 83]
Joloboff V., Lopez M., Velez F.
A Generalized Data Base Model
Rapport Interne, Centre de Recherche BULL, Grenoble, Janvier 1983.

[Jol 82]
Joloboff V.
UBIK: Un formalisme de modèlisation
Séminaire Bases de Données, ADI, Toulouse, Novembre 1982.

[Jol 84]
Joloboff V.
An interactive illustration editor for document preparation
ACM Conference on small systems and personal computers
San Diego, Dec.1983

[Ken 78]
Kent W.
Data and reality
North Holland, 1978

[KL 82]
Kowarski I., Lopez M.
The Document concept in a database
Proc. of the ACM SIGMOD Conference
Orlando, Florida, 1982

[KLMP 83]
Kim W., Lorie R., McNabb D., Plouffe W.
Nested transactions for engineering design databases
IBM Research Report RJ 3934 (44534) Juin 1983

[Lar 83]
Larson P.
A method for speeding up text retrieval
Proc. ACM SIGMOD, Mai 1983

[LKMP 84]
Lorie R., Kim W., McNabb D., Plouffe W.
User interface and access techniques for engineering databases
IBM Research Report RJ 4155 (45943) Jan. 1984

[LMWW 79]
Lockemann P., Mayr H., Weil W., Wohlleber W.
Data abstractions for Data Base Systems
ACM TODS vol. 4, N° 1, March 1979

[LP 82]
Lorie R., Plouffe W.
Complex objects and their use in design transactions
Research Report RJ 3706 (42922)
IBM Research Laboratory, San Jose, California, August 1982

[LPV 83]
Lopez M., Palazzo-Oliveira J., Velez F.
The TIGRE Data Model
Rapport de Recherche TIGRE N° 2
Laboratoire IMAG et CII Honeywell Bull, Grenoble, Sept. 1983

[LS 83]
Lum V., Schek H.
Complex data objets: text, voice, images, can DBMS manage them?, Proc of the 9th. Conf. on VLDB, Florence, Nov. 1983

[LST 81]
Lien E., Shopiro J., Tsur S.
DSIS - A database system with Interrelational semantics
Proc. of the 7th. Int. Conf. on VLDB, Cannes, 1981

[LV 84]
Lopez M., Velez F.
Modeling and Handling generalized data in the TIGRE Project
Proc. of the 8th. Honeywell Int. Conf. on Computer Science
Bloomington, Minnesota, Mai 1983

[McG 81]
McGregor D., Malone J.
The fact Data Base: A System based on Inferential Methods
in [Odd 81]

[MR 83a]
Markowitz V.M., Raz Y.
A modified Relational Algebra and its use in an Entity-Relationship environment, in [DJNY 83]

[MR 83b]
Markowitz V.M., Raz Y.
An Entity-Relationship, role oriented query language
in [DJNY 83]

[MBW 80]
Mylopoulos J., Bernstein P.A., Wong H.K.T.
A language facility for designing data base intensive applications
ACM TODS 5,2 (June 1980)

[Mic 82]
Fernandez F., Ferrat L., Lee Y. J., Nguyen G. T.
MICROBE, Manuel de reference
Laboratoire IMAG, Grenoble, 1982

[Odd 81]
Oddy R., Robertson S., Van Rijsbergen J., Williams P.
Information Retrieval Research, Butterworths, London, 1981

[Pal 84]
Palazzo Oliveira J.
Le Modèle de Données et sa Representation Relationnelle dans un Système de Gestion de Bases de Données Généralisées
Thèse de Docteur Ingénieur, INP de Grenoble, Juin 1984

[Poo 79]
Poonen G.
CLEAR: A conceptuel language for entities and relationships reprinted in Chu, W. and Chen P.P-S, eds., Tutorial : Centralized and distributed data base systems, IEEE, 1979, pp. 194-215

[PR 78]
Prenner C., Rowe L.
Programming languages for relational data base systems
Proc. AFIPS 1978 NCC, Vol. 47, pp. 849-855

[PV 83]
Palazzo-Olíveira J., Velez F.
La correspondance de schemas entre les modèles TIGRE et Relationnel, Rapport de Recherche TIGRE N° 5
Laboratoire IMAG et CII Honeywell Bull, Grenoble, Sept. 1983

[PV 83a]
Pedraza E., Vargas M.H.
Interpretation d'un langage de bases de données généralisées
Rapport de DEA, I.N.P. de Grenoble, Juin 1983

[Qui 83]
Quint V.
Un système Interactif pour la production de documents mathématiques, T.S.I., Vol. 2 N°. 3, Mai/Juin 1983

[Rij 81]
Van Rijsbergen C.J.
Information retrieval, 2nd. ed., Butterworths & Co., London, 1979

[RS 79]
Rowe L., Shoens K.
Data abstractions, views and updates in RIGEL
Proc. of the ACM SIGMOD Conference, Boston, Mass.,1979.

[RS 82]
Rowe L., Shoens K.
A Form Application Development System.
Proc. of the ACM SIGMOD conference.
Orlando, USA, May, 1982.

[SAHR 84]
Stonebraker M., Anderson E., Hanson E., Rubinstein B.
QUEL as a data type
Proc. of the ACM SIGMOD Conf., Boston, Mass., Juin 1984

[Sac 84]
Sacco G. M.
OTTER - An information retrieval sdystem for office Automation
Proc. of the 2nd. ACM - SIGOA Conf. on Office Information Systems, Toronto, June 1984

[Sal 83]
Salton G., McGill M.J.
Introduction to modern information retrieval
McGraw Hill, New York, N.Y., 1983

[Sch 77]
Schmidt J.
Some high level constructs for data type relation.
ACM TODS, Vol. 2, Number 3, September 1977.

[Sho 79]
Shoshani A.
CABLE: A chain-based language for the Entity-Relationship model
Abstract and presentation in [Che 79]

[SNF 79]
Santos C., Neuhold E., Furtado A.
A Data Type approach to the Entity Relationship model
in [Che 79]

[SS 77]
Smith J.M., Smith D.C.P.
Database abstractions: Aggregation and Generalization.
ACM TODS, June 1977.

[SS 80]
Smith J.M., Smith D.C.P.
A data base approach to software specification
in W.E. Riddle and R.E. Fairley (eds.), Software development tools,
Springer Verlag, New York (1980)

[SSLK 83]
Stonebraker M., Stettner H., Lynn N., Kalash J., Guttmann A.
Document processing in a Relational Database System
ACM TOOIS, Vol. 1 N°.2, Avril 1983

[SSW 79]
Scheuermann P., Schiffner G., Weber H.
Abstraction capabilities and invariant properties modelling within the ER model, in [Che 79]

[TCEF 83]
Tsichritzis D., Christodoulakis S. et. al.
A Multimedia Office Filing System
Proc. of the 9th. Int. Conf. on VLDB

[TZ 84]
Tsur S., Zaniolo C.
An Implementation of GEM - Supporting a Semantic Data Model on a Relational back-end
Proc. of the ACM SIGMOD Conf., Boston, Mass., Juin 1984

[Vel 82]
Velez F.
LAMBDA : Un Langage de Définition et de Manipulation de Bases de Données Generalisées
Séminaire ADI de Bases de Données, Toulouse, Nov. 1982

[Vel 84]
Velez F.
Définition formelle du langage LAMBDA
Rapport de Recherche TIGRE N° 11
Laboratoire IMAG et Centre de Recherche Bull, Feb. 1984

[Zan 83]
Zaniolo C.
The Database Language GEM
Proc. of the ACM SIGMOD Conf., San José, Mai 1983

RAPPORTS DE RECHERCHE TIGRE

[TIGRE 1]

Présentation générale du projet TIGRE

Janvier 1983

[TIGRE 2]

The Tigre data model

M. Lopez, J. Palazzo Oliveira, F. Velez - Novembre 1983

[TIGRE 3]

Proposition de modèle pour la normalisation des documents

G. Bogo, H. Richy, I. Vatton - Mars 1983

[TIGRE 4]

Recommandations pour l'ergonomie d'un poste de travail attaché à un système bureautique

I. Forestier - Mars 1983

[TIGRE 5]

La correspondance de schémas entre les modèles TIGRE et relationnel

J. Palazzo Oliveira, F. Velez - Novembre 1983

[TIGRE 6]

Description des éléments du modèle de document

G. Bogo, H. Richy, I. Vatton - Septembre 1983

[TIGRE 7]

Programmation en logique pour une base de données généralisées

Nguyen G.T., J. Olivares, P. Winninger Novembre 1983

[TIGRE 8]

Machine base de données - Etat de l'Art

M. Burnier - Novembre 1983

[TIGRE 9]

Caractérisation des formulaires pour une base de données généralisées

C. Collet - Novembre 1983

[TIGRE 10]

Accès concurrent aux documents

S. Baltas - Janvier 1984

[TIGRE 11]

Définition formelle du langage LAMBDA

F. Velez - Mars 1984

[TIGRE 12]

Logic programming for a generalized data management system

M. Adiba, Nguyen G.T. - Janvier 1984

[TIGRE 13]

Modèle et fonctionnalités pour un système intégrant outils BD et outils de conception

D. Rialhe - Janvier 1984

[TIGRE 14]

SAGE : Un système d'autorisation généralisé

M. Adiba, F. Azrou - Janvier 1984

[TIGRE 15]

Coopération de Prolog et d'un SGBD généralisé : Principes et Applications

Nguyen G.T., J. Olivarès, P. Winninger - Avril 1984

[TIGRE 16]

Two-dimensional editing

V. Joloboff, V. Quint - Juin 1984

[TIGRE 17]

Aspects logiciels de la communication homme-machine sur les postes de travail individuels

V. Joloboff - Juin 1984

[TIGRE 18]

Information processing for CAD/VLSI on a generalized data management system

G. Nguyen, M. Adiba - Juin 1984

[TIGRE 19]

Représentation de la sémantique et des méta-connaissances dans une BD généralisées.

G. Nguyen, M. Adiba - Juin 1984

[TIGRE 20]

Description de l'éditeur TIGRE - Fonctionnalités, Architecture, Interfaces.

I. Vatton, H. Richy - Juillet 1984

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974,

VU les rapports de présentation de Messieurs

. M. ADIBA, Professeur
. F. BANCILHON, Professeur

Monsieur VELEZ JARA Fernando

est autorisé à présenter une thèse en soutenance pour l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité "Informatique".

Fait à Grenoble, le 28 août 1984

Le Président de l'I.N.P.-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,