MASTER EN SCIENCES INFORMATIQUES

LEA : un Système de Développement d'Applications Informatiques basé sur le Modèle Entité-Association

Feraille, Patrick; Tomasi, Mathias

Award date:
1989

Awarding institution:
Universite de Namur

LEA:

un Système de Développement d'Applications Informatiques basé sur le Modèle Entité-Association

Mémoire
présenté par

Patrick FERAILLE
et Mathias TOMASI

en vue de l'obtention du titre de
Licencié et Maître en Informatique
(Promoteur: Jean-Luc Hainaut)

Année académique 88-89

Septembre 1989

Institut d'Informatique des Facultés Universitaires de Namur

RESUME

Le sujet de ce mémoire est une interface de développement entité-association. Nous avons développé une interface de programmation et un dictionnaire de données basés sur un modèle entité-association. La machine virtuelle ainsi réalisée se base sur une couche physique, le SGBD PYRAMIDE, réalisé par D. Rossi dans le mémoire intitulé "PYRAMIDE, a Physical Engine for the Management of Entity-Relationship Databases.". Nous avons développé cette machine dans l'objectif de fournir une première étape vers un Système de Gestion de Bases de Données Entité-association. C'est pourquoi notre système est conçu de manière à pouvoir être étendu. Certaines de ces extensions sont étudiées dans ce travail, dont la génération dynamique de schémas par l'utilisateur. Le système offre néanmoins un langage de manipulation et d'interrogation d'une base de données entité-association complet.

ABSTRACT

The subject of this dissertation is an entity-relationship development interface. We have developed a programmation interface and a data dictionary based on an entity-relationship model. The realized virtual machine is based on a physical layer, the PYRAMIDE DBMS, realized by D. Rossi in the memoire entitled "PYRAMIDE, a Physical Engine for the Management of Entity-Relationship Databases.". We developed this machine with the objective to provide a first step towards an Entity-Relationship Data Bases Management System. This is why our system is conceived so as to be extensible. Some of these extensions are studied in this work, such as dynamic generation of schemas by the user. The system provides nevertheless a complete entity-relationship database query and manipulation language.

REMERCIEMENTS

Il est très difficile de remercier toutes les personnes qui ont contribué de près ou de loin à la réalisation de ce mémoire. Nous nous excusons donc auprès de celles que nous aurions bien involontairement oubliées.

C'est avec un grand plaisir que nous remercions d'abord toute l'équipe Badine de l'IUT de Dijon pour leur accueil. Nous voulons notamment remercier Monsieur Spaccapietra, ainsi que Christine Parent, et tous les chercheurs que nous avons pu côtoyer durant notre stage. Ils nous ont beaucoup aidé dans notre travail de familiarisation avec le domaine des langages entité-association. Nous n'oublierons pas comment ils ont su nous intégrer dans leur environnement de travail.

Nous ne pouvons pas oublier non plus nos écoles secondaires respectives, l'Institut Saint-Berthuin de Malonne et l'Institut Saint-Aubain de Namur, pour la qualité de l'enseignement qu'ils nous ont donné et qui nous a permis de mener à bien des études supérieures.

Nous tenons également à remercier tout le personnel de l'Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix de Namur pour son enseignement reconnu dans bon nombre d'entreprises. Au sein de celui-ci, nous ne pouvons oublier le Professeur Jean-Luc Hainaut avec qui nous avons travaillé pour ce mémoire et qui nous a bien souvent aidé par ses conseils judicieux.

Mais, nous ne serions pas allé bien loin sans le soutien de nos parents qui nous ont permis d'accomplir nos études et qui nous ont soutenus lorsque nous en avions besoin.

Je tiens également à remercier ma soeur Christine et mes amis du Patro de Temploux pour leur soutien immesuré dans la réalisation de ce travail. Sans eux, je ne serais peut-être encore nulle part.

Merci à tous.

TABLE DES MATIERES

III.LE SYSTEME LEA

V.L'EXTENSION DYNAMIQUE D'UN SCHEMA

VI.CONCLUSION ET EXTENSIONS POSSIBLES

BIBLIOGRAPHIE

Introduction

Lors de la réalisation d'applications informatiques, un concepteur est quasi toujours amené, d'une manière ou d'une autre, à devoir structurer, modéliser l'ensemble des informations pertinentes pour le problème considéré. Depuis de nombreuses années, des modèles de structuration des informations ont été développés pour faciliter ces tâches conceptuelles de structuration des informations. Leur fonction est d'apporter une aide et un support à la définition précise des informations de l'organisation. Une fois les informations structurées selon un de ces modèles, on obtient ce qu'on appelle un **"schéma de données"** de l'application.

Le plus célèbre de ces modèles est sans doute le **modèle relationnel**. Il existe cependant un autre modèle beaucoup plus puissant, surtout au niveau de la représentation de la sémantique des informations, c'est le **modèle entité-association** appelé aussi parfois modèle entité-relation. Ses concepts principaux sont ceux d'entité et d'association (ou relation) entre des entités.

L'étape suivante pour le concepteur sera "d'entrer" son schéma de données dans l'ordinateur et d'écrire les programmes qui vont travailler sur les données. Il utilisera pour ce faire ce qu'on appelle un **Système de Gestion de Bases de Données** (SGBD). Celui-ci offre également un ensemble d'opérations qui permettent à l'utilisateur de manipuler ses données. Mais entrer un schéma de données dans l'ordinateur à l'aide d'un SGBD demande souvent de transformer le schéma , les concepts utilisés par le SGBD étant différents de ceux du modèle de structuration employé (le modèle physique du SGBD est différent du modèle conceptuel). Cela signifie également que les opérations que le concepteur voulait exécuter sur son schéma de données devront être converties en un ensemble d'opérations à exécuter sur le schéma physique des données du SGBD et dont le résultat devra être équivalant au résultat attendu sur le schéma de départ. Le concepteur trouverait cependant intéressant de pouvoir développer ses applications directement avec son schéma de données, ç'est-à-dire en utilisant les concepts relatifs au modèle de structuration employé.

Pour le modèle relationnel, il existe de nombreux SGBD dont le modèle physique de représentation des données est équivalant ou du moins très proche du modèle relationnel (la représentation physique des données est la même que la représentation conceptuelle de ces mêmes données). On les appelle d'ailleurs des **SGBD relationnels**. Ils sont toujours accompagnés d'un langage, d'une interface qui permet de travailler directement sur les données. Les opérations que l'on souhaite effectuer sur les données conceptuelles seront donc plus ou moins identiques à celles réellement effectuées sur les données (physique).

En ce qui concerne le modèle entité-association, on s'aperçoit qu'il n'existe pas encore grand chose de semblable. En effet, on ne trouve pas encore beaucoup de SGBD dont le modèle physique de représentation des données soit le modèle entité-association. Bien souvent, les **SGBD** que l'on dit **entité-association** se basent sur un modèle physique qui est relationnel ou parfois entité-association "réduit". Il faut donc, dans ce cas, convertir les schémas de données conceptuels en des schémas de données physiques. De plus, les opérations que le concepteur voudra effectuer sur son schéma de données dans le cadre de son application devront être converties en un ensemble d'opérations sur le schéma de données physiques. Ces transformations seront dans certains SGBD réalisées automatiquement et dans d'autres devront l'être manuellement.

Certains SGBD offrent une autre caractéristique importante; c'est le **dynamisme.** Le dynamisme est la possibilité donnée au concepteur de modifier son schéma de données (qu'il soit conceptuel ou physique selon les SGBD) au sein même d'un programme et plus précisément lors de son exécution. Il est donc nécessaire lors de cette exécution de pouvoir accéder à une description du schéma de données qui doit bien entendu être constamment mise-à-jour. Contenir cette description est un des rôles remplis par ce que l'on appelle un **dictionnaire de données**.

Le travail que nous avons réalisé s'insère dans le cadre général du développement d'un SGBD entité-association complet. Son objectif est de définir et de réaliser une **machine virtuelle entité-association** destinée aux développeurs d'applications informatiques. Cela signifie que l'on va donner à un développeur qui a modélisé les informations nécessaires à une application à l'aide du modèle entité-association la possibilité de travailler directement sur sa modélisation. Cela sera possible au moyen d'une interface , (ou un langage), de programmation entité-association.

Cette machine sera réalisée en deux couches. La première est constituée d'un moteur physique réalisé dans le cadre du mémoire de D. Rossi intitulé "Pyramide: a physical engine for the management of entity-relationship databases.".

et la 2ème ?

Les caractéristiques demandées à la machine virtuelle sont les suivantes:

- offrir au développeur un modèle conceptuel des données qui soit plus puissant que le modèle physique des données sous-jacent au SGBD Pyramide. Ce modèle sera appelé modèle EAC (pour Entité-Association Complet).

- offrir un langage de programmation complet et simple d'utilisation au concepteur et basé sur le modèle entité-association. Il sera appelé Langage Entité-Association (LEA)

- les transformations nécessaires pour le passage de la machine virtuelle au SGBD physique seront automatiques et transparentes au concepteur

- le dynamisme sera étudié.

Le travail réalisé se compose de six parties:

1ère partie: le modèle entité-association théorique.

On y fait un rappel des concepts du modèle entité-association d'après F. Bodart. [BODA83]

2ème partie: analyse des besoins.

Cette partie présente l'état actuel du développement d'applications informatiques et ses carences qui ont mené à diverses tentatives pour compléter le modèle entité-association d'une partie permettant de manipuler ses concepts (un langage entité-association). Certains des travaux déjà réalisés sont étudiés et comparés. Leurs manques mènent à l'élaboration d'un certain nombre de choix et d'objectifs pour la réalisation du système LEA.

3ème partie: le système LEA.

On y décrit d'une manière complète le système LEA qui sera réalisé, aussi bien au point de vue de son environnement que du langage LEA proprement dit.

4ème partie: le précompilateur.

Cette partie plus technique concerne l'implémentation du système LEA.

5ème partie: l'extension dynamique d'un schéma.

Cette partie a pour objectif de définir le problème de l'extension dynamique d'un schéma, et les modifications à apporter au système afin qu'il bénéficie de cette facilité.

6ème partie: conclusion et extensions possibles

Cette partie présente les extensions à apporter au système LEA afin d'en faire un réel SGBD entité-association.

1ère partie

Le modèle entité-association théorique

qui l'a inventé ?

Introduction

Cette partie a pour seul objectif de rappeler les concepts qui sont à la base du modèle entité-association (modèle EA), défini de manière théorique et générale d'après [BODA83].

I.1. Objectifs et utilité

Selon F. Bodart, le modèle EA est un modèle qui permet d'exprimer la sémantique des informations mémorisables et/ou véhiculables à l'aide des concepts d'entité, association, attribut, et du mécanisme des contraintes d'intégrité.

L'objectif essentiel d'un modèle de structuration des informations, tel le modèle EA, consiste à fournir une aide et un support à la définition précise des informations de l'organisation. La tâche de définition s'intègre dans la fonction de gestion de l'information. Cette fonction est inhérente à la prise de conscience par l'organisation que son efficacité dépend en partie de la qualité de son système d'information.

Le modèle EA est un modèle conceptuel. Il permet de représenter la sémantique des informations. Tous les problèmes de représentation physique des données sont exclus de cette représentation.

I.2. Les concepts du modèle

I.2.1. Le concept d'entité

Une entité est une chose du réel perçu qu'un individu ou un groupe décide de considérer comme un élément autonome, afin d'enregistrer des informations à son propos. Par exemple, une personne, un cours, un employé,... peuvent être considérés comme des entités. Une entité peut avoir des attributs. — ??

Dans un processus de description, on s'intéressera plutôt à des classes d'éléments qu'à chaque élément individuellement. On parlera donc plutôt d'un type d'entité que d'une entité. Un type d'entité (TE) est caractérisé par son nom, la définition constitutive du type, la classe de toutes les entités possibles vérifiant la définition constitutive du type. Ces entités deviennent alors des occurrences du type générique.

Par exemple, on peut caractériser le type d'entité client par
-son nom: client,
-sa définition constitutive: toute personne physique ou morale dont au moins un ordre de commande passé à la firme a permis de l'identifier,
-sa classe: toutes les entités possibles vérifiant cette définition.

I.2.2. Le concept d'association

Une association est une correspondance entre deux ou plusieurs entités (non nécessairement distinctes), où chacune assume un rôle donné, et à propos de laquelle on veut enregistrer de l'information . Une association peut posséder des attributs. Par exemple, l'achat (association) de la voiture de numéro 1234 (entité) par le client de nom Durant (entité) dans le garage de nom Dupond (entité). On aura donc une association ternaire entre les trois entités, dans laquelle les rôles sont "achète" pour le client Durant, "est_achetée" pour la voiture 1234, et "vend" pour le garage Dupond.

De même que pour le concept d'entité, on distingue le type d'association (TA) qui est la classe de toutes les associations possibles du réel perçu vérifiant la définition constitutive du type, et qui est donc également caractérisé par son nom, la définition, et la classe de ses éléments. Ces éléments sont donc des associations ou occurrences du type d'association. Le degré d'un TA est le nombre de TE non nécessairement distincts, sur lesquels le TA est défini.

I.2.3. Le concept d'attribut

Un attribut est une caractéristique d'une entité ou d'une association. Il peut prendre un(e) ou plusieurs valeurs ou groupe de valeurs. L'attribut est caractérisé entre autres par un nom. Par exemple, l'attribut nom_personne de l'entité personne prend la valeur "Durant"; l'attribut prénom_personne prend les valeurs "Georges" et "Albert"; l'attribut adresse_personne prend le groupe de valeurs (14, rue du pré fleuri, 5000, Namur). Un attribut est simple si pour une entité ou une association, il ne peut prendre qu'une valeur. Il est répétitif s'il peut en prendre plusieurs du même type.

Un attribut est décomposable si, à une entité ou une association, il fait correspondre un groupe de valeurs de types différents, et peut être décomposé en autant d'attributs qu'il y a de types différents dans le groupe de valeurs. Par exemple, l'attribut adresse_personne peut être décomposé en les attributs numéro, rue, code_postal, localité. Un attribut non décomposable est dit élémentaire. Un attribut est obligatoire s'il doit prendre une valeur pour toute occurence du type qu'il caractérise. Il est facultatif s'il peut ne pas prendre de valeur pour certaines occurences du type qu'il caractérise.

Les notions de répétitivité et facultativité sont représentées par le concept de répétitivité d'un attribut. La répétitivité est un couple de valeurs qui peut prendre les valeurs suivantes:

- attribut simple obligatoire : 1-1
- attribut simple facultatif : 0-1
- attribut répétitif obligatoire :1-n
- attribut répétitif facultatif : 0-n

On peut introduire la valeur "inconnu" lorsqu'une valeur d'un attribut existe pour une occurence mais est inconnue lors de l'observation.

I.2.4. Les contraintes d'intégrité

Une contrainte d'intégrité (CI) est une propriété non représentée par les concepts de base du modèle, mais que doivent satisfaire les informations stockées.

F. Bodart considère que les contraintes de connectivité et d'identification sont des CI obligatoires.

I.2.4.1. Contraintes de connectivité

Une contrainte de connectivité est représentée par un couple d'entiers: (connectivité minimale, connectivité maximale). La connectivité minimale (maximale) indique le nombre minimum (maximum) de fois que, à tout moment, toute entité doit assumer un rôle dans une association, c'est-à-dire le nombre minimum (maximum) d'associations auquel toute entité doit participer.

I.2.4.2. Contraintes d'identifiant

Identification d'un type d'entité

Un identifiant d'un TE est un groupe d'attributs et/ou de rôles tel que à chaque combinaison de valeurs prises par ce groupe correspond au plus une occurence de ce TE. Un TE peut être doté de plus d'un identifiant.

F. Bodart distingue 4 cas d'identification d'un TE:

- l'identifiant est formé de un ou plusieurs attributs du TE à identifier. Un attribut identifiant ne peut pas être facultatif. Il est déconseillé de considérer un attribut pouvant prendre la valeur "inconnu" comme identifiant;

- "un TE peut être identifié par les rôles assumés par des TE dans le cadre de TA auxquels participe le TE à identifier. Ce TA est considéré comme identifiant. Pour des raisons de simplicité et d'efficacité, le TA identifiant qui relie le TE à identifier et le TE dont le rôle est identifiant est un TA non-cyclique et la connectivité du rôle du TE à identifier au sein du TA identifiant est (1,1)";

- le TE est identifié par un groupe formé de un ou plusieurs de ses attributs et par un ou plusieurs rôles;

- si un TE ne peut pas être identifié, ou qu'on ne veuille pas l'identifier par un fait de l'organisation, on peut néanmoins l' identifier par un attribut substitut du TE." La valeur de cet attribut correspondra à n'importe quel signe du système susceptible d'individualiser de façon permanente l'existence d'une entité dans le système d'informations".

Identification d'un type d'association

L'dentification d'un TA est considérée comme implicite. En effet, une association est identifiée par les rôles assumés par les entités sur lesquelles elle est définie, puisque son existence leur est contingente.

I.2.4.3. Contraintes facultatives

F. Bodart aborde également d'autres contraintes qu'il considère comme facultatives:

- CI d'existence: la validité de l'existence d'une occurence d'un TE ou d'un TA est liée à d'autres éléments de la structure de données;

- CI d'inclusion, exclusion, égalité de rôles: si une entité assume un rôle, elle doit aussi assumer les rôles inclus, elle ne peut assumer les rôles exclus, elle doit aussi assumer les rôles égaux et réciproquement;

- CI d'inclusion, exclusion, égalité d'association: le principe est le même que pour les rôles, si ce n'est que cette fois ce sont toutes les occurences de TE participant aux associations qui sont visés par la contrainte;

- CI de sous-typage: correspond à la association de généralisation entre des TE;

- CI de valeur: restriction de l'ensemble des valeurs que peut prendre un attribut d'un TE ou d'un TA;

- dépendances fonctionnelles: étant donné un TE ou un TA, un attribut B dépend fonctionnellemnt d'un attribut A si, à tout moment, à chaque valeur de A correspond au plus une valeur de B; on peut étendre cette notion aux dépendances fonctionnelles entre rôles;

- dépendances multivaluées: correspond à une dépendance d'un attribut simple ou d'un groupe d'attributs simples vers un attribut répétitif

I.2.5. Le concept de schéma

Bien que ce concept ne soit pas défini dans le modèle théorique de F. Bodart, et qu'il constitue un concept central de ce travail, il nous semble utile de l'introduire à ce niveau. Le lecteur peut considérer ce concept comme une extension apportée au modèle entité-association de F. Bodart.

Un schéma est l'agrégation des TE et TA, ainsi que de leurs attributs, identifiants, des rôles et connectivités de toute participation d'un TE à un TA, modélisant le réel perçu par le concepteur dans le cadre de son application. Un schéma est caractérisé par un nom, et l'agrégation des concepts qui le constituent.

I.3. Représentation graphique

Il est courant de donner au modèle entité-association une représentation graphique, qui a pour objectif de faciliter la communication. Cette représentation graphique est incomplète et doit être accompagnée de la spécification textuelle des éléments représentés. Nous adopterons les conventions décrites dans [BODA83] aux pages 17, 29, 33.

On trouvera à la page suivante un exemple représentant le schéma de l'application suivante:
Une usine est identifiée par son nom. Elle fabrique des produits ayant chacun un numéro. Un produit est toujours fabriqué par au moins une usine. Le coût de fabrication est dépendant de l'usine qui le fabrique.

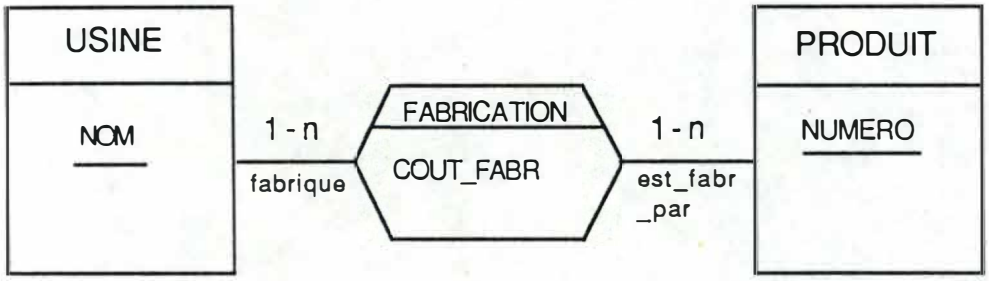
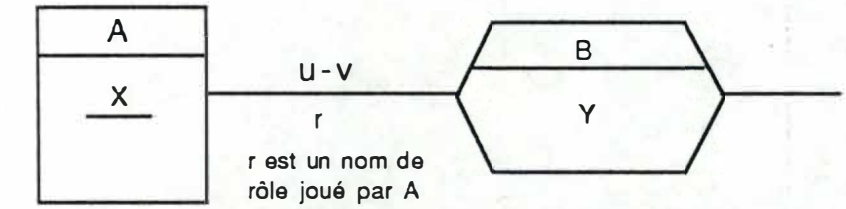


schéma de nom FABRIC exprimé dans le formalisme du modèle EA théorique

LEGENDE:



type d'entité de nom A possédant un attribut de nom x, par lequel il est identifié

r est un nom de rôle joué par A dans B avec les connectivités u-v

type de relation de nom B possédant un attribut de nom y

2ème partie

Analyse des besoins

INTRODUCTION

L'objectif de cette partie est de montrer pourquoi nous avons choisi de concevoir le système LEA. On y trouvera donc les sources de nos motivations.

Nous allons d'abord présenter un aperçu du développement actuel des applications informatiques. Nous rappellerons succinctement une méthode courante de développement d'applications. Notre objectif ne sera pas de rappeler dans les détails cette méthode, mais plutôt d'en présenter les grandes étapes, leurs produits et leur enchaînement. Nous mettrons en évidence les problèmes liés à l'utilisation de cette méthode. Nous verrons que la base de ces problèmes est l'inexistence d'un Système de Gestion de Bases de Données supportant le modèle conceptuel, c'est-à-dire le modèle entité-association dans notre cas. En effet, un travail important du développeur d'applications consistera à transformer les produits de sa spécification en produits exécutables. Il sera pour cela nécessaire de transformer le schéma modélisant l'application, ainsi que les traitements sur les données modélisées, d'un modèle vers un autre. Une étape importante d'une méthode de développement sera donc une étape de traduction qui pourra prendre beaucoup de temps selon l'ampleur de l'application modélisée.

Nous comparerons ensuite différents travaux menés afin de résoudre ce problème. L'objectif commun à la base de ces travaux est d'augmenter le modèle entité-association d'une partie de manipulation permettant d'agir sur les concepts modélisés.

Nous verrons que ces travaux ne sont pas entièrement convaincants dans le contexte du développement d'applications. Ceux-ci auront en effet souvent des objectifs particuliers qui parfois ne seront pas en accord avec notre optique de réaliser un réel outil de développement d'applications.

Nous arriverons à la conclusion que ces travaux ont des objectifs différents de ce que nous voulons faire, en l'occurrence, réaliser une première étape représentative vers un SGBD entité-association complet.

II.1. Aspects du développement actuel d'applications informatiques

Ce projet s'insère dans le cadre général du développement d'applications de gestion de bases de données.

Le développement d'applications informatiques est une tâche complexe qui nécessite l'utilisation d'une méthode. Notre objectif n'est pas de répéter une nouvelle fois les arguments favorables à l'emploi d'une méthode de développement, ni d'en décrire une quelconque. Le lecteur peu familier avec ce domaine pourra consulter [LAMS87],[HAIN86-a], et [BODA83].

Toutefois, il est nécessaire de rappeler certaines choses pour la bonne compréhension de ce travail. L'information est évidemment la base de tout processus de traitement et de communication de l'information dans une organisation. La qualité et l'efficacité de ces processus vont dépendre de la qualité des informations utilisées. On constate d'ailleurs de plus en plus l'émergence d'un souci de gestion de cette ressource que devient l'information pour l'organisation qui la possède. Ce souci passe inévitablement par la nécessité de définir de façon rigoureuse les informations de la mémoire du système d'informations de l'organisation (ensemble des Informations, et des traitements qui lui sont appliqués par des processeurs). Un point important est donc de modéliser les informations et leur structure, afin de fournir une représentation conceptuelle, sémantique des informations de l'organisation, et de leurs structurations.

Cette représentation possède certains avantages

- à l'analyse et à la conception: elle met en évidence la signification donnée par l'organisation à ses données (rappelons qu'une donnée à laquelle on donne une signification devient une information, une donnée ne dispose donc pas de sens en soi);

- à la réalisation du système d'informations (S.I.): elle aide à la compréhension de cette signification pour construire un S.I. efficace;

- à l'exploitation et à la maintenance: elle procure une définition précise des informations manipulées et de leurs conditions d'utilisation, elle constitue donc une documentation très appréciable.

Pour fournir cette représentation des informations, il faut utiliser un modèle de structuration des informations. Celui-ci doit posséder certaines qualités. Il doit permettre d'exprimer le sens des informations à l'exclusion de tout problème de représentation physique. Cette représentation doit être claire, lisible et structurée. Le modèle entité-association (modèle EA) décrit dans la première partie possède ces qualités.

De même qu'il est nécessaire de modéliser la structure des informations de l'organisation dans une démarche de conception d'applications informatiques, il faut aussi modéliser les traitements propres à l'application à effectuer sur ces informations. Le lecteur pourra se reporter aux références déjà citées dans ce chapitre pour les modèles de représentation des traitements.

A ce point du développement, on possède donc une représentation conceptuelle des informations manipulées par l'application et de leur structure ainsi que d'une représentation modélisée des traitements à leur appliquer.

Dans toute démarche de développement d'applications, on trouve évidemment aussi une étape d'implémentation de l'application. L'objectif est de fournir une version exécutable de l'application sur une machine. Pour cela, on a souvent recours à un logiciel spécialisé: le système de gestion de bases de données (SGBD). Ce type de logiciel offre en général trois grands types de fonctionnalités. Il permet

- de définir la structure, le format des données manipulées dans l'application; le résultat de cette opération est appelé la catalogue des données, et représente la structure que prendra la base de données, qui contiendra les données manipulées par l'application;

- de stocker des données dans la base conformément au format préalablement défini, et de modifier ou effacer ces données;

- d'accéder aux données dans la base en faisant référence à leur structure ou leur format.

Le format utilisé pour définir les structures des données dans la base dépend du SGBD utilisé, et plus particulièrement du modèle supporté par cet outil. Par exemple, un SGBD relationnel ne permettra de définir des données que sous forme de relations ou tables relationnelles. On ne pourra donc insérer dans la base que des données sous forme de tables. De même, on pourra accéder aux données en faisant référence à leur structure tabulaire et aux différents éléments permettant de mettre en relation ces tables entre elles (contraintes référentielles, par exemple).

Il n'existe pas encore à l'heure actuelle de véritable SGBD supportant un modèle EA qui soit satisfaisant dans un cadre tout à fait général. Une conséquence immédiate est qu'il ne sera pas possible de passer directement de la représentation conceptuelle des informations, et des traitements sur ces informations, à l'implémentation de l'application proprement dite. En effet, le modèle utilisé à la conception est différent du modèle qui sera utilisé à l'implémentation.

Au niveau de la méthodologie employée pour développer l'application, cela signifie qu'il faudra séparer les différentes phases. Durant l'analyse conceptuelle, on construira la représentation conceptuelle des informations (ou schéma), et les traitements sur cette représentation. Aux phases de conception logique et physique, il faudra transformer les produits de l'analyse conceptuelle en produits exécutables, c'est-à-dire notamment conformes au modèle supporté par le SGBD. On utilise pour cela le principe des transformations.

Dans une première phase, appelée conception logique, on transforme les produits de l'analyse conceptuelle en "produits logiques", c'est-à-dire en une solution exécutable sur une machine abstraite, non réelle. Cette solution est constituée du schéma des accès possibles exprimé dans le modèle d'accès généralisé [HAIN86-a], et issu de la transformation du schéma conceptuel, ainsi que d'une architecture de modules de traitement avec leur algorithme exprimé en LDA (Langage de Description d'Algorithme) [HAIN86-a], issue de la descripion conceptuelle des traitements. Cette solution doit être correcte: le schéma des données doit reprendre toute la sémantique du schéma copnceptuel, et les algorithmes doivent satisfaire à la spécification des modules de traitement conceptuel. Elle doit aussi être efficace: les accès aux données doivent être optimisés.

La deuxième phase va consister à produire une solution qui aura les mêmes caractéristiques que la solution logique (correcte, efficace), mais qui sera exécutable sur une machine réelle. Il faudra donc transformer le schéma et les algorithmes logiques pour qu'ils soient conformes au modèle du SGBD et aux autres outils de développement utilisés (compilateurs,...).

Il faut donc faire subir au schéma et aux traitements conceptuels deux séries de transformation. Ces processus de transformation produisent une solution exécutable, mais beaucoup moins claire et lisible que la solution conceptuelle. Par exemple, pour maintenir la sémantique du schéma conceptuel dans le schéma transformé, il faudra souvent ajouter des contraintes d'intégrité qui vont alourdir la représentation.

Ces processus alourdissent aussi la conception et la maintenance. Ils nécessitent en effet la manipulation de plusieurs versions d'une même solution, la connaissance de nombreux formalismes (EA, MAG, modèle du SGBD, formalisme de représentation conceptuelle ds traitements, LDA, langage de programmation, interface de programmation du SGBD,...). Lors de la maintenance, si un changement intervient au niveau conceptuel, il faudra connaître les transformations effectuées à la conception pour répercuter le changement sur la solution physique.

Il va de soi que plus le schéma conceptuel est complexe, que ce soit au niveau du nombre des structures représentées ou de leur richesse (récursivité,...), plus les tâches de transformation seront longues et donc coûteuses, et moins le schéma transformé en sera lisible. Il va aussi de soi que plus un schéma s'alourdit, et plus l'expression des traitements transformés pour se conformer à ce nouveau schéma s'alourdissent également.

Toutefois, étant donné que les processus de transformations de schémas sont systématiques, il a déjà été envisagé de traduire un schéma de données dans les différents formalismes de manière quasi-automatique. Cependant, ces outils ne permettent pas encore de faire suivre aux traitements les processus de transformations associés.

Nous constatons donc que la démarche actuelle de conception d'applications informatiques, bien qu'ayant le mérite de fonctionner, n'est pas entièrement satisfaisante, et nécessite énormément de travail de traduction et de programmation de la part du développeur.

L' approche que nous allons suivre consiste à construire un SGBD supportant un modèle conceptuel, en l'occurrence, le modèle EA. Le développeur aura ainsi à sa disposition un outil lui permettant

- de définir la structure des données directement dans le formalisme EA;

- de stocker les données dans ce formalisme, des les modifier, et de les supprimer

- d'accéder aux données stockées dans la base en faisant explicitement référence au formalisme EA.

Il ne lui sera donc plus nécessaire de transformer le schéma des données. Le développeur pourra produire à partir de l'analyse conceptuelle des traitements des algorithmes qui travailleront directement sur le formalisme EA. La solution conceptuelle sera une solution quasiment exécutable par la machine EA. La conception sera donc plus efficace. Le développeur devra passer moins de temps à la traduction de sa solution pour passer plus de temps à la spécification. De plus, la solution finale, déduite directement de la solution conceptuelle, sera visible plus rapidement.

Cette solution semble très prometteuse. Cependant, elle n'est valable que dans le cas où le développeur dispose d'un SGBD supportant un réel modèle EA, avec toute sa capacité d'expression sémantique. Cette voie a déjà été suivie. Nous présentons dans la partie suivante les principaux efforts déjà accomplis dans ce domaine.

II.2. Etat de l'art sur les langages entité-association.

Introduction.

Tout modèle se doit de permettre deux tâches importantes. Il doit permettre de capturer les caractéristiques du réel, propres à une application particulière. Il doit aussi permettre la manipulation des concepts qui le composent. Le modèle comprend donc non seulement une dimension de description, mais également une dimension d'action sur les concepts. Le modèle entité-association s'est imposé comme la tête de pont parmi les modèles sémantiques permettant une description conceptuelle du réel. Cependant, son essor est bloqué par le manque d'une partie de manipulation. On peut supposer que la prochaine étape dans le développement des systèmes de gestion de bases de données serait la réalisation d'un SGBD basé sur le modèle entité-association. Cette étape passe d'abord par la conception d'un langage de manipulation, au moins théorique, qui soit unanimement reconnu comme satisfaisant par la communauté informatique. De nombreuses propositions ont déjà été faites dans ce domaine, mais aucune d'elles n'a encore obtenu la satisfaction de tous les intéressés. Notre objectif ici est de présenter un certain nombre de travaux ou langages représentatifs qui ont déjà été réalisés dans cette matière.

On trouvera dans ce chapitre d'abord une approche synthétique. Un certain nombre de caractéristiques communes des langages sont dégagées, et sont comparées au niveau de chacun des langages.

Le propre d'un langage est de permettre la manipulation de données. On trouvera donc comme première partie la description des différents types de modèle entité-association employés.

Une seconde partie compare entre elles les différentes philosophies mises en oeuvre, et sous-jacentes au développement des différents langages.

On trouvera enfin la description comparée des grandes fonctionnalités des langages. Afin de maintenir une certaine uniformité dans cette étude, nous avons d'abord essayé de regrouper les aspects communs aux langages dans une approche algébrique. Nous présentons donc une série d'opérateurs de manipulation, de type projection, sélection, jointure,...,avec pour chacun d'eux ce qu'il permet de réaliser dans chaque langage. Une deuxième partie reprend les caractéristiques propres à certains langages, telles que la définition de vues, la manipulation de méta-données.

Nous avons porté en annexe une description analytique de chaque langage étudié. Le lecteur désirant approfondir son étude d'un certain langage trouvera dans cette annexe une description plus développée des langages avec aspects syntaxiques et exemples d'utilisation.

Nous voudrions enfin souligner certains aspects de l'étude de langages. La condition idéale d'étude d'un langage est évidemment de pouvoir s'en servir. Il faut pour cela disposer d'une version exécutable de ce langage, ainsi que d'une bonne documentation. Nous n'avons pas bénéficié de ces conditions, étant donné que bien souvent les langages E/A ne sont pas développés, ou sont en cours de développement. De plus, la plupart de ceux qui sont développés le sont sur des machines spécifiques. On est donc bien souvent contraint d'étudier un langage par l'intermédiaire de sa syntaxe ou des exemples. Cela risque évidemment d'entraîner un certain manque d'information. Nous espérons tout de même avoir pu souligner les principaux aspects de ces langages.

à partir d'ici, ça devient nettement plus complexe et moins clair

II.2.1 Modèles de travail.

Il n'existe pas encore à l'heure actuelle une norme permettant d'établir un consensus sur les concepts qui doivent être intégrés au modèle entité-association. Cela se reflète dans le manque d'uniformité des modèles employés par les concepteurs de langage. Une seconde raison vient du fait que les chercheurs emploient des modèles conformes à leurs objectifs. Ceux-ci sont multiples. Certains concepteurs visent d'abord la simplification, ce qui leur permet de développer des langages offrant moins de fonctionnalités mais étant plus faciles à implémenter, et pouvant être étendus par la suite à des concepts plus élaborés. C'est le cas pour HIQUEL, GORDAS, EXL, RRA, ERROL, le langage relationnellement complet, CLEAR, QBD. D'autres, plus rares, veulent d'emblée établir un cadre théorique complet, c'est le cas de ERC. Certains se définissent un modèle propre à une application particulière, c'est le cas de LAMBDA, avec le modèle TIGRE. Certains, enfin, se préoccupent de soucis d'efficacité, comme LMD.

De tous les langages rencontrés, c'est ERC qui nous semble offrir le modèle le plus complet. En effet, il permet les associations n-aires cycliques, les rôles, les contraintes de connectivité. Son apport principal est de permettre les attributs optionnels, multivalués et récursivement complexes. De plus, la théorie des amas employée pour ce modèle permet la manipulation d'ensembles contenant desoccurrences identiques. Les notions de généralisation et de spécialisation y sont également introduites. Ce type de modèle sera évidemment plus complexe à mettre en oeuvre que des modèles plus simplifiés.

Le langage LAMBDA se distingue par l'emploi d'un modèle particulier TIGRE. Il inclut ,outre les notions de rôle et de connectivité, la notion de documents structurés. TIGRE peut être considéré comme un modèle externe, spécifique pour des applications précises, mais il peut être transformé en modèle entité-association général avec des attributs complexes et multivalués.

La préoccupation d'efficacité de LMD se retrouve dans le modèle employé, qui est proche d'une implémentation physique. On y définit en effet des structures d'accès indexées, hachées,... On y emploie également des mécanismes introduisant de la redondance.

Parmi les langages basés sur un modèle plus simple, GORDAS est celui qui offre le plus de possibilités. En effet, bien que le modèle de base soit assez simplifié (il ne comprend pas les notions de rôle et de connectivité), plusieurs versions étendues sont développées. Ainsi, dans l'extension graphique de GORDAS sont inclues les notions de rôle et de connectivité. Il existe aussi des versions incluant la généralisation et les structures IS-A (généralisation).

QBD inclut d'emblée les structures IS-A mais ne supporte pas la notion de rôle. CLEAR accepte les rôles et les connectivités, tout comme HIQUEL, qui est cependant réduit aux associations binaires.

EXL, modifié pour satisfaire la propriété de complétude simplifiée, ne prend pas en compte les notions de rôle et de connectivité.

Le modèle entité-association sous-jacent au langage de manipulation DESPATH reprend la notion de signification d'une association (elle est déterminée en fonction des rôles joués par les entités participantes à l'association.). De plus, une association peut être vue comme une entité mais avec des propriétés spéciales. Cela permet d'avoir le concept d'objectification des associations (On transforme une association en une entité avec de nouvelles associations plus primitives). On a également les concepts d'associations hiérarchiques (binaires de type 1-N) et sous-typées (généralisation, spécialisation).

Finalement, certains langages se basent sur la théorie relationnelle. RRA, support sémantique d'ERROL, s'en inspire directement. [ATZ81] se base sur la complétude relationnelle. Ces modèles incluent donc des concepts facilitant le lien entre le modèle entité-association et le modèle relationnel. Ces concepts sont ceux de "surrogate keys" et de correspondance. RRA admet la notion de rôle et admet les types de relation récursifs.

II.2.2. Philosophie du langage.

Parmi les langages proposés, plusieurs démarches différentes sont explorées. Chacune a ses avantages et ses inconvénients. On ne peut les dissocier des objectifs de leurs auteurs.

Un premier objectif consiste à vouloir d'abord développer un cadre théorique et formel précis, sans considération sur la forme future du langage, du point de vue de sa syntaxe ou de son implémentation. Les auteurs suivant cette démarche sont amenés à développer des calculs (EXL modifié), ou des algèbres, comme RRA, ERC, le langage relationnellement complet. On pourra par la suite, en se basant sur une algèbre ou un calcul bien défini, développer un langage satisfaisant certains critères de convivialité et d'efficacité. Cela a été réalisé dans le cas d'ERROL, basé sur RRA, et EXL, basé sur le calcul défini dans [ATZ81].

En ce qui concerne les algèbres, RRA souffre d'un gros défaut. Cette algèbre n'est en fait qu'une reformulation de l'algèbre relationnelle, avec des ajouts pour satisfaire aux concepts du modèle entité-association. Dès lors, la richesse du modèle est sacrifiée à la simplicité. L'approche ERC nous paraît plus intéressante car elle se base sur un modèle complet et permet de manipuler des concepts plus élaborés. Un gros défaut de ERC est qu'à chaque application d'un opérateur, un nouvelle entité est créée, qui hérite des associations des opérandes. L'approche ERC ne conserve donc pas le base de données dans son état initial. A ce point de vue, l'algèbre proposée dans [CAM83] a la propriété de conservation.

Un autre objectif est de fournir directement un langage satisfaisant des critères d'efficacité (LMD), ou de convivialité (QBD, CLEAR, HIQUEL, GORDAS). Un inconvénient de cette approche est que les auteurs manquent d'un cadre formel. Dès lors, le langage est bien souvent défini par sa syntaxe ou par des exemples. Un avantage est que l'utilisateur dispose de suite d'un langage convivial. De plus, plusieurs de ces langages ont déjà été implémentés ou sont en cours d'implémentation.

Dans cette approche, beaucoup de concepteurs tirent parti de la représentation graphique du modèle entité-association. Ces langages travaillent sur base de schémas. Nous les désignerons par langages de mapping. Parmi ceux-ci, on en trouve où l'utilisateur exprime ses interrogations en manipulant directement des schémas, comme QBD.

D'autres imposent l'existence d'une représentation schématique de la base de données, et permettent à l'utilisateur d'écrire ses requêtes dans un langage donné, comme CLEAR et GORDAS.

Le désavantage de GORDAS par rapport aux autres langages de ce type est qu'il impose la transformation du diagramme entité-association en une représentation qui lui est propre, appelée graphe du schéma. Il faut noter, toujours pour GORDAS, qu'une extension est étudiée dans [ELM85] permettant à l'utilisateur d'exprimer ses requêtes par manipulation de schémas, tout comme QBD.

Il existe d'autres langages se basant sur le système query-by-example, et sur une représentation tabulaire du modèle entité-association, c'est le cas d'HIQUEL.

On trouve également des langages qui s'inspirent directement de SQL, en offrant les mêmes primitives, mais en les modifiant pour inclure le concept d'association. C'est le cas de DESPATH, LAMBDA et CLEAR.

ERROL se caractérise par une approche linguistique. Ce langage tente de reproduire les mécanismes du langage naturel par des concepts, tel celui des symboles de corrélation.

Enfin, il faut noter que la plupart des langages rencontrés permettent d'exprimer des requêtes, mais pas de manipuler la base de données. LMD permet la manipulation de concepts du modèle ER, mais il ne permet pas d'exprimer des requêtes puissantes. LAMBDA permet la manipulation du concept de document structuré. Le seul langage que nous ayons rencontré et qui permette à la fois la manipulation et l'interrogation d'une base de données est DESPATH.

II.2.3. Les fonctionnalités

Comme il a déjà été dit, il existe plusieurs types de langages basés sur le modèle entité-association. Notre problème dans ce chapitre, est d'aborder les fonctionnalités de ces langages, tout en essayant de dégager certaines d'entre elles que tout langage de manipulation basé sur le modèle entité-association devrait reproduire. A notre avis, une bonne façon d'y arriver est de se baser sur les composantes algébriques des langages, en faisant abstraction de la façon dont elles sont exprimées. On trouvera donc dans une première partie une série d'opérateurs algébriques, avec pour chacun une comparaison au niveau des différents langages. Dans une seconde partie, nous analyserons d'autres possibilités (manipulation de vues, de méta-données,...) offertes seulement par quelques langages.

II.2.3.1. Les opérateurs algébriques.

Une première remarque doit être faite à leur sujet. En effet, ces opérateurs sont complètement cachés au programmeur dans certains langages. On y explique comment on peut effectuer une requête sur un schéma souvent à partir d'exemples. Il n'est donc pas toujours facile à partir des articles lus de découvrir toutes leurs possibilités et donc tous les opérateurs algébriques sous-jacents à ces langages (Ils sont cachés derrière des instructions du genre SELECT... FROM... WHERE...).

a) L'union, l'intersection, la différence.

Nous considérons comme opérateurs d'union, de différence ou d'intersection, les opérateurs qui réalisent l'union, la différence ou l'intersection:

- soit de deux entités ou de deux associations (compatibles). Cela peut signifier la création d'une nouvelle entité ou d'une nouvelle association qui contiendrait l'union, la différence ou l'intersection des occurrences des deux entités ou associations de départ.

RRA permet l'union, l'intersection et la différence de deux relations. Etant donné que dans le cadre de cette algèbre, les relations sont aussi utilisées pour représenter les concepts du modèle EA, RRA permet l'union, l'intersection et la différence d'entités indirectement (par l'intermédiaire des relations qui les représentent).

Cependant, on ne retrouve pas clairement ces opérateurs dans ERROL qui est basé sur cette algèbre.

L'algèbre ERC, qui est une algèbre d'entités, permet l'union et la différence sur des types d'entités compatibles.

Le langage d'interrogation relationnellement complet offre également deux opérateurs add-relationship-union et add-relationship-difference qui permettent de réaliser l'union ou la différence de deux types d'entités compatibles.

- soit de deux sous-ensembles d'occurrences (compatibles) définis par des sous-requêtes. Les opérateurs sont implémentés sous la forme de connecteurs logiques "AND" et "OR". C'est le cas des langages tels que QBD, CLEAR, GORDAS, EXL, LAMBDA et DESPATH. Dans leur cas, ces opérateurs sont en réalité cachés derrière une instruction plus vaste qui permet d'exprimer une requête complète. On entend ici par sous-requête, une requête dans laquelle on accède à partir du concept d'intérêt, à un autre concept pour comparer des attributs des deux concepts ou former une condition sur le concept d'arrivée. Ce sont donc des connecteurs "AND" et "OR" plus généraux que ceux liant des sous-conditions dans une condition de sélection portant sur les attributs du concept d'intérêt.

ex: les clients qui ont une voiture de nom="GOLF" et qui sont mariés. On considèrera qu'il y a un opérateur d'intersection entre les deux sous-ensembles d'occurrences de clients définis par les deux sous-requêtes.

EXL, GORDAS et ERROL permettent aussi la négation, ou la possibilité de considérer le complémentaire du sous-ensemble défini par une sous-requête. Notons que QBD emploie la notation ensembliste U, plutôt que les connecteurs logiques.

b) La projection.

RRA permet évidemment la projection puisqu'il reprend les opérateurs relationnels en les adaptant aux modifications apportées au modèle pour supporter les concepts EA.

CLEAR et le langage d'interrogation relationnellement complet permettent également la projection d'une entité ou d'une association sur un sous-ensemble d'attributs, tout comme l'algèbre ERC et HIQUEL (seulement sur les entités). Ce dernier offre aussi la possibilité de conserver ou non les doubles. EXL n'offre pas la projection.

LMD permet de définir une entité ou une association "fictive" qui contient un sous-ensemble des attributs d'une entité ou d'une association.

Les autres langages offrent la projection dans le cadre d'une sélection (dans le sens de la définition d'une requête complète). GORDAS, QBD, ERROL et LAMBDA permettent de ne considérer qu'un sous-ensemble des attributs dans le sous-ensemble d'occurrences sélectionné lors d'une requête. DESPATH possède une instruction d'affichage qui permet de ne conserver que certains attributs (elle s'applique sur une sélection).

c) Le produit cartésien.

Le seul langage offrant cette fonctionnalité est HIQUEL.

d) La division.

Seul RRA offre cette possibilité.

e) La jointure.

La jointure apparaît dans RRA. CLEAR permet aussi l'expression de la jointure naturelle entre des entités. HIQUEL offre également des fonctionnalités analogues à une jointure externe et une jointure naturelle entre des entités.

ERC offre deux jointures: une r-jointure et un produit. La r-jointure pemet de joindre à une occurrence d'un type d'entité toutes les occurrences d'un autre type d'entité qui lui sont liées par un type d' association donné R. Le produit permet d'adjoindre à chaque occurrence d'un type d'entité E, toutes les occurrences d'un autre type d'entité. C'est le complément logique de la r-jointure.

QBD, GORDAS, EXL, ERROL, DESPATH et LAMBDA ne connaissent pas la jointure au sens de l'existence d'une instruction particulière pour la réaliser. Cependant dans ces différents langages, on

retrouve dans la sélection la possibilité d'exprimer des jointures "implicitement". En effet, contrairement au relationnel où pour exprimer des conditions sur deux relations il faut les regrouper par une jointure, on peut avec le modèle entité-association se servir de l'information des associations qui existent dans le schéma pour connecter des entités (ex: client qui est relié à voiture). La connection entre les deux entités ayant été définie dans le schéma, il suffit de la nommer. Cela se fait de plusieurs manières: 1) en utilisant un nom de connection comme GORDAS ou le nom de l'association comme QBD, 2) le nom des rôles uniquement comme HIQUEL, EXL, ERROL. CLEAR utilise les deux. DESPATH et LAMBDA utilisent le nom d'une entité et le nom du rôle joué par l'autre dans l'association entre les deux. Mais ils permettent aussi d'utiliser le nom de l'association dans certains cas.

De plus DESPATH, QBD et LAMBDA permettent également d'effectuer une jointure entre des entités non connectées par une association dans le schéma de départ. On pourra dès lors comparer des attributs de ces deux entités. Cette jointure est également cachée dans l'opération générale de sélection.

f) La réduction, la compression, le renommage et la simplification.

Ces opérateurs sont propres à l'algèbre ERC.

g) La sélection.

La sélection est le seul opérateur fourni par tous les langages (sauf LMD). Cela est tout à fait normal étant donné qu'ils sont tous des langages de requête.

Le LMD est différent. Etant donné qu'il existe des structures d'accès définies par l'utilisateur, le seul moyen d'accéder aux éléments c'est de demander directement l'accès au premier élément, au suivant, au dernier (accès séquentiel) ou un accès par une valeur de clé.

Le véritable opérateur de sélection permet à partir d'un ensemble d'occurrences et de conditions sur celui-ci produire un sous-ensemble d'occurrences inclus dans l'ensemble de départ et satisfaisant les conditions. Mais que peut être cet ensemble de départ ? Il peut être:
- l'ensemble des occurrences d'une entité ou d'une association,

- le résultat d'une autre sélection,
- le résultat d'une jointure implicite ou non entre deux entités
- le résultat de l'union, de la différence ou de l'intersection de sous-ensembles d'occurrences compatibles.

Il faut distinguer deux types de sélection. Il existe un l'opérateur de sélection simple comme dans ERC, RRA et le langage relationnellement complet qui permet de ne garder d'un type d'entité ou d'association (dans le langage relationnellement complet) que certaines occurrences en fonction d'un critère (défini par un prédicat sur les attributs du type opérande). Dans le cas où cet opérateur est utilisé, l'utilisateur doit explicitement nommer les autres opérateurs du langage comme la projection, la jointure,... pour créer un nouveau type d'entité ou d'association sur lequel on effectuera une sélection. Mais il y a aussi un opérateur de sélection beaucoup plus complexe et qui apparaît dans la plupart des langages (comme dans DESPATH, LAMBDA,...). Cet opérateur permet de sélectionner des occurrences d'un ou de plusieurs types d'entités et/ou d'association, de ne garder que certains attributs, en exprimant des contraintes de sélection non seulement sur les attributs des types opérandes mais aussi sur ceux de types liés par des TA ou non. En résumé, cette sélection mêle plusieurs opérateurs à la fois comme la projection, la jointure (souvent implicite), l'union,...

On peut voir trois étapes dans l'opérateur de sélection général:

- le choix des types de données sur lequel va porter l'opérateur de sélection.
- l'expression des conditions de sélection
  - simples
  - sur des jointures implicites ou non
  - la combinaison de conditions de sélection ("and", "or")
- la sélection des attributs que l'on veut conserver

1) Le choix des opérandes (TE ou TA).

Certains langages permettent de sélectionner plusieurs concepts à la fois comme GORDAS, DESPATH, LAMBDA et ERROL. QBD, CLEAR, EXL ne permettent de sélectionner des données que sur un type à la fois. Tous ces langages offrent la sélection sur des types d'entités ainsi que sur des types d'association, sauf HIQUEL.

2) Les conditions de sélection.

A) Conditions internes au concept. (sélections simples)

Tous les langages permettent d'exprimer des conditions simples sur les valeurs d'attributs du ou des concepts sélectionnés (=, >, <, >=, <=). LAMBDA permet de spécifier des parties de string (*model* pour modeler, remodeler,...). On retrouve également partout la présence de connecteurs logiques entre des conditions simples (and, or, not). Pour les langages permettant la sélection de plusieurs objets, on a la possibilité de comparer des attributs de ces objets entre eux (jointure).

B) Conditions avec des objets non sélectionnés.

Le problème est ici de savoir si le langage permet d'exprimer des conditions de sélection mettant en jeu des objets rattachés ou pas à des autres par des associations. Il suffit pour ce faire d'exprimer un prédicat de jointure implicite (si la jointure existe) entre les différents objets (voir jointure implicite).

Il faut aussi signaler qu'un certain nombre de langages ne permettent pas la comparaison d'objets non connectés. Ce sont CLEAR, HIQUEL, EXL, GORDAS et ERROL. QBD, DESPATH, LAMBDA le permettent (voir la jointure).

C) Combinaison de conditions.

Un autre point est de savoir si le langage permet de lier des conditions entre elles. Il existe pour cela divers mécanismes. Le branchement à l'aide de connecteurs logiques "AND", "OR". Tous les langages le permettent. Le deuxième mécanisme est l'imbrication (des termes d'un prédicat font appel aux résultats de sous-requêtes, pour obtenir le résultat de la requête il faut d'abord évaluer la sous-requête). Il est offert par QBD, CLEAR, GORDAS, EXL, ERROL, DESPATH et LAMBDA . Le troisième mécanisme est le chaînage des requêtes où l'ensemble objet d'une première requête est le sujet de la suivante,... (la deuxième requête se base sur le résultat de la première).

Il est offert par EXL et ERROL.

3) Le choix des attributs.

Une autre possibilité est celle de pouvoir projeter le résultat sur un sous-ensemble des attributs des objets sélectionnés. EXL et DESPATH ne permettent de sélectionner que l'entité ou l'association en entier. Dans le cas de DESPATH, il suffirait d'appliquer sur la sélection l'opérateur de projection (retrieval).

Une fonctionnalité intéressante est la présence dans le langage de manipulateurs d'ensembles (ex: includes,...). GORDAS possède des opérateurs fondamentaux de création et de manipulation d'ensembles. QBD n'en a pas. HIQUEL, CLEAR, EXL et ERROL, DESPATH, LAMBDA, ERC et l'extension graphique de GORDAS en possèdent également. Cela permet en général de comparer une valeur à un ensemble ou deux ensembles entre eux. Les opérateurs de comparaison sont =, > (contient), < (inclus), IN, INCLUDES, ou <>. LAMBDA permet même l'indexation des ensembles en utilisant le principe de correspondance. ERC et l'extension graphique de GORDAS permettent d'inclure des quantificateurs dans leurs ensembles et donc dans leurs requêtes. GORDAS et ERROL expriment la quantification à partir de manipulateurs d'ensembles.

Un autre point est la présence de fonctions agrégées. Elles sont présentes dans QBD, HIQUEL, GORDAS, LAMBDA, DESPATH, LMD, ERC, ERROL et CLEAR.

Il est intéressant également de pouvoir écrire des formules agébriques liant les attributs des différents objets dans la condition de sélection. Cela est possible pour QBD. Ce dernier offre même la possibilité de définir de nouveaux termes ainsi dérivés et de les inclure comme nouveaux attributs de l'objet sélectionné.

GORDAS permet aussi de définir de nouveaux termes dans un autre contexte: on peut définir un nouvel attribut dans une entité, qui est dérivé de la participation de l'entité à une association.
ex: ajout du nom de la ville connectée au client dans le client même.
HIQUEL permet aussi l'écriture de formules algébriques. Les autres langages n'ont pas cette facilité.

Plusieurs langages permettent de présenter les résultats de manière groupée ou ordonnée. CLEAR et GORDAS le font implicitement. HIQUEL utilise pour cela le splitting. DESPATH permet de définir un ordre sur un attribut (lors de l'affichage).

Lors de l'utilisation d'un langage se pose inévitablement le problème des références. Celui-ci est résolu par l'introduction de variables (EXL, RRA, ERROL, DESPATH, LAMBDA, ERC). Dans LAMBDA, ERC et DESPATH les ensembles peuvent contenir des variables de désignation. D'autres ne résolvent pas le problème (QBD, GORDAS, CLEAR).

En conclusion:

L'opérateur qui apparaît comme le plus important dans les langages est celui de sélection. Ces langages n'ont d'ailleurs dans la plupart des cas qu'un pouvoir d'interrogation, mais pas réellement de manipulation. Peu d'entre eux (GORDAS, RRA, DESPATH et LMD) permettent de définir un schéma EA ou de le modifier sauf par l'intermédiaire de vues (cela sera abordé plus loin).

DESPATH est le seul langage étudié où nous sont présentés effectivement les aspects d'interrogation et de manipulation. On peut afficher (en sélectionnant les attributs et l'ordre d'affichage), effacer, modifier (comme en SQL). Ces différentes opérations s'effectuent sur une sélection. On peut également insérer une entité ou une association entre des sous-ensembles d'entités.

II.2.3.2. Autres fonctionnalités.

a) La définition de vues.

Une vue est une représentation externe du schéma conceptuel central de la base de données, ou d'une partie de celui-ci, propre à un utilisateur, ou à une application particulière. Il existe deux mécanismes permettant la définition de vues:

- en se servant du Data Definition Language pour définir des sous-schémas

- en définissant des concepts dérivés du schéma conceptuel avec le Data Manipulation Language.

Despath utilise une mixture de ces deux méthodes. QBD permet également la définition de vues. En effet, l'utilisateur peut se définir des sous-schémas, via la phase de sélection (premier mécanisme), qu'il peut par la suite modifier via un langage de transformation (deuxième mécanisme). L'objectif final dans ce langage est pour l'utilisateur de définir un schéma entité-association qui soit une représentation graphique de la requête qu'il veut formuler. Les autres langages qui permettent la définition de vues (Hiquel et Gordas) leur donnent une forme hiérarchique.

L'expression de requêtes sur des schémas externes nécessite différentes étapes. Ces étapes peuvent être entachées de contraintes de succession, comme dans Hiquel et Gordas. Elles peuvent aussi se dérouler dans l'ordre que l'utilisateur voudra bien leur donner, selon son besoin, comme dans QBD. L'utilisateur a de plus la possibilité d'interrompre une étape pour passer à une autre, et d'y revenir ensuite au point où il l'avait quittée.

Parmi ces étapes, on retrouvera toujours la sélection, qui permet à l'utilisateur de garder les concepts qu'il considère pertinents par rapport à sa tâche, c'est-à-dire à la requête qu'il veut formuler. Dans Hiquel, cette étape se fait par des choix dans des menus. Dans Gordas, cela se fait par désignation sur le schéma. QBD permet en plus la sélection en exprimant des requêtes sur le schéma de départ, en sélectionnant des chemins, ou tous les concepts autour d'un autre, en accédant à des librairies de schémas personnels, ou de schémas de raffinements successifs historiques. QBD permet également beaucoup plus de possibilités au niveau de la transformation du schéma contenant les concepts retenus par l'utilisateur. Les auteurs ont défini un langage de transformation permettant d'activer les différentes primitives de transformation de schéma. Gordas et Hiquel, en-dehors des primitives de base telles que count,... qui permettent de définir de nouveaux attributs comme dérivés de l'agrégation d'autres, ne permettent pas la transformation de schémas.

Vues hiérarchiques

Les langages offrant le concept de hiérarchie permettent à l'utilisateur d'exprimer des requêtes de la manière suivante. Il définit d'abord un concept central. Ensuite, la requête s'articule autour de ce concept, en naviguant au travers des associations et en exprimant des conditions sur les attributs des entités rattachées. Les langages offrant ce service tirent parti de leur représentation graphique: Hiquel, Gordas. Pour ce dernier, nous considérons l'extension, qui ne fait que donner un support graphique à l'esprit hiérarchique qui existe déjà dans le langage de base.

Les systèmes hiérarchiques offrent d'autres étapes, outre la sélection et la transformation, afin de construire une vue hiérarchique. La première est la sélection d'un concept central. Gordas est plus complet qu'Hiquel à ce point de vue. Il offre, outre la sélection par désignation, la possibilité de sélectionner le type d'entité qui a le plus d'attributs à afficher,... Hiquel et Gordas (version étendue) ne permettent la sélection que d'un seul concept central, alors que Gordas (version de base) permet de sélectionner plusieurs concepts centraux dans la même requête.

La deuxième étape consiste à poser des conditions de sélection sur les attributs du concept central. Un désavantage d'Hiquel par rapport à Gordas est l'obligation de mettre les conditions composées sous forme disjonctive normale.

La troisième étape consiste à naviguer du TE-racine vers les TE-feuilles reliées par des TA. Ces deux langages permettent l'affichage de la structure hiérarchique des requêtes. Pour Hiquel, celui-ci se fait au fur et à mesure de la sélection des concepts par l'utilisateur. Gordas accepte d'abord un schéma entité-association sans structure hiérarchique, mais le réarrange selon cette structure dès que l'utilisateur a choisi un concept central. Lors de la construction des hiérarchies, les deux systèmes intègrent d'office les attributs des TA liant le TE central aux TE feuilles dans ces dernières. Ces deux systèmes ne permettent malheureusement que l'emploi de TA binaires. De plus, ils ne permettent d'exprimer des requêtes que sur des objets reliés par des TA. En effet, le mécanisme de hiérarchisation n'est possible que par l'utilisation des TA liant les TE. Deux TE reliés par un TA de type 1:1 sont toujours assimilés à un seul TE. Dans Hiquel, cela a pour effet de mettre ces deux types d'entité sur le même niveau hiérarchique.

Les tables décrivant les types d'entité sont ainsi acollées, donnant à l'utilisateur l'impression d'une table unique. Dans Gordas, le système regroupe dans le type d'entité les attributs des deux types de départ.

La dernière étape consiste à spécifier des conditions de sélection sur les TE-feuilles.

Un avantage des systèmes hiérarchiques est de pouvoir plus facilement exprimer qu'une condition doit être remplie dans l' ensemble des entités-feuilles pour sélectionner des occurrences de l'ensemble des entités-racines. L'utilisateur peut exprimer des requêtes du type:

"sélectionner entité-racine si toutes les (au moins une des, aucune des) entités-feuilles satisfont la condition p"

Les deux langages permettent cette facilité: Gordas en utilisant des comparaisons d'ensemble, Hiquel par des quantificateurs.

b) La manipulation de méta-données.

Une possibilité intéressante offerte par le langage DESPATH est la manipulation des méta-données (i.e. les données qui décrivent le schéma entité-association sur lequel on travaille) par les primitives standards du langage. Le schéma peut donc être modifié de manière incrémentale et dynamique. Les méta-données sont pour ce faire stockées dans la base de données .

c) Le respect de contraintes d'intégrité.

DESPATH est le seul langage à assurer la vérification des contraintes d'intégrité implicites (dans le schéma) qu'elles soient statiques ou dynamiques et des contraintes d'intégrité explicites définies par l'utilisateur de deux manières possibles: par formulation d'assertions comme en SQL ou par l'utilisation de procédures de modification prédéfinies (concept semblable à celui de type de données abstrait)

II.3 Conclusion

Nous avons pu constater durant notre étude des langages entité-association que peu de ceux-ci semblaient avoir réellement implémentés. Les langages qui l'ont été ont souvent été conçus sur du matériel spécifique. En opposition à cet état de fait, nous voulons construire un système de développement d'applications informatiques utilisables le plus largement possible. Ce système sera donc implémenté sur du matériel standard et répandu. La solution la plus acceptable nous semble être l'environnement Ordinateur Personnel. Pour ce qui est du logiciel, nous utiliserons PASCAL. Nous espérons de cette façon bénéficier de l'étendue de ces outils.

De plus, la plupart des documents que nous avons consultés présentent un langage. Nous nous situerons dans le cadre plus général d'un prototype de SGBD entité-association, dont le langage sera une partie parmi d'autres, qui seront détaillées dans la suite de ce travail.

Un premier point sera de se définir un modèle supporté par notre système. Nous choisirons un modèle que nous considérerons comme une bonne base pour la réalisation d'un prototype. Nous ne simplifierons donc pas trop ce modèle, mais nous n'implémenterons pas non plus tous les aspects que peut recouvrir le modèle entité-association. Cela sera particulièrement évident pour les contraintes d'intégrité et les identifiants. Ces aspects seront relégués aux extensions du système.

De nombreux travaux se limitent à l'élaboration d'un cadre théorique bien précis. Cela débouche sur l'élaboration d'algèbres ou de calculs qui définissent un environnement formel de manipulation des concepts du modèle entité-association.

A l'opposé, d'autres travaux réalisent de véritables langages entité-association dans un réel souci de convivialité, sans se définir au préalable un cadre théorique précis. Nous avons opté pour la réalisation d'un langage, plutôt que d'une algèbre ou d'un calcul. La réalisation d'une algèbre aurait eu plus d'intérêt en ce qui concerne l'approche théorique. Cependant, notre souci est d'abord de réaliser un système utilisable. Or, l'implémentation d'une algèbre nous aurait posé certains problèmes assez complexes. Citons notamment le problème de la conservation de l'état de la base de données.

Dans le cadre de la réalisation de notre langage, la convivialité ne sera pas notre premier souci. Comme nous manquerons du cadre formel qu'aurait pu nous fournir une algèbre ou un calcul, nous devrons porter notre attention à deux critères importants:

- le langage devra permettre de manipuler tous les concepts du modèle,

- il devra être non ambigu.

Ces aspects seront développés dans la partie propre au langage.

Un point qui est rarement abordé pour les langages non interactifs est de savoir si le langage devra être utilisé seul, ou s'il devra être utilisé comme complément d'un autre langage. Comme nous voulons construire un système utilisable, nous devrons aussi aborder cet aspect des choses. Nous avons choisi un langage complément du langage de programmation PASCAL. Il faudra donc définir complètement comment vont s'effectuer les interactions entre ces deux langages. Un de nos critères est que le système soit le plus simple possible à utiliser.

En ce qui concerne le langage, le principe sera le même. Nous proposerons les opérateurs de base pour manipuler les concepts de notre modèle. D'autres fonctionnalités pourront être ajoutées par la suite. Il s'agira en effet d'abord de voir si notre prototype est satisfaisant dans un environnement didactique. Une fois ce fait acquis, on pourra l'étendre afin d'en faire un réel SGBD entité-association.

3ème partie

Le système LEA

III. Le Système LEA

Introduction

L'objectif de cette partie est de définir complètement le système LEA.

Nous commencerons par donner un aperçu général du système et de son architecture en deux couches. Nous donnerons les rôles de chaque couche dans le système.

La première couche sera constituée du SGBD PYRAMIDE. Nous rappellerons donc dans une première étape les concepts de PYRAMIDE et de son interface de programmation. Nous rappellerons également quelques éléments qui nous sont utiles pour l'utilisation de ce SGBD. Notons que nous ne donnerons que les éléments ayant un intérêt pour le système LEA en entier. Pour les aspects plus particuliers à PYRAMIDE, le lecteur pourra se reporter au mémoire de D. Rossi.

Nous aborderons ensuite la deuxième couche de l'architecture qui fait l'objet de ce travail. Nous donnerons d'abord la description du modèle de données supporté par cette couche. Ce sera évidemment un modèle entité-association, défini par rapport au modèle entité-association théorique de F. Bodart.

Nous décrirons ensuite le moyen par lequel le système permet le stockage d'un schéma conforme à ce modèle. Ce moyen sera le dictionnaire de données dont la structure de méta-schéma sera expliquée en détail.

Ensuite seront abordés les mécanismes de base de ce travail: les transformations. On verra pourquoi et comment il est possible, à partir de la description d'un schéma conforme à notre modèle dans le méta-schéma, de produire un schéma équivalent conforme à PYRAMIDE.

Nous définirons ensuite l'interface de programmation du système LEA. Les concepts du langage LEA seront définis. Nous donnerons des exemples d'utilisation de cette interface.

En conclusion, nous décrirons le fonctionnement général du système LEA, c'est-à-dire les différentes étapes de l'utilisation du système, et pour chacune, la marche à suivre, les produits, et un aperçu du fonctionnement interne du système.

III.1. Architecture en deux couches

La machine virtuelle LEA est constituée de deux couches. La première est le SGBD PYRAMIDE, développé par D. Rossi, dans le mémoire :"PYRAMIDE : A Physical Engine for the Management of Entity-Relationship Databases". La deuxième couche est l'interface de développement entité-association, développée dans le cadre de ce mémoire.

Nous avons choisi une architecture en deux couches afin d'obtenir deux parties plus ou moins indépendantes au niveau de leur réalisation.

Ainsi, PYRAMIDE a été conçu de façon à garder la compatibilité avec le SGBD NDBS, déjà développé par D. Rossi. Les concepts manipulés et l'interface de programmation sont semblables. Ce travail a pu être réalisé avec des critères assez indépendants de la couche supérieure. PYRAMIDE est de plus réutilisable pour d'autres travaux.

D'autre part, la réalisation d'une interface de développement EA est facilitée par le fait du modèle supporté par PYRAMIDE, qui facilite le processus de transformation de la couche supérieure vers PYRAMIDE. Une deuxième raison est donc que nous ne voulions pas partir de rien pour réaliser un véritable SGBD entité-association, et non plus une machine virtuelle. PYRAMIDE nous est donc apparu comme une bonne base vu la proximité du modèle qu'il supporte, avec notre modèle entité-association. De plus, la couche supérieure ne fait appel qu'à l'interface de programmation de PYRAMIDE, qui est assez simple d'utilisation, et à rien d'autre.

III.1.1. Rôles de la première couche: le SGBD PYRAMIDE

Nous allons d'abord rappeler brièvement ce qui constitue l'environnement PYRAMIDE.

PYRAMIDE est un SGBD supportant un modèle entité-association restreint, notamment à des associations binaires sans attribut. Ce type de modèle est souvent appelé modèle de type réseau.

Pour manipuler les concepts du modèle qu'il supporte, PYRAMIDE offre une interface de programmation. Celle-ci est constituée d'un ensemble de primitives PASCAL pouvant être insérées dans tout programme écrit avec ce langage. Une primitive manipule une seule entité à la fois. Ces primitives sont entièrement conformes à la syntaxe PASCAL. Elles constituent donc un ensemble de procédures PASCAL accessibles dans le programme d'application par des appels. Pour ces appels, il est nécessaire d'avoir recours à des paramètres. Ceux-ci sont des constantes, des variables directement issus de la phase de production d'une base de données opérationnelle par PYRAMIDE (voir plus bas pour cette phase). Pour pouvoir utiliser PYRAMIDE dans un programme PASCAL, l'utilisateur doit donc y insérer l'unité "PYRAMIDE", contenant la déclaration des procédures de PYRAMIDE. Il devra aussi insérer le fichier des types de données et constantes PASCAL, suffixé par .TYP. Ces types permettront à l'utilisateur de déclarer des variables qu'il pourra passer en paramètre aux primitives de PYRAMIDE. Une variable dbstatus permet de recevoir un code de retour suite à l'exécution d'une primitive de PYRAMIDE.

L'utilisation de PYRAMIDE requiert évidemment, avant de travailler sur la base que celle-ci existe. Le processus à suivre pour créer une base de données opérationnelle sera abordé plus loin dans le cadre du fonctionnement général du système.

Quand la base de données est créée, l'utilisateur peut manipuler les données au moyen des primitives de PYRAMIDE. Celles-ci permettent

- d'ouvrir une ou plusieurs bases de données en même temps, de les fermer;

- d'accéder séquentiellement à toutes les entités d'un type;

- d'accéder directement à une entité par sa référence ("adresse où elle est stockée dans la base"), ou par une valeur d'identifiant;

- d'accéder séquentiellement aux entités liées à une autre par une association (chemin);

- de créer, effacer, modifier une entité d'un type;

- d'insérer, enlever une entité dans un chemin;

- de manipuler des variables d'entité ou de référence;

- d'ouvrir, fermer, annuler une transaction.

PYRAMIDE permet donc de manipuler des données dans une base par l'intermédiaire de son interface de programmation. Il reste à voir comment il est possible de créer une base de données au moyen de l'environnement PYRAMIDE.

Tout d'abord, pour assurer le stockage de la description d'un schéma entité-association, PYRAMIDE gère une structure de dictionnaire de données. Celui-ci est structuré de façon à permettre le stockage de la description de schémas EA, qu'ils soient dans une modèle restreint, comme celui supporté par PYRAMIDE, ou dans une modèle EA, comme celui supporté par la couche supérieure. Le dictionnaire de données est structuré lui-même sous forme d'un schéma EA réduit, c'est-à-dire une agrégation de types d'entité (dbschema, entity-type,...) et de types d'association les liant. C'est pourquoi cette structure est nommée méta-schéma, c'est un schéma contenant des renseignements (les méta-données) sur le schéma de l'utilisateur.

Une particularité de la gestion du dictionnaire de données par PYRAMIDE est que celui-ci est inclus dans toute base de données. Une base de données PYRAMIDE n'est donc jamais vide, elle contient toujours un noyau. Ce noyau est constitué du dictionnaire de données qui contient la description de schémas entité-association (notamment, celui de la base de données). Notons que ce dictionnaire n'est lui-même jamais vide, puisqu'il contient toujours la définition du méta-schéma. Un effet immédiat est que toute base de données à l'exploitation contiendra aussi bien les méta-données que les données. Ces méta-données seront donc accessibles par un programme d'application manipulant la base de données, au même titre que les données de l'utilisateur.

PYRAMIDE est spécifié de façon à permettre la mise à jour des méta-données, à tout moment dans un programme d'application. Cette fonctionnalité est souvent appelée extension dynamique du schéma de la base de données. Elle implique qu'à tout moment, les données peuvent être réorganisées dans la base afin de se conformer au nouveau schéma. Cette fonctionnalité n'est cependant pas implémentée. Si l'utilisateur modifie le schéma de la base de données lors de son exploitation, une nouvelle base de données vide est créée avec le nouveau schéma. Les anciennes données se trouvent toujours dans l'ancienne base, dont l'extension est changée.

L'utilisateur devra lui-même adapter les anciennes données au nouveau schéma.

Le stockage d'un schéma EA dans le dictionnaire de données doit s'effectuer à l'aide d'un programme d'application, en utilisant l'interface de programmation de PYRAMIDE. Celle-ci est d'ailleurs strictement identique, que le programme agisse sur les données ou les méta-données, le seule différence se situant au niveau des types de données manipulés.

Si le méta-schéma contient la description d'un schéma conforme au modèle de PYRAMIDE, un utilitaire (métacomp) permet de le compiler pour produire une base de données opérationnelle conforme à ce schéma, ainsi que le fichier de types de données (suffixé par .TYP) qu'il faudra insérer dans tout programme d'application travaillant sur cette base, et faisant usage de l'interface de programmation de PYRAMIDE. Ces types de données sont la traduction sous forme de types PASCAL des structures définies dans le dictionnaire de données.

Outre le fait de pouvoir stocker le schéma de la base, le dictionnaire de données a une autre fonction. Il est utile à l'interface de programmation de PYRAMIDE pour contrôler les opérations et transformer les opérations logiques de PYRAMIDE (sur des entités, associations,...) en opérations agissant directement sur la structure physique de la base de données.

En résumé, le rôle de PYRAMIDE sera d'assurer la gestion physique de la base de données, à partir d'un schéma entité-association réduit dont la description est contenue dans le dictionnaire de données. Cette gestion physique comprend donc la gestion du dictionnaire de données, la création d'une base de données opérationnelle conforme au schéma transformé dont la description est stockée dans le dictionnaire de données. PYRAMIDE se charge enfin d'exécuter effectivement les opérations sur les données dans la base. Ces opérations sont spécifiées dans la syntaxe de l'interface PYRAMIDE, et prennent donc la forme d'appels à des procédures PYRAMIDE. Ces appels résultent des processus de transformation mis en jeu dans la couche supérieure.

III.1.2. Rôles de la deuxième couche

Le rôle principal de la couche supérieure est d'offrir une interface de développement d'applications supportant un modèle entité-association plus complet que le modèle restreint.

Ce modèle comporte les même concepts que le modèle supporté par PYRAMIDE, mais les contraintes d'utilisation sont moins strictes. Par exemple, une association pourra avoir un degré supérieur à deux et pourra posséder des attributs. Ce modèle sera nommé par la suite modèle EA complet (EAC), pour plus de facilité.

Une première fonctionnalité de la couche LEA est donc de permettre à l'utilisateur le stockage d'un schéma EA conforme à ce modèle. Cette fonction est assurée par le dictionnaire de données de PYRAMIDE. En effet, celui-ci peut contenir la description de plusieurs schémas non nécessairement restreints. Donc, la couche LEA doit permettre la mise à jour du dictionnaire de données de PYRAMIDE.

Notons que le dictionnaire de données est légèrement modifié par la couche supérieure. Le méta-schéma contient en effet toujours sa description dans un formalisme conforme au modèle du niveau supérieur. Le fichier standard.dtb qui contenait donc le dictionnaire de données (vide de toute description de schéma) est donc augmenté automatiquement dans le système LEA de telle sorte que le dictionnaire de données contienne sa propre description dans un formalisme plus complet. Cela est nécessaire de façon à ce que l'utilisateur puisse travailler dans le même formalisme (EAC), aussi bien pour son propre schéma que pour le méta-schéma.

Pour assurer cette fonction de stockage, l'utilisateur doit écrire un programme d'application utilisant l'interface de programmation LEA. Cette interface est composée d'une série d'ordres permettant de manipuler de la même façon les concepts du méta-schéma et ceux du schéma de l'application. Tout comme pour PYRAMIDE, la seule différence intervient au niveau des types de donées manipulés. Ces ordres doivent être insérés dans un programme écrit en PASCAL. Les particularités de l'interface de programmation seront détaillées dans la partie qui lui est propre.

L'interface avec le programme hôte est assuré par des variables d'entité ou d'association, qui doivent être déclarées par l'utilisateur grâce à un ordre du langage. Ces variables permettent au programme d'application de passer ou recevoir des valeurs d'attribut et/ou de référencer une entité ou une association. Les erreurs à l'exécution peuvent être détectées par le programme en consultant la variable erstatus, qui sert de code de retour suite à l'exécution de tout ordre LEA. Les différentes valeurs que peut prendre cette variable ainsi que leur signification sont données dans la partie consacrée au langage.

L'analyse des ordres du langage, ainsi que les processus de transformation sont assurés par un précompilateur. La tâche principale de ce programme est de transformer tout ordre du langage LEA dans un programme PASCAL en une séquence d'instructions PASCAL et des appels aux primitives de PYRAMIDE. Cette séquence devra évidemment notamment produire l'effet attendu par l'utilisateur de l'ordre LEA.

Pour transformer les ordres LEA en primitives de PYRAMIDE, il est aussi nécessaire de transformer le schéma EAC vers un schéma EA équivalent supporté par PYRAMIDE. Le précompilateur fait appel pour ce processus au principe des transformations. Le schéma transformé sera lui-aussi stocké dans le dictionnaire de données. Les transformations n'y sont donc pas stockées. Celui-ci ne contient que le schéma de départ (EAC), et son équivalent transformé. Les transformations pour passer de l'un à l'autre ne sont connues que du précompilateur. Cela est possible du fait de la proximité des deux modèles, et de la relative simplicité pour transformer un schéma EAC en un schéma supporté par PYRAMIDE.

Nous avons choisi un processus de précompilation pour permettre au langage EA de dépasser la syntaxe de PASCAL, et offrir des ordres aux fonctionalités étendues, qui auraient compliqué fortement le passage des paramètres. Par exemple, l'ordre de création, pour assurer le respect des connectivités minimales non nulles, doit pouvoir permettre de créer un nombre non fixe d'entités et de relations, selon la structure à créer. Ce problème de la propagation des mises-à-jour est abordé plus longuement dans la partie consacrée au langage.

De plus, comme le précompilateur a seul la connaissance des transformations qui ont été réalisées pour passer du schéma EAC au schéma correspondant dans le modèle de PYRAMIDE. Il peut se charger de gérer le passage d'un ordre travaillant sur le schéma EAC vers les

primitives de PYRAMIDE, qui ne doit donc pas mémoriser ces renseignements, et est donc plus performant.

Le précompilateur se charge aussi de vérifier la conformité du schéma EAC par rapport au modèle de la couche supérieure, en consultant le dictionnaire de données. Il produit aussi le code permettant de vérifier la cohérence de la base de données suite à l'exécution des ordres du langage.

En résumé, les rôles de la couche supérieure consistent donc à

- assurer la gestion du langage LEA, au niveau de sa syntaxe, de son dialogue avec le programme d'application,...
- assurer la cohérence des ordres par rapport à la description du schéma EAC stocké dans le méta-schéma,
- assurer le processus de transformation des opérations sur un schéma EAC vers des opérations sur un schéma transformé

III.2. Le SGBD Pyramide.

Introduction

Comme il a déjà été signalé, la machine virtuelle entité-association réalisée s'appuye sur un moteur physique, le SGBD PYRAMIDE.

Il est donc nécessaire de rappeler certaines choses concernant celui-ci. Nos rappels se limiteront aux éléments qui nous sont indispensables pour pouvoir utiliser PYRAMIDE comme couche inférieure de la machine virtuelle.

L'objectif de ce chapître est donc d'étudier rapidement les caractéristiques de ce moteur physique, et ce à deux niveaux,
- son modèle de données sous-jacent (en vue de connaître les concepts supportés),
- son interface de programmation.

Pour plus de détails sur ce SGBD on pourra consulter le mémoire de D. ROSSI intitulé "PYRAMIDE: a physical engine for the management of entity-relationship databases".

III.2.1. Modèle de données supporté par le SGBD PYRAMIDE

Le modèle de données supporté par le SGBD PYRAMIDE est un sous-ensemble du modèle entité-association (modèle entité-association réduit). En effet, c'est un modèle binaire. Ce modèle réduit est inspiré du modèle d'accès généralisé décrit dans [HAIN86-a]. On y retrouve les mêmes concepts que dans le modèle entité-association, à savoir ceux de schéma, de type d'entité, de type d'association, d'attribut et d'identifiant, mais avec, cependant, des contraintes relatives à leur utilisation qui y sont plus strictes.

Ces contraintes sont les suivantes:

- un type d'entité a un nom unique dans un schéma donné.
- un type d'association a un nom unique dans un schéma donné.
- un attribut a un nom unique dans un type d'entité ou d'association .

- *type d'association*

Dans le modèle entité-association réduit, le type d'association ne peut être que binaire (entre deux types d'entité non nécessairement distincts) et sans attribut. De plus, un type d'association ne peut être que de type 1-N (et N-1 dans l'ordre inverse).

Supposons les deux types d'entité CLIENT et VOITURE. Si le type d'association PROPRIETAIRE est de type 1-N (one to many) de CLIENT vers VOITURE, cela signifie que chaque entité CLIENT peut être associée avec n'importe quel nombre (appelé N) d'entités VOITURE, et que chaque entité VOITURE peut être associée avec seulement une entité CLIENT. Il correspond donc au type d'entité CLIENT des connectivités (0,N) et au type d'entité VOITURE des connectivités (0,1). Ce sont les seules connectivités possibles dans le modèle entité-association réduit. Elles sont "cachées" derrière le terme "type d'association de type 1-N".

Le point important est la "position" de chaque type d'entité par rapport au type d'association. En effet, parler de type d'association de type 1-N de CLIENT vers VOITURE est différent de type d'association de type 1-N de VOITURE vers CLIENT. Dans le premier cas, les connectivités de CLIENT et de VOITURE sont respectivement (0,N) et (0,1). Dans le second cas, elles sont inversées, (0,1) pour CLIENT et (0,N) pour VOITURE. Pour éviter toute confusion on dira que CLIENT est le type d'entité origine du type d'association PROPRIETAIRE (connectivités (0,N)) et VOITURE en est le type d'entité cible (connectivités (0,1)). Cela ne signifie nullement que l'on ne puisse plus accéder au CLIENT à partir de VOITURE. Un type d'entité origine d'un type d'association aura donc toujours des connectivités de (0,N) et un type d'entité cible, des connectivités de (0,1). L'accès dans les deux sens reste toujours possible.

Au lieu de parler de type d'association de type 1-N d'un type d'entité (CLIENT) vers un autre (VOITURE), on peut tout aussi bien parler de type d'association de type N-1 d'un type d'entité (VOITURE) vers un autre (CLIENT). La signification est strictement identique. VOITURE en serait l'entité origine et CLIENT l'entité cible. Dans le modèle entité-assocation réduit, on définira les types d'association uniquement de type 1-N en en spécifiant l'entité origine et l'entité cible.

- *type d'un attribut.*

Tout attribut peut être simple ou répétitif, élémentaire ou décomposable. Un attribut ne peut être facultatif. Il sera donc toujours obligatoire ("mandatory").

Un attribut élémentaire peut être de type entier, caractère, booléen, réel et chaîne de caractères.

- *les identifiants.*

On ne peut définir des identifiants que sur des types d'entité. Un type d'entité ne peut avoir qu'au plus un identifiant. Un identifiant est composé d'un et d'un seul attribut du type d'entité sur lequel il est défini. Cet attribut doit être simple et élémentaire (et obligatoire).

Représentation:

Voici les conventions prises pour la représentation graphique des différents concepts de ce modèle:

- un type d'entité avec ses attributs sera représenté de la manière suivante (représentation identique à celle du modèle entité-association):
Soit un type d'entité CLIENT avec les attributs NOM et PRENOM, en voici la représentation

Figure III.2.1: représentation d'un type d'entité.

- un type d'association (de type 1-N) entre deux types d'entité sera représenté de la manière suivante:
Soit les types d'entité CLIENT et VOITURE et le type d'association PROPRIETAIRE de type 1-N de CLIENT vers VOITURE. CLIENT est l'entité origine et VOITURE est l'entité cible de ce type d'association. En voici la représentation:

Figure III.2.2: représentation d'un type d'association de type 1-N.

Le triangle symbolise la "direction" du type d'association de type 1-N. Le sommet indique l'entité origine, CLIENT, (côté 1 du type d'association) et la base, l'entité cible, VOITURE (côté N du type d'association).

III.2.2 Les primitives offertes par le SGBD PYRAMIDE

L'interface de programmation offerte par PYRAMIDE est destinée aux programmeurs d'application travaillant en Pascal tel qu'il apparaît dans la version 5 de Turbo-Pascal. En effet, elle est constituée d'un ensemble de procédures Pascal génériques incluses dans une "Unit" Turbo-Pascal. Il suffit donc pour pouvoir utiliser ces primitives d'inclure cette "Unit" dans le programme d'application par un ordre "Uses Pyramide;". L'interface se soumettant aux règles Pascal, elle doit également se soumettre à ses types de données. On trouvera donc également au début de tout programme utilisant l'interface un ordre "(*$I file-name.typ*)" ,où file-name est un nom de fichier. Ce fichier permet de disposer de l'ensemble des types Pascal correspondant aux types d'entité et d'association du schéma entité-association réduit considéré. Ce fichier ".typ" résulte d'un mapping des structures de données du SGBD PYRAMIDE vers les structures de données Pascal correspondantes et sur lesquelles travailleront les primitives de l'interface. Ce mapping est réalisé par un utilitaire du SGBD, le METACOMPILATEUR. C'est lui qui génère le fichier ".typ" à partir de la description d'un schéma entité-association réduit.

Les procédures de l'interface permettent d'ouvrir et de fermer une ou plusieurs bases de données en même temps, d'accéder à des entités d'un type donné séquentiellement ou directement sur base d'une valeur d'identifiant, d'accéder séquentiellement à des entités liées à d'autres via une association, de modifier, créer, supprimer des entités et des association et de manipuler des variables.

Les **concepts généraux** de l'interface sont ceux de:

- référence de base de données: pointeur qui référence une base de données qui a été ouverte. Il permet de sélectionner lors de l'ouverture simultanées de plusieurs bases de données celle sur laquelle on veut effectivement travailler.

- référence: type de données (DbRef) dont la valeur peut désigner une entité. La valeur NULL signifie qu'aucune entité n'est référencée.

- variable de référence: variable Pascal de type DbRef. Elle peut désigner une entité de n'importe quel type. En effet, chaque entité est identifiée par une valeur (référence) système unique (surrogate-key).

- variable entité: variable Pascal d'un des types de variable d'entité générés automatiquement par le métacompilateur dans le fichier ".typ".

-désignateur: le type d'entité ou d'association sur lequel une primitive agit est désigné par son nom. Celui-ci est en réalité une constante entière définie au niveau du fichier ".TYP".

Les **arguments** des procédures sont les suivants:

- DataBase: c'est un string ou une constante donnant le nom sans extension d'une base de données (incluant éventuellement un chemin).

- DbDesc: c'est une variable de type DDB.

- Entity_type: c'est une expression entière donnant le code numérique d'un type d'entité valide de la base de données. Ces codes sont générés dans le fichier ".typ" sous la forme de constantes entières prédéfinies nommées selon les types d'entité qu'elles désignent.

- Path_type: c'est une expression entière donnant le code numérique d'un type d'association valide de la base de données. Ces codes sont générés dans le fichier ".typ" sous forme de constantes entières prédéfinies nommées selon les types d'association qu'elles désignent.

- Ent_var: variable entité de type TEntity_type_name. Ce type est prédéfini dans le fichier ".TYP" et permet de passer au SGBD les valeurs d'attributs et/ou la référence de l'entité sur laquelle agit la primitive.

```
Ex:  tclient = record
                    xxkey : dbref;
                    xxcode : char;
                    name : string[30];
               end;

  ...
    var cli : tclient;      (cli est une "Ent_var")
```

- Origin: variable entité de type TEntity_type_name désignant une entité origine du type d'association, ç'est-à-dire l'entité à partir de laquelle l'utilisateur a décidé de "traverser" le type d'association.

- Target: variable entité de type TEntity_type_name désignant une entité cible du type d'association ç'est-à-dire l'entité vers laquelle on a décidé d'aller en "traversant" le type d'association.

- Ref_Var: variable de référence de type Dbref.

- R/E_Var: désigne soit une Ref_Var soit une Ent_var.

L'exécution des procédures de l'interface provoque la mise-à-jour d'un code de retour contenu dans une variable d'erreur Dbstatus qui indique comment la procédure s'est terminée. En voici les valeurs possibles et leur signification:

0 : opération bien terminée.
1 : objet requis (entité, association, base de données) introuvable.
2 : violation d'une contrainte d'identifiant
10 : code de type d'entité incorrect
11 : code de type d'association incorrect
30 : valeur de référence donnée en entrée incorrecte
70 : mémoire principale insuffisante
80 : espace disque insuffisant
90 : base de données corrompue
99 : erreur système ou d'entrée-sortie

Les **procédures** de l'interface sont les suivantes:

- ouverture et fermeture d'une base de données

DBOPEN (Database, var dbdesc) : ouvre la base de données et place sa référence dans dbdesc.

DBCLOSE (var dbdesc) : ferme la base de données référencée par dbdesc.

- accès séquentiel sur un type d'entité

DBFIRST (Entity_type, var Ent_var) : recherche la première entité du type Entity_type et la place dans la variable Ent_var.

DBLAST (Entity_type, var Ent_var) : recherche la dernière entité du type Entity_type et la place dans la variable Ent_var.

DBNEXT (Entity_type, var Ent_var): accède à l'entité suivante à celle référencée par Ent_var et de type Entity_type. Elle est stockée dans Ent_var. Si Ent_var ne référence aucune entité, l'opération est équivalente à un dbfirst.

DBPRIOR (Entity_type, var Ent_var): accède à l'entité précédant celle référencée par Ent_var et de type Entity_type. Elle est stockée dans Ent_var. Si Ent_var ne référence aucune entité, l'opération est équivalente à un dbfirst.

- accès à une entité par sa valeur d'identifiant

DBID (Entity_type, var Ent_var) : accède à l'entité de type Entity_type identifiée par la valeur identifiante stockée dans la variable Ent_var.

- accès direct à une entité

DBDIRECT (Entity_type, var Ent_var, var R/E_var): accède à l'entité de type Entity_type référencée par la variable Ent_var ou par la variable de référence R/E_var et stocke ses valeurs dans la variable Ent_var.

- parcours d'un type d'association

DBFPATH (var target, var origin, Path_type) : accède à la première entité connectée à l'entité origin via le type d'association Path_type. Ses valeurs d'attributs et sa référence sont stockées dans target.

DBLPATH (var target, var origin, Path_type) : accède à la dernière entité connectée à l'entité origin via le type d'association Path_type. Ses valeurs d'attributs et sa référence sont stockées dans target.

DBNAPTH (var target, var origin, Path_type) : accède à l'entité qui suit l'entité target parmi celles qui sont connectées à l'entité origin via le type d'association Path_type. Ses valeurs d'attributs et sa référence sont stockées dans target.

DBPPATH (var target, var origin, Path_type) : accède à l'entité qui précède l'entité target parmi celles qui sont connectées à l'entité origin via le type d'association Path_type. Ses valeurs d'attributs et sa référence sont stockées dans target.

Remarque: une valeur négative pour Path_type signifie qu'au lieu de se diriger de l'entité qui joue le rôle "ORIGIN" du type d'association vers l'entité qui joue le rôle "TARGET", on se dirige du type d'entité "TARGET" vers le type d'entité "ORIGIN" du type d'association . On "traverse" le type d'association dans le sens inverse (N-1).

- création d'une entité

DBCREATE (Entity_type, var Ent_var) : insére une entité de type désigné par Entity_type dont les valeurs sont contenues dans la variable Ent_var. La référence de l'entité créée est stockée dans Ent_var.

- suppression d'une entité

DBDELETE (Entity_type, var Ent_var) : efface l'entité de type Entity_type désignée par la variable Ent_var. La référence contenue dans Ent_var est mise à la valeur NULL (Ent_var ne référence plus d'entité dans la base de données).

- modification d'une entité

DBMODIFY (Entity_type, var Ent_var): modifie l'entité de type Entity_type désignée par Ent_var avec les valeurs contenues dans Ent_var.

- création d'une connection

DBINSERT (var target, var origin, Path_type) : connecte l'entité référencée par target à l'entité référencée par origin via le type d'association Path_type.

- suppression d'une connection

DBREMOVE (var target, var origin, Path_type) : efface la connection qui existait entre l'entité référencée par target et l'entité référencée par origin via le type d'association Path_type.

- fonctions de diagnostic

DBFOUND et DBNOTFOUND : boolean : indique si la dernière primitive exécutée s'est bien teminée ou non.

- manipulation de variables

DBCLEAR (var Ent_var) : met la référence de la variable Ent_var à NULL.

DBCOPYATT (Entity_type, var Ent_var1, var Ent_var2) : copie les valeurs de tous les attributs contenus dans Ent_var1 vers Ent_var2. La référence et le type restent inchangés.

DBCOPYALL (Entity_type, var Ent_var1, var Ent_var2) : copie entièrement la variable Ent_var1 vers Ent_var2.

DBCOPYREF (var Ent_var1, var Ent_var2) : copie la référence contenue dans la variable Ent_var1 vers Ent_var2.

DBEQUAL (var Ent_var1, var Ent_var2) : boolean : teste si les références contenues dans Ent_var1 et Ent_var2 sont égales.

DBNULL (var Ent_var) : boolean : teste si la référence contenue dans Ent_var vaut NULL.

- manipulation de plusieurs bases de données

DBSELECT (dbdesc) : la base de données référencée par dbdesc devient la base active.

- intégrité

DBBGTR (var ta_id : integer) : une transaction est commencée et son identifiant est placé dans ta_id.

DBENDTR (ta_id : integer) : la transaction identifiée par ta_id ainsi que toutes ses transactions "descendantes" sont cloturées.

DBABTR (ta_id : integer) : la transcation identifiée par ta_id ainsi que toutes ses transactions "descendantes" sont annulées.

Remarque: ces primitives concernant l'intégrité de la base de données ne pas implémentées dans la version actuelle de Pyramide.

III.3. Le modèle entité-association complet

III.3.1. Utilité

Il n'existe pas de norme qui permette de définir un modèle entité-association commun à tous. Ainsi, en général, les développeurs commencent par définir leur propre modèle. Comme on l'a déjà vu dans l'état de l'art (chapitre II.2.), cette définition est liée à un objectif. Si l'objectif est la simplification, le modèle qui sera utilisé sera un modèle astreint à certaines contraintes sur l'utilisation de ses concepts. Par exemple, pour le langage Hiquel, le modèle entité-association est réduit à un modèle binaire, avec des associations sans attribut. Le modèle est donc en susbstance un modèle de type réseau. Cela a évidemment comme effet la perte d'une partie importante de la puissance sémantique du modèle entité-association.

Une autre approche est de définir un modèle étendu. L'objectif est alors l'inverse du précédent, c'est-à-dire, fournir un outil basé sur un modèle puissant intégrant généralement des structures qui sont considérées comme des extensions du modèle entité-association. Par exemple, le modèle ERC, servant de base à l'algèbre du même nom intègre des structures de généralisation/spécialisation, des structures d'attributs complexes, multivalués, ... Le problème se situe alors au niveau de l'implémentation qui est évidemment d'autant plus complexe que l'est le modèle.

On voit donc que la première chose à faire est de se donner un cadre de travail, c'est-à-dire un modèle entité-association, d'une part au niveau des concepts qui le composeront, et d'autre part au niveau des contraintes que l'on imposera sur l'utilisation de ces concepts. Avant cela, il convient de se fixer un objectif.

Notre objectif est de fournir un prototype d'outil de programmation intégrant les concepts du modèle entité-association. Le modèle doit donc être tel que sa puissance sémantique soit suffisante pour que l'on puisse le qualifier d'outil entité-association. Par exemple, le modèle ne réduira pas les associations au seul degré binaire. Cependant, la vocation de prototype du langage qui devra se baser sur ce modèle engendre certaines conséquences. Ainsi, certaines constructions qui sont considérées comme des extensions au modèle ne seront pas implémentées. Dans ces extensions, on trouvera les structures de spécialisation/généralisation, certaines contraintes d'intégrité,...

III.3.2 Particularités par rapport au modèle théorique

III.3.2.1. Le concept de schéma

On peut considérer que la tâche de modélisation d'une application en utilisant le modèle EA consiste à concevoir un schéma. Dès lors, il est indispensable que le modèle inclue la notion de schéma, afin que l'utilisateur puisse accéder à un schéma qu'il a conçu, c'est-à-dire à tous les type d'entité et type d'association qui le composent.

Un schéma possède un nom obligatoire, et une définition facultative. Deux schémas ne peuvent avoir le même nom. Un schéma peut être constitué d'un ensemble vide de TE et TA.

III.3.2.2. Le concept d'entité

Un type d'entité est caractérisé par un nom obligatoire, et une définition facultative qui est la définition constitutive du type. Plusieurs types d'entité d'un même schéma ne peuvent avoir le même nom.

III.3.2.3. Le concept d'association

Un type d' association relie au moins deux, et au plus n types d'entité (n n'étant limité que par les possibilités du SGBD). Un même type d'entité peut être relié plusieurs fois au même type d'association. Un type d'association possède un nom obligatoire, et une définition facultative. Plusieurs types d'association d'un même schéma ne peuvent avoir le même nom. Le modèle ne gère pas les contraintes et les mécanismes d'héritage liés à des structures de généralisation entre type d'entité ou type d'association.

Dans l'ensemble des noms de TE et TA d'un schéma, il ne peut exister deux noms identiques.

III.2.3.4. Le concept d'attribut

Un attribut possède un nom obligatoire. Un type d'entité ou d'association peuvent ne pas avoir d'attribut. Si un TE ou un TA possèdent plusieurs attributs, les attributs à l'intérieur de ce TE ou de ce TA doivent être de nom unique.

Plusieurs types d'entité peuvent avoir un attribut de même nom, ainsi que plusieurs types d'association, et qu'une combinaison de types d'entité et de types d'association. Un attribut peut être simple, répétitif, élémentaire, ou décomposable. Il peut être obligatoire ou facultatif.

Un attribut possède également une définition facultative, ainsi qu'un type de valeur, qui vaudra 'group' pour un attribut décomposable, et qui pourra prendre les valeurs suivantes sinon:

- c(n), avec n compris entre 1 et 256 pour les valeurs de type texte, où n est la longueur de la chaîne de caractères
- n(i,j), pour les valeurs numériques, i représente le nombre de chiffres avant la virgule et j représente le nombre de chiffres après la virgule (une valeur entière sera donc représentée par n(4,0)
- date pouvant prendre une valeur valide pour une date selon le format annéemoisjour (par exemple,760123)

Un attribut ne peut pas prendre la valeur inconnue.

III.3.2.5. Rôles et connectivités

La participation d'un type d'entité à un type d'association est caractérisée par l'existence d'un rôle, d'une connectivité minimale et d'une connectivité maximale. Un rôle possède un nom. Le nom d'un rôle est identifiant d'un rôle, à l'intérieur d'un schéma. Les connectivités sont représentées par deux valeurs. Dans tous les cas de figure, ces valeurs doivent vérifier les contraintes suivantes:

-connectivité minimale: 0 ou 1
-connectivité maximale: 1 ou n (n n'est limité que par les possibilités du SGBD)

III.3.2.6. Contraintes d'identifiant

Un type d'entité ou d'association peut être identifié par un de ses attributs, qui doit être obligatoire, simple, et élémentaire. Sa répétitivité doit donc être de 1-1. Tout type d'entité ou d'association ne

peut posséder qu'un seul identifiant au maximum. Rappelons que le dictionnaire de données permet le stockage de structures plus complètes (par exemple, identifiant comprenant des rôles, plusieurs identifiants par type d'entité ou d'association). L'utilisateur ne devra cependant pas stocker de telles structures non admises par le modèle entité-association complet, sous peine de voir la précompilation annulée.

Pour ce type de contrainte, nous avons choisi de nous limiter aux fonctionnalités de PYRAMIDE. En effet, tout identifiant ou structure d'identifiant non gérés par PYRAMIDE devrait l'être par la couche supérieure. Cette gestion des valeurs identifiantes se résumerait à deux choses:

- un accès séquentiel sur le TE ou TA identifié avec test sur la valeur identifiante pour simuler un accès direct à l'entité ou l'association identifiée,
- le refus d'accepter une valeur déjà présente pour l'attribut identifiant.

Ces deux fonctionnalités n'apportent pas beaucoup du point de vue des performances et sont facilement programmables par l'utilisateur. Nous avons donc estimé que le rapport coût/apport n'était pas suffisant pour inclure ces extensions au modèle. L'utilisateur pourra néanmoins facilement simuler par exemple un identifiant secondaire pour un TE ou un TA.

Notre choix de n'accepter que les attributs obligatoires, simples, et facultatifs comme identifiant vient du fait que PYRAMIDE n'accepte que cette structure. Si le modèle acceptait un attribut décomposable ou répétitif comme identifiant, la couche supérieure devrait transformer cet identifiant en une structure acceptée par PYRAMIDE, or ce mécanisme de transformation serait assez complexe, étant donné les manques de PYRAMIDE à ce niveau.

D. Rossi ne donne pas d'indication sur la possibilité d'étendre PYRAMIDE pour accepter des attributs décomposables ou répétitifs dans un identifiant. Il ne nous est donc pas possible d'assurer que le système LEA peut être étendu pour accepter ces fonctionnalités.

Ici aussi donc, l'utilisateur devra transformer ses identifiants afin de les conformer au modèle entité-association complet.

Enfin, nous avons choisi de ne stocker dans le dictionnaire que des structures acceptées par le modèle EAC. Cela facilite le travail du précompilateur. Par exemple, s'il était permis de stocker plusieurs identifiants par TE ou TA dans le dictionnaire, sachant qu'un seul sera transformé, il faudrait déterminer un moyen qui permette au précompilateur de choisir l'identifiant à transformer. Il serait possible d'ajouter un "signe" dans le dictionnaire indiquant l'identifiant à transformer, cependant, cela aurait pour effet d'ajouter au dictionnaire des éléments propres au processus de transformation. Cela va à l'encontre de notre choix de limiter le processus de transformation au précompilateur. Il serait possible aussi d'ajouter des conventions telles que par exemple le premier identifiant transformé est le premier stocké dans le dictionnaire. Ce choix ne nous paraît pas intéressant, car il ajouterait au précompilateur des éléments qui n'auraient pour effet que de le compliquer inutilement.

Notons que d'après D. Rossi, il serait possible assez facilement que PYRAMIDE gère plusieurs identifiants. Il serait alors aussi facile de modifier la couche supérieure pour récupérer cette fonctionnalité.

III.3.2.7. Contraintes non incluses dans le modèle

Etant donné la vocation de prototype du système LEA, il nous a semblé risqué d'inclure dans le modèle les contraintes facultatives du modèle théorique. Si l'utilisateur désire bénéficier de ce type de contraintes, il pourra les programmer dans le cadre de son application.

III.4. Le méta-schéma

Introduction

L'objectif de cette partie est de présenter le mécanisme utilisé par le système LEA pour permettre le stockage de schémas EAC. Ce mécanisme est le dictionnaire de données, dont la structure est celle d'un méta-schéma. On va d'abord voir comment il est possible de stocker les concepts du modèle EAC aussi bien que les concepts du modèle entité-association réduit grâce aux concepts du méta-schéma.

Comme le dictionnaire de données est au départ une ocmposante de PYRAMIDE, on verra les rôles et la gestion du méta-schéma pour PYRAMIDE.

On verra enfin de quelle façon est généralisée la notion de méta-schéma pour le système LEA.

III.4.1. Rôle principal d'un méta-schéma

La fonction essentielle d'un méta-schéma est de pouvoir contenir la description de schémas de données d'applications. Un méta-schéma est donc un "super" schéma dont les données constituent la description de schémas. Ces données sont appelées des méta-données. En particulier, un méta-schéma peut contenir sa propre description.

III.4.2. Les concepts du méta-schéma

Chaque concept du modèle EAC, et du modèle entité-association réduit, sera considéré individuellement. On verra comment on peut représenter chacun d'entre eux dans un schéma conforme au modèle réduit (le méta-schéma). Ces concepts sont ceux de schéma, d'entité, d'association avec les notions de rôle et de connectivité, d'attribut d'une association ou d'une entité, et d'identifiant.

III.4.2.1. Le concept de schéma

Le concept central est celui de schéma. Un schéma est identifié par un nom unique (pour permettre de distinguer plusieurs schémas dans le méta-schéma). Il peut également avoir une description dont la longueur en nombre de caractères n'est pas limitée.

Représentation dans le méta-schéma:

On trouve dans le méta-schéma un type d'entité de nom DBSCHEMA qui a un attribut de nom NAME de type string de 32 caractères. A un schéma correspond une occurrence de ce type d'entité dont la valeur de l'attribut NAME est égale au nom du schéma. Comme au niveau entité-association il faut obligatoirement donner un nom à un schéma, il est donc obligatoire que l'attribut NAME du type d'entité DBSCHEMA ait une valeur. De plus, un méta-schéma pouvant contenir la description de plusieurs schémas , il est donc indispensable de pouvoir distinguer les occurrences du type d'entité DBSCHEMA. Pour cela, l'attribut NAME est identifiant.

On a également un type d'entité de nom DB_DESC avec un attribut de nom DESCRIPTOR de type string de 8O caractères. Le type d'entité DBSCHEMA sera lié au type d'entité DB_DESC par un type d'association de type 1-N appelé DB_DBDESC. Les valeurs de l'attribut DESCRIPTOR du type d'entité DB_DESC sont égales à des morceaux de 8O caractères de la description du schéma. Cette façon de procéder permet de disposer pour une occurrence du type d'entité DBSCHEMA d'un nombre quelconque d'occurrences du type d'entité DB_DESC et donc ainsi d'avoir une description d'un schéma aussi longue que l'on veut tout en minimisant la place occupée sur le disque (la taille stockée est plus ou moins égale à la taille de la description). A une occurrence du type d'entité DBSCHEMA correspondent de 0 à N occurrence(s) du type d'entité DB_DESC. A une occurrence du type d'entité DB_DESC correspond une et une seule occurrence du type d'entité DBSCHEMA.

Figure III.4.1: Représentation du concept de schéma dans le méta-schéma.

La représentation d'un schéma conforme au modèle entité-association réduit sera identique à la représentation d'un schéma conforme au modèle EAC.

Remarque: il faut noter que pour des raisons techniques (entre autre pour pouvoir distinguer un schéma EAC de son schéma entité-association réduit correspondant) le nom d'un schéma EAC sera préfixé automatiquement du caractère "$". Cela est réalisé automatiquement par le précompilateur (voir 4ème partie).

III.4.2.2. Le concept d'entité

Dans un schéma EAC on peut trouver un ensemble de types d'entité. Chaque type d'entité est identifié par un nom unique dans le schéma. On peut donc, dans le méta-schéma, représenter l'ensemble des types d'entité d'un schéma entité-association par un type d'entité ENTITY_TYPE dont un attribut est NAME de type string de 32 caractères. A chaque type d'entité d'un schéma EAC correspond une occurrence du type d'entité ENTITY_TYPE dont la valeur de l'attribut NAME est égale au nom du type d'entité dans le schéma EAC.

Au niveau du schéma EAC, un type d'entité peut également avoir une description. Elle est représentée dans le méta-schéma de la même manière que la description d'un schéma (voir III.4.1). On a donc un type d'entité ET_DESC avec un attribut DESCRIPTOR de type string de 80 caractères. Un type d'association ET_ETDESC de type 1-N relie le type d'entité ENTITY_TYPE au type d'entité ET_DESC. A chaque occurrence du type d'entité ENTITY_TYPE correspondent de 0 à N occurrence(s) du type d'entité ET_DESC. A chaque occurrence du type d'entité ET_DESC correspond une et une seule occurrence du type d'entité ENTITY_TYPE.

Un problème est qu'un méta-schéma peut contenir la description de plusieurs schémas en même temps (au moins celles du schéma EAC et du schéma entité-association réduit correspondant). Il n'est donc plus possible d'identifier les types d'entité uniquement par leur nom. Deux types d'entité de deux schémas différents peuvent avoir des noms identiques. Il faut donc, pour pouvoir identifier les différents types d'entité, relier chaque type d'entité au schéma auquel il appartient. Pour ce faire, le méta-schéma contient un type d'association de type 1-N appelé DBSCHEMA_ET entre le type d'entité DBSCHEMA et le type d'entité

ENTITY_TYPE. A chaque occurrence du type d'entité DBSCHEMA correspondent de 0 à N occurrence(s) du type d'entité ENTITY_TYPE. A chaque occurrence du type d'entité ENTITY_TYPE correspond une et une seule occurrence du type d'entité DBSCHEMA. Toutes les occurrences du type d'entité ENTITY_TYPE liées à la même occurrence du type d'entité DBSCHEMA ont toutes une valeur de leur attribut NAME différente (c'est la notion d'identifiant par rapport à un schéma).

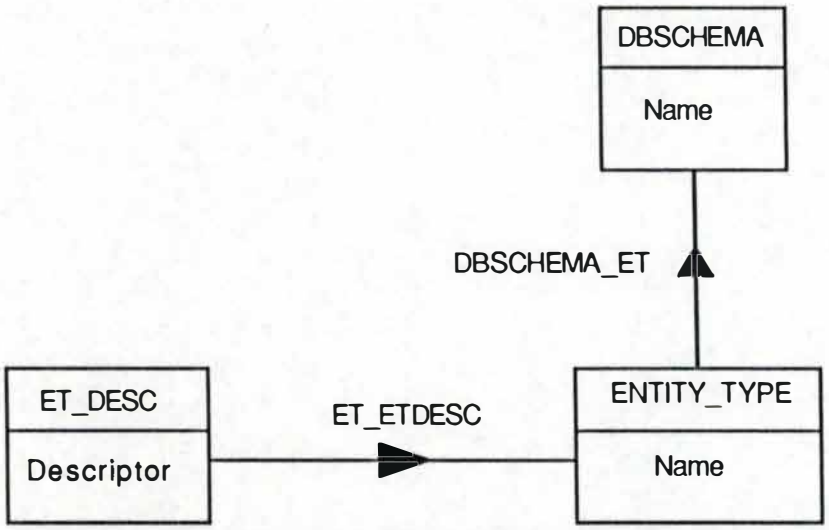


Figure III.4.2: Représentation du concept de type d'entité dans le méta-schéma.

Pour un type d'entité d'un schéma conforme au modèle entité-association réduit, c'est exactement la même chose. Il lui correspondra une occurrence du type d'entité ENTITY_TYPE liée à l'occurrence du type d'entité DBSCHEMA correspondant au schéma auquel il appartient. Des occurrences du type d'entité ET_ETDESC peuvent être liées à cette occurrence de ENTITY_TYPE.

III.4.2.3. Le concept d'association

On peut également trouver dans un schéma EAC un ensemble de types d'association. Chaque type d' association est identifié par un nom unique au schéma et est relié à deux ou plusieurs types d'entité non nécessairement distincts avec pour chacun un rôle et des connectivités minimales et maximales.

On peut représenter dans le méta-schéma l'ensemble des types d'association d'un schéma EAC par un type d'entité REL_TYPE dont un attribut est NAME de type string de 32 caractères. A chaque type d'association dans le schéma EAC, identifié par un nom, correspond une

occurrence du type d'entité REL_TYPE dont la valeur de NAME est égale au nom du type d'association dans le schéma.

Au niveau du schéma EAC, un type d'association peut également avoir une description. Elle est représentée dans le méta-schéma de la même manière que celle d'un schéma ou d'un type d'entité (voir III.4.1 ou III.4.2). On a donc un type d'entité RT_DESC avec un attribut DESCRIPTOR de type string de 80 caractères. Un type d'association de type 1-N appelé RT_RTDESC relie le type d'entité REL_TYPE au type d'entité RT_DESC. A chaque occurrence du type d'entité REL_TYPE correspondent de 0 à N occurrence(s) du type d'entité RT_DESC. A chaque occurrence du type d'entité RT_DESC correspond une et une seule ocurence du type d'entité REL_TYPE.

Comme pour les types d'entité d'un schéma il faut pouvoir distinguer des types d'association ayant des noms identiques mais dans des schémas différents. Pour ce faire, on a un type d'association DBSCHEMA_RT de type 1-N du type d'entité DBSCHEMA vers le type d'entité REL_TYPE. A chaque occurrence du type d'entité DBSCHEMA correspondeont de 0 à N occurrence(s) du type d'entité REL_TYPE. A chaque occurrence du type d'entité REL_TYPE correspond une et une seule occurrence du type d'entité DBSCHEMA. Toutes les occurrences du type d'entité REL_TYPE liées à une même occurrence du type d'entité DBSCHEMA ont une valeur de NAME différente (notion d'identifiant par rapport à un schéma).

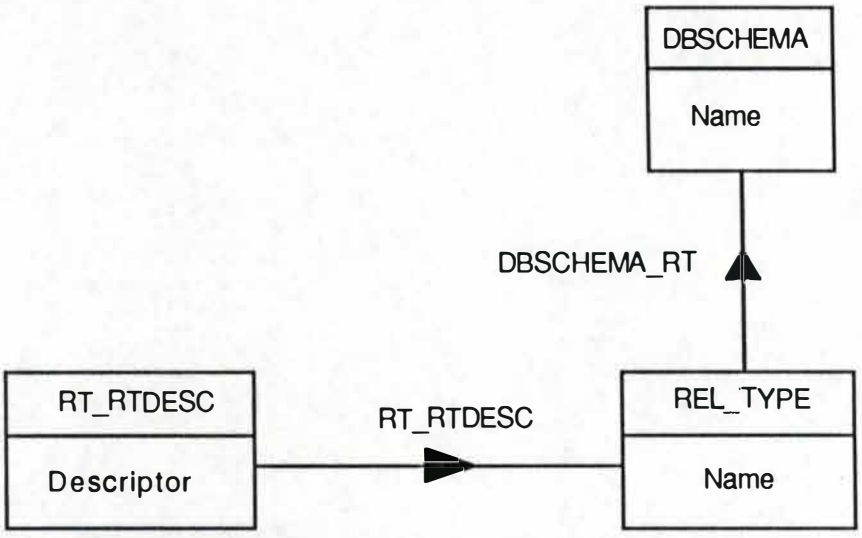


Figure III.4.3: Représentation du concept de type d'association dans le méta-schéma

Il faut également représenter la participation des différents types d'entité non nécessairement distincts à un type d'association. Ce fait est modélisé par le concept de rôle. Le méta-schéma contient donc un type d'entité appelé ROLE qui est composé de trois attributs

- NAME de type string de 32 caractères dont la valeur représente un nom de rôle dans le schéma
- MIN_CON de type entier dont la valeur représente la connectivité minimale du rôle
- MAX_CON de type entier dont la valeur représente la connectivité maximale du rôle.

On a un type d'association de type 1-N appelé ET_ROLE entre les types d'entité ENTITY_TYPE et ROLE et un autre appelé RT_ROLE entre les types d'entité REL_TYPE et ROLE. Pour chaque rôle d'un schéma EAC on trouve une occurrence du type d'entité ROLE qui y correspond (valeur de l'attribut NAME égale au nom du rôle). Elle est liée via le type d'association ET_ROLE à l'occurrence de ENTITY_TYPE correspondant au type d'entité qui joue ce rôle et via le type d'association RT_ROLE à l'occurrence de REL_TYPE qui correspond au type d'association pour lequel ce rôle est joué.

A chaque occurrence du type d'entité ENTITY_TYPE correspondent de 0 à N occurrence(s) du type d'entité ROLE. A chaque occurrence du type d'entité REL_TYPE correspondent de 2 à N occurrences du type d'entité ROLE. A chaque occurrence du type d'entité ROLE correspondent une et une seule occurrence du type d'entité ENTITY_TYPE et une et une seule occurrence du type d'entité REL_TYPE.

Cette manière de procéder résoud le problème des types d'entité qui participent plusieurs fois à un même type d'association (ex: type d'association père-fils entre le type d'entité personne et le type d'entité personne). En effet, une contrainte d'utilisation du modèle EAC est que si le même type d'entité participe plusieurs fois au même type d'association, il y joue des rôles différents. Entre une même occurrence du type d'entité ENTITY_TYPE et une même occurrence du type d'entité REL_TYPE on peut avoir deux occurrences du type d'entité ROLE. Chacune de ces occurrences du type d'entité ROLE est caractérisée par la valeur unique de son attribut NAME.

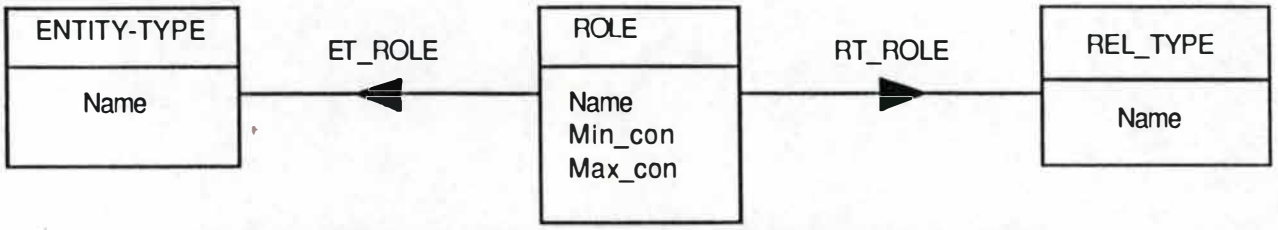


Figure III.4.4: Représentation de la notion de rôle dans le méta-schéma.

Pour un type d'association d'un schéma conforme au modèle entité-association réduit, la représentation est pratiquement identique. Il lui correspond une occurrence du type d'entité REL_TYPE liée à l'occurrence du type d'entité DBSCHEMA correspondant au schéma auquel il appartient. Des occurrences du type d'entité RT_RTDESC peuvent être liées à cette occurrence de REL_TYPE.

La seule différence se situe au niveau du type d'entité ROLE. En effet, le type d'association ne peut être que binaire, entre deux types d'entité qui jouent des rôles bien spécifiques à savoir ORIGIN et TARGET (qui indiquent la "direction" du type d'association de type 1-N) et dont les connectivités sont respectivement (0,N) et (0,1). Toute occurrence du type d'entité ROLE a donc soit pour valeur de l'attribut NAME la valeur "ORIGIN" et dans ce cas les valeurs des attributs MIN_CON et MAX_CON seront 0 et N, soit pour valeur de l'attribut NAME la valeur "TARGET" et dans ce cas les valeurs des attributs MIN_CON et MAX_CON sont 0 et 1. A une occurrence du type d'association sont toujours liées deux et seulement deux occurrences du type d'entité ROLE.

III.4.2.4. Le concept d'attribut

Un autre concept du modèle est celui d'attribut, que ce soit d'un type d'entité ou d'un type d'association. En effet, à chaque type d'entité ou d'association d'un schéma EAC peuvent être associés de 0 à N attributs.

On peut représenter dans le méta-schéma un attribut par un type d'entité appelé ATTRIBUTE dont les attributs sont NAME de type string de 32 caractères, VAL_TYPE de type caractère, VAL_LENGTH de type entier, DEC de type entier, MIN_REP et MAX_REP de type entier. A chaque attribut d'un type d'entité ou d'association dans le schéma EAC correspond une occurrence du type d'entité ATTRIBUTE dont la valeur des attributs est fonction de la description de l'attribut dans le schéma EAC.

Voici la signification des différents attributs du TE ATTRIBUTE:

- NAME représente le nom de l'attribut
- VAL_TYPE représente le type de valeur que peut contenir l'attribut. Les types de valeur possibles sont C (caractère), B (booléen), N (numérique) et G (group). Cette valeur G indique que l'attribut est décomposable.
- VAL_LENGTH représente la longueur maximale de l'attribut.
- DEC représente le nombre de décimales d'un attribut de type numérique
- MIN_REP représente la répétitivité minimale de l'attribut
- MAX_REP représente la répétitivité maximale de l'attribut

On a également un type d'association de type 1-N appelé ET_ATT entre les types d'entité ENTITY_TYPE et ATTRIBUTE, un type d'association de type 1-N appelé RT_ATT entre les types d'entité REL_TYPE et ATTRIBUTE et un type d'association de type 1-N appelé ATT_ATT entre le type d'entité ATTRIBUTE et lui-même. Ils permettent de lier une occurrence de ATTRIBUTE soit à une occurrence de ENTITY_TYPE (attribut d'un type d'entité), soit à une occurrence du type d'entité REL_TYPE (attribut d'une association), soit à une occurrence du type d'entité ATTRIBUTE (attribut fils d'un attribut décomposable).

A chaque occurrence du type d'entité ENTITY_TYPE ou REL_TYPE correspondent de 0 à N occurrence(s) du type d'entité ATTRIBUTE.

A une occurrence du type d'entité ATTRIBUTE peuvent être liées de 0 à N occurrence(s) du type d'entité ATTRIBUTE (0 si attribut non décomposable et de 1 à N si attribut décomposable). c'est-à-dire avoir de 0 à N descendant(s). S'il y a des descendants la valeur de l'attribut VAL_TYPE de l'occurrence du type d'entité ATTRIBUTE qui a des descendants est égale à G.

Toute occurrence du type d'entité ATTRIBUTE liée à une autre occurrence de ce type d'entité (attribut fils d'un attribut décomposable) est liée à la même occurrence du type d'entité ENTITY_TYPE ou REL_TYPE que l'attribut "père".

On peut donc dire que chaque occurrence du type d'entité ATTRIBUTE est liée

- soit à une et une seule occurrence du type d'entité ENTITY_TYPE (attribut "direct" d'un type d'entité)

- soit à une et une seule occurrence du type d'entité REL_TYPE (attribut "direct" d'un type d'association)
- soit à une et une seule occurrence du type d'entité ATTRIBUTE dont la valeur de l'attribut VAL_TYPE est égale à G (attribut descendant d'un attribut décomposable) et à une occurrence du type d'entité ENTITY_TYPE ou REL_TYPE (la même que celle à laquelle est liée l'attribut "père").

Au niveau du schéma EAC, un type d'attribut peut également avoir une description. Elle est représentée au niveau du méta-schéma de la même manière que celle d'un schéma, d'un type d'entité ou d'un type d'association (voir III.4.1, III.4.2, ou III.4.3). On a donc un type d'entité ATT_DESC avec un attribut DESCRIPTOR de type string de 80 caractères. Un type d'association appelé ATT_ATTDESC de type 1-N relie le type d'entité ATTRIBUTE au type d'entité ATT_DESC. A chaque occurrence du type d'entité ATTRIBUTE correspondent de 0 à N occurrence(s) du type d'entité ATT_DESC. A chaque occurrence du type d'entité ATT_DESC correspond une et une seule ocurence du type d'entité ATTRIBUTE.

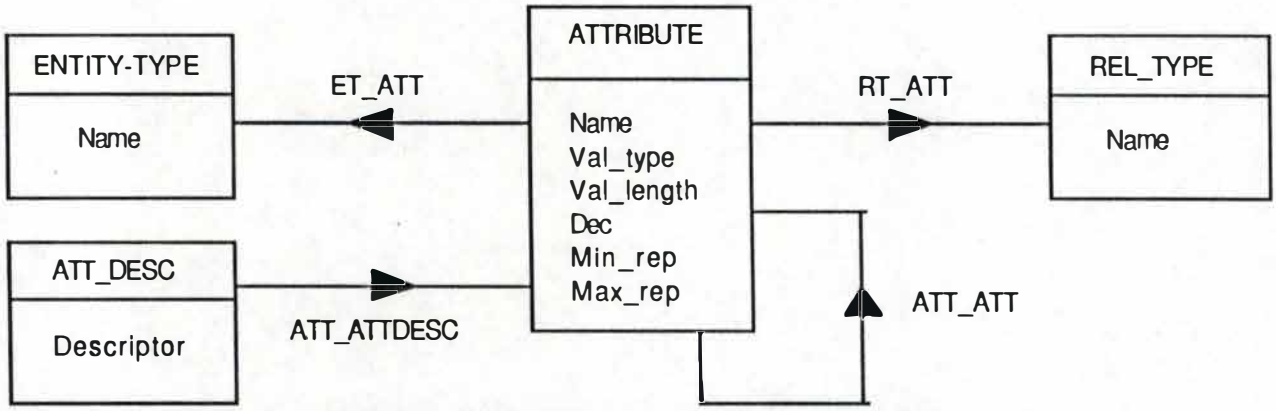


Figure III.4.5: Représentation du concept d'attribut dans le méta-schéma.

Dans le cas d'un attribut d'un schéma conforme au modèle entité-association réduit, le principe est identique mis à part le fait qu'une occurrence du type d'entité ATTRIBUTE ne peut pas être liée à une occurrence du type d'entité REL_TYPE car dans ce modèle un type d'association ne peut pas avoir d'attribut.

III.4.2.5. Le concept d'identifiant

On peut également trouver dans un schéma EAC un ensemble d'identifiants. Cela est représenté dans le méta-schéma par un type d'entité de nom IDENTIFIANT.

Remarque: Pour pouvoir intégrer le schéma dans un cadre plus général ce type d'entité sera en réalité appelé GROUP.

A chaque identifiant du schéma EAC correspond une occurrence du type d'entité GROUP dans le méta-schéma.

Un identifiant pouvant être aussi bien défini sur un type d'entité que sur un type d'association, il faut indiquer dans le méta-schéma pour chaque identifiant le type d'entité ou d'association sur lequel il est défini. Pour ce faire, on utilise deux types d'association de type 1-N:
- le premier va du type d'entité ENTITY_TYPE vers le type d'entité GROUP et est appelé ET_GROUP.
- le second va du type d'entité REL_TYPE vers le type d'entité GROUP et est appelé RT_GROUP. (voir figure III.4.6)

On peut donc dire qu'à chaque occurrence du type d'entité GROUP représentant un identifiant du schéma EAC correspond une et une seule occurrence
- soit du type d'entité ENTITY_TYPE représentant un type d'entité.
- soit du type d'entité REL_TYPE représentant un type d'association.

A chaque occurrence du type d'entité ENTITY_TYPE ou REL_TYPE peuvent correspondre de 0 à N occurrence(s) du type d'entité GROUP car dans un schéma EAC un type d'entité ou d'association peut avoir de 0 à N identifiant(s). Il faut donc pour une même occurrence du type d'entité ENTITY_TYPE ou REL_TYPE pouvoir distinguer les différentes occurrences du type d'entité GROUP qui lui sont liées. Pour ce faire, on place dans le type d'entité GROUP un attribut NUMBER de type entier. Toutes les occurrences du type d'entité GROUP liées à la même occurrence du type d'entité ENTITY_TYPE ou REL_TYPE ont une valeur de leur attribut NUMBER différente.

Dans le contexte d'un modèle entité-association général, tout identifiant peut être composé d'attributs et de rôles. Chaque identifiant est donc composé d'un certain nombre de composants. Chacun de ses composants est soit un attribut, soit un rôle. Pour pouvoir lier un identifiant (occurrence du type d'entité GROUP) avec les attributs (occurrence du type d'entité ATTRIBUTE) et les rôles (occurrences du type d'entité ROLE) qui le constituent, on dispose d'un type d'entité appelé COMPONENT. Un type d'association de type 1-N appelé GR_COMP lie le type

d'entité GROUP au type d'entité COMPONENT, un type d'association de type 1-N appelé ROLE_COMP lie le type d'entité ROLE avec le type d'entité COMPONENT et un type d'association de type 1-N appelé ATT_COMP lie le type d'entité ATTRIBUTE avec le type d'entité COMPONENT.

A une occurrence du type d'entité GROUP correspondent de 1 à N occurrence(s) du type d'entité COMPONENT (au moins une car un identifiant possède au moins un attribut ou un rôle), une occurrence par composant de l'identifiant.

A chaque occurrence du type d'entité COMPONENT correspond une et une seule occurrence du type d'entité GROUP.

A chaque occurrence du type d'entité COMPONENT correspond une et une seule occurrence, soit du type d'entité ATTRIBUTE, soit du type d'entité ROLE.

A chaque occurrence du type d'entité ATTRIBUTE et du type d'entité ROLE peuvent correspondre de 0 à N occurrence(s) du type d'entité COMPONENT.

Pour distinguer les différentes occurrences du type d'entité COMPONENT liées à la même occurrence du type d'entité GROUP, on ajoute au type d'entité COMPONENT un attribut NUMBER de type entier. Les valeurs de cet attribut sont différentes pour toutes les occurrences du type d'entité COMPONENT liées à la même occurrence du type d'entité GROUP.

Chaque identifiant est composé d'un certain nombre de composants. Chacun de ses composants est, soit un attribut, soit un rôle.
A une occurrence du type d'entité GROUP correspondent de 1 à N occurrence(s) du type d'entité COMPONENT (au moins une car un identifiant possède au moins un attribut ou un rôle).

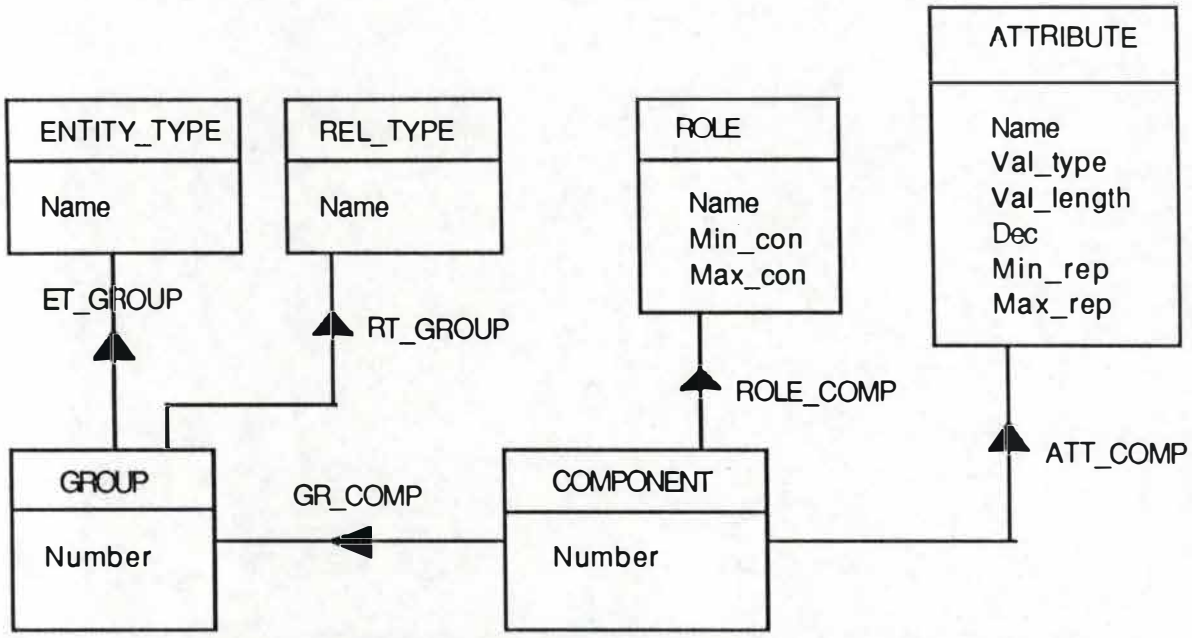


Figure III.4.6 : Représentation du concept d'identifiant dans le méta-schéma

Le concept d'identifiant dans le méta-schéma est beaucoup plus général que ce qui est réellement permis par le modèle EAC. En effet, on ne peut, dans la version actuelle, stocker qu'un et un seul identifiant par type d'entité ou type d'association. Celui-ci ne peut être composé que d'un seul attribut simple, élémentaire et obligatoire. Cet attribut doit appartenir au TE ou TA qu'il identifie. On a donc pour chaque occurrence de ENTITY_TYPE ou de REL_TYPE au plus une occurrence du type d'entité GROUP. Pour chaque occurrence du type d'entité GROUP on a une et une seule occurrence du type d'entité COMPONENT. Chaque occurrence du type d'entité COMPONENT est liée à une et une seule occurrence du type d'entité ATTRIBUTE. De plus, un identifiant ne peut comprendre de noms de rôle. Chaque occurrence du TE COMPONENT ne peut donc être liée à une occurrence du TE ROLE par le TA ROLE_COMP.

Dans le cas d'un identifiant d'un type d'entité d'un schéma conforme au modèle entité-association réduit, le principe est le même que ci-dessus.

III.4.2.6. Schéma PYRAMIDE du méta-schéma

Le schéma général du méta-schéma exprimé dans le modèle entité-association réduit est représenté à la page suivante:

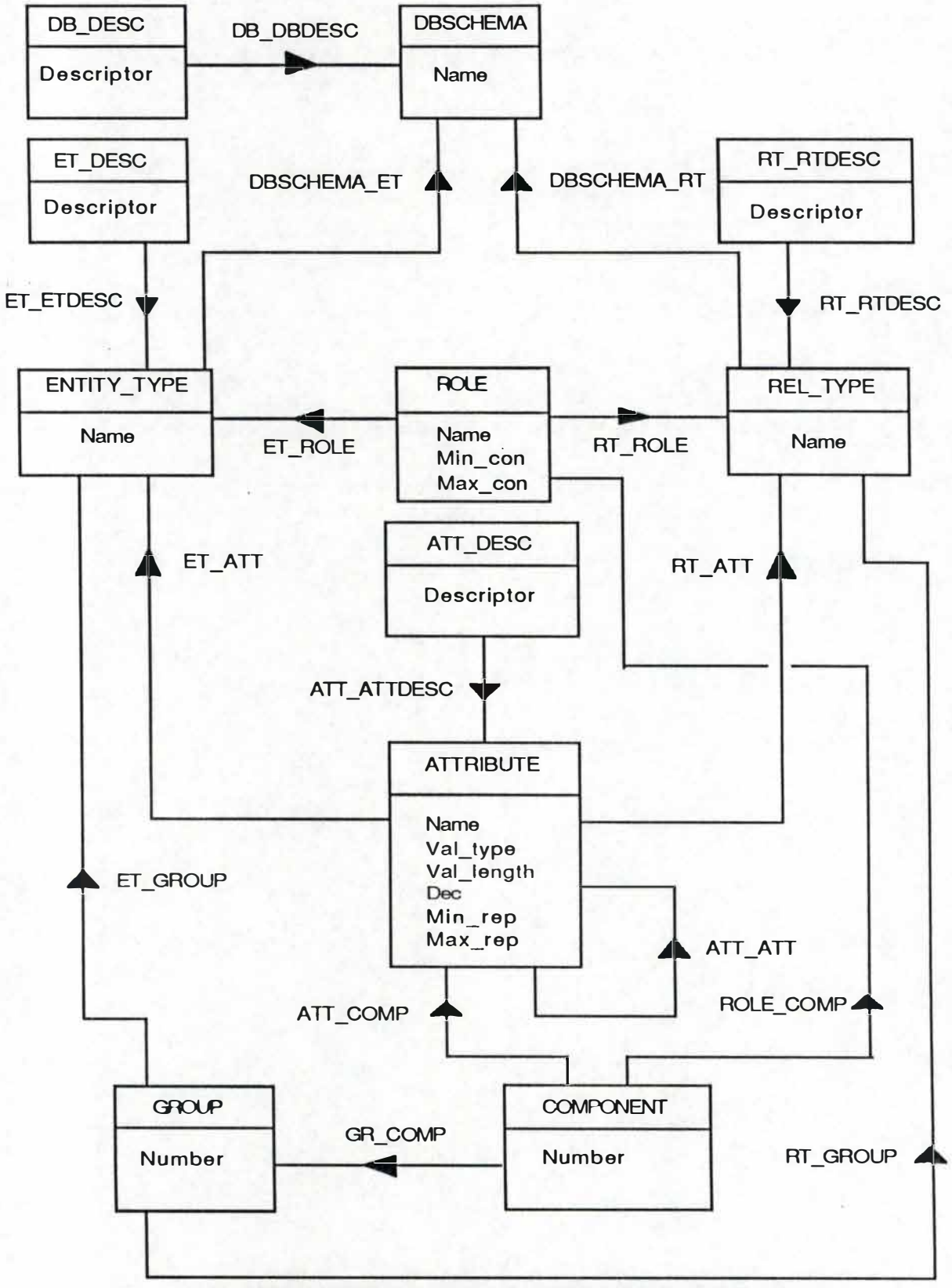


Figure III.4.7 : Schéma entité-association réduit du méta-schéma.

III.4.2.7. Exemples

Les exemples suivants, qui sont extraits de l'exemple donné dans l'annexe II, vont nous permettre de montrer comment ce schéma peut être stocké dans le méta-schéma.

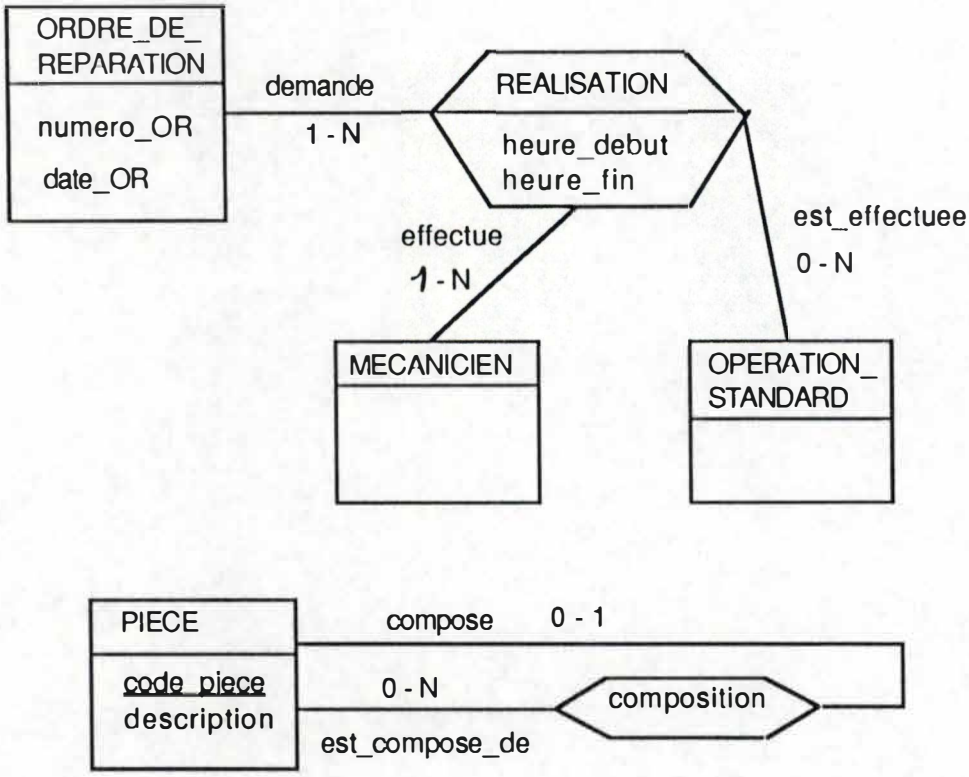


Figure III.4.8 : exemples de schémas EAC

Remarque: on représente dans les figures suivantes les différentes occurrences contenues dans les différents types d'entité du méta-schéma. Pour chaque occurrence, on donne les valeurs de ses différents attributs dans le type d'entité. Une ligne reliant deux occurrences indique l'existence d'une occurrence du type d'association qui relie les deux types d'entité dans le méta-schéma. Chaque tableau représente un type d'entité. Chaque colonne représente un attribut. Chaque ligne des tableaux représente une occurrence.

Par exemple dans la figure III.4.9:

Représentons dans le méta-schéma la définition du schéma de l'exemple:

Soit le type d'entité DBSCHEMA qui a un attribut NAME. On a pour ce type d'entité une occurrence dont la valeur de l'attribut NAME est égale à "$garage". Elle est liée à une occurrence du type d'entité DB_DESC dont la valeur de l'attribut DESCRIPTOR est "Schéma décrivant...".

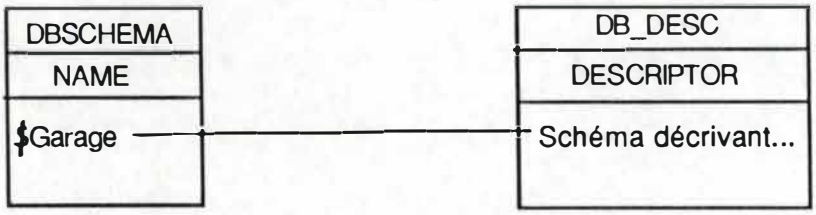


Figure III.4.9 : description du schéma EAC

Représentons les 4 TE de l'exemple:

Soit le type d'entité ENTITY_TYPE avec l'attribut NAME. On a pour ce type d'entité quatre occurrences dont les valeurs de l'attribut NAME sont respectivement "ORDRE_DE_REPARATION", "PIECE", "MECANICIEN" et "OPERATION_STANDARD". Elles sont liées à l'occurrence du type d'entité DBSCHEMA dont la valeur de l' attribut NAME est "$GARAGE" . Deux de ces occurrences (ordre_de_reparation et piece) sont liées à quatre occurrences du type d'entité ATTRIBUTE dont les valeurs des attributs NAME et TYPE sont respectivement ("numero_or","N"), ("date_or","D"), ("code_piece","N") et ("description","C"). Ce sont les différents attributs de ces deux types d'entité.

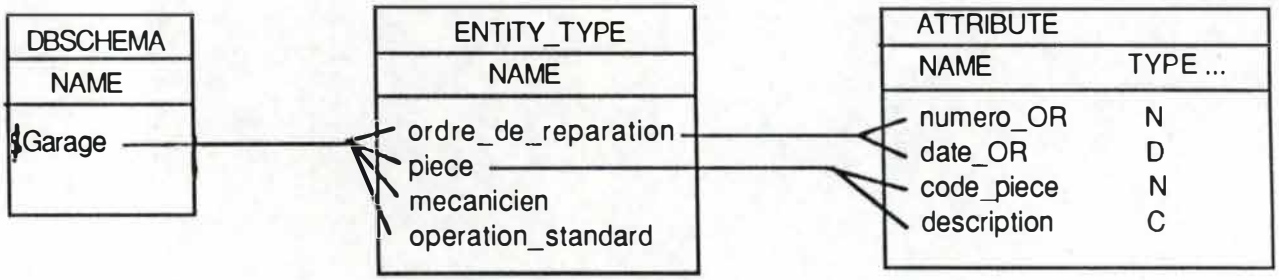


Figure III.4.10 : description des types d'entité EAC avec leurs attributs

Représentons les 2 TA de l'exemple:

Soit type d'entité REL_TYPE avec l'attribut NAME. On a pour ce type d'entité deux occurrences dont les valeurs de l'attribut NAME sont respectivement "COMPOSITION" et "REALISATION". Elles sont liées à l'occurrence du type d'entité DBSCHEMA qui a pour valeur de son attribut NAME "$GARAGE". L'occurrence "REALISATION" est liée à deux occurrences du type d'entité ATTRIBUTE dont les valeurs des attributs NAME et TYPE sont respectivement ("heure_debut","N") et ("heure_fin","N"). Ce sont les deux attributs du type d'association "REALISATION" dans le schéma EAC.

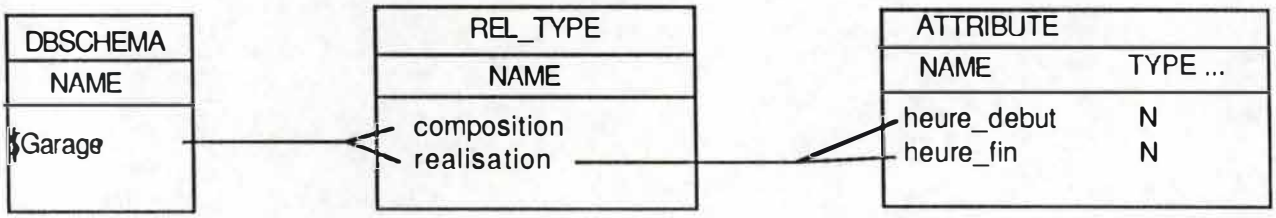


Figure III.4.11 : description des types d'association EAC avec leurs attributs

Représentons les participations des TE aux TA de l'exemple:

Soit le type d'entité ROLE avec les attributs NAME, MIN_CON et MAX_CON. On a pour ce type d'entité cinq occurrences dont les valeurs des attributs sont respectivement ("compose",0,1), ("est_compose_de",0,N), ("demande",1,N), ("effectue",2,N) et ("est_effectuee", 0,N). Les deux premières occurrences sont liées à l'occurrence "COMPOSITION" du type d'entité REL_TYPE et à l'occurrence "PIECE" du type d'entité ENTITY_TYPE. "COMPOSITION" est dans le schéma EAC une association binaire entre le type d'entité "PIECE" et lui-même (association récursive) qui y joue les rôles "COMPOSE" et "EST_COMPOSE_DE". Les trois dernières occurrences du type d'entité ROLE sont liées à l'occurrence "REALISATION" du type d'entité REL_TYPE (association ternaire) et sont chacune liée à une occurrence du type d'entité ENTITY_TYPE (type d'entité qui joue ce rôle dans le type d'association "REALISATION"), à savoir "DEMANDE" lié à "ORDRE_DE_REPARATION", "EFFECTUE" lié à "MECANICIEN" et "EST_EFFECTUEE" lié à "OPERATION_STANDARD".

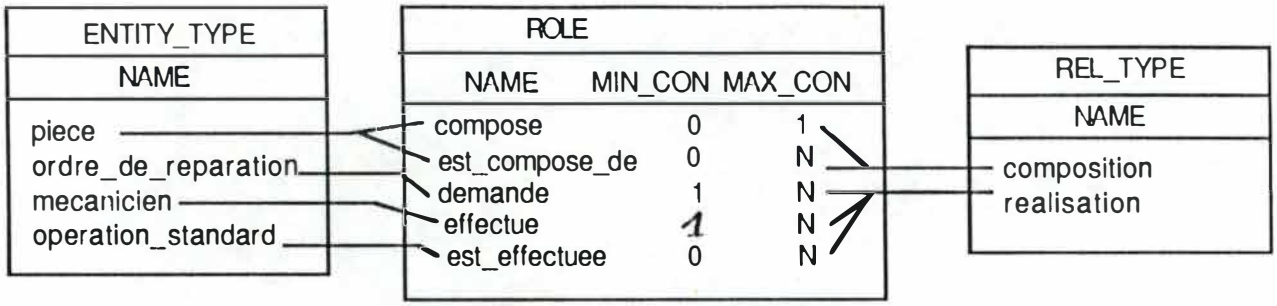


Figure III.4.12 : description des rôles EAC

Représentons l'identifiant du TE de nom pièce:

Soit les types d'entité GROUP et COMPONENT avec les attributs NUMBER pour chacun. On a pour le type d'entité GROUP une occurrence dont la valeur est "1". Elle est liée à l'occurrence "PIECE" du type d'entité ENTITY_TYPE. Cela représente l'existence d'un identifiant pour le type d'entité PIECE dans le schéma EAC. Cette occurrence du type d'entité GROUP est liée à une et une seule occurrence du type d'entité COMPONENT dont la valeur est "1". Cela signifie que l'identifiant est composé d'un seul élément. L'occurrence du type d'entité COMPONENT est liée à l'occurrence "CODE_PIECE" du type d'entité ATTRIBUTE. Cela signifie que l'identifiant du type d'entité PIECE est basé sur l'attribut CODE_PIECE. Cette occurrence du type d'entité ATTRIBUTE est liée à l'occurrence "PIECE" du type d'entité ENTITY_TYPE. Cela signifie que CODE_PIECE est un attribut du type d'entité PIECE dans le schéma EAC.

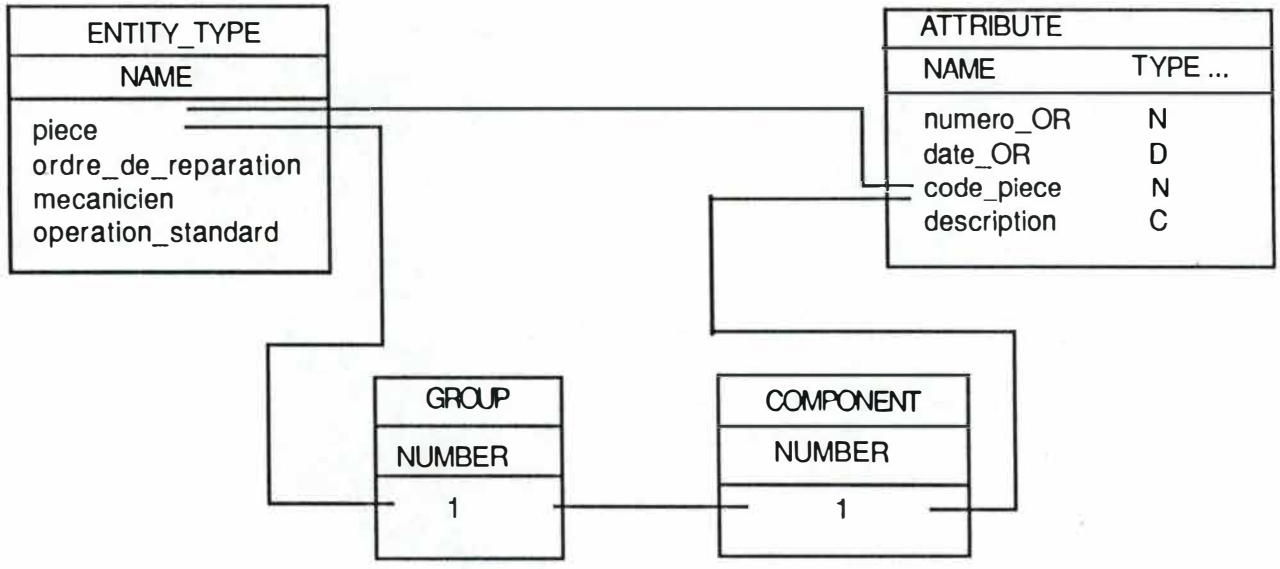


Figure III.4.13 : description des identifiants EAC

III.4.3. Gestion du méta-schéma par PYRAMIDE

Une des particularités de Pyramide est que chaque base de données inclut, dans son schéma, le méta-schéma qui contient des données décrivant aussi bien le schéma de l'utilisateur (à partir duquel la base de données est construite) que le méta-schéma. Chaque base de données contient donc sa propre description. Par exemple, dans le cas de schémas conformes au modèle entité-association réduit, les méta-données spécifieront qu'il y a dans le schéma de l'utilisateur des types d'entité "CLIENT" et "VOITURE", que le type d'association "PROPRIETAIRE" est défini entre ces deux types d'entité, que "NOM" est un attribut de "CLIENT",...

Un schéma de base de données inclut donc les objets du méta-schéma (par exemple, "ENTITY_TYPE", "REL_TYPE",...) plus ceux du schéma utilisateur (par exemple, "CLIENT", "VOITURE",...) qui sont décrits par les méta-données. Une base de données contient donc des méta-données et des données utilisateurs.

Au niveau de Pyramide, il n'y a pas de différence technique entre les méta-données et les données utilisateurs, hormis qu'elles sont de types différents. Les méta-données peuvent donc être accédées par un programme d'application en utilisant l'interface de programmation standard de Pyramide. Il est donc indispensable que les objets de ce méta-schéma soient conformes au modèle entité-association réduit supporté par Pyramide. Il faut pour cela que les données et les méta-données soient conformes au même modèle. C'est pourquoi le méta-schéma est défini conformément au modèle entité-association réduit. Il est donc constitué de types d'entité, de types d'association de type 1-N binaires et sans attribut, d'attributs de types d'entité,... Le méta-schéma étant à l'origine défini au niveau de Pyramide, il permet donc de stocker la description de schémas entité-association réduit.

Un intérêt du méta-schéma au niveau de Pyramide est de permettre la génération automatique d'une base de données opérationnelle en fonction du schéma décrit dans le méta-schéma. Cela est possible via l'utilitaire METACOMP du SGBD Pyramide. Ce méta-schéma permet également une mise-à-jour dynamique du schéma qui provoque une restructuration de la base de données. Actuellement, cette restructuration provoque la perte de toutes les données utilisateurs. L'ancienne base de données est cependant conservée dans un fichier ".odb", produit par PYRAMIDE.

III.4.4. Généralisation du méta-schéma au système LEA

Mais il est nécessaire de généraliser la notion de méta-schéma à la couche supérieure. Cette généralisation implique la nécessité de pouvoir stocker des schémas EAC dans le méta-schéma. Cela sera très simple puisque le méta-schéma est conçu de façon à pouvoir stocker des schémas conformes à des modèles plus complets que EAC.

III.4.4.1. Schéma EAC du méta-schéma

Pour assurer un plus grand parrallélisme entre les deux couches, nous avons choisi de représenter, tout comme Pyramide, les méta-données et les données de la même façon au niveau EAC (pas de différence technique). Tout comme pour la couche inférieure, nous avons donc dû transformer le méta-schéma pour qu'il soit conforme au modèle EAC. Ce choix se justifie aussi par le fait que cette transformation de l'un vers l'autre est évidente. Il a suffit de rajouter des noms de rôles et d'adapter éventuellement les connectivités qui sont gérées par la couche supérieure et non par le SGBD Pyramide. L'utilisateur de la couche supérieure et donc du langage LEA ne devra utiliser que le schéma EAC du méta-schéma.

L'utilisateur pourra de cette façon manipuler les méta-données aussi bien que les données avec les ordres standards du langage LEA, et exactement de la même manière.

III.4.4.2. Rôles du méta-schéma

La fonction essentielle du méta-schéma dans le système LEA est de permettre l' accès par le précompilateur à la description du schéma EAC de l'utilisateur. Cet accès permet

- d'assurer le "mapping" entre les deux couches, ç'est-à-dire, la traduction des ordres LEA en des appels aux primitives de Pyramide. Cette traduction nécessite aussi d'assurer la transformation d'un schéma exprimé dans le modèle EAC vers son correspondant exprimé dans le modèle entité-association réduit,

- de vérifier que les ordres du langage LEA sont corrects, ç'est-à-dire qu'ils portent sur des types d'entité, d'association, sur des attributs qui

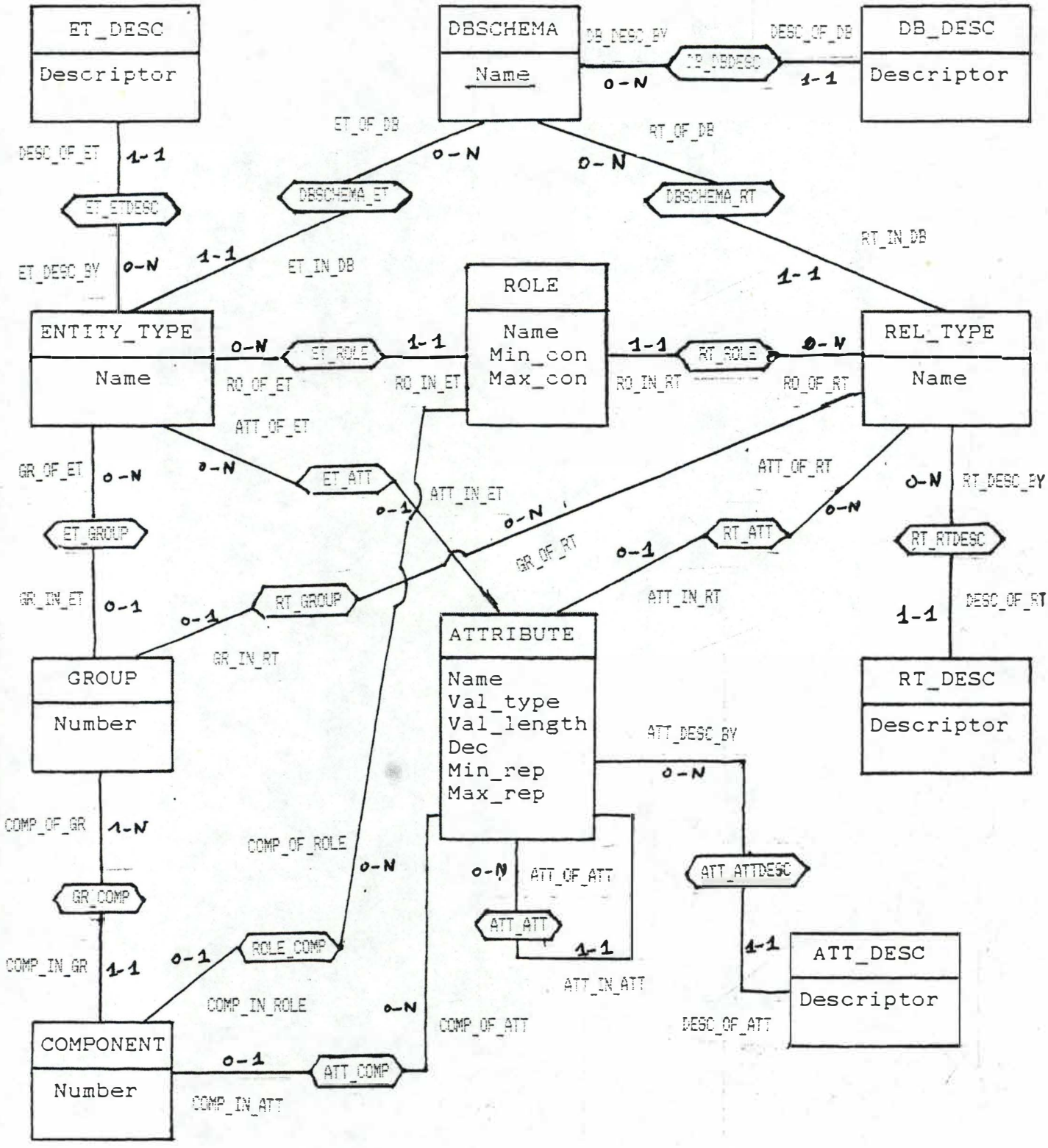


Figure III.4.14 : Schéma EAC du méta-schéma.

existent bien dans le schéma défini par l'utilisateur ou dans le méta-schéma,
- de vérifier les contraintes d'intégrité (connectivités minimales).

Un autre intérêt important du méta-schéma est de faciliter la modification dynamique d' un schéma entité-association.

III.4.4.3. Contenu du méta-schéma

Le contenu du méta-schéma est augmenté du fait de son utilisation dans le système LEA:

Le méta-schéma EAC contient

- au départ, ç'est-à-dire quand la base de données est vide de tout schéma utilisateur, des méta-données décrivant le méta-schéma d'une manière conforme au modèle entité-association réduit de Pyramide et d'une manière conforme au modèle EAC,

- quand la base de données contient un schéma utilisateur, les méta-données déjà citées plus des méta-données décrivant le schéma de l'utilisateur sous forme EAC et sous forme entité-association réduit.

III.5. Transformation de schémas

Introduction

L'objectif du chapitre est de définir les processus de transformation qui vont être mis en jeu par le précompilateur pour transformer un schéma exprimé dans le modèle EAC vers un schéma exprimé dans le modèle entité-association réduit.

Nous donnerons d'abord nos objectifs et les choix qui ont guidés ces processus. On verra par après comment chaque concept est transformé.

On donnera pour terminer un exemple de fonctionnement pratique.

III.5.1. Objectifs et choix

Comme il a déjà été signalé à de nombreuses reprises, il va falloir assurer le "mapping" entre la couche virtuelle entité-association et la couche physique de Pyramide. Ce "mapping" va consister à traduire les ordres LEA qui sont basés sur le modèle de données EAC en un ensemble de primitives Pyramide qui sont basées sur le modèle entité-association réduit. Il est donc indispensable avant de pouvoir traduire les ordres du langage de savoir comment va s'effectuer le passage d'un modèle vers l'autre. C'est donc l'objectif de ce chapitre que de définir de telles règles de transformation. Elles ne concernent que la transformation d'un schéma EAC vers un schéma entité-association réduit et en aucune manière la transformation des ordres du langage LEA.

Ces règles de transformation doivent nous permettre d'obtenir automatiquement et de manière univoque, à partir d'un schéma EAC, un schéma entité-association réduit qui lui correspond et qui n'a d'autres propriétés que d'être correct et le plus proche possible du schéma d'origine. Ces règles seront les plus simples possibles. Elles seront connues uniquement du précompilateur qui les utilisera pour effectuer la traduction des ordres LEA en des ordres Pyramide (caractère automatique de la transformation). En effet, pour effectuer le "mapping", le précompilateur ne peut pas se contenter du schéma entité-association de départ et du schéma entité-association réduit correspondant. Il doit savoir ce qui a été transformé et comment, mais aussi ce qui n'a pas été transformé. Il serait possible de stocker les règles de transformation dans

le méta-schéma, mais nous n'avons pas choisi cette solution du mapping physique. Nous voulons en effet que le méta-schéma soit indépendant de tout processus de transformation, qui sont plutôt propres au précompilateur.

Les règles de transformation n'étant pas connues du méta-schéma, et comme il est indispensable à PYRAMIDE de disposer dans le méta-schéma de la version entité-association réduite du schéma de départ pour pouvoir en utilisant l'utilitaire METACOMP produire une base de données opérationnelle, il faut que le schéma entité-association réduit soit généré d'une manière ou d'une autre. Il le sera par le précompilateur. Celui-ci, lorsqu'il traduira des ordres de création, modification, suppression de méta-données (ordres détectés par le nom des objets sur lesquels ils travaillent et qui sont différents pour les données et les méta-données), provoquera la génération du code Pyramide nécessaire à la fois pour effectuer ces opérations au niveau du schéma entité-association dans le méta-schéma mais également au niveau de son correspondant entité-association réduit. Il le fera en appliquant les règles de transformation.

Il faut noter qu'il y a deux sortes de transformation possibles:

- les transformations sans perte de sémantique et qui peuvent être réversibles ou non
- les transformations réduites avec perte de sémantique.

C'est cette dernière sorte de transformation que l'on retrouve ici. En effet, hormis la perte des noms de rôles lors de la transformation, qui ne peut pas être considérée comme une véritable perte sémantique, on perd surtout lors des transformations de la sémantique au niveau des connectivités et des identifiants.

Pour les identifiants, il n'y a cependant aucune perte dans la version actuelle. En effet, la notion et l'utilisation d'identifiants sont strictement identiques pour le modèle entité-association réduit et le modèle EAC, dans cette version du système LEA.

En ce qui concerne les connectivités, le modèle entité-association réduit ne permet que des types d'association binaires avec des connectivités (0,N) et (0,1). Toutes les valeurs de connectivités d'un schéma EAC sont donc converties en de telles valeurs. On verra

cependant que l'on peut conserver malgré tout ces informations dans le schéma entité-association réduit, ce que nous n'avons pas fait pour ce système.

Cette perte de sémantique nous posait donc un problème face auquel deux solutions se présentaient:
- ne pas gérer les connectivités minimales
- gérer les connectivités minimales au niveau de la couche supérieure (via le précompilateur).

C'est cette deuxième solution qui a été choisie mais en limitant toutefois la valeur de la connectivité minimale testée à 1 (contrainte d'existence). Cela est du à des raisons de performance et de facilité d'implémentation.

Les règles de transformation énoncées ne garantissent donc nullement la réversibilité. A partir d'un schéma final entité-association réduit, on peut retrouver une série de schémas entité-association originaux parmi lesquels on retrouvera le schéma de départ.

Remarque: on peut vérifier aisément que le passage du méta-schéma entité-association au méta-schéma entité-association réduit respecte ces règles de transformation.

III.5.2. Transformation du concept de schéma

Ce concept est identique pour les deux modèles. Il ne sera donc pas modifié.

III.5.3. Transformation du concept d'entité

Ce concept est identique pour les deux modèles. Il ne sera donc pas modifié par le processus de transformation.

III.5.4. Transformation du concept d'association

Le concept d'association diffère entre les deux modèles. En effet, le modèle entité-association réduit ne permet que des types d' association binaires et sans attribut. Il faut de plus que les connectivités liées à ces types d'association soient respectivement 0-N et O-1. Il faut donc transformer les types d'association d'un schéma EAC qui ne satisfont pas ces conditions. Pour ce faire, il faut distinguer deux cas:

- les types d'association binaires et sans attribut avec des connectivités égales à (0,1) ou (1,1) et (0,N) ou (1,N),

- les autres .

1) Les types d'association binaires sans attribut avec des connectivités égales à (0,1) ou (1,1) et (0,N) ou (1,N).

Soit deux types d'entité A et B relié par une association R.
A joue le rôle AR avec des connectivités (0,N)
B joue le rôle BR avec des connectivités (1,1)
(AR ≠ BR)

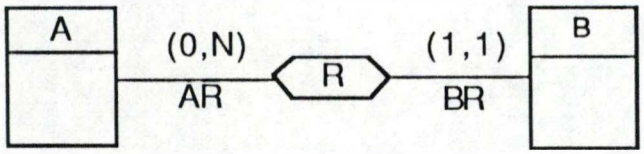


Figure III.5.1: type d'association conforme à EAC

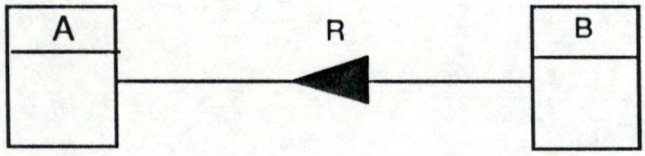


Figure III.5.2: type d'association conforme au modèle entité-association réduit

On a dans ce cas un type d'association conforme au modèle entité-association réduit. Le rôle AR est remplacé par le rôle "ORIGIN" et le rôle BR par le rôle "TARGET". Cer rôles indiquent la position du type d'entité par rapport au type d'association et par conséquent détermine les connectivités qui sont (0,N) pour le rôle "ORIGIN" et (0,1) pour le rôle "TARGET". Pour que la transformation soit complète, il faut ajouter des contraintes d'intégrité d'existence dans le schéma conforme au modèle entité-association réduit:

- si la connectivité minimale de A vaut 1, il faut que toute entité de A soit liée à au moins une entité de B
- si la connectivité minimale de B vaut 1, il faut que toute entité de B soit liée à une et une seule entité de A.

Toutefois, Pyramide ne les gérant pas, elles sont données à titre informatif. On pourrait conserver ces contraintes dans le méta-schéma pour assurer la complétude de la transformation. Comme l'utilitaire du

SGBD PYRAMIDE, qui à partir de la description d'un schéma entité-association réduit produit une base de données opérationnelle, ne considère pas la valeur des connectivités minimale et maximale (attributs MIN_CON et MAX_CON du type d'entité ROLE du méta-schéma) car il les considère toujours égales à 0 et à N pour un rôle ORIGIN et à 0 et à 1 pour un rôle TARGET, on peut placer les contraintes d'intégrité d'existence dans ces attributs.

Par exemple: on a dans un schéma entité-association réduit un type d'entité A (ORIGIN) lié par un type d'association de type 1-N appelé REL au type d'entité B (TARGET)

avec des contraintes d'intégrité d'existence:
- toute entité de B est liée à au moins une entité de A

On aurait dans le méta-schéma:
- une occurrence de ROLE ayant le groupe de valeurs (ORIGIN,0,N) liée à l'occurrence A de ENTITY_TYPE et à l'occurrence REL de REL_TYPE
- une occurrence de ROLE ayant le groupe de valeurs (TARGET,1,1) liée à l'occurrence B de ENTITY_TYPE et à l'occurrence REL de REL_TYPE

Remarque: on a perdu les noms de rôles AR et BR. Pour les retrouver il faudrait avoir une règle systématique pour leur construction. OR, le modèle EAC ne réduit pas le choix des noms de rôles à une telle règle. La transformation n'est donc pas réversible.

2) Les autres types d'association (et/ou ternaires, et/ou avec attributs, et/ou de connectivités différentes de (0,1) ou (1,1) et (0,N) ou (1,N)).

Soit R un type d'association de degré N,
Soit $E_1 \ldots E_n$ les types d'entités de REL
Soit $R_1 \ldots R_n$ les rôles joués par les types d'entités dans R
S'il existe k et I tel que $E_k = E_I$ alors $R_k \neq R_I$
Soit E_I un des types d'entité participant au type d'association R
R_I le rôle joué par E_I dans le type d'association R
la connectivité de R_I est i-j avec i=0 ou1 et j= 1 ou N

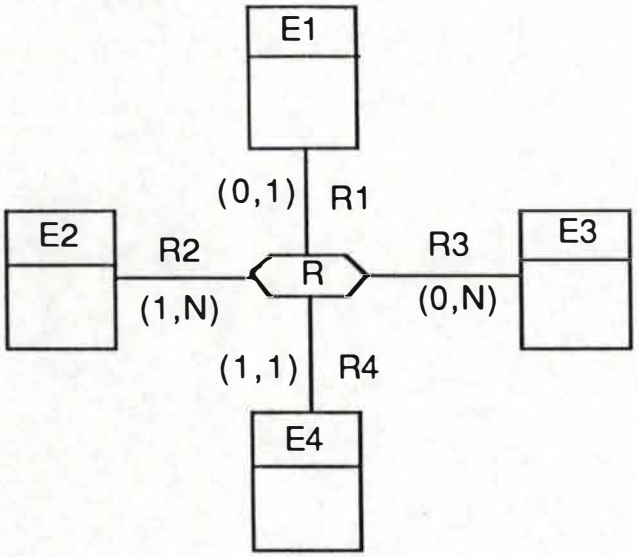


Figure III.5.3: type d'association conforme à EAC

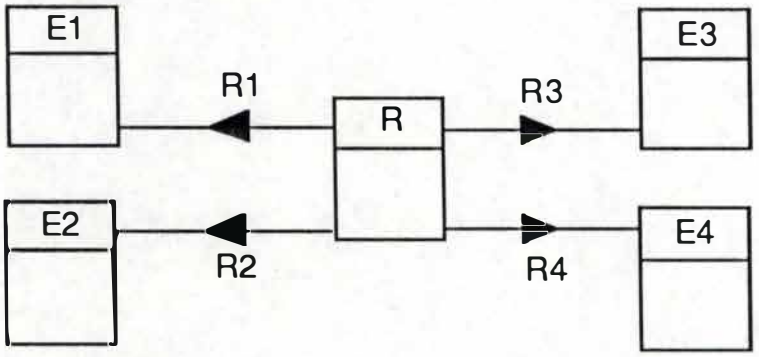


Figure III.5.4: type d'association conforme au modèle entité-association réduit

La transformation provoque la création d'un nouveau type d'entité. Le nom de ce type d'entité sera R . On aura la création pour chaque A_I participant au type d'association R d'un type d'association de type 1-N (type d'association du modèle entité-association réduit) de nom "R_I" (nom du rôle joué par par E_I dans le TA R du schéma EAC) entre le type d'entité E_I (joue le rôle "ORIGIN") et le type d'entité R (joue le rôle "TARGET"). Cela justifie le fait que les noms des types d'entité, des types d'association et des rôles doivent être uniques au schéma entité-association. En effet, de cette façon, le précompilateur ne crée pas de TE ou de TA avec un nom existant déjà dans le schéma réduit.

On peut également rajouter à titre informatif des contraintes d'intégrité d'existence pour que la transformation soit complète:

- si i>0, il faut que toute entité de A_I soit liée à au moins une entité de R,
- si j<N, il faut que toute entité de A_I soit liée à au plus j entité(s) de R.

III.5.5. Transformation du concept d'attribut

Le concept est identique pour les deux modèles, mais dans le modèle entité-association réduit un attribut ne peut être attribut que d'un type d'entité. Il n'y a pas de problème pour la transformation étant donné que tout type d'association avec des attributs devient un type d'entité. Les attributs de type d'association dans le modèle EAC deviennent des attributs de type d'entité dans le modèle entité-association réduit.

III.5.6. Transformation du concept d'identifiant

Le modèle entité-association réduit n'autorise que les identifiants sur les types d'entité, composés d'un et d'un seul attribut du TE qu'il identifie. Cet attribut doit être obligatoire, simple, et élémentaire. On ne peut définir qu'un et un seul identifiant par type d'entité. On ne pourra donc conserver lors de la transformation qu'une partie des identifiants définis sur le schéma EAC. Les autres qui ne seront pas conservés devront être gérés par l'inteface EAC sans pour autant que l'utilisateur ne doive le programmer. C'est donc le précompilateur qui générera ce qui est nécessaire pour satisfaire cette notion.

Actuellement, le modèle EAC n'accepte que les identifiants supportés par le modèle entité-association réduit. Le processus de transformation est donc simplifié pour la version actuelle du système LEA.

III.5.7. Fonctionnement pratique et exemples

D'un point de vue pratique, la transformation d'un schéma EAC en un schéma entité-association réduit s'effectue directement au sein du méta-schéma et est automatique. Quand un utilisateur décrit son schéma EAC (description stockée dans le méta-schéma), un schéma entité-association réduit correspondant est automatiquement produit (description également stockée dans le méta-schéma). Il faut noter que lors de cette transformation, les informations concernant les descriptions du schéma, des types d'entité, des types d'association et des attributs,

descriptions contenues dans les types d'entité DB_DESC, ET_DESC, RT_DESC et ATT_DESC ne sont pas reprises dans le schéma réduit.

Exemple:

Cet exemple complète celui du point III.4.2.7. , il décrit le stockage dans le méta-schéma du schéma EAC et de sa transformation en un schéma entité-association réduit.

L'utilisateur stocke dans le méta-schéma un schéma EAC de nom "GARAGE" avec une description "Schéma décrivant...". On trouvera dans le méta-schéma deux occurrences du type d'entité DBSCHEMA: "$Garage" et "Garage". La première occurrence est liée à une occurrence de DB_DESC: "Schéma décrivant...".

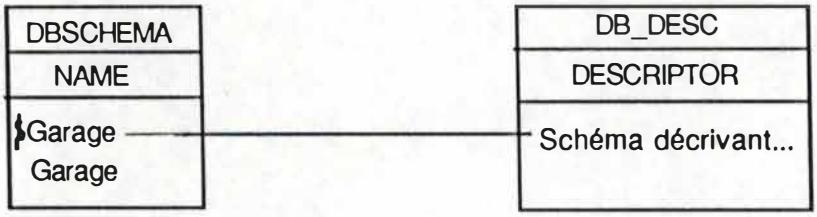


Figure III.5.5: description du schéma

L'utilisateur stocke trois types d'entité: ordre_de_reparation, mecanicien et operation_standard. On aura dans le type d'entité ENTITY_TYPE six occurrences dont les valeurs de l'attribut NAME seront "mecanicien", "ordre_de_reparation", "operation_standard", "mecanicien", "ordre_de_reparation" et "operation_standard". Les trois premières seront liées à l'occurrence "$Garage" de DBSCHEMA et les trois autres à l'occurrence "Garage" de DBSCHEMA.

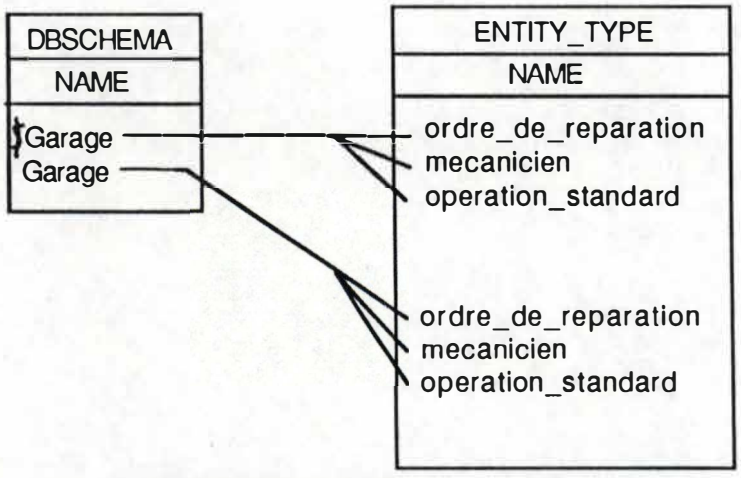


Figure III.5.6: description des types d'entité

L'utilisateur stocke une association "realisation" avec les attributs "heure_debut" et "heure_fin" entre les types d'entité "mecanicien", "ordre_de_reparation" et "operation_standard":
- "ordre_de_reparation" joue le rôle "demande" avec une connectivité 1-N
- "mecanicien" joue le rôle "effectue" avec une connectivité 0-N
- "operation_standard" joue le rôle "est_effectuee" avec une connectivité 0-N

Si on transforme cette association, on aura un nouveau type d'entité dans le schéma entité-association réduit: "realisation" et trois nouveaux types d'association "est_effectuee", "demande" et "effectue"

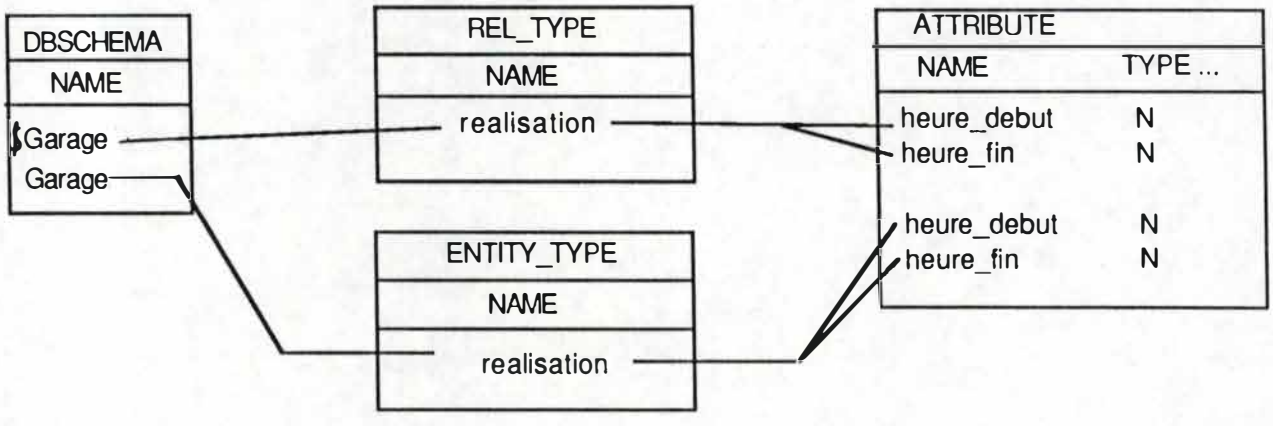


Figure III.5.7: description du type d'association avec ses attributs

Le type d'association "realisation" est transformé en un type d'entité "realisation" dans le schéma entité-association réduit.

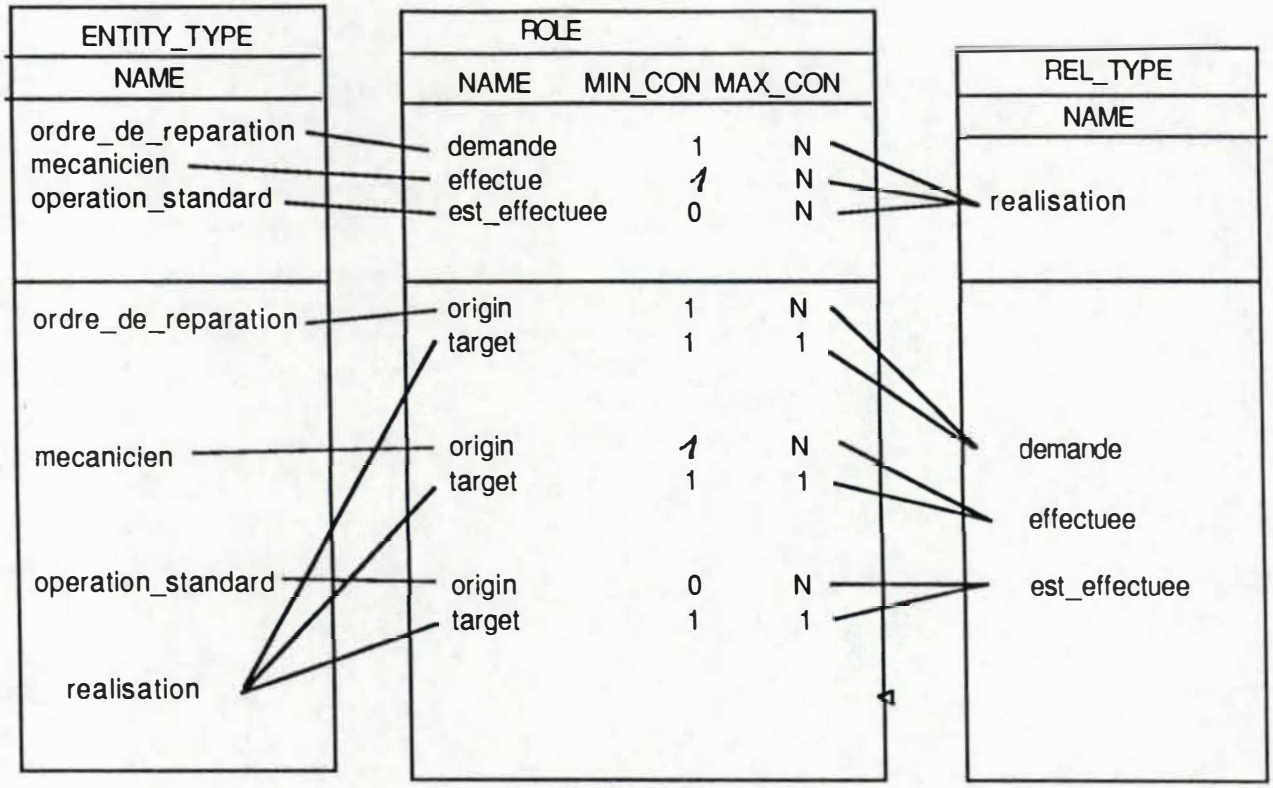


Figure III.5.8: description des rôles

Remarque: les occurrences de la première partie des types d'entités ENTITY_TYPE et REL_TYPE sont rattachées à l'occurrence du type d'entité DBSCHEMA dont la valeur de l'attribut NAME est égale à "$Garage." (schéma EAC). Les autres sont rattachées à l'occurrence du type d'entité DBSCHEMA dont la valeur de l'attribut NAME est égale à "Garage" (schéma entité-association réduit).

III.6. Définition de l'interface de programmation LEA

Introduction

L' objectif de cette partie est de définir une interface de programmation permettant de manipuler les concepts du modèle EAC. On va d'abord donner les objectifs propres à la définition de cette interface. On définira ensuite quelques notations utilisées pour définir l'interface et queleques conventions concernant son utilisation. On définira ensuite les concepts de cette interface de programmation, c'est-à-dire les types et variables, les codes de retour et enfin le langage de manipulation des concepts du modèle EAC.

III.6.1. Les choix de base

Nous allons d'abord donner les objectifs que nous avons assignés au langage LEA, qui est composé d'une série d'ordres permettant la manipulation des concepts du modèle EAC.

Le premier objectif tient évidemment au choix de notre architecture en deux couches. Comme LEA se base sur PYRAMIDE, et que tout ce qui n'est pas géré par PYRAMIDE devra l'être par la couche supérieure, nous devrons évidemment nous limiter, afin de ne pas surcharger la couche supérieure et que les transformations à effectuer par le précompilateur pour passer de LEA vers PYRAMIDE ne soient pas trop complexes.

Le langage doit en outre être non ambigu. En d'autres termes, chaque ordre du langage doit être défini de façon à ce que son effet soit défini de manière précise et soit univoque. Il ne doit pas être possible de trouver dans le langage un ordre dont l'expression mène à une confusion concernant sa sémantique.

Le langage doit être complet. Ce qui veut dire que l'ensemble des ordres doit permettre:

- d'assurer le stockage dans le dictionnaire de données de toute description de schéma conforme au modèle EAC, ainsi que sa modification;

- de créer, modifier, effacer dans une base de données toute occurrence de types définis dans le dictionnaire;

- d'accéder à toute structure préalablement stockée dans la base de données, et donc également dans le dictionnaire de données.

LEA doit assurer la cohérence de la base de données par rapport à sa définition. Notons que le méta-schéma se définissant lui-même, il faudra aussi assurer la cohérence des modifications sur le méta-schéma par rapport à sa définition. On aura donc que, si avant l'exécution d'un ordre LEA, la base est cohérente, elle le sera également après, à moins q'une erreur grave ne soit intervenue pendant son exécution, mais dans ce cas l'exécution de l'ordre ne s'est pas terminée. LEA ne gère pas le "crash recovery". Il nous semble en effet que cette gestion est d'un niveau trop physique que pour pouvoir être prise en charge par le couche supérieure.

Un autre objectif pour LEA est la simplicité d'utilisation. Comme PYRAMIDE doit être utilisé dans un environnement PASCAL, ce qui facilite la programmation des applications, il nous a semblé adapté de reprendre cette caractéristique. Les ordres du langage devront donc être insérés dans des programmes PASCAL. L'utilisateur dispose ainsi, en même temps que d'un langage de manipulation de bases de données, de toutes les facilités qu'offre PASCAL pour la programmation.

L'implémentation d'un précompilateur nous a permis de dépasser la syntaxe PASCAL. Cela est surtout vrai pour la désignation des éléments de la base de données sur lesquels on veut agir. Pour les opérations d'accès et de manipulation, nous sommes malgré tout restés assez proches des structures algorithmiques de PASCAL (assignation, boucle d'accès,...). Ce qui facilite de nouveau l'utilisation du langage.

Pour assurer la communication entre PASCAL et LEA, nous avons eu recours au même mécanisme que PYRAMIDE, c'est-à-dire l'utilisation de variables. Celles-ci, ainsi que les types de données sont évidemment adaptés à LEA, mais peuvent être facilement transformés vers PYRAMIDE par le précompilateur.

Le langage est également simplifié par l'absence de courants ou d'arguments implicites. Si l'utilisateur désire bénéficier de courants, il devra les déclarer lui-même, au niveau du programme d'application, et en assurer la gestion.

Dans une optique de simplification du langage, nous avons également spécifié le langage de façon à ce que les manipulations des concepts du schéma de l'application et du méta-schéma soient strictement identiques. Celles-ci se feront donc au moyen des mêmes ordres, et de la même façon, la seule différence intervenant au niveau des types manipulés. De cette façon, l'utilisation d'un ordre est définie de manière standard, quelque soit l'objet manipulé. Rappelons que la manipulation du méta-schéma consiste à définir le structure de la base de données. LEA est donc aussi un langage de définition de données, tout en étant conçu comme un langage de manipulation de données.

Etant donné que PYRAMIDE est conçu pour être assez performant, notamment en ce qui concerne les accès à la mémoire secondaire, il serait dommage que le système LEA gâche cet atout. Nous porterons donc à l'implémentation notre attention sur ce problème des performances. De cette façon, le système LEA pourrait être employé pour des applications d'ingénierie, pour lesquelles le modèle entité-association semblerait bien adapté. On se reportera au mémoire de D Rossi pour une argumentation plus précise de ce point.

Enfin, un dernier objectif est que le système LEA soit utilisable dans les limitations des ordinateurs personnels actuellement disponibles.

III.6.2. Notations et conventions

Pour définir le langage, on utilisera les notations suivantes.

<string> signification:

si string représente une structure quelconque connue du langage, <string> représente la désignation de cette structure et sera remplacé au niveau du langage par une instanciation quelconque de cette structure conforme au langage.

string signification:

si string représente une suite de caractères, cette suite sera réécrite au niveau du langage. string représente alors un mot réservé du langage.

<string>=liste signification:

la désignation <string> peut être remplacée par la liste. Cette liste peut prendre une des formes suivantes:

- liste formée d'opérateurs ET et OU:
$a_{11}...a_{1n}|a_{21}...a_{2m}|...|a_{p1}...a_{pq}$, avec n,m,p,q≥1, et pour tout i,j, a_{ij} = <$string_{ij}$> ou $string_{ij}$;
on aura alors au niveau du langage qu'une instanciation de <string> correspondra à l' instanciation ou à la recopie de (a_{11}, et de..., et de a_{1n}), ou de (a_{21}, et de ..., et de a_{2m}), ou de..., ou de (a_{p1}, et de ..., et de a_{pq});

- on a les cas particuliers d'une liste formée par l'opérateur ET si p = 1 et q = n; et par l'opérateur OU si n = m =...= q = 1

Remarque: l'opérateur ET utilisé ici comporte une notion d'ordre, il faut l'assimiler à un opérateur du type "suivi de".

[$string_1$<$string_2$>...] signification:

au niveau du langage, l'instanciation de <$string_2$> et la réécriture de $string_1$ ou de toute combinaison de $string_j$ et de <$string_i$> est possible, mais non obligatoire. Si les instanciations et réécritures sont faites, elles le sont pout tous les éléments entre [et]

Nous donnons dans la suite les diverses conventions d'utilisation de LEA.

Tout ordre du langage LEA (Langage Entité-Association) devra être précédé par le caractère suivant: $

Tout élément d'un ordre de LEA est séparé par au moins un espace des autres éléments.

Les différents mots d'un nom composé doivent être séparés par "_".

Les conventions PASCAL (; en fin d'instruction,...) doivent être respectées au sein du langage.

La portée d'un opérateur ou d'un connecteur quelconque du langage sera assurée par des parenthèses, lorsque cela est nécessaire pour empêcher une ambiguïté.

L'utilisateur ne peut redéfinir dans un ordre LEA un quelconque mot réservé de LEA ou de PASCAL.

III.6.3. Types et variables

Nous introduisons dans cette partie les concept de types de données et de variables utilisés par LEA pour assurer le dialogue entre les ordres du langage et PASCAL.

Afin de pouvoir manipuler les concepts du schéma de la base de données (méta-schéma et schéma de l'application) dans un programme, l'utilisateur dispose d'un type de données pour chaque TE et TA apparaissant dans ce schéma. Ces types de données sont prédéfinis, et ne peuvent pas être utilisés comme nom de variable. Ils peuvent être considérés comme des mots réservés du langage. L'utilisateur peut ainsi, pour chacun des TE et TA, qu'ils appartiennent au schéma de l'application, ou au méta-schéma, déclarer des variables pouvant contenir des occurrences pour ces TE et TA.

Les types de données associés aux TE sont directement dérivés des types de données PYRAMIDE se trouvant dans le fichier suffixé par .TYP. Les types associés aux TA sont différents des types de PYRAMIDE, ils doivent donc être être créés par le précompilateur dans un fichier suffixé par .VAR. Ce fichier contient également la déclaration des variables intermédiaires nécessaires au précompilateur pour assurer les processus de transformation.

On appellera variable d'entité (d'association) toute variable déclarée d'un des types de données correspondant à un TE (TA) du schéma. Une variable ne peut être déclarée que d'un seul type à la fois. Par exemple, la variable nommée "varent" ne peut être déclarée à la fois de type voiture et de type client. L'utilisation de ces variables permet au programme

d'application d'accéder ou de transmettre à LEA des renseignements (référence ou valeurs d'attributs, ou entités liées à une association) sur les objets manipulés.

III.6.3.1. Type de données associé à un TE

Pour un TE, le type de données automatiquement déclaré dans le programme PASCAL précompilé est directement dérivé du suivant. L'utilisateur peut considérer pour manipuler ses variables que le type qui figure dans les déclarations est conforme à cette syntaxe:

```
type   <nom du type d'entité> = record
                                  <nom attribut 1>: <type 1>;
                                  <nom attribut 2>: <type 2>;
                                  ...                  ...
                                  <nom attribut n>: <type n>
                                end;
```

où pour tout i <nom attribut i> désigne le nom d'un attribut du TE dont le nom est désigné par <nom du type d'entité>, et <type i> désigne la traduction PASCAL du type associé à l'attribut.

Les processus de transformation des types de données par le précompilateur seront abordés dans la quatrième partie

III.6.3.2 Type de données associé à un TA

Pour un TA, le type de données déclaré est directement dérivé de ce type. L'utilisateur peut considérer que le type déclaré est le suivant:

```
type   <nom du type d'association> = record
                                       <nom attribut 1>: <type 1>;
                                       <nom attribut 2>: <type 2>;
                                     ...                  ...
                                       <nom attribut n>: <type n>;
                                       R<nom de rôle 1>: <nom de TE 1>;
                                       R<nom de rôle 2>: <nom de TE 2>;
                                     ...                  ...
                                       R<nom de rôle n>: <nom de TE n>
                                   end;
```

où pour tout i <nom attribut i> désigne un attribut du TA dont le nom est désigné par <nom du type d'association>, et <type i> désigne la traduction PASCAL du type associé à l'attribut;<nom de rôle i> désigne un nom de rôle joué dans le cadre de l'association, <nom de TE i> désigne le nom du TE jouant ce rôle.

Il est nécessaire de préfixer les noms de rôles car lors du processus de transformation, chaque nom de rôle est employé comme nom pour le TA dans le schéma transformé vers PYRAMIDE. Or, ces noms de TA sont déclarés comme des constantes de désignation dans le fichier suffixé par .TYP, et inclus dans le programme précompilé.

III.6.3.3 Traduction des types d'attribut

Si un attribut est répétitif, son type PASCAL sera un tableau dont la taille sera égale à la répétitivité maximale de l'attribut.

Si un attribut est décomposable, son type PASCAL sera un record portant le nom de l'attribut, et dont les éléments seront les attributs qui le composent avec leur type correspondant.

Les types d'attribut simples sont traduits de la manière suivante:

type E/A	type PASCAL
n(i,j) avec j=0 et i ≤ 4	integer
n'i,j) avec j<>0 ou i>4	real
c(n) avec n=1	char
c(n) avec n>1	string[n]
d(date)	string[6]
b(booléen)	boolean

La valeur facultative est traduite de la manière suivante:

pour un type numérique (entier ou réel): constante PASCAL de nom NO_VALUE.n
pour un type caractère(s) (char ou string[n]): constante PASCAL de nom NO_VALUE.s

III.6.3.4 Exemples

```
type    client = record
                    numéro_id_client:integer;
                    nom_cli:string[20];
                    descriptif_client = record
                        prénoms_client: array [1:n] of string[40];
                        adresse_client = record
                                            numéro:integer;
                                            rue:string[20];
                                            code_postal:integer;
                                            localité:string[40]
                                        end;
                                    end;
                end;

        location = record
                        date_location:string[6];
                        Rpossède:client;
                        Rest_possédée_par:voiture;
                    end;
```

III.6.4. Codes de retour erstatus

Après qu'un ordre du langage ait été exécuté par un programme d'application, la variable entière globale de nom **erstatus** indique comment l'opération s'est effectuée. Nous détaillerons par après le fonctionnement de chaque ordre, mais, étant donné que les codes de retour ont le plus souvent la même signification, nous donnons ici la liste des valeurs possibles pour la variable erstatus, et leur signification correspondante.

0: l'ordre a été correctement exécuté
1: l'objet demandé n'a pas été trouvé
2: violation d'une contrainte d'identification lors d'une opération de mise-à-jour; l'opération n'est pas effectuée
14: il n'y a pas de schéma actif; l'opération n'est pas effectuée

19: l'exécution de cette opération entraînerait une incohérence de la base de données par rapport à sa définition; l'opération n'est pas effectuée
20: il existe déjà un schéma actif; l'opération d'ouverture d'un nouveau schéma n'a pas fonctionné
80: l'espace de la mémoire secondaire est dépassé
90: la base de données est incohérente; PYRAMIDE a détecté une référence incorrecte
99: erreur système ou d'entrée/sortie

Comme on peut le voir, plus la valeur contenue dans la variable est grande, plus l'incident intervenu est grave. Les codes à un chiffre définissent des conditions de déroulement normales. Les trois premiers codes à deux chiffres définissent des situations où l'utilisateur tente d'effectuer une manipulation non permise par le système. Les trois derniers codes à deux chiffres définissent des situations où une grave erreur externe est intervenue, que le système est incapable de prendre en charge.

III.6.5. Définition du langage

Nous allons dans cette partie décrire chacun des ordres du langage, c'est-à-dire leur syntaxe, la description de leurs effets, dans une situation normale ou pas, et les codes de retour associés.

Nous donnerons pour chaque ordre un exemple d'utilisation très simple et volontairement incomplet. Notre objectif est ici de donner une aide pour la compréhension de la syntaxe de l'ordre. Le lecteur pourra trouver des exemples représentatifs dans la partie III.6.6.

Certains éléments de syntaxe sont communs à plusieurs ordres du langage, nous les décrirons donc ici.

<var_name> représente un nom de variable conforme à PASCAL

<var_ent_name> représente un nom de variable d'entité conforme à PASCAL, et associé à un type d'entité du méta-schéma ou du schéma de l'application

<var_rel_name> représente un nom de variable d'association conforme à PASCAL, et associé à un type d'association du méta-schéma ou du schéma de l'application

<ent_type> représente un nom de type d'entité du méta-schéma ou du schéma de l'application, ent_type est donc une valeur de type string de 1 à 32 caractères

<rel_type> représente un nom de type d'association du méta-schéma ou du schéma de l'application, rel_type est donc une valeur de type string de 1 à 32 caractères

<db_name> représente un nom de base de données, db_name est donc une expression ou une valeur du type string PASCAL conteant de 1 à 32 caractères, pouvant contenir un chemin, sans extension

<sch_name> représente un nom de schéma, sch_name est donc une valeur de type string de 1 à 32 caractères, sans extension

<valeur> représente une valeur de type string, entier, réel, ou booléen

<const_name> représente un nom de constante conforme à PASCAL

<attr_name> représente un nom d'attribut d'un TE ou TA du méta-schéma ou du schéma de l'application, attr_name est donc une valeur de type string conteant de 1 à 32 caractères

<var_id_trans> représente le nom d'une variable conforme à PASCAL, de type entier

Les éléments entre syntaxe autres que ceux repris plus haut seront définis par la suite. Notons qu'ils ne le seront qu'une seule fois. S'ils apparaissent dans la syntaxe de plusieurs ordres, ils sont définis uniquement au niveau du premier ordre où ils apparaissent. Les différences d'utilisation de ces éléments apparaissant dans plusieurs ordres seront données dans les descriptions.

III.6.5.1. Déclaration des variables

A) Syntaxe

$ [VAR] <list_var> : ENTITY <ent_type> ;
$ [VAR] <list_var> : RELATION <rel_type> ;

<list_var> = <var_name> [,<list_var>]

B) Description

Toute variable d'entité ou d'association utilisée par le programme d'application doit être déclarée dans ce programme au moyen de l'ordre LEA de déclaration. Une variable ne peut être déclarée que d'un seul type au sein d'un programme. La déclaration est généralisable aux procédures et fonctions, il suffit pour cela de faire précéder la déclaration de la procédure ou de la fonction par $. Pour cette déclaration, tous les paramètres doivent être déclarés en une seule ligne, sinon, chaque ligne ne peut contenir la déclaration que d'un seul paramètre (voir exemples).

Quand la variable a été déclarée d'un type correspondant à un TE ou un TA du méta-schéma ou du schéma de l'utilisateur, elle peut recevoir du SGBD la référence et les valeurs d'attributs d'un élément du type à la fois. La référence est un concept de PYRAMIDE. Il permet d'accéder directement à cet élément , la référence sert de "surrogate key" à PYRAMIDE. Ce concept sert également au langage LEA, uniquement dans certains contextes, qui seront vus plus tard. Le langage ne permet pas à l'utilisateur de manipuler directement le concept de référence. Tout comme l'écrit D. Rossi, cela ne devrait de toute façon être manipulé que par le SGBD. Toutefois, l'utilisateur peut manipuler ces références dans son programme d'aplication en insérant des primitives de PYRAMIDE. Les concepts de PYRAMIDE sont toujours accessibles avec LEA, même si le langage ne permet pas leur manipulation. Notons que les primitives de PYRAMIDE, si elles apparaissent dans le programme d'application, ne doivent pas être préfixées par $. Ce genre de manipulation n'est toutefois pas conseillé, à moins d'avoir une bonne connaissance des deux couches et de leurs interactions.

Le langage, tel qu'il est conçu dans cette version, ne permet pas à l'utilisateur de définir des types PASCAL contenant des types de données associés à des TE ou TA. L'utilisateur peut néanmoins obtenir le même effet. Il doit pour cela d'abord se définir un type intermédiaire, qui

est la reproduction de la définition du type à insérer dans un type plus complexe. Il devra par après s'assurer que les modifications faites sur le type associé au TE ou au TA sont répercutées sur ce type intermédiaire. Il est évident que ces types intermédiaires ne peuvent être utilisés avec des ordres du langage LEA.

C) Exemples

```
program  const_array_cli;

type elt_cli = record
                    number: integer;
                    name: string[20]
               end;
       tab_cli = array [1..100] of elt_cli;

$ var cli: ENTITY client;
       num:integer;
$      propr: RELATION proprietaire;
$      voit1, voit2: ENTITY voiture;
        tabclient:tab_cli;
$ procedure exemple1 (ind:integer; $cli: entity client;...);
$ procedure exemple2 (ind2:integer;
                        $cli2: entity client;
                       ...);

begin
    ...
       (* on trouve ici un ordre du langage accédant à une entité client et
rangeant ses valeurs d'attribut dans la variable cli *)
       tabclient[1].number:=cli.number;
       tabclient[1].name:=cli.name;
       write (cli.name);
       write(tabclient[1].name);
    ...
end.
```

III.6.5.2. Spécification du schéma de travail

A) Syntaxe

$ USES DATABASE '<db_name>' [SCHEMA '<sch_name>'];

B) Description

Cet ordre est nécessaire vu l'implémentation du précompilateur. Nous avons en effet choisi un compilateur en une passe. Ce qui veut dire que le précompilateur ne parcourt qu'une seule fois le fichier à compiler pour produire le fichier compilé. Le précompilateur doit donc connaître dès le début du programme quelle base de données, et quel schéma seront ouverts par le programme. L'absence d'un nom de schéma signifie que le programme travaille uniquement sur le méta-schéma.

Cet ordre doit être la première instruction du programme d'application. Il doit figurer directement après l'instruction de déclaration du nom du programme de PASCAL. Il ne peut figurer qu'une seule fois dans un programme.

Notons que le nom de la base de données et le nom du schéma peuvent être identiques.

III.6.5.3. Ouverture d'un schéma

A) Syntaxe

$ OPEN DATABASE '<db_name>' [SCHEMA '<sch_name>'];

B) Description

Cet ordre est nécessaire afin que la couche supérieure puisse signaler à PYRAMIDE la base de données qui doit être ouverte, et afin de savoir quel schéma est modifié ou consulté par le programme d'application.

Si l'ordre ne contient pas de clause de désignation d'un schéma, le méta-schéma est de toute façon ouvert pour entrer la définition d'un nouveau schéma. Sinon, le schéma d'application dont l'utilisateur donne le nom est ouvert et les données vérifiant la définition de ce schéma peuvent être modifiées. Le méta-schéma est alors

uniquement accessible en lecture. Dans ces cas, le système a donc trouvé la base de données et le schéma et les ouvre, erstatus vaut alors 0.

Si le système ne trouve pas la base de données nommée, ou s'il trouve la base mais pas le schéma, erstatus vaut 1, et il n'y a pas de schéma ouvert. Si une erreur système ou d'E/S intervient, erstatus vaut 99, et il n'y a pas de schéma accessible.

Pour des raisons de facilité d'implémentation, la couche LEA ne permet l'ouverture que d'une seule base de données et d'un seul schéma dans un programme, si l'utilisateur tente cette manipulation, elle est refusée et erstatus vaut 20. PYRAMIDE permet d'ouvrir plusieurs bases de données à la fois. L'utilisateur peut bénéficier de cet avantage, mais il doit pour cela insérer lui-même dans le programme d'application les appels à PYRAMIDE pour gérer cette deuxième base. Ces manipulations doivent être totalement transparentes pour la couche supérieure, et donc pour le précompilateur. Le lecteur intéressé pourra se reporter au mémoire de D. Rossi pour les primitives de PYRAMIDE et leur utilisation.

C) Codes de retour

erstatus = 0; tout s'est bien passé, la base de données et le schéma sont ouverts

erstatus = 1; le schéma ou la base de données portant ce nom n'a pas été trouvé, il n'y a pas de schéma ouvert

erstatus = 20; il y a déjà une base de données et un schéma ouverts

erstatus = 99; erreur système ou d'entrée/sortie

D) Exemple

```
program ouverture_schema;
$ USES DATABASE 'garage';
(* ouverture en modification du méta-schéma de la BD 'garage'*)
...
begin
      $ OPEN DATABASE 'garage';
          if erstatus <> 0 then trt_err_open;
        ...
end.
```

III.6.5.4. Fermeture d'un schéma

A) Syntaxe

$ CLOSE;

B) Description

La base de données active est fermée (et donc aussi tous les schémas qu'elle contient), et toutes les modifications faites après le dernier OPEN sont effectives; il n'y a plus de schéma actif pour le programme, et erstatus vaut 0. S'il y a eu une erreur système ou d'entrée/sortie, erstatus vaut 99, et le système n'offre aucune garantie sur l'état de la base de données qui était ouverte.

C) Codes de retour

erstatus = 0; la BD est fermée, il n'y a plus de BD active

erstatus = 1; il n'y avait pas de BD ouverte

erstatus = 99; erreur système ou d'entrée/sortie

D) Exemple

```
program  fermeture_schema;

$ USES DATABASE 'garage' SCHEMA 'garage';
(* ouverture en modification du schéma 'garage' de la BD 'garage'
    et en accès du méta-schéma de la BD *)
...

begin

      $ OPEN DATABASE 'garage' SCHEMA 'garage';
          if erstatus <> 0 then trt_err_open;
        ...
      $ CLOSE;
          if erstatus <> 0 then trt_err_close;
end.
```

III. 6. 5.5. La sélection

Remarquons d'abord que cet opérateur n'est pas un véritable opérateur de sélection. Il ne peut être utilisé tel quel dans un programme d'application. Cet opérateur sert plutôt à désigner les éléments auxquels on veut accéder. Pour y accéder effectivement, il faudra utiliser un opérateur d'assignation ou de boucle d'accès (pour accéder successivement à tous les éléments désignés). L'accès est donc inspiré de LDA [HAIN86-a]. Ce mécanisme nous a semblé plus adapté dans une optique de langage emboîté, en l'occurrence dans PASCAL. En effet, de cette façon, la syntaxe LEA reste proche de celle de PASCAL, et la communication entre LEA et PASCAL s'effectue très simplement à l'aide de variables.

La clause de sélection est de type navigationnel. Elle permet de spécifier l'objet auquel on veut accéder en citant ses qualités, c'est-à-dire ses valeurs d'attributs, et/ou les entités auquel il est lié par des associations.

A) Syntaxe

```
<selection-entite> = <ent_type> [<var_ent_name>]
                                [WITH <attr_value_spec>]
                                [THAT<rel_cond_spec>]

<rel_cond_spec> = <rel_cond_spec> <oplog> <rel_cond_spec> |
                  <role_ent_name> [LINKED_TO <ent_cond_spec>]
                                  [THROUGH <rel_spec>]

<rel_spec> = <rel_type> WITH <attr_value_spec> |
             <rel_type> <var_rel_name>

<selection-association> = <rel_type> [<var_rel_name>]
                                     [WITH <attr_value_spec>]
                                     [BETWEEN <ent-cond-spec>]

<ent_cond_spec> = <ent_cond_spec> AND <ent_cond_spec> |
                  <selection_entite>

<attr_value_spec>= <attr_value_spec> <oplog> <attr_value_spec>
|<attr_name><opcomp><value>
```

<oplog> = AND|OR

<opcomp> = < | > | <= | >= | =| <> (l'opérateur doit être compatible avec la valeur du type d'attribut désigné par <attr-name>)

<value> = <var_name> | <const_name> | <valeur> | NO_VALUE

<role_ent_name> désigne un nom de rôle joué par <ent_type>

B) Description

Désigne l'ensemble des occurrences du type d'entité ou d'association vérifiant la clause de sélection sur lequel une des opérations suivantes pourra être effectuée: assignation, modification, effacement.

Nous allons d'abord signaler quelques considérations concernant l'utilisation de variables dans la clause de sélection.

Toute variable apparaissant dans une clause de sélection doit référencer une entité ou une association dans la base de données, elle doit donc préalablement avoir été assignée ou utilisée pour une création ou une modification.

Toute variable d'entité ou d'association contient la référence de la dernière entité ou association assignée à cette variable, ou de la dernière entité ou association créée ou modifiée à l'aide de cette variable dans un ordre de création ou de modification. Une variable peut ne pas contenir de référence si elle n'a pas encore été utilisée dans ce contexte.

Il est donc possible dans une clause de sélection de référencer directement une entité ou une association, pour peu qu'elle soit référencée par une variable. Cela permet de rendre les clauses de sélection plus lisibles, et de travailler directement avec des objets que l'on peut désigner (la dernière entité d'un type qui a été créée dans le programme,...). Cela est aussi avantageux du point de vue des performances. En effet, le concept de référence est utilisé comme clé identifiante interne à PYRAMIDE. Si la clause de sélection contient donc une variable contenant une référence, le précompilateur pourra transformer cette partie en un accès direct pour PYRAMIDE.

La référence des variables utilisées dans la sélection n'est pas modifiée lors de cette opération. Par contre, les valeurs d'attributs de l'objet référencé par la variable peuvent être mises à jour dans cette variable. Cela est dû au fait que lorsque PYRAMIDE accède à un élément, il range automatiquement ses valeurs d'attributs dans la variable qui lui est passée en paramètre.

Nous allons maintenant prendre un à un chaque élément d'une clause.

Il faut d'abord citer le nom du TE ou TA sélectionné. Ce nom de type doit toujours apparaître. On peut après donner un nom de variable si elle contient une référence Cette variable doit évidemment être déclarée du type qui précède. L'utilisateur peut donner des valeurs d'attributs de l'entité (et en plus référencer les entités participantes pour une association) à la fois à l'aide d'une variable et de clauses WITH et BETWEEN. Si ces deux moyens sont utilisés en même temps, ce sont les renseignements donnés dans les clauses qui sont retenus par le précompilateur.

La clause WITH est nécessaire pour donner les valeurs d'attributs de l'élément sélectionné, s'il y a lieu de les donner.

La clause THAT est utile si l'utilisateur veut spécifier une clause de sélection contenant des prédicats navigationnels (<rel_cond_spec>), en citant les entités attachées à celle sélectionnée.

Pour ce type de prédicat, l'utilisateur doit d'abord citer le nom du rôle joué par l'entité selectionnée. La clause LINKED_TO est utile s'il est nécessaire de spécifier l'entité-cible dans le prédicat navigationnel. Toutes les entités-cibles spécifiées après cette clause doivent être liées à l'entité sélectionnée par la même association. Si l' utilisateur veut traverser plusieurs associations à partir de l'entité sélectionnée, il faudra répéter le prédicat navigationnel au moyen de connecteurs logiques. La clause THROUGH est nécessaire si l'utilisateur veut stipuler des valeurs d'attribut de l'association traversée dans le prédicat navigationnel.

La clause BETWEEN est nécessaire si l'utilisateur veut spécifier certaines entités liées à l'association sélectionnée.

La valeur NO_VALUE ne peut être utilisée qu'avec l'opérateur de comparaison =.

Voyons maintenant si la clause de sélection permet de lever les ambiguïtés possibles.

Il existe d'abord des ambiguïtés propres à la syntaxe de l'ordre. Elles doivent être résolues par l'utilisateur à l'aide de parenthèses.

Un deuxième type d'ambiguïté existe si un des types d'entité intervenant dans la sélection exerce plusieurs rôles dans un type d'association apparaissant dans la sélection. Pour lever cette ambiguïté, l'utilisateur devra citer le nom du rôle concerné par la sélection.

Par exemple, si un TA R lie le TE A avec le rôle RA au TE B avec les rôles RB1 et RB2. Une sélection pourrait être A THAT RA LIKNKED_TO B. Cette clause est ambigüe car B participe à R avec les rôles RB1 et RB2. On lève l'ambiguïté en écrivant A THAT RA LINKED_TO B THAT RB1 (ou RB2).

De même R BETWEEN B est ambigu. On lève l'ambiguïté en écrivant R BETWEEN B THAT RB1 (ou RB2).

D) Exemples

Pour des exemples voir assignation, boucle d'accès, effacement et modification.

III.6.5.6. L'assignation

A) Syntaxe

$ <var_name> := (<selection_entite> | <selection_association>) ;

B) Description

Si la clause de sélection désigne un élément, sa référence et ses valeurs d'attributs sont rangées dans la variable assignée, et erstatus vaut 0.

S'il existe dans la base plusieurs occurrences vérifiant la clause de sélection. Après l'opération, la variable contiendra les informations concernant la première occurrence que le système a trouvée dans la base.

Le type de la variable assignée doit évidemment être compatible avec le TE ou le TA sélectionné.

Si son contenu a reçu une référence, la variable peut être utilisée dans une sélection pour référencer une entité liée à l'entité sélectionnée, pour une opération de suppression, de modification, de création, de la même façon que pour la sélection.

C) Codes de retour

erstatus = 0; la variable contient une occurrence

erstatus = 1; il n'y a pas d'occurrence vérifiant la clause de sélection

erstatus = 99; erreur système ou d'entrée/sortie

D) Exemples

```
program assignation;
$ var cli: ENTITY client;
$       loc:RELATION location;
...

begin
    ...
      $ cli:= client WITH (name = 'Dupont') or (name = 'Marcel')
            THAT(possede LINKED_TO voiture WITH
numero_plaque <> 'AA24BB');

(* accède à un client de nom Dupont ou Marcel qui possède une voiture de
numéro de plaque différent de AA24BB*)

        if erstatus <> 0 then trt_err_assign;

    ...
```

```
loc:=location BETWEEN ( client WITH name ='Dupont' )
                AND  ( voiture WITH numero_plaque <> 'AA24BB' );
```

(*accède à une location du client de nom Dupont pour la voiture dont le numéro de plaque est différent de AA24BB*)

```
if erstatus <>  0 then trt_err_assign;
```

```
loc:=location BETWEEN ( client WITH name ='Dupont' THAT
achète voiture WITH numero_plaque <> 'AA24BB' );
```

(*accède à une location du client de nom Dupont qui achète la voiture dont le numéro de plaque est différent de AA24BB*)

```
if erstatus <>  0 then trt_err_assign;          ...
end.
```

III.6.5.7. Boucle d'accès

L'assignation ne permet de considérer qu'une seule occurrence d'un type donné à la fois. Or, un critère de sélection peut être satisfait par plusieurs occurrences. Pour accéder à ces occurrences, le langage utilise une structure de boucle d'accès.

A) Syntaxe

```
$  FOR <var_name> := (<selection_entite> | <selection_association>) DO
   <sequence>
$  ENDFOR;
```

<sequence> est une suite d'instructions PASCAL et/ou LEA

B) Description

La sémantique est la généralisation de l'assignation dans le sens où toute occurrence vérifiant le critère de sélection sera assignée successivement à la variable, ainsi que sera exécutée la suite d'instructions désignée par <séquence>.

C) Codes de retour

erstatus = 0; tout s'est bien passé

erstatus = 1; pas d'occurrence vérifiant le critère de sélection

erstatus = 99; erreur système ou d'entrée/sortie

D) Exemples

```
program boucle_d'acces;
...
$ var cli:ENTITY client;
$       loc:RELATION location;
...

begin
      ...
       $ FOR cli:=client DO

(*accède à tous les clients*)
      ...
      $ ENDFOR;
          if erstatus > 1 then trt_err_boucle;
      ...
        $ FOR cli:=client WITH name ='Marcel'
                          THAT possede LINKED_TO voiture DO

(*accède à tous les clients de nom Marcel possèdant une voiture*)
      ...
      $ ENDFOR;
          if erstatus > 1 then trt_err_boucle;
      ...
        $ FOR loc:=location BETWEEN ( client WITH name ='Marcel' THAT loue)
                            AND  ( voiture THAT est_louee_par) DO

(*accède à toutes les locations d'une voiture du client de nom Marcel*)
      ...
      $ ENDFOR;
          if erstatus > 1 then trt_err_boucle;
      ...
end.
```

III.6.5.8. Création

A) Syntaxe

```
$ CREATE (<creation_entite> | <creation_association>) ;

<creation_association> = <rel_type> <var_rel_name>
                         [WITH <attr_value_spec>]
                        [BETWEEN <ent_cond_create>]

<creation_entite> = <ent_type> <var_ent_name>
                         [WITH <attr_value_spec>]
                         [THAT <rel_cond_create>]

<rel_cond_create> = <rel_cond_create> <oplog> <rel_cond_create> |
                    <role_ent_name> [LINKED_TO <ent_cond_spec>]
                                    [THROUGH <rel_spec>]
<rel_spec> =  <rel_type> <var_rel_name> WITH <attr_value_spec> |
              <rel_type> <var_rel_name>

<ent_cond_create> = <ent_cond_create> <oplog> <ent_cond_create> |
                    <creation_entite>
```

<role_ent_name> désigne un nom de rôle joué par <ent_type>

B) Description

Cet ordre est nécessaire pour créer des éléments dans la base de données.

Nous avons choisi de spécifier un ordre de création assurant la cohérence de la base de données par rapport à sa définition. Ce qui veut dire que le système doit pouvoir détecter toute incohérence dans l'ordre de création avant son exécution. Une grande partie des incohérences possibles peuvent être détectées par le précompilateur lorsqu'il analysera la syntaxe de l'ordre, notamment l'utilisation d'un rôle inconnu, la propagation des mises-à-jour,...(voir la quatrième partie pour toutes les erreurs détectées à la précompilation). D'autres incohérences ne sont détectables qu'à l'exécution. Par exemple, si l'utilisateur ne donne pas de valeur à un attribut obligatoire (s'il assigne le contenu d'une variable à l'attribut). Tout ordre susceptible d'entraîner une incohérence n'est pas exécuté et

erstatus vaut 19. Il faudra aussi vérifier à l'exécution l'usage des valeurs d'identifiant. Si l'utilisateur commet une erreur dans une manipulation d'identifiants, l'rdre n'est pas exécuté, et erstatus vaut 2.

Un cas important dans lequel il faut garantir le maintien de la cohérence est celui de la propagation des mises-à-jour.

Supposons que l'ordre de création ne permette de créer qu'un seul élément à la fois, entité ou association. Si l'utilisateur désire créer une entité E1 jouant dans une association non encore créée (car sinon une association existerait sans entité participante) un rôle de connectivité minimale à 1. Cette opération serait refusée puisque si l'on admettait la création de E1, il pourrait exister dans la base de données une entité exerçant un rôle de connectivité minimale à 1, mais n'étant pas encore créée. La contrainte de connectivité minimale serait violée.

On pourrait résoudre le problème de la façon suivante. Il faudrait supposer que l'entité attachée à E1, c'est-à-dire E2 par l'association existe déjà. Dès lors, il suffirait de spécifier un ordre de création permettant de créer une entité avec ces rôles, et en même temps les associations qui la lient à d'autres entités. Ainsi, moyennant le fait que E2 existe déjà, l'utilisateur écrirait l'ordre pour créer E1, ainsi que son rôle de connectivité minimale à 1, et l'association dans le cadre de laquelle E1 exerce ce rôle, qu'il lierait à E2.

Cependant, cette solution fait l'hypothèse que E2 existe déjà. Or, si E2 participe à l'association la liant à E1, elle aussi avec un rôle de connectivité minimale à 1, il aurait fallu avec la même solution, pour créer E2, supposer que E1 existait déjà. Or, pour créer E1, il faut supposer que E2 existe. Cette solution n'est pas praticable. Or, on ne peut restreindre le modèle à un seul TE exerçant un rôle de connectivité minimale à 1 par TA. En effet, la situation opposée peut se présenter souvent. L'exemple typique est celui des lignes de commande. Une commande comprend au moins une ligne, et toute ligne appartient à une commande. Commande exerce donc un rôle de connectivité (1,n) dans l'association la liant à ligne, et ligne une connectivité (1,1) dans l'association la liant à commande.

La seule manière de résoudre ce problème est donc d'offrir un ordre de création permettant de créer plusieurs entités et associations. Ainsi, l'utilisateur en consultant le schéma de la base de données peut écrire l'ordre de création correct pour que toutes les contraintes de connectivité minimale à 1 soient respectées, de manière récursive (car il est évident que dans notre exemple E2 pourrait également exercer des rôles de connectivités minimales à 1). L'utilisateur pourra donc, dans notre exemple, écrire l'ordre créant à la fois E1, E2, et l'association les liant.

L'utilisateur peut reprendre la description des ordres WITH, THAT,... dans la sélection. Leur principe est le même ici. Toutefois, leur utilisation est légèrement différente.

Pour l'ordre de création, il est obligatoire de citer un nom de variable pour chaque type manipulé, sauf pour les types d'association. Pour ces derniers, l'utilisateur devra citer une clause THROUGH s'il veut insérer dans l'ordre une variable référençant l'association créée ou désignant une association par sa référence.

Si des variables référençant déjà des objets dans la base de données sont utilisés dans l'ordre, ces objets ne sont pas recréés. Ces variables serviront plutôt pour désigner les objets qui doivent être mis en correspondance avec les objets à créer. Pour ces objets déjà référencés, il n'est pas nécesaire d'insérer des clauses WITH, THAT, BETWWEN, puisque ceux-ci sont désignés univoquement par leur référence.

Par contre, les variables utilisées dans l'ordre qui ne référencent pas encore d'objets dans la base, référencent après exécution de l'ordre les objets créés par l'ordre. Pour les entités citées dans l'ordre et associées à des variables ne contenant pas de référence, il est possible mais non conseillé d'utiliser une clause WITH. Notons que si l'utilisateur entre des renseignements contradictoires pour les valeurs d'attributs dans la variable et dans la clause WITH, ce sont ceux contenus dans la clause WITH qui seront retenus par le précompilateur. Le mécanisme est identique pour les associations dans le cas des clauses WITH et BETWEEN.

L'utilisation de la clause <attr_value_spec> est restreinte à l'opérateur de comparaison =.

C) Codes de retour

erstatus = 0; tout s'est bien passé

erstatus = 2; contrainte d'identifiant violée, pas de création effectuée

erstatus = 14; pas de schéma actif, pas de création effectuée

erstatus = 19;l'exécution de cette opération entraînerait une incohérence par rapport à la définition du schéma de la base de données; l'opération n'est pas effectuée

erstatus = 80; espace mémoire dépassé

erstatus = 90; le schéma est incohérent

erstatus = 99; erreur système ou d'entrée/sortie

D) Exemples

```
program creation;
$ var cli: ENTITY client;
$       loc:RELATION location;
$       voit:ENTITY voiture;
...
begin
      ...
          $ CREATE client cli WITH (numero_id_client = 1234) AND (name =
'Dupont');

(*crée un client de numéro 1234 et de nom Dupont, et range sa référence
dans cli*)
              if erstatus <> 0 then trt_err_create;
          ...
             voit.numero_chassis:=12345;
             voit.numero_plaque:='12AA24';
           $ CREATE voiture voit;

(*crée une voiture de numéro de châssis 12345 et de numéro de plaque
12AA24, et range sa référence dans voit*)
              if erstatus <> 0 then trt_err_create;
```

```
...
$ CREATE location loc BETWEEN (client cli)
                              AND (voiture voit);

(* crée une association de location entre les entités référencées par cli et
voit, et range sa référence dans loc*)
        if erstatus <> 0 then trt_err_create;
...
end.
```

III. 6. 5.9. Suppression

A) Syntaxe

$ DELETE (<selection_entite> | <selection_association>) ;

B) Description

Cet ordre est nécessaire pour pouvoir effacer des objets dans la base de données.

L'ordre de suppression est de type ensembliste. Son exécution entraîne la suppression dans la base de tous les éléments vérifiant la clause de sélection.

Les éléments désignés dans la clause de sélection du fait de leurs relations avec l'objet sélectionné ne sont effacés que si cela est nécessaire afin d'assurer le maintien de la cohérence de la base de données. Sont aussi effacés tous les éléments n'apparaissant pas dans la clause de sélection, mais dont la suppression est également nécessaire pour assurer le maintien de la cohérence de la base de données.

De cette façon, le problème de la propagation des suppressions d'une entité ou association à toutes les entités attachées par des rôles de connectivité minimale à 1, et des associations les liant est géré récursivement de façon automatique lors de l'exécution de l'ordre de suppression. La BD est donc cohérente après exécution de cet ordre.

La description de la clause de sélection est toujours valable ici.

Après l'exécution de l'ordre, les variables qui référençaient des objets dans la base, qui ont été effacés, ne référencent plus rien, mais elles contiennent toujours les valeurs d'attributs des objets effacés.

C) Codes de retour

erstatus = 0; tout s'est bien passé

erstatus = 1; il n'existe pas de telle occurrence dans le schéma

erstatus = 14; pas de schéma actif, pas de suppression effectuée

erstatus = 90; le schéma est incohérent

erstatus = 99; erreur système ou d'entrée/sortie

D) Exemples

```
program  effacement;
...
begin
    ...
      $ DELETE client WITH numero_id_client = 1234;
       if erstatus <> 0 then trt_err_remove;
    ...
end.
```

III.6.5.10. Modification

A) Syntaxe

$ MODIFY <selection_entite> USING <attr_value_spec> ;
$ MODIFY <selection_association> USING <attr_value_spec> ;

B) Description

Cet opérateur est nécessaire pour pouvoir modifier les valeurs d'attributs d'entités ou d'associations dans la base de données.

Pour changer les entités participant à une association, il faut effacer l'association et la recréer entre les entités adéquates.

Après exécution de l'ordre, toutes les entités ou associations vérifiant la clause de sélection ont leurs valeurs d'attributs modifiées par celles contenues dans la clause USING. L'ordre ne modifie que les valeurs d'attributs pour les attributs cités dans la clause USING.

Dans <attr_value_spec> dans la clause USING, le seul opérateur de comparaison permis est =.

C) Codes de retour

erstatus = 0; tout s'est bien passé

erstatus = 1; il n'existe pas de telle occurrence dans le schéma

erstatus = 2; contrainte d'identifiant violée, pas de modification effectuée

erstatus = 14; pas de schéma actif, pas de modification effectuée

erstatus = 19; la modification de cette occurrence entraînerait une incohérence dans le schéma, pas de modification effectuée

erstatus = 80; espace mémoire dépassé, pas de modification effectuée

erstatus = 90; le schéma est incohérent

erstatus = 99; erreur système ou d'entrée/sortie

D) Exemple

```
program modification;
...
begin
    ...
      $ MODIFY client WITH numero_id_client = 1234
          USING client WITH name = 'Marcel';
(*l'attribut NAME du client de numéro 1234 devient 'Marcel'*)
       if erstatus <> 0 then trt_err_modify;
    ...
end.
```

III.6.5.11. Gestion des transactions

Le langage LEA comprend les ordres nécessaires à la gestion des transactions. Cette gestion est directement dérivée de PYRAMIDE. Cependant, celle-ci n'étant pas implémentée dans PYRAMIDE, elle ne le sera pas non plus pour LEA. Toutefois, la couche supérieure est conçue de façon à ce que dès que PYRAMIDE permettra cette fonctionnalité, LEA puisse l'offrir également moyennant quelques modifications minimes dans le précompilateur.

LEA reprend donc le système des transactions emboîtées de PYRAMIDE. Chaque transaction est composée d'un ensemble de transactions-filles, qui doivent commencer après, et se terminer avant leur transaction-mère. Les transactions emboîtées permettent de distribuer le travail entre les transactions-filles et d'en annuler un nombre limité.

A) Démarrage d'une transaction

Syntaxe

$ BEGIN_TRANS <var_id_trans>;

Description

Dans la base de données ouverte, une transaction est commencée comme unité de cohérence, synchronisation, et récupération. Si l'opération réussit, la variable d'identification de la transaction reçoit une valeur. Si la transaction est commencée dans une transaction déjà en cours, elle est considérée comme sa transaction-fille.

Codes de retour

erstatus = 0; une nouvelle transaction est commencée

erstatus = 30; il est intervenu un problème lors de l'ouverture de la transaction, elle n'est pas ouverte

B) Clôture d'une transaction

Syntaxe

$ END_TRANS <var_id_trans>;

Description

Dans la base de données ouverte, la transaction identifiée par <var_id_trans> est clôturée, ainsi que toutes ses transactions-filles. Après que l'opération se soit terminée correctement, toutes les modifications effectuées dans la base de données depuis l'ouverture de la transaction clôturée sont effectifs.

Codes de retour

erstatus = 0; une nouvelle transaction est terminée

erstatus = 90; il est intervenu un problème lors de la clôture de la transaction, elle n'est pas fermée

C) Annulation d'une transaction

Syntaxe

$ ABORT_TRANS <var_id_trans>;

Description

Dans la base de données ouverte, la transaction identifiée par <var_id_trans> est annulée, ainsi que toutes ses transactions-filles en défaisant toutes les modifications effectuées depuis le commencement de la transaction annulée

Codes de retour

erstatus = 0; une nouvelle transaction est annulée

erstatus = 90; il est intervenu un problème lors de l'annulation de la transaction, elle n'est pas annulée

III.6.6. Exemples

III.6.6.1. Etude du cas 'GARAGE'

Ces exemples se basent sur l'application définie dans l'annexe I.I. Ce sont des programmes types complets qui montrent les possibilités offertes par le langage entité-association défini.

```
program selection_association;

(* ce programme affiche les heures de début et de fin des
occurrences du type d'association realisation qui sont liées à
des opérations standards de numéro égal à 3 ou à 4 ,ou à des
mécaniciens de nom 'Marcel' ou 'Nestor'*)

$ USES DATABASE 'garage' SCHEMA 'garage';

$ var varreal: RELATION realisation;

procedure trt_err_open;
begin
end;
procedure trt_err_close;
begin
end;
procedure trt_err_for;
begin
end;

begin
     $ OPEN DATABASE 'garage' SCHEMA 'garage';
     if erstatus <> 0 then trt_err_open;

     $ FOR vareal := realisation BETWEEN ( operation_standard
                  WITH (numero_standard = 3) OR (numero_standard = 4))
                                   OR ( mecanicien
                  WITH (nom = 'Marcel') OR (nom = 'Nestor')) DO

          write (varreal.heure_debut, '    ', varreal.heure_fin);

     $ ENDFOR;
     if erstatus > 1 then trt_err_for;

     $ CLOSE;
     if erstatus <> 0 then trt_err_close;
end.
```

```
program recherche;

(* ce programme affiche le nom des clients dont la voiture a été
réparée après le 01/01/1989. Le programme fait cela de deux
manières différentes *)

$ USES DATABASE 'garage' SCHEMA 'garage';
```

```
$ var varcli: ENTITY client;
$     varvoit: ENTITY voiture;

procedure trt_err_open;
begin
end;
procedure trt_err_close;
begin
end;
procedure trt_err_for;
begin
end;

begin
     $ OPEN DATABASE 'garage' SCHEMA 'garage';
     if erstatus <> 0 then trt_err_open;

     $ FOR varcli := client
                     THAT possede
                     LINKED_TO voiture THAT sujette_a
                                       LINKED_TO ordre_de_reparation
                                       WITH (date_OR > '0101I989') DO

          write (varcli.nom_cli);

     $ ENDFOR;
     if erstatus > 1 then trt_err_for;

     $ FOR varvoit := voiture THAT sujette_a
                              LINKED_TO ordre_de_reparation
                                        WITH (date_OR > '0101I989') DO
     $         FOR varcli := client THAT possede
                                   LINKED_TO voiture varvoit DO

          write (varcli.nom_cli);
          write (varvoit.numero_plaque);

     $         ENDFOR;
     if erstatus > 1 then trt_err_for;
     $ ENDFOR;
     if erstatus > 1 then trt_err_for;

     $ CLOSE;
     if erstatus <> 0 then trt_err_close;
end.
```

program creation_d_occurrences;

```
(*  ce  programme saisit au clavier des données de  l'utilisateur
pour  créer des occurrences des types d'entité client et  voiture
et du type d'association proprietaire *)

$ USES DATABASE 'garage' SCHEMA 'garage';

$ var varvoit: ENTITY voiture;
$     varcli: ENTITY client;
$     varprop: RELATION proprietaire;
i:integer;
```

```
procedure trt_err_open;
begin
end;
procedure trt_err_close;
begin
end;
procedure trt_err_create;
begin
end;

begin
     $ OPEN DATABASE 'garage' SCHEMA 'garage';
     if erstatus <> 0 then trt_err_open;

     write ('Num. id. client: '); read (varcli.numero_id_client);
     while varcli.numero_id_client <> 0 do
     begin
          write ('Nom du client: '); read (varcli.nom_cli);
          write('Prenoms du client: ');
          i:=1;
          repeat
          begin
               read (varcli.descriptif_client.prenoms_client[i]);
               i:=i+1;
          end;
          until (i>5) or (varcli.descriptif_client.prenoms_client[i] = '
          write('Adresse:');
          write('rue: ');
          read (varcli.descriptif_client.adresse_client.rue);
          write('numero: ');
          read (varcli.descriptif_client.adresse_client.numero);
          write('code postal: ');
          read (varcli.descriptif_client.adresse_client.code_postal);
          $ CREATE client varcli;
          if erstatus <> 0 then trt_err_create;
          write ('Num chassis de la voiture: ');
          read(varvoit.numero_chassis);
          while varvoit.numero_chassis <> 0 do
          begin
             write('Numero de plaque: '); read(varvoit.numero_plaque);
             $ CREATE voiture varvoit THAT est_possedee_par
                                      LINKED_TO client varcli
                                      THROUGH proprietaire varprop;
             if erstatus <> 0 then trt_err_create;
             write ('Propriétaire: ');
             writeln (varprop.rpossede.nom_cli);
             write ('Num chassis de la voiture: ');
             read(varvoit.numero_chassis);
          end;
          write ('Num. id. client: '); read (varcli.numero_id_client);
     end;

     $ CLOSE;
     if erstatus <> 0 then trt_err_close;
end.
```

```
program listage_des_TE_et_des_TA;

(* ce programme liste les types d'entité et d'association du
schéma 'garage' avec pour chaque type l'ensemble de ses
attributs*)

$ USES DATABASE 'garage';

$ var varsch: ENTITY dbschema;
$     varent: ENTITY entity_type;
$     varrel: ENTITY rel_type;
$     varatt: ENTITY attribute;
$     varrole: ENTITY role;
i:integer;

procedure trt_err_open;
begin
end;
procedure trt_err_close;
begin
end;
procedure trt_err_assign;
begin
end;
procedure trt_err_for;
begin
end;

$ procedure lire_att_dec ($ varatt1: ENTITY attribute);

$ var varatt2: ENTITY attribute;
begin
     $ FOR varatt2 := attribute THAT att_in_att
                                LINKED_TO attribute varatt1
                                          THAT att_of_att  DO

          write (varatt2.name);
          if varatt2.val_type:='G' then lire_att_dec (varatt2);
     $ ENDFOR;
     if erstatus > 1 then trt_err_for;
end;

begin
     $ OPEN DATABASE 'garage';
     if erstatus <> 0 then trt_err_open;

     $ varsch:= dbschema WITH (name = 'garage');
     if erstatus > 1 then trt_err_assign;
     write('Nom du schéma: ', varsch.name);

     $ FOR varent := entity_type THAT et_in_db
                                 LINKED_TO dbschema varsch DO

          write (varent.name);

     $ ENDFOR;
     if erstatus > 1 then trt_err_for;

     $ FOR varrel := rel_type THAT rt_in_db
                                 LINKED_TO dbschema varsch DO
```

```
          write (varrel.name);
          $ FOR varrole := role THAT ro_in_rt
                                LINKED_TO rel_type varrel DO

               write(varrole.name);
               write(varrole.min_con);
               write(varrole.max_con);
               $ varent:= entity_type THAT ro_of_et
                                     LINKED_TO role varrole;
               if erstatus > 1 then trt_err_assign;
               write (varent.name);

          $ ENDFOR;
          if erstatus > 1 then trt_err_for;

     $ ENDFOR;
     if erstatus > 1 then trt_err_for;

     $ FOR varent := entity_type THAT et_in_db
                                LINKED_TO dbschema varsch DO

          write (varent.name);
     $    FOR varatt := attribute THAT att_in_et
                                 LINKED_TO entity_type varent DO

               write (varatt.name);
               if varatt.val_type:='G' then lire_att_dec (varatt);

     $    ENDFOR;
          if erstatus > 1 then trt_err_for;

     $ ENDFOR;
     if erstatus > 1 then trt_err_for;

     $ FOR varrel := rel_type THAT rt_in_db
                               LINKED_TO dbschema varsch DO

          write (varrel.name);
     $    FOR varatt := attribute THAT att_in_rt
                                 LINKED_TO rel_type varrel DO

               write (varatt.name);
               if varatt.val_type:='G' then lire_att_dec (varatt);

     $    ENDFOR;
          if erstatus > 1 then trt_err_for;

     $ ENDFOR;
     if erstatus > 1 then trt_err_for;

     $ CLOSE;
     if erstatus <> 0 then trt_err_close;
end.
```

III.6.6.2. Manipulations du méta-schéma

```
program description_du_schéma;

(* ce programme constitue une bibliothèque de primitives qui
permettent au programmeur de décrire son schéma plus facilement*)

$ USES DATABASE 'garage';

$var schcour:ENTITY dbschema;
$    descrschcour:ENTITY db_desc;
$    entcour:ENTITY entity_type;
$    descrentcour: ENTITY et_desc;
$    relcour:ENTITY rel_type;
$    descrrelcour: ENTITY rt_desc;
$    rolecour:ENTITY role;
$    attcour,attperecour:ENTITY attribute;
$    descrattcour: ENTITY att_desc;
$    idcour:ENTITY group;
$    compcour:ENTITY component;

procedure trt_err_open;
begin
end;
procedure trt_err_close;
begin
end;
procedure trt_err_create;
begin
end;
procedure trt_err_assign;
begin
end;

                (*******************************)

$ procedure CREATE_SCH ($var varsch:ENTITY dbschema);
begin
     $ CREATE dbschema varsch;
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)

$ procedure CREATE_DB_DESC ($varsch:ENTITY dbschema;
                            $var vardescrsch:ENTITY db_desc);
begin
     $ CREATE  db_desc vardescrsch THAT desc_of_db
                                   LINKED_TO dbschema varsch;
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)

$ procedure CREATE_ET ($varsch:ENTITY dbschema;
                       $var varent: ENTITY entity_type);
begin
     $ CREATE entity_type varent THAT et_in_db
                                LINKED_TO dbschema varsch;
     if erstatus <> 0 then trt_err_create;
end;
```

```
                (*******************************)

$  procedure CREATE_ET_DESC ($varent:ENTITY entity_type;
                             $var vardescret:ENTITY et_desc);
begin
     $ CREATE  et_desc vardescret THAT desc_of_et
                                  LINKED_TO entity_type varent;
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)

$ procedure CREATE_RT ($varsch:ENTITY dbschema;
                       $varrel:ENTITY rel_type);
begin
     $ CREATE rel_type varrel THAT rt_in_db
                               LINKED_TO dbschema varsch;
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)

$ procedure CREATE_ROLE ($varsch:ENTITY dbschema;
                         $varrel:ENTITY rel_type;
                         $var varrole: ENTITY role;
                         entname:string[32]);
$ var varent:ENTITY entity_type;
begin
     $ varent:= entity_type WITH name = entname
                            THAT et_in_db LINKED_TO dbschema varsch;
     if erstatus <> 0 then trt_err_assign;
     $  CREATE role varrole THAT (ro_in_et LINKED_TO  entity_type
                                           varent)
                            AND   (ro_in_rt  LINKED_TO   rel_type
                                           varrel);
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)

$ procedure CREATE_RT_DESC ($varrel: ENTITY rel_type;
                            $var vardescrrt: ENTITY rt_desc);
begin
     $  CREATE   rt_desc  vardescrrt  THAT  desc_of_rt
                                      LINKED_TO rel_type varrel;
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)

$ procedure CREATE_ATT_ET ($varent:ENTITY entity_type;
                           $ var varatt:ENTITY attribute);
begin
     $ CREATE attribute varatt THAT att_in_et
                               LINKED_TO entity_type varent;
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)
```

```
$ procedure CREATE_ATT_RT ($varrel:ENTITY rel_type;
                           $ var varatt:ENTITY attribute);
begin
     $ CREATE attribute varatt THAT att_in_rt
                               LINKED_TO rel_type varrel;
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)

$ procedure CREATE_ATT_ATT ($varattpere:ENTITY attribute;
                            $ var varatt:ENTITY attribute);
begin
     $ CREATE attribute varatt THAT att_in_att
                               LINKED_TO attribute varattpere;
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)

$ procedure CREATE_ATT_DESC ($varatt:ENTITY attribute;
                             $var vardescratt:ENTITY att_desc);
begin
     $ CREATE  att_desc vardescratt THAT desc_of_att
                                    LINKED_TO attribute varatt;
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)

$ procedure CREATE_ID_ET ($varent:ENTITY entity_type;
                          $varatt:ENTITY attribute;
                          $var varid:ENTITY group;
                          $var varcomp:ENTITY component);
begin
     $ CREATE group varid THAT (gr_in_et
                                LINKED_TO entity_type varent)
                          AND (comp_of_gr
                               LINKED_TO component varcomp
                                   THAT (comp_in_att
                                   LINKED_TO attribute varatt));
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)

$ procedure CREATE_ID_RT ($varrel:ENTITY rel_type;
                          $varatt:ENTITY attribute;
                          $var varid:ENTITY group;
                          $var varcomp:ENTITY component);
begin
     $ CREATE group varid THAT (gr_in_rt
                               LINKED_TO rel_type varrel)
                          AND (comp_of_gr
                               LINKED_TO component varcomp
                                   THAT (comp_in_att
                                   LINKED_TO attribute varatt));
     if erstatus <> 0 then trt_err_create;
end;

                (*******************************)
```

```
var testid,i:integer;

begin
     clrscr;
     gotoxy (23,1);write('CREATION D''UN SCHEMA');
     gotoxy (23,2);write('--------------------');

(* création d'un schéma de nom *)

     gotoxy (5,3); write ('nom du schéma: ');
     readln (schcour.name);
     CREATE_SCH (schcour);
     gotoxy (5,4); write ('description du schéma: ');
     gotoxy (1,5);
     readln (descrschcour.descr);
     while descrschcour.descr<>'' do
     begin
          CREATE_DB_DESC (schcour,descrschcour);
          gotoxy (5,4); write ('description du schéma: ');
          gotoxy (1,5); write ('                                    ');
          gotoxy(1,5);
          readln (descrschcour.descr);
     end;

(* création des types d'entité *)

     gotoxy (5,6); write (nom du TE: );
     readln (entcour.name);
     while entcour.name<>'' do
     begin
          CREATE_ET (schcour,entcour);
          gotoxy (5,7); write (nom d'attribut: ');
          readln (attcour.name);
          while attcour.name<>'' do
          begin
               gotoxy (5,8); write ('type:       ');
               readln (attcour.val_type);
               gotoxy (5,9); write ('longueur:       ');
               readln (attcour.val_length);
               gotoxy (5,10); write ('dec.:       ');
               readln (attcour.dec);
               gotoxy (5,11); write ('min_rep:       ');
               readln (attcour.min_rep);
               gotoxy (5,12); write ('max_rep:       ');
               readln (attcour.max_rep);
               gotoxy (30,12); write ('id: ');
               readln (testid);
               CREATE_ATT_ET (entcour,attcour);
               if testid<>0 then
               begin
                    idcour.number:=testid;
                    compcour.number:=1;
                    CREATE_ID_TE (entcour,attcour,idcour,compcour);
               end;
               if attcour.val_type = 'G' then
               begin
                    attperecour:=attcour;
                    gotoxy (8,13); write (nom d'attribut: ');
                    readln (attcour.name);
                    while attcour.name<>'' do
```

```
                    begin
                         gotoxy (8,14); write ('type:       ');
                         readln (attcour.val_type);
                         gotoxy (8,15); write ('longueur:       ');
                         readln (attcour.val_length);
                         gotoxy (8,16); write ('dec.:       ');
                         readln (attcour.dec);
                         gotoxy (8,17); write ('min_rep:       ');
                         readln (attcour.min_rep);
                         gotoxy (8,18); write ('max_rep:       ');
                         readln (attcour.max_rep);
                         CREATE_ATT_ATT (attperecour,attcour);
                         gotoxy (30,18); write ('id: ');
                         readln (testid);
                         if testid<>0 then
                         begin
                              idcour.number:=testid;
                              compcour.number:=1;
                         CREATE_ID_TE (entcour,attcour,idcour,compcour);
                         end;
                         gotoxy (28,13); write ('                  ');
                         readln (attcour.name);
                    end;
               end;
               gotoxy (25,7); write ('                  ');
               readln (attcour.name);
          end;
          gotoxy(16,6); write ('                              ');
          gotoxy(16,6); readln (entcour.name);
     end;


(*  création de types d'association *)

     for i:= 6 to 18 do
     begin
          gotoxy (1,i);
          write('                                          ');
     end;
     gotoxy (5,6); write ('nom du TA: ');
     readln (relcour.name);
     while relcour.name<>'' do
     begin
          CREATE_RT (schcour,relcour);
          gotoxy (5,7); write (nom d'attribut: ');
          readln (attcour.name);
          while attcour.name<>'' do
          begin
               gotoxy (5,8); write ('type:       ');
               readln (attcour.val_type);
               gotoxy (5,9); write ('longueur:       ');
               readln (attcour.val_length);
               gotoxy (5,10); write ('dec.:       ');
               readln (attcour.dec);
               gotoxy (5,11); write ('min_rep:       ');
               readln (attcour.min_rep);
               gotoxy (5,12); write ('max_rep:       ');
               readln (attcour.max_rep);
               gotoxy (30,12); write ('id: ');
               readln (testid);
```

```
                CREATE_ATT_RT (relcour,attcour);
                if testid<>0 then
                begin
                     idcour.number:=idtest;
                     compcour.number:=1;
                     CREATE_ID_TA (relcour,attcour,idcour,compcour);
                end;
                if attcour.val_type = 'G' then
                begin
                     attperecour:=attcour;
                     gotoxy (8,13); write (nom d'attribut: ');
                     readln (attcour.name);
                     while attcour.name<>'' do
                     begin
                          gotoxy (8,14); write ('type:       ');
                          readln (attcour.val_type);
                          gotoxy (8,15); write ('longueur:       ');
                          readln (attcour.val_length);
                          gotoxy (8,16); write ('dec.:       ');
                          readln (attcour.dec);
                          gotoxy (8,17); write ('min_rep:       ');
                          readln (attcour.min_rep);
                          gotoxy (8,18); write ('max_rep:       ');
                          readln (attcour.max_rep);
                          CREATE_ATT_ATT (attperecour,attcour);
                          gotoxy (30,18); write ('id: ');
                          readln (testid);
                          if testid<>0 then
                          begin
                               idcour.number:=testid;
                               compcour.number:=1;
                         CREATE_ID_TA (relcour,attcour,idcour,compcour);
                          end;
                          gotoxy (28,13); write ('                    ');
                          readln (attcour.name);
                     end;
                end;
                gotoxy (25,7); write ('                  ');
                readln (attcour.name);
           end;
           gotoxy (5,20); write ('nom du rôle: ');
           readln (rolecour.name);
           while rolecour.name<>'' do
           begin
                gotoxy (5,21);write('min_con:     :');
                readln (rolecour.min_con);
                gotoxy (30,21); write ('max_con: ');
                readln (rolecour.max_con);
                gotoxy (5,22); write ('nom de l'entité');
                readln (entname);
                CREATE_ROLE (relcour,rolecour,entname);
                gotoxy (18,20); write ('                  ');
                readln (rolecour.name);
           end;
           gotoxy(16,6); write ('                              ');
           gotoxy(16,6); readln (relcour.name);
     end;
end.
```

III.7. Fonctionnement général du système LEA

Nous ferons dans la suite de cette partie référence à l'exemple de la page suivante, afin de clarifier l'exposé. L'application modélisée est la suivante:

Une usine est toujours caractérisée par un nom, un produit par son numéro. Une usine fabrique au moins un produit. Un produit est toujours fabriqué par au moins une usine. Le coût de fabrication est dépendant de l'usine qui le fabrique.

Le problème de l'extension dynamique du schéma au niveau de la couche LEA sera discuté plus loin. Cette fonction n'est cependant pas implémentée. Dès lors, l'utilisation du système se fera en deux phases distinctes:

- création d'une base vide de données,

- manipulation de la base avec possibilité d'accès à la définition des données dans le dictionnaire de données.

III.7.1. Création d'une base de données

a) l'utilisateur doit copier le fichier standard.dtb dans son fichier de travail; par exemple: **copy standard.dtb fabric.dtb**. De cette façon, fabric.dtb contient un dictionnaire de données opérationnel, qui lui-même contient une description du méta-schéma conforme au modèle EAC.

b) il faut ensuite garnir le méta-schéma avec la description du schéma EAC de l'application. Pour cela, l'utilisateur doit écrire un programme PASCAL contenant des ordres LEA nécessaires à cette tâche, dans un fichier suffixé par .lea (par exemple,**creer_fabric.lea**). Notons que ce programme travaille sur le schéma EAC décrivant le méta-schéma, en manipulant des occurrences pour les méta-types. Il ne peut pas encore travailler sur les concepts du schéma de l'application car il n'existe pas encore de base de données opérationnelle conforme à ce schéma.

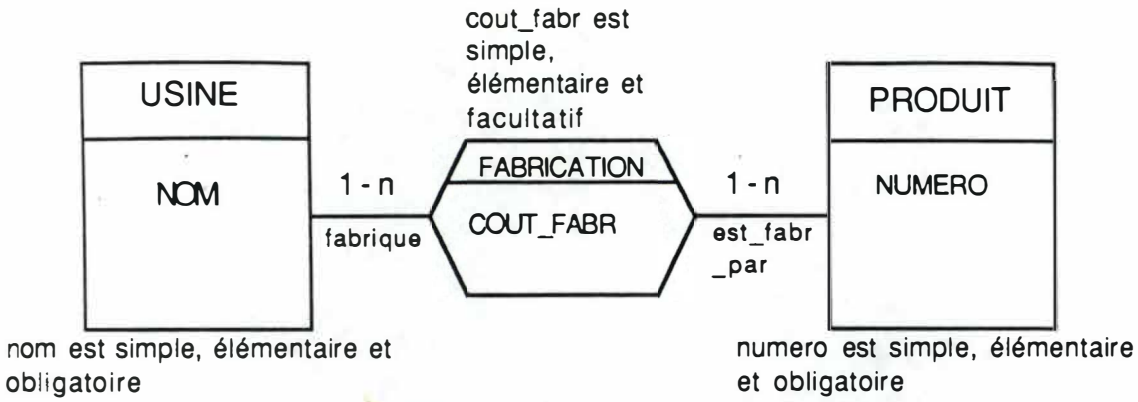


schéma de nom FABRIC exprimé dans le formalisme du modèle EAC

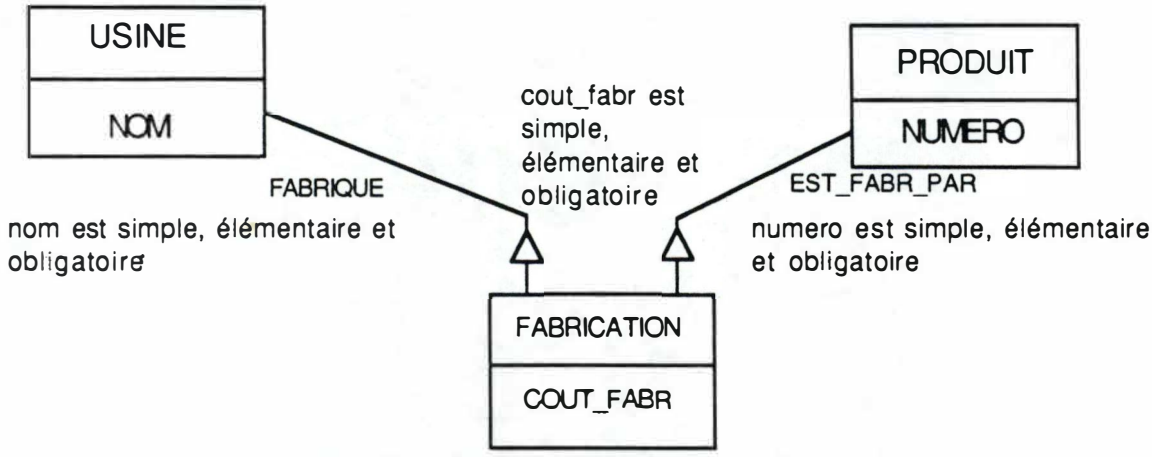
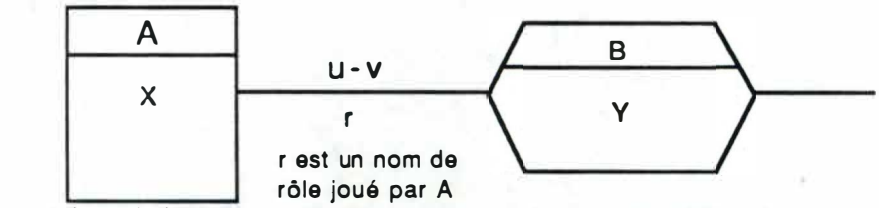


schéma transformé équivalent exprimé dans le formalisme du modèle supporté par PYRAMIDE

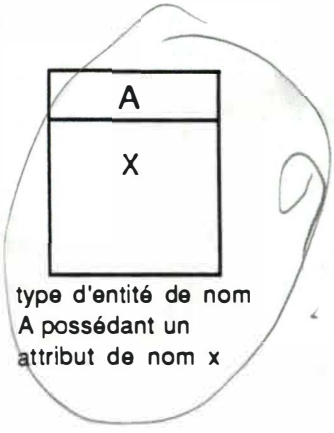
LEGENDE:



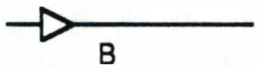
type d'entité de nom A possédant un attribut de nom x

r est un nom de rôle joué par A dans B avec les connectivités u-v

type de relation de nom B possédant un attribut de nom y

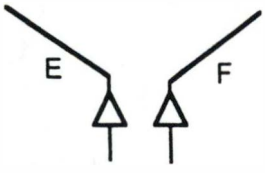


type d'entité de nom A possédant un attribut de nom x



représente un type d'association de nom B, et de connectivité 0-n dans le sens

Δ O (origine)
n (destination)



toute association du type E d exister en même temps qu'un association du type F

c) une fois le programme écrit, il faut lancer le précompilateur de LEA (par exemple: **precompi creer_fabric**). Celui-ci va générer un programme PASCAL où tout ordre du langage LEA aura été transformé en une séquence d'instructions PASCAL, avec des primitives de PYRAMIDE (par exemple: **creer_fabric.pas**). Dans ce programme sont inclus par le précompilateur la clause d'inclusion de la unit "PYRAMIDE", ainsi que le fichier de types de PYRAMIDE (standard.typ), le fichier des types associés aux TA, les déclarations des variables des types associés aux TE et aux TA, et des variables générées par le précompilateur (creer_fabric.var). L'effet de ce programme sera de charger dans le méta-schéma les descriptions du schéma EAC, et du schéma transformé correspondant. Si le schéma de l'utilisateur contient une structure non conforme au modèle EAC, la précompilation est refusée, un diagnostic est renvoyé avec le numéro de la ligne d'erreur dans le programme source. La liste des diagnostics est donnée dans la partie consacrée au précompilateur.

Fonctionnement du précompilateur

Le précompilateur ouvre le fichier source en lecture. Il crée un fichier suffixé par .pas dans lequel il va écrire le programme transformé. Il y insère d'abord l'ordre d'inclusion de la unit "PYRAMIDE", ainsi que du fichier STANDARD.TYP, et du fichier CREER_FABRIC.VAR. Toute déclaration de variable rencontrée est transformée pour être compatible avec les types de PYRAMIDE et insérée dans le fichier CREER_FABRIC.VAR.

Le précompilateur recopie toutes les lignes ne contenant pas d'ordre LEA. Pour les autres, il analyse d'abord la conformité de la ligne par rapport à la syntaxe de LEA, et aux structures du méta-schéma, en consultant le dictionnaire de données. C'est ici qu'il détectera par exemple un nom de rôle inconnu pour le méta-schéma dans l'ordre LEA.

Il va ensuite produire le code nécessaire au stockage du schéma EA de nom fabric dans le dictionnaire. En analysant les ordres, il détecte les structures à créér. Par exemple, le précompilateur va constater que l'association de nom fabrication contient un attribut. Il sait que dans ce cas il doit transformer cette structure pour produire le schéma transformé (voir schéma). Il va donc produire le code pour stocker dans le dictionnaire le schéma transformé. Le précompilateur produit aussi le code pour gérer les erreurs à l'exécution (mise-à-jour de la variable erstatus).

d) il faut ensuite compiler ce programme avec TURBO-PASCAL et l'exécuter, afin de charger les descriptions des deux schémas dans le dictionnaire de données.

e) il reste enfin à lancer l'utilitaire de PYRAMIDE (**metacomp**). Celui-ci va se charger de créer une base de données opérationnelle à partir de la description du schéma conforme à son modèle contenue dans le dictionnaire de données. Le fichier de l'utilisateur (**fabric.dtb**) est donc étendu pour contenir une base de données opérationnelle conforme à son schéma de données. Cette phase produit aussi un fichier de types de données PASCAL (par exemple, **fabric.typ**) à inclure dans tout programme d'application travaillant sur la base de données.

III.7.2. Manipuler la base de données

a) l'utilisateur doit d'abord écrire un programme d'application PASCAL manipulant la base de données à l'aide de l'interface de programmation LEA, dans un fichier suffixé par .lea (par exemple, **manip_fabric.lea**)

b) par le même processus qu'en A) (par exemple, **precompi manip_fabric**), le précompilateur transforme le programme en un programme PASCAL(**manip_fabric.pas**) conforme à PYRAMIDE, où ont été inclus le fichier des types (par exemple, fabric.typ), et le fichier des variables (par exemple, manip_fabric.var), ainsi que l'ordre d'inclusion de la unit "PYRAMIDE".

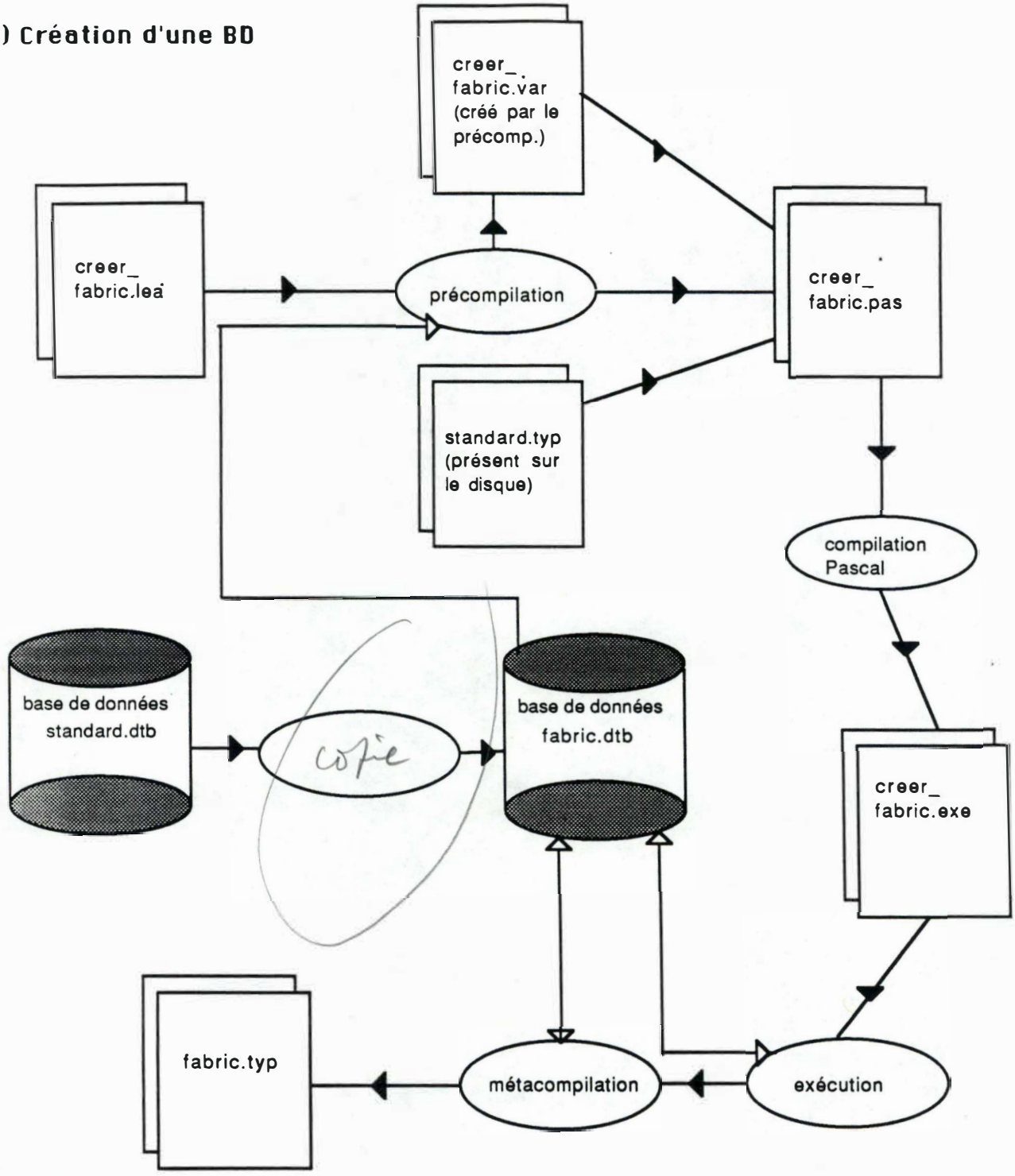
> Fonctionnement du précompilateur
>
> Le précompilateur fonctionne de la même façon que lors de la création d'une base de données. Il transforme les ordres LEA manipulant le schéma EAC, en une séquence PASCAL avec des appels à PYRAMIDE, travaillant sur le schéma transformé.
> Ainsi, par exemple si un ordre LEA contient une boucle d'accès aux produits d'une usine, le précompilateur, en consultant le dictionnaire va constater que le type d'association fabrication liant ces deux types d'entité contient un attribut. Il sait que cette structure a été transformée en un type d'entité de même nom que le type d'association de départ, et que les noms des chemins liant ces types d'entité au type d'entité intermédiaire sont les noms de rôle contenus dans le schéma EAC. A partir de là, il peut donc transformer la structure d'accès de départ en deux boucles d'accès: accès aux fabrications de l'usine, et pour chaque fabrication au produit lié par le chemin.

c) il suffit enfin de compiler ce programme avec TURBO-PASCAL et de l'exécuter.

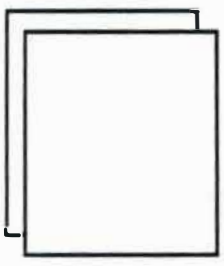
Les schémas des pages suivantes représentent les étapes successives de l'utilisation du système ainsi que leurs fichiers en entrée et sortie, les interactions avec la BD sont également signalées.

Tout produit en entrée d'un processus se retrouve également inchangé en sortie (par exemple, suite à la compilation d'un fichier sont présents en sortie le fichier compilé, mais aussi le fichier source). Ce fait n'est pas représenté sur les schémas afin de les alléger.

A) Création d'une BD



LEGENDE:



fichier

processus
lancé par
l'utilisateur

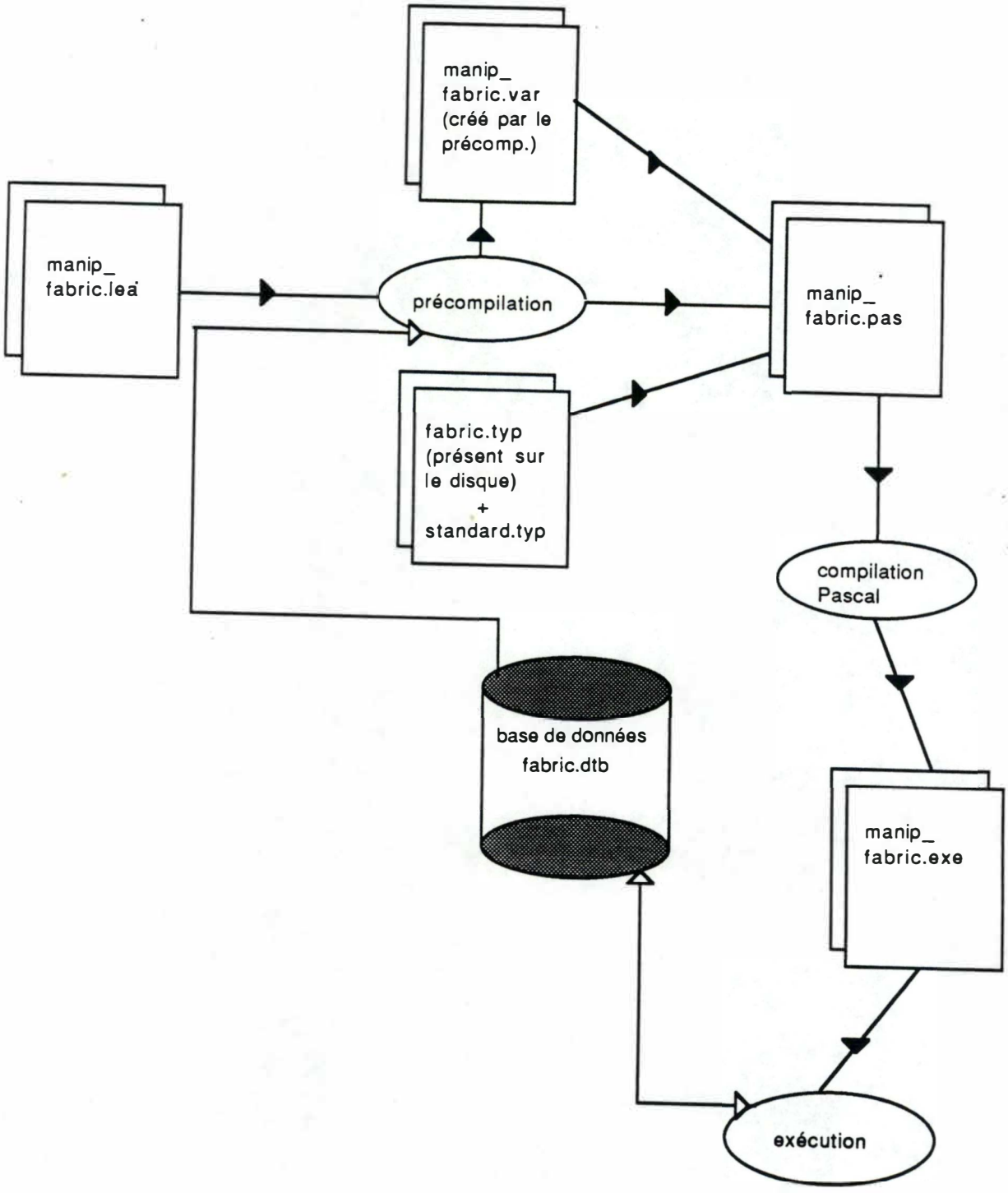
entre 2 fichiers:
inclusion du fichier
origine dans le
fichier destination

consultation de
la BD

entre un processus et
un fichier: production
ou utilisation du fichier
par le processus

mise-à-jour
de la BD

B) Manipulation d'une BD



4ème partie

Le précompilateur

Introduction

La précompilation est la phase de transformation d'un programme écrit en langage symbolique en un programme compilable. Ce procédé est souvent employé pour permettre l'extension de langages de haut niveau à l'aide de constructions syntaxiques n'appartenant pas au langage original, mais qui peuvent être traduites automatiquement en termes de celui-ci.

C'est ce qui est réalisé dans ce travail. En effet, au langage Pascal de départ a été ajouté un ensemble d'ordres LEA qui sont des constructions syntaxiques qui lui sont étrangères. Elles seront traduites en des termes Pascal (étendu par les primitives du SGBD Pyramide). Cela a donc mené à la réalisation d'un précompilateur LEA appelé PRECOMPI.

Ce qui sera décrit ici n'a pas pour but de détailler la conception du précompilateur, mais d'en donner les lignes essentielles aussi bien pour son utilisateur ("mode d'emploi") que pour celui qui serait chargé de son évolution (principes généraux de l'implémentation). On ne trouvera donc pas ici un "listing Pascal" commenté du précompilateur LEA.

IV.1. Architecture du précompilateur.

Cette étude de l'architecture du précompilateur se fera à deux niveaux :
- au niveau logique
- au niveau physique.

IV.1.1. L'architecture logique.

L'architecture logique [voir LAMS87] est constituée d'un ensemble de composants et de relations entre ceux-ci. Un composant ainsi que l'ensemble des relations qui le lient à d'autres constituent un module. Un module sera une unité de travail pour l'analyste, une unité d'affectation, de répartition du travail. Un module est donc différent d'une procédure ou d'un sous-programme. Les relations entre les composants peuvent être de différents types ("utilise", "précède", "déclenche",...).

Définir l'architecture logique consiste à structurer le système de manière hiérarchique, ç'est-à-dire l'organiser en niveaux différents et ordonnés, pour obtenir une structure maitrisable.

On a une structure hiérachisée si et seulement si il existe une et une seule relation R entre ses composants telle que:

niv_0 = { A | il n'existe pas de B : R(A,B) }

niv_i = { A | il existe B appartenant au niveau niv_{i-1} : R(A,B)
et [il existe C : R(A,C) => C est de niveau niv_{i-1} ou inférieur]}.

La hiérarchie choisie se fonde sur une relation "utilise" entre les composants. On dira que R(A,B), ç'est-à-dire "A utilise B " (A et B composants), si et seulement si la validité de A dépend de la disponibilité d'une version correcte du point de vue de sa spécification et de sa conception de B [LAMS87].

Le niveau 1 est composé des modules trt_err_synt et trt_err_sem, le niveau 2 des modules analyseur_syntaxique et analyseur_semantique, le niveau 3 du module selection, le niveau 4 du module assignation, le niveau 5 des modules transactions, uses, trt_err_lexicale, declar_var, open_close, delete, create, modify et for_endfor, le niveau 6 des modules MOTEUR, lire_ecrire_ligne et analyseur_lexical qui constituent un tout.

(voir schéma page suivante)

Remarque: Les flèches sont de types différents uniquement pour clarifier le schéma. Elles ont toutes la même signification, à savoir matérialiser une relation entre deux modules.

Il faut noter que tous ces modules "utilisent" Pascal et certains même Pyramide. Pascal et Pyramide seraient donc des modules de niveaux encore inférieurs. Ils sont absents du schéma pour des raisons de clarté.

La relation entre le module analyseur_lexical et les modules de niveaux inférieurs doit plutôt être vue comme une relation de type 'APPEL'.

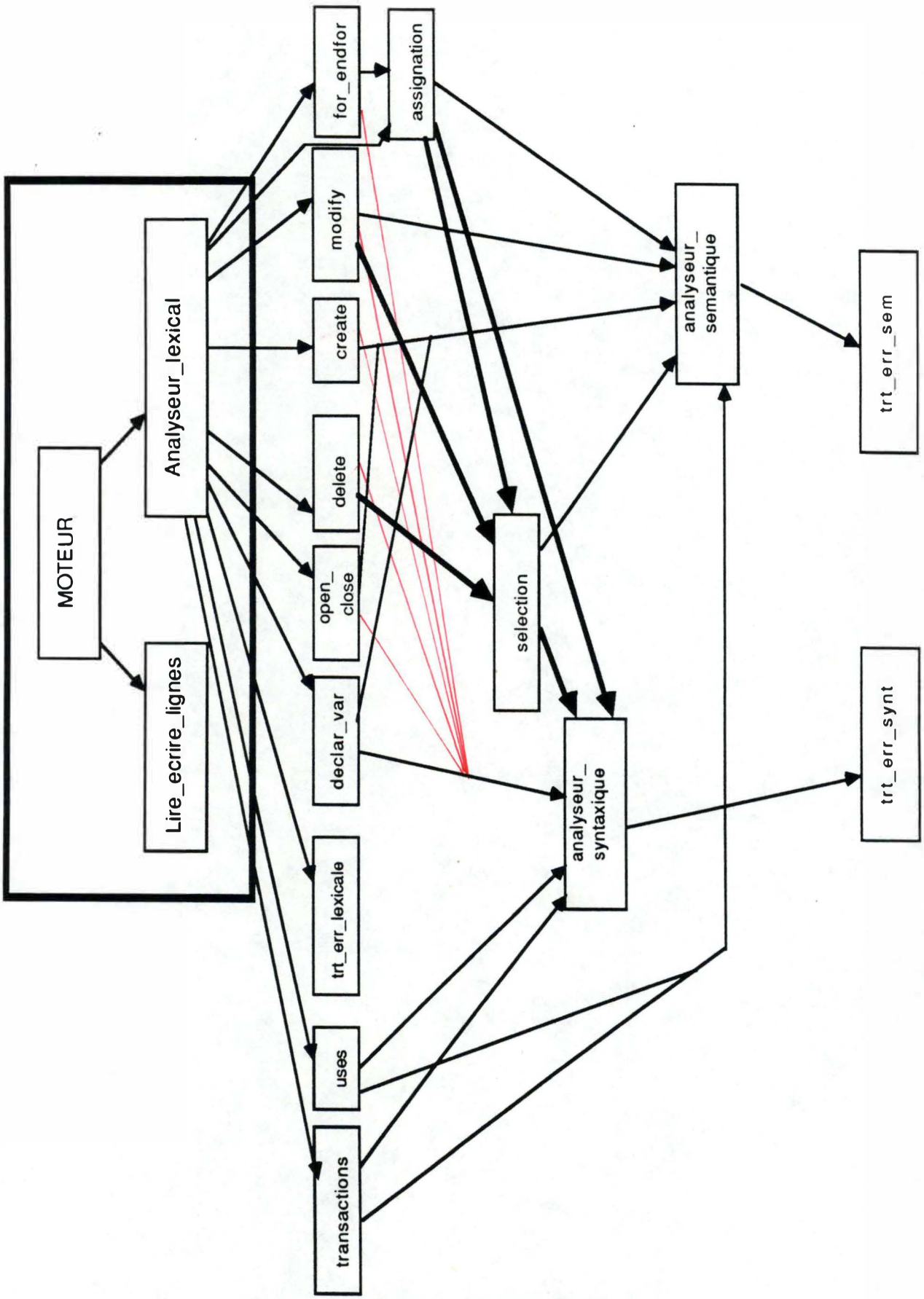


fig IV.1. : l'architecture logique.

IV.1.2. L'architecture physique.

L'architecture physique est constituée de modules qui sont des unités de travail pour la machine. Chaque module physique est donc une unité d'exécution, de compilation textuelle. Les relations possibles entre les modules physiques sont des relations de type "appel" ou "déclenche".

Un module logique ne provoquera pas nécessairement la conception d'un module physique correspondant. Un module logique peut en effet donner naissance à un ou plusieurs modules physiques. Un module physique peut résulter de un ou plusieurs modules logiques.

L'architecture physique est en réalité l'architecture du programme final (dans notre cas PRECOMPI). Elle diffère de l'architecture logique principalement au niveau des modules logiques d'analyse sémantique et syntaxique. Ceux-ci ont en effet pour des raisons de facilité d'implémentation été éclatés physiquement et se retrouvent dans les modules physiques declar_var, open_close,... qui traitent les différents ordres du langage ainsi que dans le module physique de sélection. Il faut noter également que le traitement des différents types d'erreur a été regroupé en un seul module physique.

Le programme PRECOMPI se présente de la manière suivante:

- il est éclaté en cinq fichiers distincts; un fichier principal (precompi.pas) et quatre fichiers secondaires (declarat.pas, trt_orer.pas, trt_dive.pas et trt_err.pas).
Une fois compilé, on obtient un fichier principal precompi.exe et quatre fichiers .TPU (qui sont des "unités" Pascal). Ces cinq fichiers finaux doivent être présents sur la disquette pour que le programme PRECOMPI soit complet.

- le fichier principal precompi.pas correspond aux modules logiques MOTEUR et ANALYSEUR LEXICAL. Ses fonctions sont de

+ sélectionner le programme à précompiler
+ parcourir ligne par ligne celui-ci
Si ce n'est pas un ordre LEA, il faut la recopier telle qu'elle dans le fichier final.
Sinon, il "appelera" le module (la procédure) correspondant à l'ordre LEA rencontré.
+ clôturer le programme.

- le fichier declarat.pas contient la description de toutes les constantes, de tous les types et de toutes les variables nécessaires au programme.

- le fichier trt_orer.pas correspond aux modules logiques qui traitent les ordres du langage ainsi que les modules d'analyse sémantique et syntaxique. Ses fonctions sont donc de

+ analyser syntaxiquement et sémantiquement les ordres LEA.

+ générer le code Pascal (étendu par Pyramide) correspondant dans le fichier final.

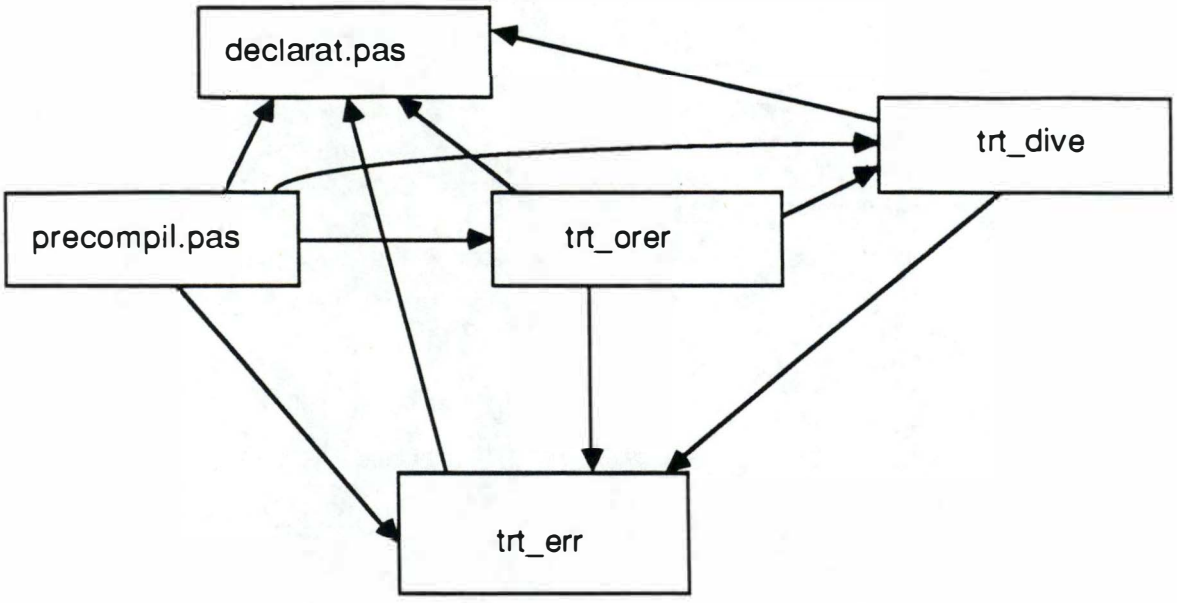
- le fichier trt_dive.pas constitue une bibliothèque de primitives qui sont nécessaires au fonctionnement du reste du programme. On y trouve principalement

+ les primitives qui gèrent le méta-schéma dans la mémoire centrale (voir ordre USES).
+ les primitives d'analyse syntaxique
+ les primitives d'analyse sémantique.

- le fichier trt_err.pas a pour seule fonction de produire à partir d'un code qui lui est fourni le message d'erreur correspondant et de clôturer le programme.

On peut représenter l'architecture du programme de la manière suivante :

Remarque: dire que A "utilise" B dans l'architecture physique signifie que B constitue une "unité" Pascal qui est "utilisée" par le programme ou l"unité" A.



A ——————▶ B signifie que A utilise ("uses") B

fig IV.2. : l'architecture physique.

IV.2. Fonctionnement du précompilateur

IV.2.1. Fichiers nécessaires.

Pour pouvoir utiliser le précompilateur LEA, appelé PRECOMPI, il faut que soient présents sur le disque et dans le même directory les fichiers suivants:

- PRECOMPI.exe
- declarat.tpu
- trt_orer.tpu
- trt_dive.tpu
- trt_err.tpu
- PYRAMIDE.tpu (primitives du SGBD Pyramide).

Mais il faut aussi:

- le fichier suffixé par ".LEA" que l'on veut précompiler (ex: cree_sch.lea)
- la base de données suffixée par ".DTB" sur laquelle on veut travailler (ex: garage.dtb)

- le fichier standard.typ qui sera inclus dans tout programme final et qui contient la déclaration des types permettant de manipuler le méta-schéma
- éventuellement (pour travailler sur un schéma déjà décrit) le fichier ".TYP" généré par l'utilitaire de Pyramide METACOMP (ex: garage.typ).

IV.2.2. Utilisation.

Pour exécuter le précompilateur, il suffit de lancer le programme PRECOMPI. Un écran de présentation apparaît alors. Il est demandé à l'utilisateur d'entrer le nom de son fichier de départ dans lequel est le programme qu'il veut précompiler.

Program name (without extension) :

Il doit entrer ce nom sans le suffixer (ex: cree_sch) Il sera automatiquement suffixé par LEA (ex: cree_sch.lea).

Si le fichier cité n'est pas présent dans le directory, le message "Unfound file-program cree_sch.lea" sera affiché et un nouveau nom de fichier sera demandé.

Pour sortir du programme, il suffit de donner un nom de fichier vide, ç'est-à-dire taper simplement un "return".

Si le fichier cité existe, le message "Found file-program cree_sch.lea" apparaît. Il est alors vérifié si le fichier résultat de la précompilation, dans lequel se trouvera le programme transformé, et qui aura pour nom le nom du fichier de départ suffixé par ".PAS" existe déjà ou non dans le directory (ex: cree_sch.pas).

Si le fichier résultat existe déjà, il sera alors demandé à l'utilisateur s'il veut oui ou non réécrire sur ce fichier.

File cree_sch.pas already exist. Overwrite (y/n) ?

Si la réponse est négative, la précompilation sera arrêtée (programme stoppé), sinon elle se poursuivra.

Si le fichier résultat n'existe pas, la précompilation se poursuit normalement.

La précompilation se fait jusqu'à ce que:

- le programme de départ ait été entièrement précompilé. On constate alors l'apparition d'un ou de deux fichiers supplémentaires sur le disque. Un fichier de nom du fichier de départ suffixé par ".PAS" (ex: crée_sch.pas) et un autre de nom du fichier de départ mais suffixé par ".VAR" (ex: cree_sch.var). Il sera inclus dans le programme final lors de sa compilation définitive. Il contient un ensemble de déclarations de types et de variables générés par le précompilateur et nécessaires au programme finel (voir IV.2.3.).

- une erreur soit détectée dans les ordres LEA (voir IV.2.3.).

IV.2.3. Les codes d'erreur.

Chaque fois qu'une erreur est détectée dans les ordres LEA analysés dans le programme de départ, qu'elle soit syntaxique ou sémantique, la précompilation est arrêtée.

Il est alors affiché à l'écran le numéro de la ligne et la ligne incriminée dans le programme de départ, ç'est-à-dire la ligne pendant l'analyse de laquelle le précompilateur a détecté l'erreur, ainsi qu'un message d'erreur explicatif.

Voici les différentes erreurs possibles à la précompilation et leur signification

- 1 : ordre du langage LEA inconnu.
- 2 : absence de l'ordre "USES" en début de programme.
- 3 : élément de la syntaxe d'un ordre manquant ou erroné.
- 4 : base de données de nom <dbname>.dtb inaccessible.
- 5 : schéma de nom <schname> introuvable dans la base de données spécifiée.
- 6 : création du fichier <file-name>.var impossible (erreur physique).
- 7 : incompatibilité entre ordre "USES" et ordre "OPEN" du point de vue du nom de la base de données <dbname>.
- 8 : incompatibilité entre ordre "USES" et ordre "OPEN" du point de vue du nom du schéma <schname>.
- 9 : ";", ",", ")", "(", ou "DO" absent dans l'ordre. Cela provoque une syntaxe incorrecte.
- 10 : type d'entité ou d'association n'existant pas dans le méta-schéma et/ou le schéma spécifié.

- 11 : variable existant déjà d'un autre type.
- 12 : variable non définie.
- 13 : rôle incorrect.
- 14 : "chemin" ("path") navigationnel incorrect.
- 15 : violation de la cohérence du méta-schéma ou du schéma possible.
- 16 : attribut inexistant.

IV.3. Implémentation

IV.3.1. Initialisation (ordre "USES")

Voici pour rappel la syntaxe de l'ordre "USES" :

$ USES DATABASE <dbname> SCHEMA <schname>;

Comme déjà signalé, l'ordre "USES" est un ordre qui sert uniquement au précompilateur. Il lui permet de connaître, dès le début du programme qu'il précompile, la base de données et éventuellement le schéma sur lequel le programme travaille. Dans la version actuelle, un programme ne peut travailler que sur une seule base de données et sur un seul schéma de celle-ci à la fois.

Sachant sur quelle base de données et quel schéma le programme travaille, le précompilateur pourra connaître les descriptions du méta-schéma et du schéma et ainsi vérifier la cohérence et la sémantique des ordres LEA présents dans le programme par rapport à ces descriptions.

La nécessité de l'ordre "USES" provient de ce que l'ordre LEA "OPEN DATABASE <dbname> [SCHEMA <schname>]" ne sera pas forcément le premier ordre LEA du programme. Il peut y avoir des déclarations de procédures et de fonctions le précédant et dans lesquelles on peut trouver des ordres LEA. Le précompilateur travaillant en une seule passe doit pourtant pouvoir analyser ces ordres LEA rencntrés avant le 'OPEN". Il lui est donc indispensable de savoir sur quelle base de données et sur quel schéma il devra se baser pour travailler dès le début de la précompilation.

L'ordre "USES" provoque la génération dans le fichier final de plusieurs lignes de code Pascal. Ce sont les lignes :

- "Uses Pyramide;" : qui permet d'inclure l'"unité" Pyramide dans le programme et donc d'autoriser l'accès à ces primitives.

- "(*$I standard.typ*)" : fichier qui contient la déclaration des types Pascal permettant de travailler dans le méta-schéma. Cette déclaration sera présente dans tout fichier final produit par PRECOMPI. Cela autorise, pour n'importe quel programme, la manipulation des méta-données.

- "(*$I <dbname>.typ*)" : fichier qui contient la déclaration des types Pascal permettant de travailler dans le schéma. Cette instruction ne sera présente que si un nom de schéma est présent dans l'instruction "USES".

- "(*$I <file-progr-name>.var*)" : fichier de déclaration de types et de variables généré par le précompilateur et inclus à la compilation finale. (voir IV.3.2).

Dans le souci d'améliorer les performances du précompilateur, la description du méta-schéma et éventuellement du schéma utilisateur (s'il y en a un de spécifié dans l'ordre "USES") sera stockée en mémoire centrale. L'analyse par le précompilateur de l'ordre "USES" provoquera donc le chargement en mémoire centrale de la description

- du méta-schéma stockée dans le méta-schéma de la base de données <dbname> avec une valeur de l'attribut NAME du type d'entité DBSCHEMA égale à "META-SCHEMA".

- du schéma de nom <schname> stockée dans le méta-schéma de la base de données <dbname> avec une valeur de l'attribut NAME du type d'entité DBSCHEMA égale à <schname>.

On dispose pour le stockage des descriptions d'une structure de données interne au précompilateur qui est plus ou moins dynamique (usage de pointeurs) pour pouvoir s'adapter à la "taille" des descriptions. Elle permet surtout de ne pas devoir décrire les informations deux fois. Par exemple, pour stocker un attribut, il ne faudra pas l'accompagner du nom du TE auquel il appartient. Le TE étant stocké ailleurs, il suffit de faire pointer le TE vers ses attributs. Cela permettra aussi d'avoir des accès relativement rapide.

Légende:

pointeur vers un élément de type B

fig. IV.3. : légende a.

signifie que A est un
tableau de pointeurs

fig. IV.4. : légende b.

le type EL est un record
composé des éléments A ,..., Z

fig. IV.5. : légende c.

Représentation de la structure:

Les types tent et trel sont constitués d'un tableau de pointeur vers des éléments ter.

Un type ter permet de représenter un type d'entité ou un type d'association du méta-schéma ou du schéma.

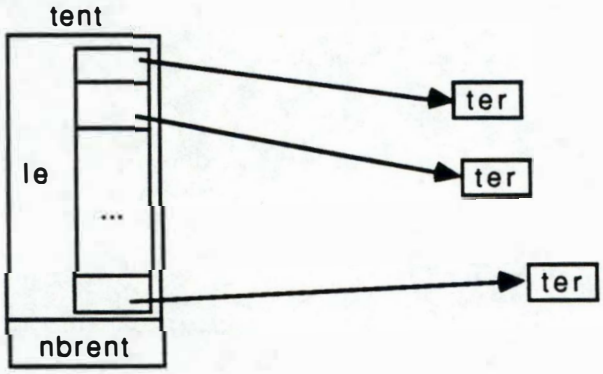


fig. IV.6.

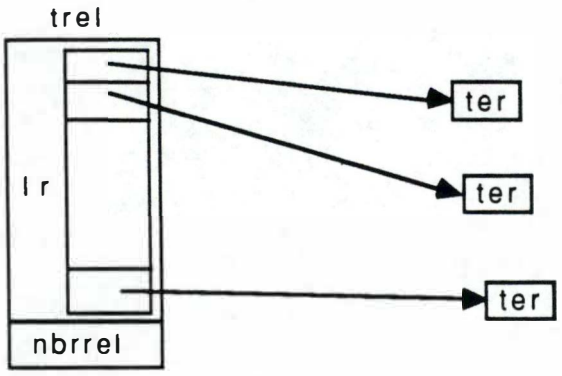


fig. IV.7.

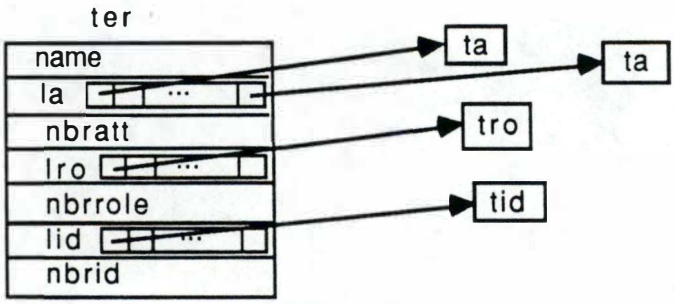


fig. IV.8.

Le type ta permet de représenter un attribut d'un TE ou d'un TA.

Le type tro permet de représenter un rôle et de le relier au TE qui le joue et au TA pour lequel il est joué.

Le type tid permet de représenter un identifiant d'un TE ou d'un TA et de le lier aux attributs et/ou rôles qui le composent.

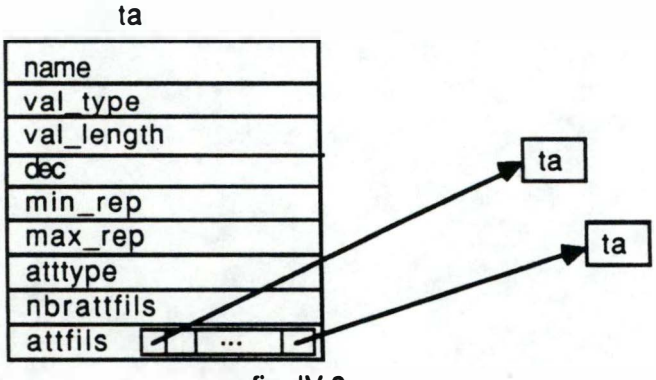


fig. IV.9.

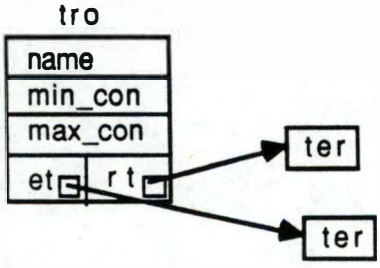


fig. IV.10.

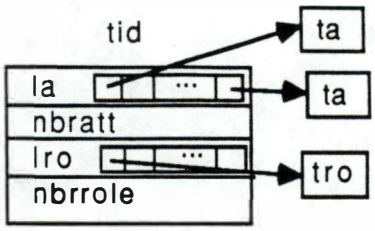


fig. IV.11.

Une extension possible et intéressante de l'ordre "USES" serait de permettre la spécification de plusieurs base de données et de plusieurs schémas. Cela permettrait alors dans un programme de transférer des données d'un schéma vers un autre, ce qui n'est pas possible actuellement. Cette extension est réalisable sans trop de difficultés. Il suffit principalement d'adapter la structure de données interne en ajoutant par exemple la structure suivante :

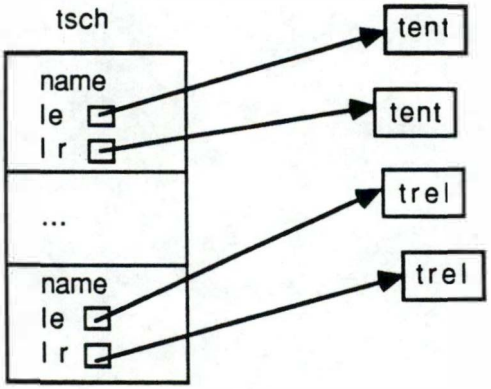


fig. IV.12.

Il est vrai que l'on peut également transformer la structure de données interne, pour gagner de la place mémoire et éviter le problème de dépassement de capacité des tableaux, en une structure totalement dynamique. Il suffit de remplacer les tableaux par des listes chaînées qui sont de taille variable.

Par exemple :

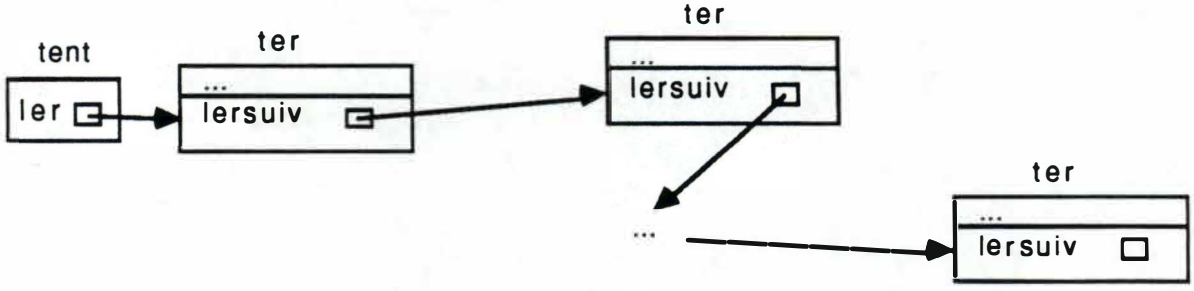


fig. IV.13.

IV.3.2. Déclaration de variables.

IV.3.2.1. Variables générées par le précompilateur.

La précompilation d'ordres LEA en des ordres Pascal et Pyramide peut nécessiter la définition de variables et de types Pascal supplémentaires à ceux définis par l'utilisateur dans son programme de départ.

Il suffit par exemple de considérer le cas d'une association de type N-N entre deux types d'entité (A et C) pour l'utilisateur. Elle sera transformée pour le précompilateur en trois types d'entité (A, B et C) avec deux types d'association de type 1-N. Un entre A et B et un entre C et B. L'accès de A vers C de l'utilisateur sera transformé par le précompilateur en un accès de A vers B, suivi d'un accès de B vers C. Cela nécessite une variable intermédiaire qui permettra de référencer B avant d'accéder à C.

La déclaration de cette variable sera générée par le précompilateur. Or toute déclaration de variable Pascal doit se faire au début du programme (ou de la procédure ou de la fonction) où la variable apparaît. Le précompilateur ne détectant le besoin de cette variable qu'au milieu d'un programme et ne travaillant qu'en une seule passe, il lui est difficile de réinsérer cette déclaration de variable au début du programme final sous peine de devoir recopier entièrement ce dernier. Il a donc fallu trouver une solution. Dans tout programme résultat d'une précompilation, on trouvera un ordre "(*$I <file-prog-name>.var*)" (voir IV.3.1.) permettant d'inclure le fichier <file-prog-name>.var à la compilation finale. Celui-ci sera garni, au fur et à mesure de la précompilation, des déclarations de variable nécessaires pour le programme final.

IV.3.2.2. Variables LEA

Comme il a déjà été signalé, il existe des variables de deux types particuliers en LEA: des variables d'entité et des variables d'association.

La déclaration d'une variable d'entité ne pose aucun problème au précompilateur. Sa traduction se fera de la manière suivante :

Soit une variable entité déclaré de la manière suivante

```
$ var cli: ENTITY client;
```

Elle sera traduite par

```
var cli : tclient;
```

On a donc que ENTITY <nom-type-entité> est traduit par T<nom-type-entité>. Ce type final se trouve déclaré dans le fichier ".TYP" généré par METACOMP et inclus dans tout programme Pascal final (voir IV.3.1.).

Pour une variable d'association, la traduction n'est pas aussi simple. En effet, il n'existe pas de type correspondant dans le fichier ".TYP". Ce sera donc au précompilateur a générer le type nécessaire à la traduction.
La déclaration d'une variable d'association,

```
$ var  <nom-var> : RELATION <nom-type-association>;
```

sera traduite de la manière suivante

```
$ var <nom-var> : TR <nom-type-association>;
```

Le précompilateur générera le type TR <nom-type-association> qui sera

```
TR <nom-type-association> = record
                            xxkey : dbref;
                            xxcode : char;
                            <nom-att1> : <type-att1>;
                            ...
                            <nom-attn> : <type-attn>;
                            R<role1> : T<nom-ent1>;
                            ...
                            R<rolem> : T<nom-entm>;
                                  end;
```

où $<role_1>$ représente le rôle joué par le type d'entité $<nom\text{-}ent_1>$ dans le type d'association.

L'ensemble des types générés par le précompilateur se retouve pour les mêmes raisons que pour les variables dans le fichier ".VAR"

Remarque: il faut noter que d'un point de vue pratique les déclarations des types sont stockées pendant la précompilation dans un fichier ".TTP". A la fin de celle-ci, les fichiers ".VAR" et ".TTP" sont fusionnés en un seul fichier ".VAR" qui contient, dans l'ordre, la déclaration des types et la déclaration des variables générées par le précompilateur.

Il n'est pas permis de créer plusieurs fois le même type TR dans un programme. Le précompilateur garde donc en mémoire la liste des types TR qu'il a déjà créé. Avant d'en créer un nouveau, il peut donc vérifier s'il n'existe pas encore.

Voici la structure de données interne qui conserve la liste des types. C'est une liste chaînée.

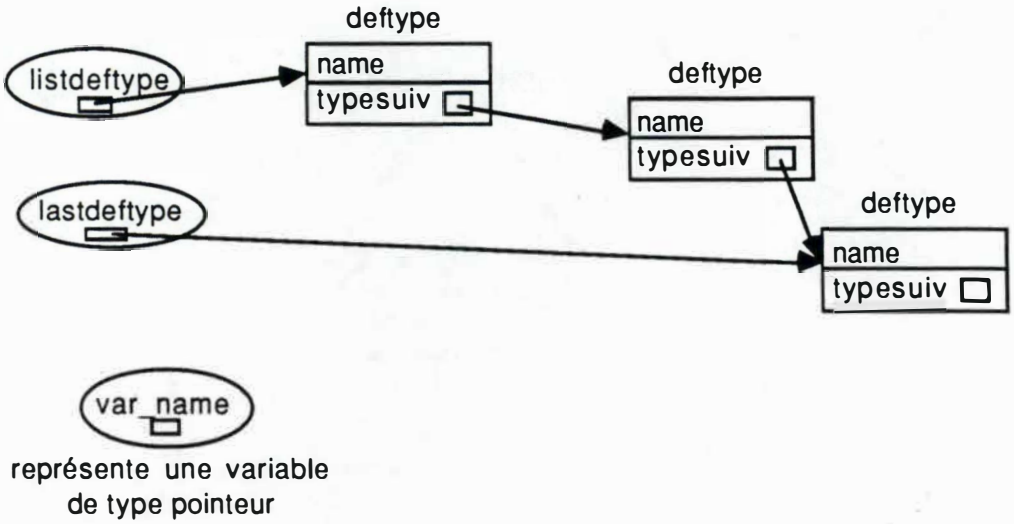


fig. IV.14.

Le précompilateur conserve également la liste de toutes les variables déclarées avec leur type. La structure de données interne est une liste chaînée.

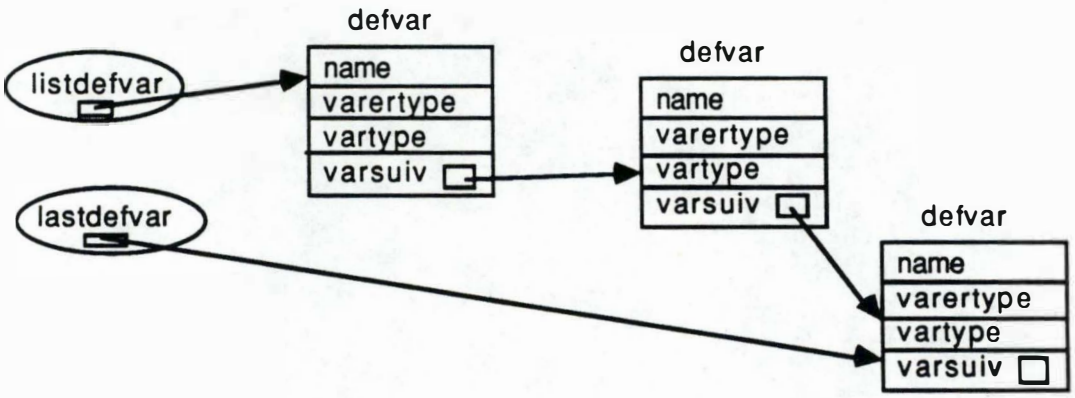


fig. IV.15.

Cette structure permet au précompilateur de
- vérifier qu'une variable n'est pas déclarée dans le programme de deux types différents
- d'assurer la compatibilité des types lors d'une assignation.
Par exemple : $ cli:= client WITH name = 'DUPONT';
Le précompilateur peut vérifier que cli est bien du type client.
- d'assurer la compatibilité des types à l'intérieur d'un ordre.
Par exemple : $ create client cli;
Le précompilateur peut vérifier que cli est bien du type client.

IV.3.3. Gestion des transactions

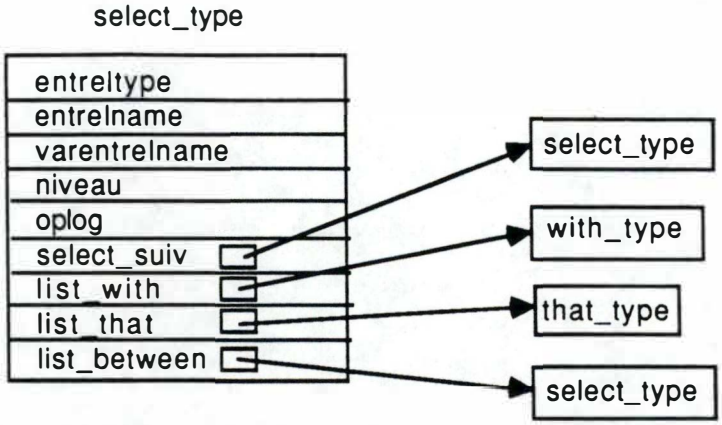
Ces ordres sont implémentés dans le précompilateur, dans le sens où celui-ci produit l'ordre Pyramide correspondant. Cependant les ordres Pyramide résultant (dbbgtr, dbendtr, dbabtr) ne sont pas implémentés. Le programme final ne pourra donc pas être actuellement compilé définitivement. Il ne faut donc pas utiliser ces ordres LEA dans cette version.

IV.3.4. Sélection

La sélection se trouve dans les ordres LEA suivants :
- assignation (ex: cli := client WITH name = 'DUPONT';)
- boucle d'accès (ex: FOR cli := client DO ... ENDFOR;)
- suppression (ex: DELETE client WITH name = 'DUPONT';)
- modification (ex: MODIFY client WITH name = 'DUPONT' USING name = 'DUPOND';)

D'une manière générale, on peut résumer le principe de la transformation d'un ordre LEA de la manière suivante:
1) analyse syntaxique
2) analyse sémantique (en parallèle avec 1)
3) génération du code final résultant de la transformation

Tous les ordres cités plus haut étant tous composés d'une partie sélection, nous avons choisi de regrouper l'analyse syntaxique et l'analyse sémantqiue de leur partie sélection dans une seule primitive. Elle vérifie la syntaxe et la sémantique de la partie sélection des ordres et garnit une structure de données interne au précompilateur qui représente une découpe de la partie sélection. C'est à partir de cette découpe et en fonction de l'ordre LEA à traduire que le précompilateur générera le code final.



entreltype: vaut E ou R et indique si le select_type représente un TE ou un TA.
entrelname: nom du TE ou du TA.
varentrelname: nom de la variable associée au TE ou au TA dans l'ordre LEA.
niveau: indique le niveau de parenthèses du select_type.
oplog: représente l'opérateur logique qui suit le select_type.
select_suiv: pointeur vers le select_suiv lié par le connecteur logique.
list_with: pointeur vers la liste des éléments du WITH.
list_that: pointeur vers la liste des éléments qui suivent le THAT.
list_between: pointeur vers la liste des éléments qui suivent le BETWEEN.

fig. IV.16.

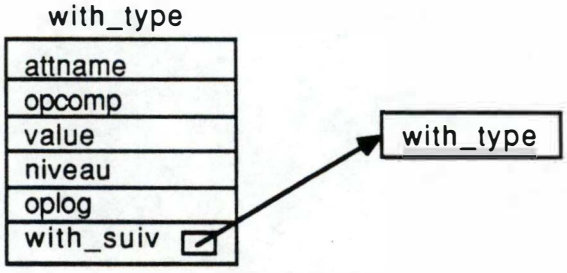
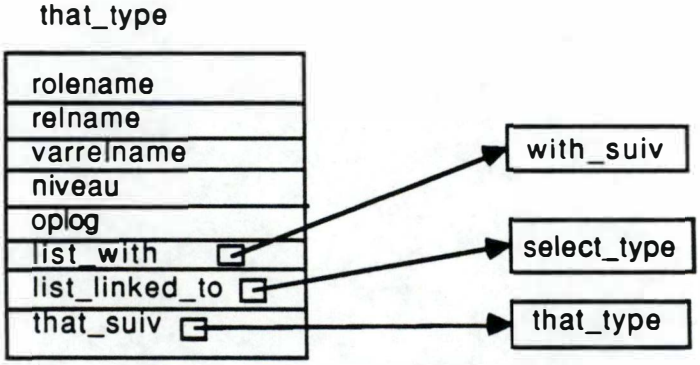


fig. IV.17.



list_linked_to: pointeur vers la liste des éléments qui suivent le LINKED_TO.

fig. IV.18.

Remarque: - Etant donné que la gestion des transactions n'est pas implémentée par Pyramide, on ne peut pas en faire usage dans le code généré dans le programme final. Cela signifie qu'il n'est pas possible de grouper l'ensemble des ordres correspondant à un ordre LEA en une transaction. On ne peut donc pas garantir la durabilité des traitements effectués et la cohérence de la base de données. On ne peut donc pas garantir non plus qu'un ordre LEA soit exécuté comme un tout ou pas du tout.

- L'ordre DELETE nécessite une analyse supplémentaire à celle réalisée dans le cadre de l'analyse de la sélection. Il faut en effet, pour pouvoir traduire cet ordre, rechercher dans la description du méta-schéma ou du schéma l'ensemble des entités et des associations qui doivent également être effacées pour garantir la cohérence de la base de données (respect des connectivités minimales).

Exemple : Un type d'association ACHAT est défini entre un type d'entité CLIENT qui joue le rôle ACHETE et à des connectivités (1,N) et le type d'entité VOITURE qui joue le rôle est_achete_par et à des connectivités (1,1).
L'ordre DELETE client WITH name = 'DUPONT' devra provoquer la suppression de:
- l'entité client dont l'attribut name vaut 'DUPONT'
- les associations achat qui lient le client 'DUPONT' à des voitures
- les entités voiture liées par achat au client 'DUPONT'.

IV.3.5. Création.

On trouve dans l'ordre LEA de création une partie se rapprochant fortement de la partie "sélection" des autres ordres LEA. Cette partie de l'ordre CREATE sera donc analysée par la même primitive que pour les autres ordres LEA. La génération du code final sera différente.

Il faut lors d'une création vérifier que l'on crée, dans cet ordre, tout ce qui est nécessaire pour assurer la cohérence de la base de données.

Exemple (voir exemple de la sélection en IV.3.4.):
L'ordre $ CREATE client WITH name = 'DUPONT' est incomplet. L'ordre correct serait:

```
$ CREATE client WITH name = 'DUPONT'
                    THAT achete LINKED_TO  voiture voit
                                        WITH (numero_plaque = 'AAB124');
```

Il faut faire attention au fait que dans un ordre CREATE on peut à la fois référencer des éléments qui existent déjà (ils seront obligatoirement référencés par des variables contenant leurs références physiques) et avoir des éléments qui n'existent pas encore. Il suffira de produire le code qui teste si une variable contient une référence d'un élément ou pas. Dans la négative, il faudra créer cet élément dans la base de données.

5ème partie

L'extension dynamique d'un schéma

V. L'extension Dynamique d'un Schéma

V.1. Définition du problème

Dynamisme signifierait, dans le cas présent, pouvoir dans le même programme travailler à la fois sur le schéma des données (le créer, le modifier) et sur les données elles-mêmes (les créer et les modifier).

Par exemple: -soit un type d'entité client avec les attributs nom et prénom,

-on a des occurrences pour ce type d'entité dont les groupes de valeurs sont ("Dupont", "Marcel") et ("Dardenne", "Jules").

Dans un même programme, le dynamisme permettrait d'ajouter un attribut adresse au type d'entité client tout en supprimant l'attribut prénom de ce type d'entité. Tout cela se ferait en conservant l'intégralité des occurrences de ce type d'entité. Elles seraient adaptées au changement intervenu dans le schéma des données (le type d'entité client est modifié) dans le sens où toutes les valeurs de ces occurrences correspondant à l'attribut prénom seront supprimées.

Supposons que l'on modifie les occurrences en donnant une adresse à Dupont et à Dardenne. On aurait dans le type d'entité client des occurrences dont les groupes de valeurs seraient ("Dupont","5, rue des Fleurs, 5000 Namur") et ("Dardenne"," 13 rue des Rys, 1000 Bruxelles").

V.2. Etat actuel et problèmes

Dans l'état actuel du système LEA, un tel dynamisme n'est pas possible et ce pour plusieurs raisons:

1) modifier un schéma de données implique une modification de la base de données physique correspondante.

Actuellement, pour modifier la base de données physique, il faut employer un utilitaire du SGBD (METACOMP) qui, à partir de la description d'un schéma de données conforme au modèle EA réduit dans le méta-schéma produit une base de données physique correspondante. Cette base de données est cependant vide des données de l'application. Toutes les anciennes données ont donc été perdues. Il faut les réintroduire. Il faut noter que l'ancienne base est conservée telle qu'elle était dans un autre fichier suffixé par ".odb". On peut alors transférer les données d'un fichier (d'une base de données) vers l'autre en les adaptant. Ce travail est à charge de l'utilisateur.

2) pour employer cet utilitaire il faut actuellement sortir du programme d'application, car il n'est pas possible de lancer METACOMP à partir d'un programme. On ne peut donc pas modifier un schéma définissant des données et travailler sur les données décrites dans le même programme.

3) à chaque base de données physique produite par l'utilitaire du SGBD est associé un fichier de types PASCAL ("<database_name>.typ"). Il est généré par le métacompilateur lors de la compilation d'un schéma de données. Ces types Pascal résultent d'un mapping des structures de données du SGBD PYRAMIDE. Pour pouvoir utiliser les primitives de Pyramide, ce fichier doit être inclus dans le programme Pascal avant la compilation finale. C'est encore une raison qui impose la séparation de la modification d'un schéma de données et du travail sur celles-ci en deux programmes distincts. Le deuxième programme ne pourra être compilé définitivement que quand le premier (modification de schéma) aura été exécuté ainsi que l'utilitaire METACOMP.

En effet, le premier programme modifie pendant son exécution la description du schéma de données. Une fois cette exécution terminée, on lance l'utilitaire METACOMP qui produit la base de données vide des données de l'application et le fichier des types Pascal correspondant. C'est seulement alors qu'il peut être inclus dans le programme qui travaille sur les données pour le compiler définitivement.

Le fonctionnement actuel du système à déjà été expliqué en III.7.

V.3 Fonctionnement souhaité

Voici le fonctionnement du système que l'on voudrait voir possible (fonctionnement dynamique) :

1) l'utilisateur écrit un programme contenant des ordres LEA de description ou de modification d'un schéma de données et en même temps qui travaille sur les données de ce schéma (création, suppression, modification, consultation).

2) l'utilisateur précompile son programme d'application, le précompilateur produit un programme Pascal conforme à Turbo-Pascal. C'est le programme de départ dans lequel les ordres propres au langage LEA ont été remplacés par une suite d'appels à des primitives de Pyramide et d'instructions standards Pascal.

3) l'utilisateur compile ce dernier programme Pascal obtenu. On a un programme exécutable.

4) l'exécution de ce programme a pour résultat

- de créer ou de modifier la description du schéma dans le méta-schéma et donc aussi la base de données physique correspondante, sans perdre de données et en les adaptant si nécessaire au nouveau schéma.

- de créer, modifier, supprimer des données dans le nouveau schéma. Il est à remarquer que la structure définitive par exemple d'un type d'entité (l'ensemble de ses attributs et de leurs types) n'est pas forcément connue à la compilation vu que le type d'entité peut être modifié à l'exécution.

V.4. Solutions

Il est évident que les solutions devront être apportées au niveau de PYRAMIDE et de la couche supérieure. En effet, il est impossible pour la couche supérieure d'assurer le dynamisme si la couche inférieure (PYRAMIDE) n'est pas dynamique. Nous allons donc aborder les solutions à apporter aux deux niveaux du système LEA. Nous ne dégagerons pas clairement les deux couches car nous considérons le système LEA comme un tout. PYRAMIDE n'est considéré qu'en tant que composante du système final, et non comme un outil indépendant.

V.4.1. L'utilitaire METACOMP

Il faudrait tout d'abord que cet utilitaire soit appelable directement dans un programme Pascal (comme une primitive), ç'est-à-dire que lors de l'exécution d'un programme Pascal on puisse demander l'exécution de METACOMP sur un schéma de données inclus dans le méta-schéma. L'utilitaire modifierait la base de données physique correspondant à ce schéma de données en fonction des modifications qui lui ont été apportés (voir fonctionnement actuel du système). Mais il faudrait aussi que l'utilitaire conserve les données incluses dans la base de données avant la modification de son schéma. Il devrait pouvoir les

adapter (Par exemple: suppression d'un attribut, changement de son type,...). La solution à ces problèmes pourrait être la suivante:

- mettre l'utilitaire METACOMP dans une unit Pascal qu'il suffirait d'inclure au début du programme

- étant donné que METACOMP conserve l'ancienne base de données dans un fichier ".ODB" lors de sa modification, il suffirait de reprendre les données dans celui-ci et de les transférer vers le nouveau fichier ".DTB" en les adaptant. C'est justement cette adaptation des données qui risque de poser bien des problèmes.

V.4.2. Les identifiants physiques de PYRAMIDE

Comme on l'a vu, dans la base de données, il existe des types d'entité et des types d'association de type 1-N entre ceux-ci. Ces différents types d'entité et d'association sont identifiés au niveau physique de la base de données. Dans le cas de Pyramide, ces identifiants sont des entiers. Les types d'entité et d'association sont identifiés d'une manière distincte.

Par exemple: le type d'entité CLIENT sera identifié au niveau physique par l'entier 1 et VOITURE par l'entier 2.
le type d'association ACHAT sera lui identifié par l'entier 1.

Ces identifiants sont cachés dans le programme Pascal derrière des constantes prédéfinies au début du programme (incluses dans le fichier ".TYP" générés par METACOMP).

Lors de l'utilisation d'une primitive de PYRAMIDE, il faut utiliser ces constantes pour désigner le(s) type(s) d'entité et/ou d'association auquel appartiennent les objets manipulés.

Par exemple:

```
...
const client =1;
      voiture =2;
      achat = 1;
...
var   cli : tclient;
...
dbfirst  (client,cli);
```

V. L'extension Dynamique d'un Schéma

Le problème, au point de vue du dynamisme, est que les identifiants physiques des différents types d'entité et d'association ne seront pas forcément connus avant la compilation. Ils peuvent, en effet, être définis au cours de l'exécution du programme. L'identifiant physique ne sera seulement connu que lors de la création du type à l'exécution. Il n'est donc plus possible de définir une constante au début du programme, comme cela était fait avant.

Par exemple: soit l'ordre LEA suivant:

```
$ CREATE client cli WITH name = 'DUPONT';
```

Il peut être traduit en des ordres Pascal (étendu par Pyramide) par

```
cli.name := 'DUPONT';
dbcreate  (client,cli);
```

CLIENT est ici une constante numérique représentant l'identifiant physique du type d'entité. Avec le dynamisme, la traduction devrait être:

```
cli.name := 'DUPONT';
dbcreate  (1,cli);
```

En effet, les identifiants physiques n'étant pas toujours connus à la compilation, on ne peut plus définir des constantes qui les représentent. Une solution peut consister à utiliser une variable de type entier qui prendra la bonne valeur à l'exécution.

```
typeentite := 1;
cli.name := 'DUPONT';
dbcreate  (typeentite,cli);
```

Le problème est alors d'obtenir la valeur de cet identifiant physique à l'exécution.

Une solution consiste à ajouter dans le méta-schéma un attribut IDTE de type entier au type d'entité ENTITY_TYPE dans lequel le métacompilateur placerait, après la création physique d'un type d'entité, la valeur de son identifiant physique. Elle devrait être placée dans le schéma entité-association réduit correspondant au schéma EAC de l'utilisateur.

On aurait alors:

```
...
cli.name := 'DUPONT';
sch.name := 'GARAGE';
dbid (dbschema,sch);
(* les méta-types n'étant pas modifiables dynamiquement, leur identifiant
physique peut faire l'objet d'une déclaration de constante au début du
programme *)
dbfpath (ent, sch, dbschema_et);
while (dbfound) and (ent.name<>'CLIENT') do
      dbnpath (ent, sch, dbschema_et);
if dbfound then typeentite := ent.idte;
dbcreate (typeentite,cli);
...
```

Le principe est identique pour les types d'association. On ajoute au méta-schéma un attribut IDTA dans le type d'entité REL_TYPE qui sera garni, comme pour IDTE, par METACOMP.

On aurait dans le programme final:

```
sch.name := 'GARAGE';
dbid (dbschema,sch);
dbfpath (rel, sch, dbschema_rt);
while (dbfound) and (rel.name<>'ACHAT') do
      dbnpath (rel, sch, dbschema_rt);
if dbfound then typeassociation := rel.idta;
dbfpath (voit, cli, typeassociation);
```

Cette solution apporte un changement au niveau du métacompilateur METACOMP qui doit "garnir" les attributs IDTE et IDTA, mais également au niveau du précompilateur. Celui-ci doit en effet générer le code Pascal et Pyramide pour lire la valeur de l'identifiant physique et l'assigner à la variable typeentite ou typeassociation. Ces deux variables étant des variables générées par le précompilateur, on trouvera leur déclaration dans le fichier ".VAR" produit par celui-ci.

En résumé, la solution jusqu'à présent consiste à:

_ pour l'utilisateur, faire précéder tout ordre manipulant les données d'un appel au métacompilateur, si cet ordre suit des ordres de manipulation du méta-schéma. Le métacompilateur met à jour les attributs IDTE et IDTA des nouveaux objets créés, et se charge de modifier la base de données conformément à la nouvelle description contenue dans le méta-schéma.

- pour le précompilateur, à générer le code nécessaire pour obtenir à l'exécution l'identifiant physique d'un TE ou d'un TA (voir ci-dessus).

V.4.3. Les types dynamiques

Au niveau du langage LEA, il est possible de définir des variables d'entité ou d'association.

Par exemple: $ VAR cli : ENTITY client;

La variable cli est une variable d'entité de type client. Le type client est dérivé directement de la description du type d'entité CLIENT dans le méta-schéma, plus précisément de l'ensemble des attributs de ce type d'entité et de leurs caractéristiques.

Ainsi, soit le TE CLIENT avec les attributs NOM = S(30) et PRENOM = S(30)

Il lui correspond donc un type de données client qui permet de déclarer des variables pour manipuler des occurrences du type d'entité CLIENT:

```
type client = record
                NAME : string [30];
                PRENOM : string [30];
              end;
```

En pratique, le type client est dérivé directement du fichier des types Pascal générés par METACOMP (fichier .TYP). On a dans celui-ci le type tclient qui a la structure suivante:

```
type tclient = record
                 xxref : dbkey;
                 xxcode : char;
                 NAME : string [30];
                 PRENOM : string [30];
               end;
```

Les attributs xxref et xxcode servent uniquement aux primitives du SGBD Pyramide et au précompilateur mais pas directement aux ordres LEA. Le programmeur peut donc ignorer leur présence.

Le processus de traduction de la déclaration d'une variable d'entité est donc:

```
$ VAR cli : ENTITY client est traduit par
   VAR cli : tclient;
```

Pour les variables d'association, la transformation est un peu plus complexe. En effet, le fichier des types Pascal .TYP ne contient pas nécessairement la déclaration d'un type associé à un TA. On trouvera un type associé à un TA seulement si celui-ci a été transformé en un TE dans le schéma entité-association réduit. De plus, au niveau du système LEA, les types de variables d'association contiennent des informations supplémentaires sur les entités liées.

Soit le type d'association ACHAT, contenant un attribut , entre CLIENT et VOITURE.

Ce TA étant transformé en un TE suite au processus de transformation de schémas, il lui correspondra dans le fichier .TYP le type de données suivant:

```
type tachat = record
                xxref : dbkey;
                xxcode : char;
                date : string[6];
              end;
```

Si le TA n'est pas transformé en un TE, il ne lui correspond aucun type de données dans le fichier .TYP. En effet, le METACOMP ne génère des types de données que pour les TE.

V. L'extension Dynamique d'un Schéma

La déclaration d'une variable d'association suivante, au niveau du langage LEA:

```
$ VAR ach : RELATION achat;
```

sera traduite par le précompilateur en:

```
VAR ach : TRachat;
```

Le type TRachat aura la forme suivante:

```
type TRachat = record
                 xxref : dbkey;
                 xxcode : char;
                  date : string[6];
                  Rachete : tclient;
                   Rest_achetée_par : tvoiture;
               end;
```

Ce type sera généré par le précompilateur. Il se trouvera dans le fichier .VAR. La première partie de ce type est dérivée directement du type tachat déclaré dans le fichier .TYP. Lorsqu'il ne correspond pas de type de données pour un TA dans le fichier .TYP, la première partie du type TR (éléments xxref, xxcode et les attributs du TA) est absente.

Le précompilateur doit donc se charger d'assurer la correspondance entre les manipulations de variables de types TR... par l'utilisateur, et l'appel à des primitives PYRAMIDE qui, elles, utilisent des variables des types T... Le problème est que ces processus de traduction supposent que les types PYRAMIDE existent avant la précompilation du programme d'application travaillant sur les données. Si le programme permet de modifier le contenu du méta-schéma durant son exécution, il est impossible de disposer avant la précompilation des types de données correspondants à des TE ou TA créés pendant l'exécution. Ceux-ci ne sont pas accessibles, même si créés lors de l'appel au métacompilateur, à moins de recompiler le programme en incluant le nouveau fichier .TYP. Cela serait absurde puisque le programme est en phase d'exécution.

Comme tout TE est susceptible d'être modifié dynamiquement, une solution consiste à définir un type de données standard, valable pour stocker les occurrences de n'importe quel TE, et ce indépendamment de sa structure. Ce type est appelé type d'entité

dynamique (TED). De même, pour les TA, on aura un type standard appelé type d'association dynamique (TAD).

Outre les éléments propres à PYRAMIDE, ce type ne contient qu'un seul élément capable de stocker les valeurs de tous les attributs du TE qu'il référence. La structure de ce type est:

```
type TED = record
              xxref : dbkey;
              xxcode : char;
               att : string[255];
           end;
```

La traduction de la déclaration d'une variable d'entité:

$ VAR cli: ENTITY client;

sera:

VAR cli : TED; (au lieu de cli : tclient)

L'élément att reprend les attributs NOM et PRENOM du type d'entité CLIENT. On avait avant cli.name = 'DUPONT' et cli.prenom = 'MARCEL'. On aura après cli.att = 'DUPONT MARCEL';

Pour les types d'association, on a un élément att qui reprend l'ensemble des attributs du type et un élément role, qui reprend l'ensemble des rôles des TE liés au TA:

```
type TAD = record
             xxref : dbkey;
             xxcode : char;
               att : string [255];
               role : array [1..maxrole] of record
                                              rolename : string[32];
                                            roletype : TED;
                                              end;
             end;
```

Ces types pourraient être générés par METACOMP dans le fichier .TYP, ils sont de toute façon défini une fois pour toutes.

L'utilisation de types de données standards pour manipuler les occurrences de tous les TE et TA ne va cependant pas sans poser quelques problèmes. En effet, il n'est plus possible de référencer directement lors de l'utilisation d'une variable un nom d'attribut, puisque celui-ci n'est plus défini.

Il faudra donc rétablir dynamiquement la concordance entre l'élément att unique et les différents attributs du TE ou du TA. On aura par exemple pour le type d'entité CLIENT la valeur de l'attribut NAME entre la 1ère et la 30ème position de ATT et pour l'attribut PRENOM entre la 31ème et la 60ème.

Pour décomposer dynamiquement cet élément, il faut donc savoir pour chaque attribut la position correspondante dans l'élément ATT. Il suffit de connaître la première position dans ATT et la longueur de l'attribut. On ajoutera donc un attribut RANG au type d'entité ATTRIBUTE dans le méta-schéma. On y mettra la valeur de la première position de l'attribut dans l'élément ATT. Elle sera garnie à l'exécution par le métacompilateur lors de la création ou de la modification d'un attribut dans le méta-schéma.

Lorsque l'utilisateur manipulera des valeurs d'attributs par l'intermédiaire d'une variable, le précompilateur devra donc générer le code afin d'assurer la concordance entre l'élément att, et les attributs stockés dans le méta-schéma. Pour cela, le langage LEA sera augmenté de deux ordres: un pour obtenir une valeur d'un attribut, un pour transmettre une valeur à un attribut.

D'abord, pour obtenir la valeur d'un attribut:

On pouvait avant déclarer cli:

```
$ VAR cli: ENTITY client;
```

Si cli référencait un client dans la base de données, il suffisait de faire WRITE (cli.name) pour afficher la valeur de son attribut NAME.

Avec la nouvelle solution pour les types, cela n'est plus possible. Faire WRITE (cli.att) affiche la valeur de tous les attributs à la suite les uns des autres. On doit donc ajouter un ordre au langage LEA qui permette d'accéder à un attribut d'un TE ou d'un TA:

Syntaxe: $ <var_name>:=<att_name> OF <entity_type_name> <var_name_TED>;

<var_name> est de type string

<var_name_TED> est de type TED

<att_name> est le nom de l'attribut dont on veut la valeur

Par exemple: $ name_cli := name OF client cli;

Cet ordre serait traduit par une fonction Pascal GIVEATT qui renverrait comme résultat dans la variable assignée un string (même pour les entiers, les booléens,...) qui correspondrait à la valeur de l'attribut dans la variable de type TED considérée, et qui référence une entité. On trouverait la déclaration de cette fonction au début de chaque programme précompilé.

<var_name>:= GIVEATT(<att_name>,<entity_type_name>,<var_name_TED>);

La traduction de l'ordre LEA précedant serait donc:

name_cli := GIVEATT (name, client, cli);

Voici le principe de fonctionnement de la fonction Pascal GIVEATT:

- accéder au bon type d'entité ou d'association dans le méta-schéma,
- accéder au bon attribut. On en obtient la longueur et la position dans l'élément ATT, en consultant les attributs val_length, et rang du type d'entité ATTRIBUTE dans le méta-schéma,
-transformer la valeur dans TED en une valeur en caractères et la transférer.

Pour transmettre une valeur à un attribut:

Il n'est plus possible avec le dynamisme d'effectuer l'opération suivante:

cli.name:= 'DUPONT';
$ CREATE client cli;

Si l'on veut rendre cela possible, il faut ajouter un ordre au langage LEA qui permette de donner une valeur à un attribut. Cet ordre aurait la forme:

$ <att_name> OF <entity_type_name> <var_name>:= <value>;

Cet ordre serait traduit par une procédure PASCAL qui aurait la forme suivante:

CREATEATT (<att_name>, <entity_type_name>, <var_name>, <value>);

Cette procédure va donc se charger de donner à l'attribut <att_name> du type d'entité <entity_type_name> référencé par la variable du type TED <var_name> la valeur <value>.

Cette procédure sera déclarée au début de tout programme précompilé.

Par exemple: CREATEATT (name, client,cli,'DUPONT');

En conclusion, pour tout ordre du langage LEA, deux cas sont possibles:

- soit l'utilisateur référence un attribut par son nom. Dans ce cas, le précompilateur peut effectuer les transformations nécessaires et produire le code pour stocker ou accéder aux bonnes valeurs aux bons endroits.

- soit il utilise des variables. Dans ce cas, il doit d'abord utiliser les deux ordres du langage définis ci-dessus afin de les garnir conformément à la structure décrite dans le méta-schéma (position correcte des différents attributs dans l'élément att des variables).

Cas des variables d'association:

Dans une variable d'association, on dispose de renseignements concernant les entités liées par l'association référencée par la variable.

Par exemple:

var ach : RELATION achat;
(entre client et voiture)

On aura ach.Rachete qui contient tous les renseignements sur l'entité client et ach.Rest_achete_par qui contient tous les renseignements sur l'entité voiture.

Pour les types dynamiques, cela sera légèrement différent. On ne peut pas connaître le nom des rôles toujours avant l'exécution. Le principe doit donc être modifié.

On aura:

```
Role: array [1..maxrole] of record
                              rolename : string[32];
                              roletype : TED;
                            end;
```

Quand on sélectionnera une association par un ordre LEA et qu'elle sera affectée à une variable association, il faudra garnir celle-ci convenablement.

Sans le dynamisme, il suffisait au programmeur pour obtenir d'une variable <var_name> les renseignements concernant une entité liée y jouant un rôle précis <role_name> (par exemple: le client qui achète), de spécifier <var_name>.R<role_name> pour les obtenir.

Avec le dynamisme, le type de données associé à un TA est différent. Il faudra pour obtenir ces mêmes renseignements réaliser la série d'opérations suivantes:

- soit un nom de rôle <role_name> et une variable référençant une association <var_name>. <var_name> est de type TAD.

- rechercher dans le tableau role de la variable <var_name> (<var_name>.role) l'élément dont rolename vaut <role_name>. On obtient ainsi l'indice i de cet élément dans le tableau.

- accéder aux renseignements demandés dans <var_name>.role[i].roletype.

V.4.4. Vérification sémantique

Dans un contexte dynamique, un certain nombre de vérifications sémantiques ne peuvent plus se faire à la précompilation. En effet, comment vérifier à la précompilation qu'un type d'entité existe bien dans la base de données si celui-ci n'est créé qu'à l'exécution. Ces vérifications devront donc être effectuées à l'exécution. Le précompilateur devra se charger de générer le code nécessaire à ces opérations.

Dans l'implémentation non dynamique, nous avons décidé de créer une structure interne de données capable d'accueillir la description d'un schéma contenue dans le méta-schéma, afin d'optimiser le travail du précompilateur. Pour une implémentation dynamique, un choix doit être fait. Soit on abandonne cette structure, et dans ce cas tous les contrôles devront se faire par accès direct au méta-schéma. Soit on garde cette structure, et dans ce cas, quelques adaptations doivent être opérées.

D'abord, il faudra modifier cette structure afin qu'elle puisse accueillir les nouveaux attributs définis précédemment (idte, idta, rang). Ensuite, comme les contrôles se feront à l'exécution, cette structure devra faire partie intégrante du programme précompilé. Enfin, il faudra disposer d'une série de primitives permettant de charger cette structure interne, à partir de la description contenue dans le méta-schéma, ainsi que de primitives permettant de la consulter. Il suffira ainsi que le précompilateur, dès qu'il détecte un appel au métacompilateur inséré par l'utilisateur, traduise cet appel, et le fasse suivre d'appels aux primitives qui se chargeront, à l'exécution, d'adapter la structure interne à la nouvelle description dans le méta-schéma.

Ces primitives de chargement et d'accès à la structure interne existent déjà dans le précompilateur. Il suffira de les adapter aux modifications faites à la structure de données.

6ème partie

Conclusion et Extensions possibles

VI. Conclusion et Extensions possibles

Nous espérons avoir réalisé dans ce travail ce que l'on pourrait appeler un prototype de SGBD entité-association.

Ce prototype dispose néanmoins de fonctionnalités qui en font un outil utilisable pour le développement d'applications informatiques sur PC. Citons notamment la gestion d'un dictionnaire de données stocké dans la base de données même, le support d'un modèle entité-association déjà assez complet.

Toutefois, de nombreuses extensions pourraient être apportées au système LEA afin d'en faire un réel SGBD entité-association. Ces extensions pourraient être de deux types.

VI.1. Extensions au modèle supporté

Comme le lecteur a pu le constater, le modèle EAC comprend tous les concepts de base du modèle entité-association. Certaines extensions pourraient cependant lui être apportées.

VI.1.1. Extension de l'utilisation des identifiants

Des quatre cas d'identification de F. Bodart (voir première partie), nous n'en avons conservé que le premier, étant donné les limitations de PYRAMIDE. Il ne serait pas difficile d'étendre le système LEA pour qu'il accepte plusieurs identifiants par TE ou TA. En effet, d'après D. Rossi, PYRAMIDE pourrait l'admettre sans grosses modifications. Il est clair que si PYRAMIDE gère plusieurs identifiants par TE, le système LEA peut facilement en gérer autant par TE et TA.

Pour ce qui est de la composition des identifiants, nous ne connaissons pas les possibilités d'extensions de PYRAMIDE à ce niveau. Il serait probablement possible de gérer des identifiants complexes au niveau de la couche supérieure, mais les performances ne seraient certainement pas aussi intéressantes que si cette fonction était reléguée à PYRAMIDE.

VI.1.2. Extension de l'utilisation des connectivités

Nous nous sommes également restreints à la gestion des connectivités minimales égales à un. Il serait intéressant d'étendre cette notion afin de gérer les connectivités minimales dont la valeur ne serait plus limitée à un. Il faudrait alors généraliser le travail qui a été fait sur la propagation des mises-à-jour.

VI.1.3. Autres contraintes

Enfin, le modèle entité-association théorique comprend de nombreux autres types de contraintes d'intégrité (voir première partie), que EAC ne reprend pas. Il serait évidemment intéressant que le SGBD entité-association permette la gestion automatique d'au moins certains d'entre eux. Par exemple, si le système devait être utilisé pour des applications d'ingénierie, il serait intéressant qu'il puisse gérer les contraintes associées aux mécanismes de généralisation/spécialisation.

VI.2. Extensions au système LEA

VI.2.1. Extension dynamique d'un schéma

La plupart des SGBD actuels considérent encore la définition des données comme étant statique. La modification du schéma des données entraîne énormément de modifications à apporter aux données déjà créées, et est donc une tâche que l'on évitera le plus possible.

Ainsi, dans la plupart des systèmes existants, les structures de données sont définies complètement, et ne peuvent être modifiées par après. Certains SGBD permettent de modifier le schéma des données à la condition que ces modifications soient compatibles avec les données déjà créées.

L'extension dynamique d'un schéma et des données suppose que l'utilisateur puisse accéder au dictionnaire, et le modifier dynamiquement.

Le premier point est déjà possible avec le système existant. Le problème de la modification dynamique du dictionnaire est étudié de manière théorique. Le système LEA ne permet pas encore cette fonctionnalité, mais pour peu que PYRAMIDE le permette, l'extension ne devrait pas poser de problème. Le langage est de toute façon spécifié de façon à être compatible avec cette extension future.

VI.2.2. La gestion des transactions

Cette fonctionnalité s'appuie entièrement sur PYRAMIDE. La couche supérieure est spécifiée de façon à ce que, dès que PYRAMIDE permettra cette facilité, elle puisse être récupérée par la couche supérieure sans problème. Les ordres sont déjà spécifiés, et le code pour les transformer est écrit.

VI.2.3. Puissance d'expression du langage

Le langage ne comprend pas la notion de quantificateur permettant de manipuler des conditions de sélection sur des ensembles d'occurrences. Il ne permet pas non plus la négation, ni les fonctions agrégées. Tous ces aspects pourraient être utiles dans une version étendue du système LEA.

VI.3. Conclusion

Il reste donc encore du travail afin de disposer d'un vrai SGBD entité-association. Le système LEA est néanmoins un premier pas, que nous espérons convaincant.

BIBLIOGRAPHIE

[ATZ81] ATZENI P. , CHEN P.P. : *Completeness of query language for the entity-relationship model,* Proc. 2nd International Conference on Entity Relationship Approach, Washington, D.C., October 1981, pp. 111-124.

[BER79] BERGERON R.F. , CHIANG T.C. : *A database management system with an E-R conceptual model,* Proc. 1st International Conference on Entity Relationship Approach, Los Angeles, California, December 1979, pp. 467-475.

[BODA83] Bodart F. , Pigneur Y. : *Conception assistée des applications informatiques, 1- Etude d'opportunité et analyse conceptuelle,* Masson éditions & Presses Universitaires de Namur, 1983.

[CAM85] CAMPBELL D.M., CZEJDO B. , EMBLEY D.W. : *A relationally complete query language for an entity-relationship model,* Proc. 4th International Conference on Entity Relationship Approach, Chicago, Illinois, October 1985, pp. 90-99.

[CAT88] CATARCI T. , SANTUCCI G. : *Query by diagram : a graphic query system,* Proc. 7th International Conference on Entity Relationship Approach, Roma, Italy, November 1988, pp. 157-174.

[CHE76] CHEN P.P. : *The entity-relationship model : towards a unified view of data,* ACM Transactions On Database Systems, vol. 1, n°1, mars 1976.

[DATE83] Date, C. J. : *Introduction to database systems, Volume II,* Addison-Wesley, 1983.

[ELM81] ELMASRI R. , WIEDERHOLD G. : *GORDAS : a formal high-level query language for the entity-relationship model,* Proc. 2nd International Conference on Entity Relationship Approach, Washington, D.C., October 1981, pp. 49-70.

[ELM85] ELMASRI R.A. , LARSON J.A. : *A graphical query facility for ER databases,* Proc. 4th International Conference on Entity Relationship Approach, Chicago, Illinois, October 1985, pp. 236-245.

[HAIN86-a] Hainaut, J-L. : *Conception assistée des applications informatiques, 2- conception de la base de données*, Masson éditions & Presses Universitaires de Namur, 1986.

[HAIN86-b] Hainaut, J-L. : *Technologie des fichiers, Notes de cours,* Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix à Namur, 1986.

[HAIN87] Hainaut, J-L. :*NDBS- a simple database system for small computers,* Document interne, Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix à Namur, 1987.

[LAMS87] Van Lamsweerde, A. : *Méthodologie de développement de logiciels, Notes de cours,* Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix à Namur, 1987.

[LAR83] LARSON J.A. , DWYER P.A. : *Defining external schemas for an entity relationship database,* Proc. 3rd International Conference on Entity Relationship Approach, Anaheim, California, October 1983, pp. 347-364.

[LUS79] LUSK E.L. , OVERBEEK R.A. : *A DML for entity-relationship models,* Proc. 1st International Conference on Entity Relationship Approach, Los Angeles, California, December 1979, pp. 445-461.

[MAR83] MARKOWITZ V.M., RAZ Y. : *ERROL : an entity-relationship, role oriented, query language,* Proc. 3rd International Conference on Entity Relationship Approach, Anaheim, California, October 1983, pp. 329-345.

[MIS86] MISSALA M. , SUBIETA K. : *Semantics of query languages for the entity-relationship model,* Proc. 5th International Conference on Entity Relationship Approach, Dijon, France, November 1986, pp. 291-310.

[PAR87] PARENT C., SPACCAPIETRA S. : *Un modèle et une algèbre pour les bases de données de type entité-relation,* Université de Bourgogne, Rapport de recherche n° 8701, janvier 1987.

[POO78] POONEN G. : *CLEAR : a conceptual language for entities and relationships,* Proc. ICMOD, Milan, Italy, 1978, pp. 95-113.

[RAZ83] MARKOWITZ V.M. , RAZ Y. : *A modified relational algebra and its use in an entity-relationship environment,* Proc. 3rd International Conference on Entity Relationship Approach, Anaheim, California, October 1983, pp. 315-328.

[ROE85] ROESNER W. : *DESPATH : an ER manipulation language,* Proc. 4th International Conference on Entity Relationship Approach, Chicago, Illinois, October 1985, pp. 72-81.

[ROSS89] ROSSI D. : Pyramide : a physical engine for the management of entity-relationship databases, Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix à Namur, 1989.

[SHO79] SHOSHANI A. : *CABLE : a chain-based language for the entity-relationship model,* Proc. 1st International Conference on Entity Relationship Approach, Los Angeles, California, December 1979, p. 465.

[TURB88] *Turbo Pascal, User's guide and programmer's reference*. Borland, 1988.

[URS83] URSPRUNG P. , ZEHNDER C.A. : *HIQUEL : an interactive query language to define and use hierarchies,* Proc. 3rd International Conference on Entity Relationship Approach, Anaheim, California, October 1983, pp. 299-314.

[VEL85] VELEZ F. : *LAMBDA : an entity-relationship based query language for the retrieval of structured documents,* Proc. 4th International Conference on Entity Relationship Approach, Chicago, Illinois, October 1985, pp. 82-89.