MASTER EN SCIENCES INFORMATIQUES

La Rétro-Ingénierie des Bases de Données, Application à IMS

Richard, Philippe

Award date:
1995

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
NAMUR
Institut d'Informatique

La Rétro-Ingénierie des Bases de Données, Application à IMS

Mémoire présenté pour l'obtention du grade de Licencié et Maître en Informatique

Philippe RICHARD

Promoteur : Jean-Luc HAINAUT

Année académique 1994-1995

Remerciements

Je tiens tout d'abord à exprimer ma profonde gratitude envers mon promoteur Jean-Luc Hainaut pour ses judicieux conseils, sa patience et l'attention qu'il m'a portée durant toute cette année.

Je souhaite aussi remercier cordialement Jean Henrard ainsi que toute l'équipe DB-MAIN pour l'aide, l'assistance et les conseils donnés tout au long de mon travail.

C'est également un plaisir pour moi de remercier Bénédicte Delvaux, Marc Ervier, Mr Hanotiau, Michel Roba et Philippe Esquenet pour l'accueil dans leur entreprise et pour leur dévouement. De la même façon, je désire présenter mes remerciements à Ignace Van Waes et Camille Brichard de chez IBM pour leur aide et leurs éclaircissements sur IMS.

De façon générale, je tiens à remercier toutes les personnes qui, directement ou indirectement, ont participé à l'élaboration de ce mémoire.

Enfin, je voudrais profiter de l'occasion qui m'est offerte pour remercier affectueusement toutes les personnes (professeurs, assistants, parents et amis) qui m'ont soutenu et aidé durant toutes ces années d'études.

Résumé

La rétro-ingénierie des bases de données consiste à retrouver dans la documentation disponible les spécifications fonctionnelles et techniques desquelles une application est née. Ce mémoire traite de la rétro-ingénierie appliquée aux bases de données IMS.

Le mémoire analyse d'abord la phase d'extraction des structures de données IMS. Ensuite, nous effectuons une analyse du code de manipulation de la base de données, des programmes et des données. Nous avons alors un premier schéma conceptuel.

Suite à cela, la phase de conceptualisation des structures de données IMS débute. Durant cette phase, nous transformons le schéma afin de le mettre sous une forme plus présentable. Nous obtenons alors le schéma conceptuel final.

Pour illustrer notre analyse, nous présentons finalement une étude de cas dans laquelle un processus complet de rétro-ingénierie d'une base de données IMS est expliqué.

Abstract

Data Base Reverse Engineering of is a process wich recover from available documentation the functional and technical specifications from wich an application is born. This thesis deals with reverse engineering of IMS data bases.

The thesis first analyses the IMS data structures extraction process. Afterwards, we proceed an analysis of data base manipulation codes, programs and datas. We then obtain a first conceptual schema.

After that, the IMS data structures conceptualisation process begins. During this process, we tranform the schema in order to make it more convenient. We then obtain the last conceptual schema.

To illustrate our analysis, we present finally a case study in wich a whole reverse engineering process of an IMS data base is explained.

Table des matières

Introduction

La Rétro-Ingénierie de Bases de Données consiste à retrouver dans la documentation disponible les spécifications fonctionnelles et techniques desquelles une application est née.

Cette documentation disponible est constituée du code source de définition des bases de données, du code source de manipulation des bases de données, des programmes applicatifs, du contenu de ces bases de données ou encore de la documentation de développement ou de la culture d'entreprise. Sur base de ces sources, le rétro-ingénieur essaye de retirer le maximum d'informations conceptuelles et techniques. Certaines sources comme le code de définition des bases de données sont analysées aisément et permettent de trouver bon nombre d'informations. Par contre, d'autres sources comme les programmes applicatifs sont beaucoup plus difficiles à examiner mais peuvent également cacher des informations importantes. Comme résultat à cette phase, nous obtenons un schéma conceptuel brut dont la structure est conforme au système de gestion de bases de données cible et qui contient également des optimisations.

Afin d'avoir une vue plus abstraite et ne tenant compte d'aucun paramètre technique, il convient de transformer ce schéma. Le résultat de ces transformations donne un schéma conceptuel final. Et si le rétro-ingénieur a bien travaillé, ce schéma final doit correspondre au schéma initial qui avait donné naissance à l'application.

En résumé, nous pouvons définir le processus de rétro-ingénierie comme l'inverse du processus de conception, ou d'ingénierie, de bases de données. La question à se poser est de savoir pourquoi le processus de rétro-ingénierie est beaucoup moins étudié et appliqué que son homologue de la conception.

Comme nous l'avons précédemment annoncé, une des réponses se trouve dans le matériel de départ. En effet, si l'ingénierie part de spécifications fonctionnelles bien définies, la rétro-ingénierie exploite, elle, une gamme de documentations très diverses où l'information utile est noyée dans toute une série d'autres informations. Le processus de rétro-ingénierie est donc plus difficile et moins formel que son inverse. Ceci explique sans doute le désintéressement des chercheurs à l'égard de la rétro-ingénierie.

Nous sommes toutefois convaincus du bien fondé et de la nécessité de recherches dans le domaine de la rétro-ingénierie. Effectivement, il est regrettable que les études menées jusqu'à maintenant dans le domaine de la conception ne se préoccupent pas du cycle de vie complet d'une application. Ces études ne prennent en compte que le développement de l'application et laisse de côté tout ce qui concerne sa maintenance. Hors, il est reconnu que la majorité du temps consacré à une application se passe à essayer de maintenir cette application. Dans ce cadre là, la rétro-ingénierie est d'une grande utilité car elle permet de faciliter cette maintenance. En effet, les concepteurs peuvent, grâce à la rétro-ingénierie, retrouver le schéma conceptuel initial de l'application et accélérer ainsi la maintenance. D'un autre côté, il est tout aussi évident que le coût lié à cette maintenance diminue en proportion ce qui est un avantage de poids pour les entreprises confrontées à ces maintenances.

Le but de ce mémoire est donc d'approfondir l'étude de la rétro-ingénierie. Pour ce faire, nous nous sommes fixés comme objectif de réaliser une étude complète sur le processus de rétro-ingénierie appliqué à deux systèmes de gestion de bases de données à savoir IMS et DATACOM/DB.

Le défi est d'autant plus grand que ces deux systèmes sont relativement anciens. Ils n'ont donc pas cette qualité d'abstraction conceptuelle que possèdent les systèmes plus actuels. Par conséquent, ils mélangent beaucoup d'informations techniques et conceptuelles et rendent ainsi la tâche du rétro-ingénieur plus difficile. Par contre, cette ancienneté fait qu'ils sont fort répandus dans le monde des bases de données. En effet, il est courant de voir de grandes sociétés utiliser encore ces systèmes. De ce fait, une étude sur la rétro-ingénierie de ces deux systèmes est naturellement nécessaire.

Pour mener à bien notre étude, nous pouvons également compter sur le soutien de deux entreprises utilisant ces deux systèmes. Il s'agit pour IMS de la banque CGER et pour DATACOM/DB de la société des 3 Suisses. Ces deux sociétés vont nous permettre d'avoir une vue plus pratique des applications que celle évoquée dans la littérature. Cette vue pratique est capitale car elle montre les astuces et les optimisations utilisées dans le développement des applications.

En ce qui concerne la structure de ce mémoire, nous allons dans un premier chapitre analyser la rétro-ingénierie de façon générale. Nous en dégagerons alors les différentes phases et les différents sous-processus. Ces phases et processus serviront par la suite de structure pour le reste du mémoire. Pour terminer, nous illustrerons le tout par une étude de cas.

Chapitre 1 : La rétro-ingénierie des bases de données

1.1. Introduction

Notre but dans ce chapitre n'est pas de faire un examen approfondi du sujet mais seulement de le décrire au sens large. Le lecteur aura ainsi une vision plus claire du domaine pour la suite de l'exposé. De même, cela permet de jeter les bases de nos outils d'analyse et d'établir la façon dont nous allons travailler.

Ce chapitre est basé principalement sur des documents écrits par Jean-Luc Hainaut et son équipe et dont les références sont [HAINAUT, 93a], [HAINAUT, 94], [HAINAUT, 95]. Ces écrits sont le résultat de plusieurs années de recherche dans le domaine de la rétro-ingénierie des bases de données. Cette recherche s'est déroulée dans trois programmes cadres. Le premier projet se nommait TRAMIS et était destiné à l'étude du design de base de données (BD) basé sur les transformations. Ensuite, la rétro-ingénierie (*reverse engineering*) de BD a pris le relais dans le projet PHENIX[1]. L'objectif de ce projet était de développer une approche système-expert pour la rétro-ingénierie des BD. Le projet DB-MAIN[2] a ensuite pris la relève et est encore en développement aujourd'hui. Ce dernier projet universitaire/industriel de recherche a pour but de mettre en évidence les concepts et les méthodologies inclus dans l'évolution des BD et de développer un outil CASE intégrant les résultats de la recherche.

Nous avons principalement basé ce chapitre sur ces documents car le milieu scientifique en général ne s'est, jusqu'à présent, pas fort intéressé à ce domaine de recherche. Cependant, il est sûr que les secteurs économiques où l'information et son stockage revêtent une grande importance sont, quant à eux, très attachés à la rétro-ingénierie. Une preuve en est le nombre de partenaires industriels impliqués dans le projet DB-MAIN. La raison de cet engouement provient du fait que, dans beaucoup de cas, faire évoluer les Systèmes d'Information anciennement développés sans documentation est d'un coût trop élevé.

---

[1] PHENIX est un programme de recherche université/industrie de quatre ans (1989-1993) développé conjointement par les FUNDP et le BIKIT (Babbage Institute for Knowledge and Information Technology, Université de Ghent), et est supporté par un consortium de quatorze industriels et par IRSIA/IWOLN, une agence de support à la recherche belge (contrat n° 5421).

[2] Le projet DB-MAIN est partiellement supporté par ACEC-OSI, ARIANE-II, Banque UCL, centre de recherche public H. Tudor, CGER, Cockrill-Sambre, CONCIS, D'Ieteren, DIGITAL, EDF, Groupe S, IBM, OBLOG Software, ORIGIN, Winterthur, 3 Suisses, BBL. Il a été développé en collaboration avec le laboratoire de base de données de l'EPFL (Lausanne).

1.2. Définition et état actuel de l'art

Pour définir en quelques mots la Rétro-Ingénierie (RI), citons [HAINAUT, 93a]:*la RI essaie de répondre à la question " Quels sont les spécifications possibles de cette implémentation? "*. La RI tente donc de retrouver dans la documentation disponible les spécifications fonctionnelles et techniques ainsi que la stratégie desquelles l'application est née. Ces spécifications servent ensuite à restructurer, maintenir, faire évoluer ou migrer les applications analysées.

La documentation servant de base à l'analyse est constituée de divers supports: le code du Data Definition Language (source principale), les programmes sources et le Data Manipulation Language, la documentation du design, les structures organisationnelles ou encore la culture d'entreprise. Les sources sont diverses et tout est bon pour y trouver la moindre information utile.

Comme nous l'avons dit plus haut, les recherches dans ce domaine ne sont pas légions. Signalons cependant à titre d'indication les références suivantes ([HAINAUT, 93a]):

- RI des fichiers standards : [CASANOVA, 83], [NILLSON, 85], [DAVIS, 85];
- RI des bases de données IMS : [NAVATHE, 88], [WINANS, 90];
- RI des bases de données CODASYL : [BATINI, 92];
- RI des bases de données relationnelles : [CASANOVA, 84], [NAVATHE, 88], [DAVIS, 88], [SPRING, 90], [FONKAM, 92].

Toutefois, la plupart de ces études apparaissent limitées et trop rigides. Elles sont basées sur des hypothèses trop fortes au point de vue de la qualité de la documentation et de la stratégie de design. Citons par exemple [HAINAUT, 93a]:

- toutes les spécifications conceptuelles ont été traduites en structures de données et contraintes;
- la translation est assez rigide (pas d'astuce de représentation);
- le schéma n'a pas été restructuré pour des objectifs de performances ou autres;
- un schéma physique complet de la base est disponible;
- les noms ont été choisis rationnellement (par exemple une *foreign key* et sa clé de référence portent le même nom).

1.3. Développement d'un système d'information

D'un point de vue pragmatique, l'ingénierie des programmes dans le développement d'un système d'information de gestion apparaît comme beaucoup moins réfléchie que l'ingénierie des bases de données. Les outils CASE supportant ces activités sont d'ailleurs pauvres en ce qui concerne le génie logiciel. Cette différence s'explique par deux faits édictés par [HAINAUT, 93a]. D'une part, les spécifications de données sont mieux comprises que les spécifications de programmes et le domaine d'implémentation des structures de données est plus restrictif que celui des programmes. D'autre part, une théorie réaliste et compréhensive d'ingénierie des BD a été mise au point, au contraire du génie logiciel. Les outils transformationnels ont d'ailleurs contribué largement à la sûreté et au crédit apportés à cette théorie d'ingénierie. Pour un examen détaillé de l'approche transformationnelle appliquée à la RI, on peut consulter [HAINAUT, 93b].

Bien que l'ingénierie des BD possède maintenant une grande expérience, le secteur de la rétro-ingénierie ne s'est pas encore fort développé. Ceci est dû en partie au matériel de départ. En effet, si l'ingénierie part de spécifications fonctionnelles bien définies, la rétro-ingénierie exploite, elle, une gamme de documentations diverses où l'information utile est noyée dans toute une série d'autres informations. Par ailleurs, le processus de rétro-ingénierie doit non seulement parvenir au schéma conceptuel initial mais aussi retrouver la stratégie employée lors de l'ingénierie. Ce processus apparaît donc comme moins rigide et structuré que son inverse.

Ces deux constations nous amènent à regarder la rétro-ingénierie comme le reflet dans le miroir du processus initial mais aussi comme un processus de conception à part entière. En outre, son niveau d'automatisation n'est en aucune mesure comparable à celui de l'ingénierie. En effet, si le processus de *design* d'une BD peut se voir comme partiellement interactif, voire même hautement automatisé, le processus de rétro-ingénierie ne peut, lui, se réaliser sans une interaction très élevée entre l'outil CASE et le rétro-ingénieur. Ce dernier fait appel aux fonctions de l'outil et par les résultats obtenus, émet des hypothèses, fait des tests, revient en arrière, refait appel aux fonctions de l'outil et ainsi de suite. De plus, au contraire du *design* normal qui est vu par tous comme séquentiel, le *rétro-design* peut être vu soit comme un processus séquentiel soit comme un ensemble de processus réalisés en parallèle. Ce parallélisme est dû à la difficulté de

séparer les informations physiques et logiques dans la documentation et à la non-rigidité naturelle du processus.

Une des questions à se poser est de savoir pourquoi la recherche en général et les outils CASE en particulier ne se sont pas penchés sur le domaine de la rétro-ingénierie. Une réponse est que la plupart des méthodologies ignorent l'évolution des bases de données après leur naissance. Les produits actuels supposent que les développements concernent uniquement des nouvelles applications alors que la majorité des développements effectifs touchent à l'évolution des applications existantes. Ces anciennes applications sont appelées en anglais des *Legacy Systems* (systèmes légués). Elles posent le problème de savoir les faire évoluer sous peine de perdre un nombre important d'informations et de travail. La rétro-ingénierie met donc en évidence ici son but principal qui est de permettre la réexploitation des anciennes applications.

En ce qui concerne l'évolution et la maintenance, des problèmes comme la migration, l'extension, la conversion, la ré-ingénierie ne sont pas non plus pris en compte dans les produits actuels. De plus, lors de l'ingénierie initiale, certains outils ne gardent pas trace de la stratégie adoptée. Cette perte d'information aussi essentielle qu'un schéma conceptuel est appelée **absence de traçabilité** et traduit bien le manque profond de ces outils dans la prise en compte du cycle de vie complet d'une base de données.

Il faut donc considérer la rétro-ingénierie comme faisant partie du cycle de vie complet d'une BD, et par la même l'intégrer dans le domaine de l'ingénierie et des outils CASE supportant le développement des BD.

De ce qui a été dit plus haut, on en retire que le processus de rétro-ingénierie est un processus complexe et difficile avec comme conséquence qu'il n'est pas réalisé la plupart du temps. Toutefois, [HAINAUT, 93a] nous fournit un ensemble d'arguments permettant d'atténuer cette difficulté:

- la distance sémantique entre les spécifications conceptuelles et leurs implémentations physiques est souvent étroite;
- les données sont généralement la partie la plus stable des applications;
- même dans les très vieilles applications, la structure sémantique qui sous-tend la structure des fichiers est souvent indépendante des procédures.

Après avoir replacé la rétro-ingénierie dans son contexte général, nous allons, au point suivant, décrire en détail son mécanisme et les processus sous-jacents.

1.4. Méthodologie générale de la rétro-ingénierie de BD

Tout d'abord, nous pouvons dire que la rétro-ingénierie de bases de données comprend deux processus principaux à savoir l'**extraction des structures de données (*Data Structure Extraction: DSE*)** et la **conceptualisation des structures de données (*Data Structure Conceptualization: DSC*)**. Ces deux processus généraux requièrent différents matériels de recherche, différentes méthodes et différents raisonnements. Ils peuvent se voir comme la contrepartie de l'ingénierie initiale. Le DSE correspond à l'inverse du *design* physique et le DSC à l'inverse du *design* logique. Ces deux processus fournissent par conséquent les deux produits attendus d'une rétro-ingénierie à savoir le schéma optimisé orienté Système de Gestion de Bases de Données cible pour le DSE et le schéma conceptuel pour le DSC. Nous avons illustré la rétro-ingénierie en détail dans la Figure 1 fournie par [HAINAUT, 95].

1.4.1. Extraction des structures de données

Cette phase sert à retrouver le schéma optimisé orienté vers le SGBD cible. Pour ce faire, elle se base sur la documentation formée du code Data Definition Language (DDL) - Data Manipulation Language (DML), du schéma physique, des programmes et des données.

Trois problèmes ont été mis en évidence par [HAINAUT, 95] dans cette phase.

Le premier de ces problèmes est la **dissimulation de structure (*structure hiding*)**. Elle s'applique à des structures de données ou des contraintes qui *peuvent être implémentées dans le SGBD cible*. Ces contraintes ou structures sont implémentées par des contraintes ou structures plus générales afin de satisfaire d'autres exigences telles que la simplicité, la généricité ou l'efficacité. Par exemple, un attribut composé est déclaré comme un champ atomique regroupant les différents champs composants, une relation *one-to-many* est implémentée sous une relation *many-to-many* ou encore une relation est implémentée par une *foreign key*. Ce type de problème se retrouve beaucoup dans les structures IMS.

Les **structures non-déclaratives** sont le deuxième problème. Ce sont des structures ou contraintes qui *ne peuvent être implémentées dans le SGBD cible*. Elles

sont alors représentées dans d'autres sources comme les programmes ou les données. Dans ce type de problème figure par exemple la transcription de contraintes référentielles dans les fichiers standards.

Enfin, le troisième problème se retrouve sous la forme de **perte de spécifications**. Cette perte se fait lorsque des structures ou des contraintes *ne sont implémentées ni dans le SGBD cible ni dans les programmes*. Elles se retrouvent alors dans les structures organisationnelles ou la culture d'entreprise. En conséquence, ces structures ou contraintes ne peuvent être retrouvées dans les sources disponibles sur papier.

Ces trois problèmes mettent en évidence la difficulté de retrouver toute l'information cachée dans les sources. Toutefois, une grande majorité de ces informations peut être issue des analyses qui constituent le DSE et sont décrites ci-après.

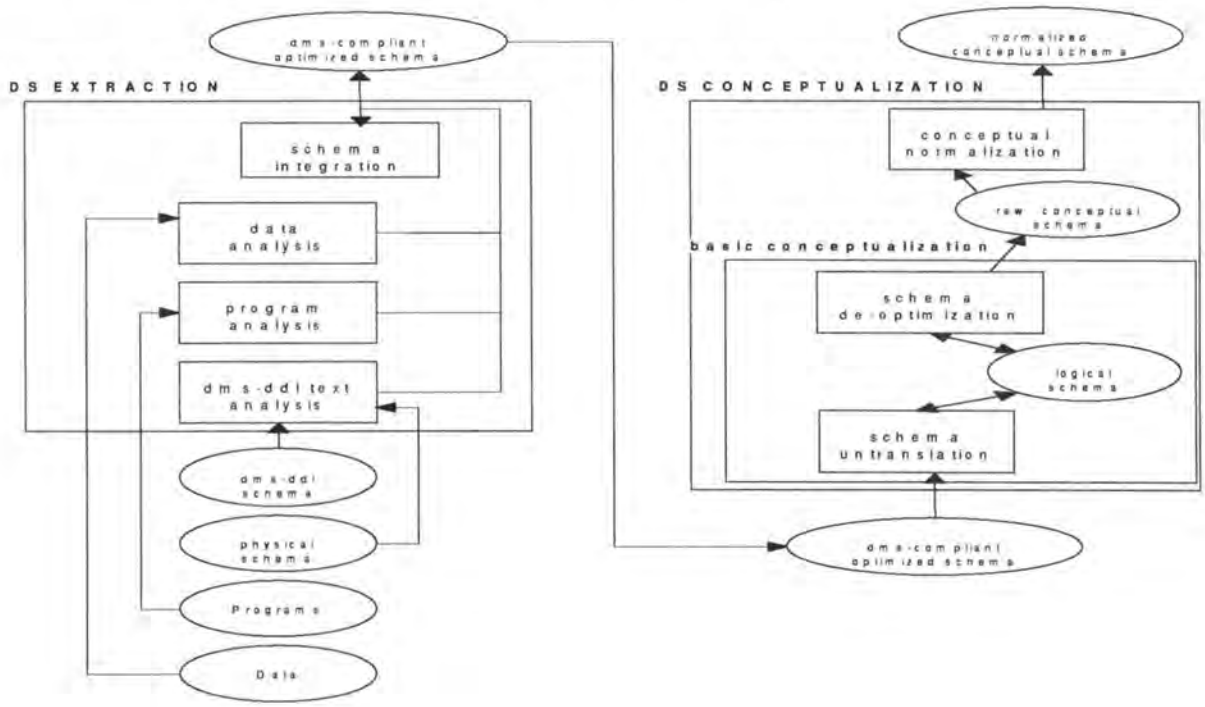

Figure 1 : La rétro-ingénierie de bases de données

L'**analyse du code DMS-DDL et du schéma physique** est réalisée en premier lieu car c'est la source la plus regorgeante d'informations. Cette première analyse n'est donc pas sans raison. En effet, de par sa richesse, elle permet de donner un premier squelette au schéma. Les autres sources d'informations sont plus difficiles à analyser et apportent proportionnellement moins d'informations. Leur analyse est par conséquent effectuée ensuite et sert à affiner le schéma déjà obtenu. Ces analyses portent sur les programmes et sur les données stockées. Toutefois, comme nous l'avons déjà dit plus haut, ces différentes analyses peuvent se faire également en parallèle. Par exemple, une même information peut avoir été traduite dans deux sources différentes (une contrainte d'intégrité traduite dans le DDL et dans les programmes). Dès lors, il serait appréciable de confirmer une information trouvée en analysant directement une autre source.

L'**analyse des programmes** est un processus beaucoup plus complexe. Il retrouve dans les programmes les informations qui permettent de confirmer ou de détailler les structures découvertes par l'analyse du DDL-DML et du schéma physique. Par exemple, un champ peut se révéler être un attribut composé par l'analyse d'une *working-area* en

COBOL qui montrerait la décomposition de ce champ. Cette analyse peut donc régler une partie des problèmes engendrés par la dissimulation de structure et les structures non-déclaratives.

L'**analyse des données** examine le contenu de la base de données pour détecter des structures ou des contraintes non encore trouvées et pour tester des hypothèses (est-ce que ce champ c peut être une *foreign key* pour cet autre fichier f?). On peut, par le biais de cette analyse, trouver des solutions aux trois problèmes évoqués ci-dessus.

Finalement, l'**intégration de schéma** sert à intégrer, sans perte d'information, les différents schémas issus des analyses précédantes.

Le produit sortant de ce processus est un schéma intégré. Ce schéma est orienté vers le SGBD cible et comporte des astuces de représentation provenant d'une possible optimisation.

1.4.2. Conceptualisation des structures de données

Ce second processus se base sur le résultat du premier processus, à savoir le schéma optimisé orienté vers le SGBD. A partir de ce schéma, le processus a pour but de fournir un schéma conceptuel normalisé. Pour cela, il détecte et transforme les structures non-conceptuelles, les redondances, les structures résultant d'une optimisation et les structures résultant de la dépendance au SGBD cible. Ce processus de conceptualisation repose donc fortement sur les techniques transformationnelles. Le schéma résultant sera alors une interprétation conceptuelle possible des sources analysées.

Ce processus consiste en deux phases: la conceptualisation de base et la normalisation conceptuelle.

La **conceptualisation de base (*basic conceptualization*)** a pour objectif principal d'extraire tous les concepts sémantiques sous-jacents au schéma source. Ce sous-processus se divise également en deux nouveaux sous-processus: la détraduction du schéma et la déoptimisation du schéma.

La **détraduction du schéma (*schema untranslation*)** consiste à retrouver les structures conceptuelles desquelles le développeur a dérivé les structures logiques du schéma source. Pour cela, il faut d'abord identifier les transformations symétriquement réversibles qui ont permis cette translation entre les deux schémas. On retrouve alors naturellement les structures conceptuelles à la base des transformations.

La **déoptimisation du schéma (*schema de-optimization*)** a pour but de trouver les structures provenant d'une optimisation. Trois familles d'optimisations sont à considérer: la dénormalisation, la redondance structurelle et la restructuration. On peut parler de déoptimisation dépendante et indépendante du SGBD.

De ce sous-processus, il ressort un nouveau schéma appelé le schéma conceptuel de base.

La **normalisation conceptuelle (*conceptual normalization*)** améliore le schéma conceptuel de base dans le but de lui donner les qualités attendues d'un schéma conceptuel initial. Ces qualités sont par exemple l'expressivité, la simplicité, la minimalité, la lisibilité, la généricité ou l'extensibilité. Citons par exemple le remplacement de types d'entités par des attributs ou la standardisation des noms.

Le schéma final est donc une interprétation possible du schéma initial compte tenu des choix de stratégie pris en hypothèse lors de la rétro-ingénierie. Si nous avions à notre disposition une documentation fournie sur la stratégie qui avait été employée lors de l'ingénierie initiale, il est évident que le schéma résultant de notre processus serait presque totalement identique au schéma conceptuel initial.

1.4.3. Recherche de la stratégie de conception

Suite à la dernière remarque de la section précédente, nous en venons à ce dernier point que nous considérons comme différent des deux précédents. En effet, cette recherche de stratégie ne figure pas sur le schéma de la Figure 1. Ceci provient du fait qu'elle est présente à toutes les étapes de la conceptualisation des structures de données. Plusieurs fois durant le processus, le rétro-ingénieur se retrouve devant un choix de conceptualisation à faire. Il est évident que son homologue de l'ingénierie aura dû précédemment faire ces mêmes choix. Dès lors, si nous voulons arriver à un schéma proche de celui dont était parti l'analyste, il faut aussi s'astreindre à rechercher la stratégie de conception de ce dernier. Si on oublie cette recherche, il est fort à parier que notre schéma final sera, bien que correct, très loin du schéma de l'ingénierie initiale. C'est pour cette raison que nous avons voulu différencier clairement ce troisième processus.

Maintenant, pour retrouver cette stratégie, nous pouvons tabler sur plusieurs ressources. Premièrement, une documentation de l'analyse peut être disponible. Si ce n'est pas le cas, un entretien avec l'analyste (s'il est encore présent dans l'entreprise) permet de voir quelle a été sa façon de travailler et dès lors de reconstituer une stratégie acceptable. D'autre part, la prise en compte des habitudes de l'entreprise quant à l'ingénierie peut être d'une aide précieuse. Ces sources permettent donc de constituer, si pas une stratégie complète, au moins un ensemble de règles de décisions.

L'analyste peut alors, à chaque point de décision du DSC, se reporter à la stratégie de référence et opter pour des choix judicieux et non hasardeux. Toutefois, à défaut de stratégie disponible, l'analyste doit faire preuve d'une certaine continuité logique dans ses choix.

De notre avis, la recherche de la stratégie doit donc, si les sources le permettent, se situer en avant de la conceptualisation. Cette dernière est alors grandement facilitée par la stratégie disponible. Par ailleurs, le processus de conceptualisation peut, au fur et à mesure de son avancement, améliorer ou restructurer la stratégie. La conclusion en est que la recherche de la stratégie constitue un processus important à ne pas négliger.

Pour terminer, signalons que cette vue du processus de rétro-ingénierie est surtout là pour mettre en évidence quels sont les problèmes à résoudre et de quelle façon les résoudre. Elle ne doit pas être considérée comme une méthode formelle d'exécution des différentes tâches.

Chapitre 2 : Etude du système de gestion de bases de données IMS

2.1. Présentation du produit

Information Management System (IMS) est un Système de Gestion de Bases de Données (SGBD) hiérarchique qui apparut vers la fin des années 60. La première version, " IMS/360 version 1 ", résulta d'un projet de développement conjoint entre IBM et North American Rockwell et fut mise sur le marché en 1968. Par la suite, d'autres versions suivirent et parmi elles "IMS/ESA Version 4" qui constitue actuellement la dernière version commercialisée.

Pour [GALASCI, 89], IMS est issu des premières structures séquentielles de données décrites à l'aide de langages tels que COBOL. Dans [IBM/IMS, 76], IBM indique avoir développé ce système en réponse à une demande des utilisateurs pour une meilleure capacité et flexibilité à développer des applications de bases de données et de communications. Sur cette base, IBM a développé un ensemble de fonctions de définitions de base de données (BD) et de techniques de manipulation. L'utilisateur peut donc organiser ses données dans une meilleure compréhension de l'information et de manière plus efficace.

D'autre part, IBM a également voulu développer un système indépendant et ne se rattachant donc à aucune considération hardware ou software. Il est bien sûr évident que l'émergence d'IMS provient aussi de la demande sans cesse croissante de l'intégration des différents fichiers en une seule unité de stockage et donc, de l'émergence même des bases de données.

Pour l'anecdote, signalons enfin qu'une telle demande est venue principalement de la NASA. Celle-ci avait grandement besoin d'un tel système de gestion de données afin d'administrer efficacement l'impressionnante quantité d'informations que nécessitait le projet de conquête lunaire APOLLO. On peut consulter à ce sujet la référence [BARNETT, 74].

IMS fut dès lors un des premiers SGBD à être commercialisé et il est encore actuellement un des plus utilisés. Sa force réside dans sa capacité à traiter de grands volumes de données, principalement sur les *mainframes*. Etant donné que ces derniers sont contrôlés par un ou plusieurs systèmes d'exploitation, IMS est donc disponible sur plusieurs systèmes d'exploitation (*Operating System* : OS).

IMS est un SGBD à langage hôte. Cela signifie qu'il ne fournit pas son propre langage de programmation pour le développement des programmes d'application. Ceux-ci sont écrits notamment en COBOL, PL/1, FORTRAN et en assembleur dans une

moindre mesure. Des modules d'interfaces sont fournis pour que ces langages puissent faire appel au Data Manipulation Language (DML) d'IMS. Il faut pour cela que l'administrateur de la base de données définisse préalablement la BD par l'intermédiaire du Data Definition Language (DDL). Le DDL et le DML d'IMS sont regroupés dans le Data Language One (DL/1) qui constitue le coeur d'IMS.

Avec le temps, IMS s'est enrichi et possède maintenant, autour des fonctionnalités de base, tout un ensemble d'outils intégrant les avancées technologiques majeures. Parmi ces outils figurent des programmes utilitaires pour décrire la structure de la base, créer la base, réorganiser la base, analyser la charge du système. On peut citer également une interface de requêtes, un générateur d'application, des aides à la conception, aux tests et au déboguage. Ceci a permis à IMS de rester bien présent sur le marché et de continuer à être performant.

Notre but dans ce chapitre est de nous familiariser avec les concepts fondamentaux d'IMS. Nous serons ainsi mieux armés pour illustrer et comprendre la rétro-ingénierie des bases de données IMS. Tout d'abord, il convient de présenter le modèle hiérarchique de façon abstraite (point 2.2. Le Modèle hiérarchique) pour garder un cadre général à notre analyse. Ayant établi les bases du modèle hiérarchique, nous en viendrons naturellement à IMS (au point 2.3. IMS). Dans cette section, nous analyserons en premier lieu son architecture. Suite à cette architecture, nous passerons en revue les éléments fondamentaux de description d'IMS (Data Base Description, Program Communication Block, *logical* Data Base Description, Program Specification Block, Programme d'Application et les index secondaires) ainsi que les différentes bases de données permises (en référence à la méthode d'accès de la base de données).

2.2. Le Modèle hiérarchique

Le modèle hiérarchique ne fut pas conçu initialement sur base d'un modèle de donnée abstrait prédéfini. Au contraire du modèle relationnel formellement défini ou du modèle réseau basé sur un ensemble de recommandations du rapport CODASYL 1971, le modèle hiérarchique fut défini par abstraction des systèmes implémentés. Il est donc principalement constitué des concepts et structures introduits par IMS, ce dernier étant le principal système hiérarchique. Bien sûr, il existe d'autres systèmes commercialisés qui se différencient au niveau des détails mais l'approche essentielle du modèle reste identique. Parmi ces systèmes, notons, à titre d'information, SYSTEM 2000 développé par SAS et qui constitue également un système hiérarchique très populaire.

Nous allons maintenant définir ce modèle et ses concepts. Pour ce faire, nous utiliserons la terminologie IMS et non celle prise en compte par certains auteurs ([DATE, 90], [BATINI, 92]). Ceci est justifié du fait que nous nous intéressons spécialement à IMS.

2.2.1. Concepts de base

Le modèle hiérarchique repose sur deux concepts essentiels : le type segment et le type relation physique parent-enfant.

Le **type segment** correspond à la définition d'un ensemble de segments qui regroupent le même type d'informations élémentaires. Un type segment contient donc une ou plusieurs occurrence(s): le(s) **segment(s)**. Un type segment est constitué d'une collection de **champs** qui sont les données élémentaires. Ces champs peuvent être de type entier, réel, chaîne de caractères ou autre suivant le système utilisé.

Le **type relation physique parent-enfant** est un type d'association binaire *one-to-many* entre un **type segment parent physique** et un **type segment enfant physique**. Une occurrence d'un type relation physique consiste en une occurrence du type segment parent physique et une ou plusieurs occurrence(s) du type segment enfant physique.

Une **base de données hiérarchique** contient un certain nombre de hiérarchies. Une **hiérarchie** consiste en un ensemble ordonné de types segments et de types relations physiques arrangés de façon à former un arbre. Cette pure hiérarchie de types segments

est appelée en IMS une **base de données physique**. Une base de données hiérarchique est donc composée d'une ou plusieurs base(s) de données physique(s).

Une hiérarchie, ou **arbre**, consiste en un **type segment racine** et un ensemble de zéro, un ou plusieurs sous-arbre(s). Ces **sous-arbres** sont eux-mêmes constitués d'un type segment racine et d'un ensemble de zéro, un ou plusieurs sous-arbre(s), et ainsi de suite. Un arbre possède donc plusieurs **niveaux**. Le premier niveau est celui du type segment racine. Les sous-arbres du type segment racine sont du deuxième niveau, et ainsi de suite. Le nombre de niveaux correspond donc à la profondeur de l'arbre.

Pour illustrer notre propos, prenons l'exemple de la Figure 2. Cet exemple contient les informations sur la constitution des départements d'une entreprise. Dans l'exemple, chaque *département* possède un numéro, un nom, zéro, un ou plusieurs *budget(s)* et zéro, une ou plusieurs *classe(s) de personnes*. Chaque *budget* possède un numéro et un montant. Chaque *classe-personne* possède un numéro, un nom et zéro, une ou plusieurs *personne(s)*. Chaque *personne* possède un numéro, un nom, zéro, une ou plusieurs *expérience(s)* et zéro, un ou plusieurs *diplôme(s)*.

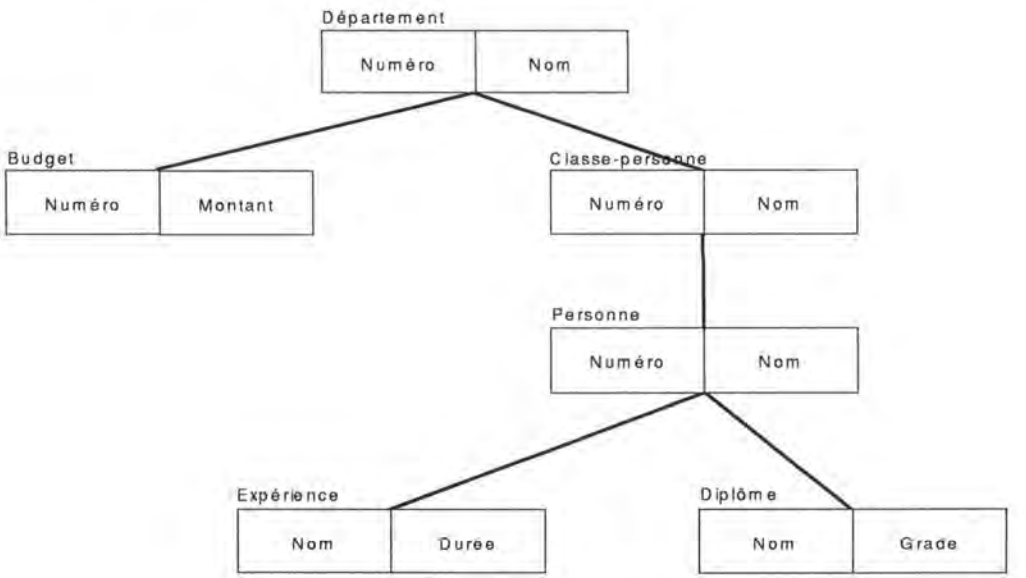

Figure 2 : Exemple de hiérarchie de types segments

Dans cet exemple, l'arbre a comme racine le type segment *département* et possède deux sous-arbres ayant comme racine le type segment *budget* et le type segment *classe-personne*. Notons ici que l'ordre dans lequel sont placés les sous-arbres est important. Dans l'exemple, le sous-arbre *budget* précède le sous-arbre *classe-personne*. Le sous-arbre *budget* ne possède que sa racine alors que le sous-arbre *classe-personne* possède un sous-arbre (*personne*). Ce sous-arbre *personne* possède, quant à lui, deux sous-arbres (*expérience* et *diplôme*). Cet arbre a quatre niveaux: 1er niveau (*département*), 2ème niveau (*budget* et *classe-personne*), 3ème niveau (*personne*) et 4ème niveau (*expérience* et *diplôme*).

La BD contient six types segments : *département*, *budget*, *classe-personne*, *personne*, *expérience* et *diplôme*. Le type segment *département* est appelé type segment racine et les autres types segments sont appelés **types segments dépendants**. Plus exactement, nous pouvons illustrer clairement le concept de type relation physique parent-enfant. La BD contient cinq types relations physiques : (*département*, *budget*), (*département*, *classe-personne*), (*classe-personne*, *personne*), (*personne*, *expérience*) et

dans le type relation (*classe-personne*, *personne*), *classe-personne* est le type segment parent physique et *personne* est le type segment enfant physique. Finalement, notons que tous les types segments ne faisant partie d'aucun type relation physique en tant que parent peuvent être appelés des **types segments feuilles**, à l'exception du type segment racine évidemment.

Nous allons maintenant nous tourner vers les occurrences. Bien entendu, tout ce que nous avons déjà dit pour les types est aussi valable pour les occurrences . Chaque **arbre d'occurrence** consiste en un arbre dont la racine est une occurrence du type segment racine, et possédant zéro, une ou plusieurs occurrence(s) des types segments immédiatement dépendants du type segment racine. De même, les sous-arbres ainsi formés répondent à la même règle. Un exemple d'arbre occurrence est illustré à la Figure 3.

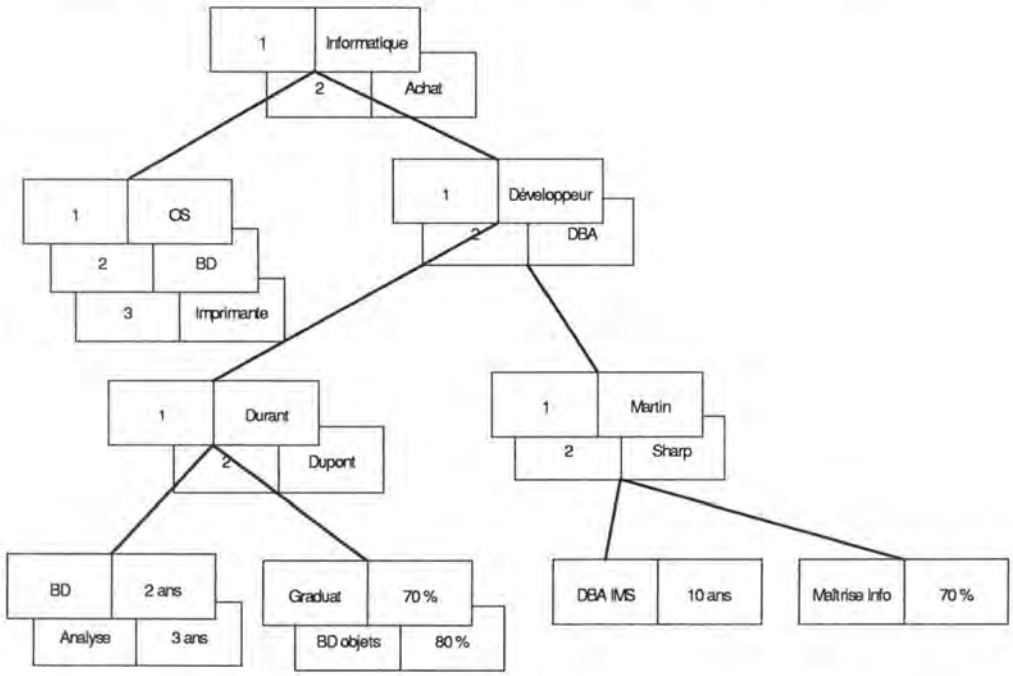

Figure 3 : Arbre d'occurrence pour la hiérarchie de la Figure 2

En IMS, un arbre d'occurrence est appelé **enregistrement physique** (*physical record*).

Dans l'exemple, le département *informatique* possède trois budgets et deux classes de personnes. Les classes de personnes possèdent deux personnes toutes les deux. Dans les personnes appartenant à la classe de personnes *DBA (Data Base Administrator)*, seule la personne *Sharp* possède une expérience et un diplôme. La personne *Martin* ne possède ni expérience ni diplôme. Dans le sous-arbre des *développeurs*, la personne *Durant* possède deux expériences et deux diplômes mais la personne *Dupont* ne possède ni expérience ni diplôme. Cet arbre d'occurrence dont la racine est le département *Informatique* constitue donc, en IMS, un enregistrement physique.

Nous pouvons maintenant aborder un nouveau concept propre aux arbres d'occurrences: le concept de **jumeaux**. Deux segments (occurrences d'un même type segment) sont dits jumeaux lorsqu'ils ont comme parent la même occurrence du type segment parent. Dans l'exemple, les trois budgets sont jumeaux, de même que les deux expériences de *Durant*. Par contre, les deux expériences de *Durant* ne sont pas jumeaux avec les deux diplômes du même *Durant*, car ils n'appartiennent pas au même type

segment bien qu'ils aient le même parent. Dans [IBM/IMS, 76], ils sont alors qualifiés de ***siblings*** (enfants du même parent).

2.2.2. Caractéristiques du modèle

Maintenant que nous avons défini en détail les concepts du modèle et la façon dont ceux-ci interagissent entre eux, nous pouvons en ressortir les caractéristiques fondamentales.

Pour ce faire, nous allons nous baser sur les trois propriétés que [BATINI, 92] a mis en évidence:

Un schéma hiérarchique de types segments et de types relations physiques parent-enfant doit suivre les trois propriétés suivantes :

1. Chaque type segment, excepté le type segment racine, participe, en tant que type segment enfant, dans exactement un et un seul type relation physique;
2. Un type segment peut participer en tant que type segment parent dans aucun, un ou plusieurs type(s) relation(s) physique(s);
3. Si un type segment participe en tant que parent dans plus d'un type relation physique, alors ses types segments enfants sont ordonnés. L'ordre est montré, par convention graphique, de gauche à droite dans un schéma hiérarchique.

De ces trois propriétés , on peut déduire que chaque type segment, excepté la racine, possède un et un seul parent mais peut avoir zéro ou plusieurs enfant(s). Nous sommes donc limités à la modélisation des types d'associations. Concrètement, on peut modéliser des types d'associations *one-to-many* (1-N) mais pas des types d'associations *many-to-many* (N-N). Nous verrons cependant au point suivant comment ce problème peut être résolu.

Les deux caractéristiques suivantes ont été énoncées par [GALASCI, 89].

La première caractéristique est que plusieurs liens (rôles) entre deux types segments sont représentés par une redondance de types segments. Un exemple nous est fourni par [GALASCI, 89]: Soient deux rôles entre personne et voiture, représentés par le schéma conceptuel général de la Figure 4. La structure hiérarchique correspondante est illustrée à la Figure 5. Cette caractéristique nous montre donc la redondance que peut amener le modèle hiérarchique.

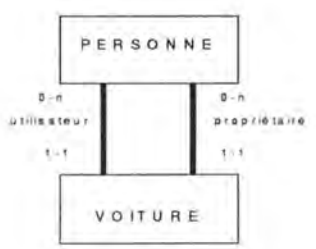

Figure 4 : Schéma conceptuel avec deux rôles entre deux types d'entités

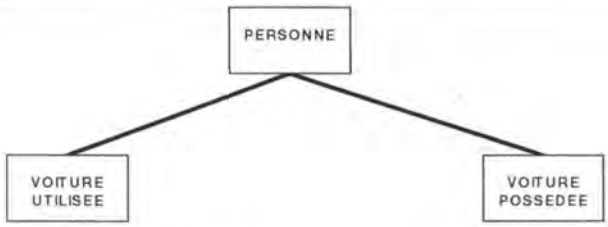

Figure 5 : Structure hiérarchique correspondante au schéma de la Figure 4

La deuxième caractéristique concerne la manipulation de la BD. En effet, pour rechercher l'information, nous sommes obligés de passer par le segment racine et de parcourir le chemin de l'arbre qui mène à notre information. Le choix du type segment racine et des différents niveaux de l'arbre est donc fonction des interrogations de la base, c'est-à-dire de directives de conception techniques et non conceptuelles.

Plusieurs inconvénients sont alors dégagés par [GALASCI, 89] et illustrés par l'exemple que nous fournit [THALMAN, 84]. Cet exemple décrit la base de données des commandes d'un magasin. La hiérarchie est la suivante (Figure 6):

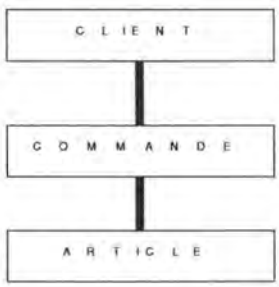

Figure 6 : Structure hiérarchique sur les commandes d'un magasin

Les différents inconvénients sont les suivants :

- **L'ajout d'un segment ne peut s'effectuer que si les segments supérieurs dont il dépend existent dans l'arbre**. Dans l'exemple, on ne peut introduire un article dans la base sans devoir créer un client et une commande. Un article ne peut donc exister seul;
- **La suppression d'un segment entraîne la suppression des segments inférieurs qui lui sont rattachés**. Dans l'exemple, si nous supprimons un client qui est le seul à avoir commandé un article précis, nous perdons, sans pour autant le vouloir, la commande et l'article qui y étaient rattachés;
- **La modification d'un segment entraîne la recherche de tous les segments qui contiennent l'information recherchée**. Comme exemple, il suffit de prendre celui de la Figure 5. Si la couleur d'une voiture change, il faut la changer dans les deux sous-arbres;

D'autres inconvénients peuvent encore se greffer à cette liste:

- **Il est facile de voyager en descendant dans l'arbre mais il est beaucoup moins évident de partir de ses feuilles**. Dans l'exemple, obtenir les articles d'un client est aisé mais trouver les clients qui ont commandé un article précis l'est beaucoup moins.

Toutefois, il faut voir que ces inconvénients peuvent tout aussi bien être pris pour des avantages en terme de contrôle d'intégrité de la base. En effet, l'obligation, lors de

l'ajout d'un segment, d'ajouter également les segments supérieurs correspond à un renforcement de l'intégrité de la base. Il en est de même lors de la suppression d'un segment. De plus, cet exemple est beaucoup trop simpliste et ne reflète en aucune façon la réalité. Mais il permet de mettre en évidence certaines carences du modèle.

D'autre part, [DATE, 90] nous fait remarquer que dans une BD hiérarchique, certaines informations, qui seraient représentées dans le modèle relationnel par des clés étrangères, sont ici instanciées par des types relations physiques parent-enfant. Par exemple, dans la Figure 2, la connexion entre *Département* et *Budget* est représentée par un type relation physique, au contraire d'une duplication de l'identifiant de *Département* dans le type segment *Budget*.

2.2.3. La relation logique parent-enfant

Au vu des caractéristiques édictées ci-dessus, le modèle hiérarchique apparaît comme assez faible pour la modélisation conceptuelle. Cette faiblesse est due à la rigueur de la structure hiérarchique. Pour palier à cette insuffisance, le modèle admet une entrave à cette structure rigide. Cette entrave, ou exception, s'appelle **relation logique** et permet d'établir, entre deux types segments appartenant à la même arborescence ou à deux arborescences différentes des relations hiérarchiques parent-enfant. Pour être précis, ces relations logiques sont appelées des **types relations logiques parent-enfant**. Les deux types segments concernés se voient attribuer le qualificatif de **type segment parent logique** et **type segment enfant logique**. De même, la notion de **jumeaux logiques** apparaît dans l'arbre d'occurrence et suit la même règle que pour les jumeaux physiques.

Illustrons ceci par un exemple en partant du schéma conceptuel général de la Figure 7. En IMS, la solution redondante est celle représentée par la Figure 8. Dans cette solution, nous avons une redondance des types segments qui se retrouvent dans les deux BD. De plus, nous avons une perte sémantique car le lien entre les deux types d'entités A et B, formé par le type d'entité C, n'est plus visible. Pour combler cette perte, le programmeur doit prendre en compte dans ses programmes, une vérification de la cohérence et des contraintes attachées aux deux ensembles des segments C.

La solution représentée avec une relation logique (matérialisée par un **pointeur logique**) permet d'éviter la redondance (Figure 9). Dans cette solution, le trait discontinu entre le type segment C et le type segment B représente le pointeur logique. Le type segment B (graphiquement le trait discontinu part du bas du rectangle) est le *parent logique*, et le type segment C (graphiquement, le trait discontinu arrive sur le haut du rectangle) est *l'enfant logique*. Toutes les occurrences du type segment C, qui ont la même occurrence du type segment B comme parent logique, sont dits *jumeaux logiques*.

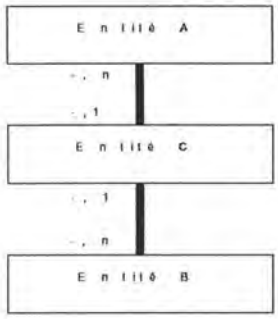

Figure 7 : Schéma conceptuel sur trois types d'entités et deux types d'associations *one-to-many*

Figure 8 : Solution redondante de la Figure 7

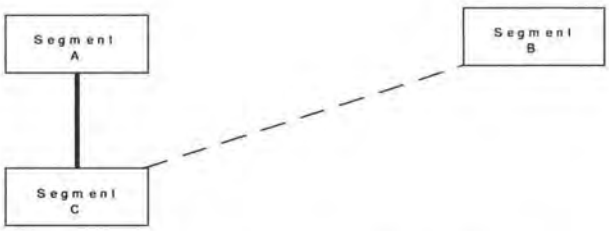

Figure 9 : Solution non-redondante de la Figure 7

Les pointeurs logiques permettent donc d'introduire des liens entre plusieurs structures d'arbres ou entre types segments d'un même arbre, et ainsi de ne pas respecter l'ordre hiérarchique. Ces types relations logiques sont transparents pour l'utilisateur de la BD. Dans le cas où les deux types segments reliés logiquement appartiennent à deux BD différentes, **ces BD sont dites liées logiquement**.

Les types relations logiques entre types segments sont soumis aux restrictions suivantes ([HENNEBERT]) :

1. Un enfant logique doit toujours avoir un parent physique (c'est-à-dire qu'il ne peut pas être la racine d'une base de données). Il peut exister à n'importe quel niveau de la hiérarchie de l'arbre;
2. Un enfant logique ne peut avoir qu'un parent logique et qu'un parent physique;
3. Un parent logique peut être la racine d'une BD ou se trouver à n'importe quel niveau de hiérarchie dans la structure arborescente de sa BD;
4. Un parent logique peut avoir un ou plusieurs enfant(s) logique(s);
5. Un type segment ne peut être à la fois enfant logique et parent logique;
6. Un enfant logique ne peut pas avoir des enfants physiques qui sont aussi enfants logiques d'un autre type segment.

Les règles énoncées ci-dessus nous permettent de résoudre le problème de la Figure 4. Ce problème est résolu à la Figure 10. Toutefois, si la Figure 4 possédait un troisième type d'association *one-to-many*, il ne serait pas possible de le modéliser en raison de la restriction 2.

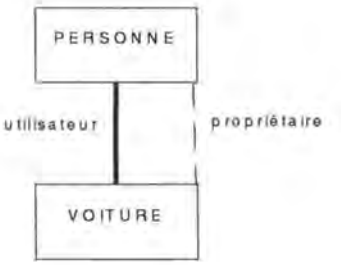

Figure 10 : Résolution de la Figure 4 par un type relation logique

D'autre part, comme nous l'annoncions plus haut, ce système de **type relation logique** permet de résoudre un grand problème de modélisation. En effet, il permet de modéliser simplement les types d'associations *many-to-many*. Pour l'exemple, considérons un type d'association N-N entre Employé et Département. La solution non redondante la plus simple est illustrée par la Figure 11.

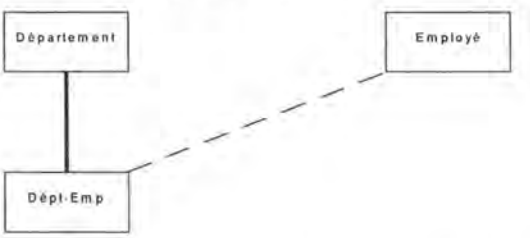

Figure 11 : Type relation logique unidirectionnel

Lorsque certaines informations sont dépendantes des deux types d'entités, comme par exemple le nombre d'heures qu'un employé travaille dans un département, elles peuvent être incluses dans le type segment formant le pointeur logique. Ces informations sont couramment appelées des **données d'intersection** par les utilisateurs de BD hiérarchiques.

Nous pouvons maintenant définir une BD logique: "*Une BD logique, comme une BD physique, consiste en un arrangement hiérarchique de types segments. Cependant, les types segments en question, quoique accessibles par cette BD logique, appartiennent réellement à une ou plusieurs BD physique(s).*" [DATE, 90]

Pour ce faire, les BD logiques implémentent la notion de type relation logique vue auparavant. Dans la construction d'une BD logique, nous prenons comme type segment racine le type segment racine d'une BD physique. Ensuite, nous prenons comme types segments dépendants n'importe quel type segment enfant physique. Si le type segment enfant physique sélectionné est aussi un type segment enfant logique, nous pouvons alors le concaténer avec son type segment parent logique. La relation est ainsi établie. En se basant sur la Figure 11, nous pouvons établir la BD logique de la Figure 12:

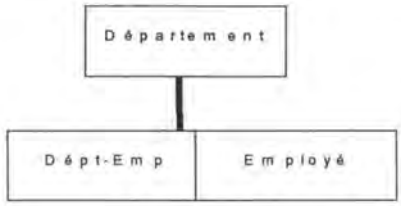

Figure 12 : Exemple de BD logique basée sur la Figure 11

Nous avons ainsi modélisé un type d'association N-N. Cependant, il n'est accessible que dans le sens Département-Employé. C'est pour cela qu'il est appelé un type relation logique **unidirectionnel**. Bien sûr, le pointeur logique aurait tout aussi bien pu être l'enfant physique d'Employé et l'enfant logique de Département. De cette façon, l'accès serait dans le sens Employé-Département. Toutefois, il se peut que l'on veuille parcourir la relation dans les deux sens. Il faut alors créer deux types segments pointeurs. Ceci est représenté à la Figure 13:

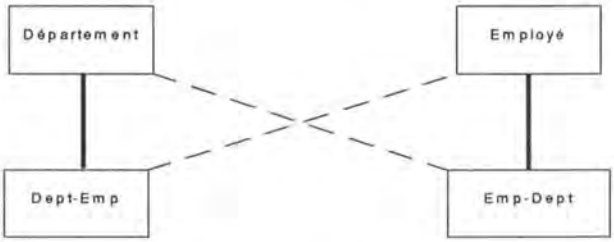

Figure 13 : Type relation logique bidirectionnel

En IMS, les types segments Dept-Emp et Emp-Dept sont appelés des **types segments paires**. Nous pouvons alors établir deux BD logiques (Figure 14) qui permettront l'accès dans les deux sens.

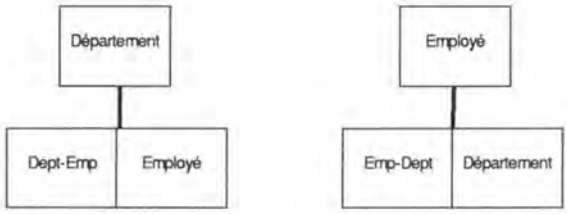

Figure 14 : Exemple de BD logiques basées sur la Figure 13

Le type relation logique faisant appel à des types segments paires est appelé un type relation logique **bidirectionnel** car il est possible de le parcourir dans les deux sens. Reste à savoir comment les deux types segments paires sont stockés car ils ont la même fonction et possèdent donc les mêmes données.

Premièrement, nous pouvons les déclarer tous les deux comme types segments enfants logiques avec la même description. Ainsi, il seront tous les deux réellement stockés et sont appelés des types segments enfants logiques **réels**. IMS se charge alors de l'intégrité des segments lors de manipulation, c'est-à-dire que lors d'une insertion d'un des deux segments, IMS se charge d'insérer également l'autre segment. Le type relation est dans ce cas un type relation logique **bidirectionnel physique**.

Deuxièmement, nous pouvons en déclarer un comme type segment enfant logique **réel** et l'autre comme type segment enfant logique **virtuel**. La description des données

d'intersection se trouve dans l'enfant logique réel. L'autre type segment n'existant pas, celui-ci est matérialisé par une suite de pointeurs logiques sur les segments du type segment enfant logique réel. Le type relation logique est alors appelé un type relation logique **bidirectionnel virtuel**.

2.2.4. La séquence hiérarchique

Pour en finir avec la présentation du modèle hiérarchique, nous allons décrire la façon dont une base de données physique est stockée et par la même la notion de **séquence hiérarchique**.

Tout d'abord, rappelons le fait qu'un arbre possède plusieurs occurrences et chacune de ces occurrences correspond à un enregistrement physique.

En ce qui concerne le rangement des segments d'un enregistrement physique, nous citerons la règle édictée par [DATE, 90] :

> *Considérons un arbre T avec un segment racine R et des sous-arbres S1, S2, ...,Sn (dans cet ordre). Soit t une occurrence de T, avec une racine r (une occurrence de R) et des sous-arbres s1, s2, ..., sn (respectivement occurrences de S1, S2, ..., Sn). La séquence hiérarchique pour t (récursivement) est la séquence obtenu en prenant d'abord le segment r, suivi par tous les segments de s1 en séquence hiérarchique, suivi par tous les segments de s2 en séquence hiérarchique, ..., suivi par tous les segments de sn en séquence hiérarchique.*

En se basant sur l'exemple de la Figure 3, cela donne la Figure 15:

DEPARTEMENT Informatique
BUDGET OS
BUDGET BD
BUDGET Imprimante
CLASSE_PERSONNE Développeur
PERSONNE Durant
EXPERIENCE BD
EXPERIENCE Analyse
DIPLOME Graduat
DIPLOME BD objets
PERSONNE Dupont
CLASSE_PERSONNE DBA
PERSONNE Martin
PERSONNE Sharp
EXPERIENCE DBA IMS
DIPLOME Maîtrise Info
DEPARTEMENT Achat

Figure 15 : Séquence hiérarchique de l'exemple de la Figure 3

Chaque enregistrement physique peut être vu comme une occurrence du sous-arbre d'un arbre système, ceci afin de voir la BD entière comme un seul arbre et donc une seule séquence hiérarchique. Par conséquent, la notion de séquence hiérarchique définit un ordre total pour l'ensemble de tous les segments de la BD physique, et cette dernière peut se voir comme stockée suivant cet ordre total.

2.3. IMS

Nous allons maintenant rentrer dans le sujet proprement dit. Pour ce faire, nous allons décrire l'architecture d'un environnement IMS. Pour chaque élément de cette architecture, nous verrons ses caractéristiques.

2.3.1. Architecture

L'architecture d'un environnement IMS peut se voir comme celle donnée dans l'exemple de la Figure 16. Nous allons, dans les points suivants, détailler les éléments du schéma en partant du bas (les BD physiques) et en remontant progressivement vers le haut (les Programmes d'Application).

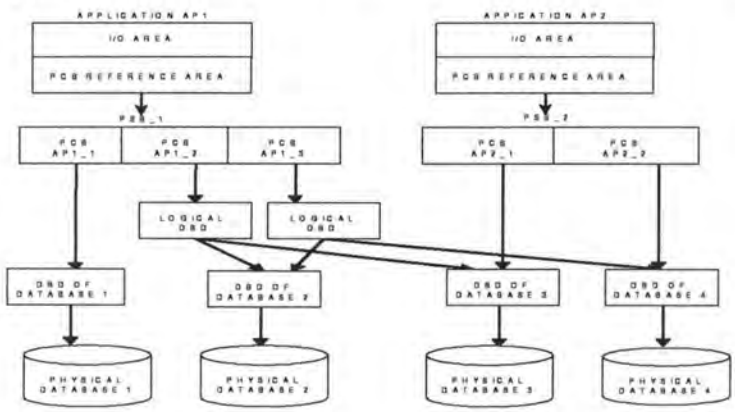

Figure 16 : Exemple d'architecture d'un environnement IMS

Notons pour commencer que les Data Base Description physiques, les Data Base Description logiques et les Program Communication Block constituent le DDL d'IMS. Le Programme d'Application fait quant à lui appel aux instructions du DML.

2.3.2. La Data Base Description

La base de données stockée constitue logiquement l'élément de base du système. Cette base de données générale est en fait constituée d'une ou plusieurs bases de données physiques (quatre dans l'exemple). Les BD physiques sont définies à la construction par

le DBA (Data Base Administrator) et ces définitions sont consignées, en même temps que les méthodes d'accès, dans les Data Base Description (DBD).

Les DBD, écrites en DL/1, donnent donc une description de la base de données. Etant donné que la base stockée est constituée de plusieurs BD physiques, chaque DBD correspond à une et une seule BD physique. La description est accompagnée de la méthode d'accès utilisée pour cette BD. Une DBD peut également décrire une BD logique. Dans ce cas, cette DBD logique repose sur une ou plusieurs DBD physique(s) précédemment définie(s) dans lesquelles nous retrouvons en plus de la description normale les types relations logiques.

2.3.3. Les différentes bases de données

IMS fournit une grande variété de méthode de stockage, ou d'accès, pour ses BD physiques. Il est d'ailleurs commun de qualifier ces BD par leur méthode de stockage. Nous ne nous étendrons pas ici sur ces méthodes mais il nous semble pertinent d'en faire une brève description.

Premièrement, nous pouvons diviser les principales méthodes en deux groupes: les méthodes séquentielles et les méthodes en accès direct. Les premières citées représentent la séquence hiérarchique des segments dépendants par la contiguïté et les secondes par des pointeurs. Deuxièmement, ces deux groupes se redivisent en deux méthodes suivant l'utilisation ou non d'un index pour accéder aux segments racines. Nous avons alors nos quatre principales méthodes: HSAM (Hierarchical Sequential Access Method), HISAM (Hierarchical Indexed Sequential Access Method), HDAM (Hierarchical Direct Access Method) et HIDAM (Hierarchical Indexed Direct Access Method). L'accès au segment racine, pour les méthodes en accès direct, se fait soit par hachage (HDAM) soit par index (HIDAM). Une particularité existe encore pour les méthodes HSAM et HISAM quand la BD ne contient qu'un seul type segment. De ce fait, le préfixe attaché à chaque segment n'est plus obligatoire. Ces deux méthodes supplémentaires sont alors appelées respectivement SHSAM (Simple HSAM) et SHISAM (Simple HISAM).

Pour être complet, ajoutons encore les méthodes DEDB (Data Entry DataBase), MSDB (Main Storage DataBase) et GSAM (Generalized Sequential Access Method). Les deux premières méthodes sont utiles dans les environnements transactionnels lourds nécessitant de grandes performances. La méthode DEDB peut être vue comme une extension de la méthode HDAM qui partitionne la hiérarchie en plusieurs *area*. La méthode MSDB est utilisée pour les BD n'ayant qu'un seul segment, ces derniers étant gardés en mémoire primaire. Cette méthode convient bien, par exemple, pour des tables de références. Enfin, la méthode GSAM permet d'utiliser les fonctions de récupération d'IMS pour des fichiers séquentiels conventionnels. Par conséquent, ces BD GSAM ne contiennent pas de types segments et de champs.

2.3.4. Le Program Communication Block

En IMS, il existe deux types de vues: les Program Communication Block (PCB) et les DBD logiques. Tout d'abord, analysons le premier type de vue c'est-à-dire le PCB.

Chaque PCB correspond donc à une vue d'une DBD. Il peut bien sûr exister plusieurs PCB pour une seule DBD. En fait, cette vue est la plus simple des deux car

elle ne repose que sur une et une seule BD physique. Pour définir ce type de vue, reprenons les règles de [ELMASRI, 89] ci-dessous:

Un PCB est une sous-hiérarchie d'une DBD qui observe ces quatre règles :

1. Le type segment racine doit faire partie de la vue;
2. N'importe quel type segment non racine peut être omis;
3. Si un type segment est omis, alors tous les types segments enfants doivent être omis également;
4. Pour les types segment repris, n'importe quel champ peut être omis.

On en retire que le type segment racine fait à chaque fois partie de la vue et que nous pouvons sélectionner les types segments et les champs désirés. Les types segments et les champs ainsi sélectionnés sont dits **sensibles** (ou sensitifs).

Pour terminer, les avantages des PCB sont, entre autre, de garantir un certain degré de contrôle pour la sécurité des données et de protéger l'utilisateur de changements ne le concernant pas dans la base.

2.3.5. La Logical Data Base Description

Les DBD logiques (deux dans l'exemple) constituent le deuxième type de vue en IMS. Une base de données logique est définie à l'aide d'une DBD dont on spécifie l'accès comme logique. Elle se base sur une ou plusieurs BD physique(s) contenant des types relations logiques, ce qui lui confère le statut de vue.

Etant donné qu'IMS nous fournit deux types de vues opposés, il est intéressant de mettre à jour leurs différences. Ceci nous est donné par [DATE, 90] :

- Une base de données logique peut être définie sur plusieurs DBD physiques, tandis qu'un PCB doit être défini sur une seule DBD;
- Une vue définie par une DBD logique peut diverger considérablement de la structure physique sous-jacente, au contraire d'une vue définie par un PCB;
- Contrairement à la vue d'un PCB, la vue d'une DBD logique est directement supportée par ses propres chaînes de pointeurs physiques. Par conséquent, il faut voir les BD logiques comme de vraies structures physiques bien que les types segments appartiennent à des BD physiques.

Notons aussi que sur le schéma de la Figure 16, les DBD logiques sont rattachées à un PCB. On doit donc définir un PCB sur une DBD logique. Cela peut paraître déroutant car nous venons de les différencier et de les classer distinctement. Pour comprendre cela, il ne faut pas seulement voir les DBD logiques comme des vues mais aussi comme des BD à part entières, même si elles sont virtuelles. Un programme d'application ne pouvant accéder à une DBD qu'au moyen d'un PCB, il est donc normal d'avoir un PCB sur une DBD logique.

2.3.6. Le Program Specification Block

Le Program Specification Block (PSB) contient un ou plusieurs Program Communication Block (PCB).

Il sert d'interface entre le programme d'application (PA) et les DBD et indique, par l'intermédiaire de ses PCB, les DBD auxquelles le PA peut accéder. Le PSB ne doit pas

seulement se voir comme le collecteur des différentes vues du programme. C'est un élément important car il sert de tampon entre le PA et IMS. Il est donc chargé à chaque exécution du PA.

2.3.7. Le Programme d'Application

Le Programme d'Application est le siège d'interrogation de la base. Il est écrit en langage hôte et fait appel aux instructions du Data Manipulation Language (DML) par le biais de CALL. Prenons un exemple (Figure 17) basé sur [DATE, 90] et prenant comme langage hôte le PL/1:

```
CALL PLITDLI (SIX, GU, DEPPCB, PERSONNE_AREA, DSSA, CSSA, PSSA) ;
```

Figure 17 : Exemple de Call basé sur la Figure 2

Ce *call* a pour but de retrouver une personne dans la base de données de la Figure 2. On peut décortiquer l'appel comme suit :

- PLITDLI (" PL/1 To DL/1 ") identifie le point d'entrée du programme;
- SIX représente une variable numérique en PL/1 dont la valeur (supposée 6) représente le nombre d'arguments du *call*, excluant le compteur lui-même;
- GU représente une variable chaîne de caractère dont la valeur (supposée GU) spécifie l'opération DL/1 à exécuter;
- DEPPCB représente ce que l'on appellerait en SQL un curseur. En IMS, ce type de zone est défini pour chaque PCB et contient différentes informations sur le déroulement de la requête;
- PERSONNE_AREA correspond à une variable PL/1 dans laquelle le segment PERSONNE désiré sera renvoyé. C'est donc la zone d'entrée/sortie du programme (I/O area);
- DSSA, CSSA et PSSA sont des **Segment Search Arguments (SSA)**. Ce sont des variables PL/1 dont la valeur est le nom d'un segment et un ensemble de restrictions. Ces variables permettent donc de sélectionner un segment particulier. Par exemple :

```
DSSA : DEPARTEMENT (NUMERO = '1')
CSSA : CLASSE-PERSONNE (NOM = 'Développeur')
PSSA : PERSONNE (NUMERO = '1')
```

Figure 18 : Exemple de Segment Search Arguments basé sur la Figure 2

Les *calls* avec un ou des SSA(s) sont dits **qualifiés** et les *calls* sans SSA sont dits **non qualifiés**.

En ce qui concerne les opérations de DML permises par DL/1, en voici la liste avec une brève description:

Get Unique (GU): renvoie le premier segment correspondant à la recherche;
Get Next (GN): renvoie le segment suivant la position courante dans la BD;
Get Next Within Parent (GNP): renvoie le segment enfant du même parent suivant la position courante;
Get Hold Unique (GHU), **Get Hold Next (GHN)** et **Get Hold Next Within Parent (GHNP)**: agissent comme les trois précédantes mais en gardant le segment pour un effacement ou une mise à jour;
Insert: insère un nouveau segment;
Delete: efface le segment retiré avec un Get Hold;
Replace: remplace le segment retiré avec un Get Hold par le nouveau segment modifié.

2.3.8. Les index secondaires

Afin de ne pas limiter l'accès aux segments par le seul moyen de la hiérarchie, IMS fournit une possibilité d'index secondaires. Il est alors possible de mettre un index sur un type segment quelconque et ainsi de ne plus avoir l'obligation de passer par la structure hiérarchique pour atteindre les segments.

IMS fournit deux types d'index secondaires:

- un index qui donne l'accès à n'importe quel type segment sur base de la valeur de n'importe quel(s) champ(s) de ce type segment;
- un index qui donne l'accès à un type segment donné sur base de la valeur de n'importe quel(s) champ(s) d'un autre type segment d'un niveau inférieur.

Une particularité amenée par les index secondaires est la restructuration de la hiérarchie. En effet, quand le type segment indexé est un type segment dépendant, alors l'index secondaire est structuré selon les règles suivantes [DATE, 90]:

- le type segment indexé devient le type segment racine;
- les types segments ancêtres de ce type segment deviennent les types segments dépendants les plus à gauche dans la hiérarchie, et en ordre inverse. Cela veut dire que le type segment parent du type segment indexé devient son type segment enfant le plus à gauche. On ajoute ensuite à ce dernier comme type segment enfant son type segment parent, et ainsi de suite jusqu'au type segment racine qui est alors un type segment feuille;
- les types segments dépendants du type segment indexé apparaissent comme avant, excepté qu'ils se trouvent à la droite des types segments introduits par la règle précédente;
- enfin, aucun autre type segment n'est inclus.

Ces règles sont édictées pour la structuration des instructions du PCB qui définit une vue sur l'index secondaire. La vue du PCB n'est alors plus une stricte sous-hiérarchie de la hiérarchie de la BD. Ceci est illustré par la Figure 19 où la restructuration est due à un index secondaire visant le type segment Personne de la hiérarchie de la Figure 2.

Un index secondaire est une BD à part entière qui peut être traitée indépendamment des données qu'elle indexe. Il est alors possible d'ignorer la BD indexée. Pour ce faire, certains champs du type segment indexé peuvent être copiés dans la BD index secondaire. IMS se charge alors de mettre à jour ces champs dupliqués quand le segment source est modifié.

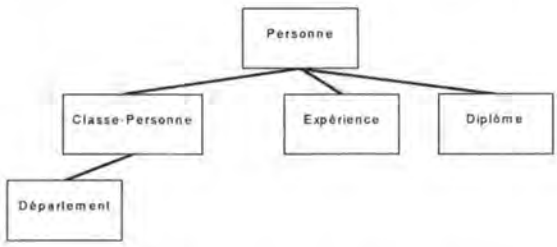

Figure 19 : Restructuration par un index secondaire

Chapitre 3 : L'extraction des structures de données

3.1. Introduction

Dans ce chapitre, nous allons tenter d'établir la correspondance entre les éléments d'IMS et leur contrepartie conceptuelle. Ce chapitre découle donc de la phase d'extraction des structures de données. Durant cette phase, l'analyste retire toutes les informations utiles des sources à sa disposition. La source principale est le code DDL. Ce code est analysé au point suivant et permet de jeter les bases du schéma optimisé orienté SGBD IMS. En effet, les informations que l'on peut y trouver sont essentielles et relativement complètes. Nous essayerons donc de retrouver le plus d'informations possibles de cette source. Les autres sources apporteront, ensuite, des informations supplémentaires au schéma ou viendront confirmer certaines informations déjà présentes.

Pour notre analyse, nous avons pris comme environnement de référence l'outil Case DB-MAIN développé ici à l'Institut. Nous nous limiterons donc à la modélisation offerte par cet outil qui est basée sur le modèle Entité_Association. Bien sûr, le lecteur avisé pourra faire la correspondance avec tout autre modèle de sa connaissance.

Dans la description des instructions, la syntaxe répond aux conventions suivantes:

- les mots en lettres capitales et les caractères spéciaux doivent apparaître tel qu'écrits;
- les mots en lettres minuscules doivent être remplacés par une valeur de l'utilisateur. Ces valeurs sont des valeurs numériques ou des noms alphanumériques de un à huit caractères;
- [] indique des mots clés ou des paramètres facultatifs;
- {} indique qu'un choix de paramètres doit être fait. Les paramètres sont listés verticalement, ou horizontalement avec un blanc comme séparation;
- ,... indique que plusieurs ensembles de paramètres peuvent être décrits pour le mot clé;
- Dans le cas d'un choix à faire (ex:{FIRST NEXT LAST}), le paramètre par défaut est souligné.

3.2. Apport du DDL pour l'extraction

Le DDL est essentiel en IMS. C'est par lui que le DBA (Data Base Administrator) décrit les bases de données utilisées et les éléments les composant.

Par mesure de clarté, nous avons séparé les situations dans lesquelles se trouvent les différentes instructions en quatre grandes parties: les bases de données physiques, les bases de données logiques, les bases de données index et les vues (PCB). Pour chaque instruction, nous nous efforçons d'en donner la syntaxe la plus précise possible ainsi que sa fonction dans le système. Une analyse détaillée de cette syntaxe est alors exposée. Suite à cela, nous précisons les apports au schéma que cette instruction procure.

Notons également que certains mots clés possèdent une abréviation équivalente. En voici la liste avec les abréviations respectives entre crochet: POINTER[PTR], FIRST[F], LAST[L], HERE[H], KEY[K], DATA[D], VIRTUAL[V], PHYSICAL[P], HIER[H], HIERBWD[HB], TWIN[T], TWINBWD[TB], NOTWIN[NT], LTWIN[LT], LTWINBWD[LTB], LPARNT[LP], CTR[C], SINGLE[S], MULTIPLE[M], YES[Y] et NO[N].

Une hypothèse importante que nous avons pris en compte pour l'extraction est la conformité et la correction des sources analysées par rapport au langage DL/1. Nous entendons par là que ces sources sont passées à la compilation et ont été reconnues sans erreurs. Cela implique que nous ne traitons pas les erreurs , et donc que l'extracteur s'attend à recevoir une syntaxe correcte pour les instructions.

D'autre part, suivant la méthode d'accès de la BD, la syntaxe des instructions de la description s'en trouve modifiée. En effet, certains mots clés seront obligatoires suivant telle méthode d'accès et seront facultatifs ou inexistants pour telle autre méthode d'accès. Afin de simplifier la syntaxe exposée, nous avons indiqué comme obligatoires les mots clés qui étaient obligatoires pour toutes les méthodes. Pour les autres mots clés (obligatoires, facultatifs ou absents suivant la méthode d'accès), ils seront simplement indiqués comme facultatifs. Etant donné l'hypothèse de correction des sources évoquée ci-dessus, l'extracteur ne cherche pas à savoir si le mot clé doit être présent ou pas. Signalons également qu'une description complète des syntaxes précises exigées par les différentes méthodes d'accès se trouve en Annexe A.

3.2.1. Les bases de données physiques

Le schéma de la Figure 20 montre la séquence d'enchaînement des instructions DL/1 dans la description d'une BD physique.

```
[PRINT]
DBD                    - Figure 21
{DATASET/AREA}         - Figure 22/Figure 23
                       ┌──────────────────────────┐
                       │SEGM      - Figure 24     │
                       │LCHILD    - Figure 31     │
                       │FIELD     - Figure 27     │
                       │...                       │
                       │FIELD                     │
                       │...                       │
                       └──────────────────────────┘

                       ┌──────────────────────────┐
                       │SEGM                      │
                       │...                       │
                       │...                       │
                       └──────────────────────────┘
                       ...

DATASET
                       SEGM
                       ...
                       ...
                       ...

DBDGEN
[FINISH]
END
```

Figure 20 : Enchaînement des instructions DL/1 pour une DBD physique

Plusieurs remarques sont à faire pour éclaircir le schéma. Les instructions PRINT, DBD, DBDGEN, FINISH et END n'apparaissent qu'une seule fois. La première et les trois dernières ne seront pas étudiées car elles correspondent à des instructions de contrôle et de génération lors de la compilation du schéma. La description de la BD commence obligatoirement par DBD et se termine par END. Il convient également de signaler que l'ordre des instructions est important. Ainsi, la suite des instructions SEGM indique l'ordre hiérarchique de la BD. Autre remarque: les instructions FIELD et LCHILD sont attachées à la définition d'un type segment. L'instruction DATASET, à laquelle sont rattachées les instructions SEGM, peut apparaître plusieurs fois dans la description de la BD. Pour terminer, il faut rajouter quelques informations de limitation. Le nombre maximum de niveaux dans la structure hiérarchique est de 15, et il ne peut y avoir plus de 255 types segments dans la DBD. Il peut y avoir de 0 à 255 champs dans chaque type segment, avec un maximum de 1000 champs (FIELD et XDFLD) dans la DBD.

3.2.1.1. L'instruction DBD

Cette instruction identifie la base de donnée physique et en décrit l'organisation. Son format est représenté à la Figure 21 ci-dessous.

```
DBD
                NAME=(dbname1)
               ,ACCESS={HSAM
                        SHSAM
                        (HISAM [,VSAM])
                        (SHISAM [,VSAM])
                        (GSAM [,{BSAM VSAM}])
                        (HDAM [,{OSAM VSAM}])
                        (HIDAM [,{OSAM VSAM}])
```

```
                    MSDB
                    DEDB
                    (INDEX[,VSAM][,DOSCOMP])
                    }
[,EXIT=((
   [{*
     exitname [,{KEY NOKEY}]
                [,{NOPATH PATH}]
                [,{DATA NODATA}]
   }]
   [,{CASCADE
      (CASCADE [,{KEY NOKEY}]
               [,{DATA NODATA}]
               [,{NOPATH PATH}])
      NOCASCADE
   }]
   [{LOG NOLOG}]),...)]
[,VERSION='n']
[,RMNAME=(random[,numrap
          ,sizeblock,sizeraa])]
[,PASSWD={YES
          NO}]
```

Figure 21 : Syntaxe de l'instruction DBD

NAME: *dbname1* est le nom de la BD. Ce nom doit être unique dans tout l'environnement DL/1.

ACCESS: spécifie la méthode d'accès DL/1 et la méthode d'accès de l'OS à utiliser avec cette base de données. Le schéma montre clairement les différentes possibilités offertes et l'explication des différentes méthodes n'entre pas dans le cadre de l'exposé. Nous ne nous étendrons donc pas sur le fonctionnement de ces méthodes.

EXIT: spécifie qu'une ou plusieurs routine(s) de capture de données est(sont) utilisée(s) sur l'ensemble des types segments de la BD et permette(nt) à des utilisateurs finals DB2 d'accéder à la BD. Les options paramètrisent l'appel à cette routine.

VERSION: permet de donner une version à la DBD par la chaîne de caractères *n*. Si cette dernière n'est pas présente, le générateur produit automatiquement un estampillage.

RMNAME: est utilisé pour les bases de données en accès HDAM. Ce mot clé spécifie les paramètres liés au hachage:

1. *random* indique le nom de la fonction de hachage.
2. *numrap* indique le nombre de Root Anchor Point (RAP) dans un Control Interval (CI).[3]
3. *sizeraa* indique le nombre de CI de la RAA (Root Adressable Area).[4]
4. *sizeblock* indique le nombre maximum d'octets d'un enregistrement qui sont stockés dans le CI lors de l'insertion.[5]

---

[3] Le CI correspond en VSAM au bloc de données dans lequel sont stockés les enregistrements. Dans le cas où la fonction de hachage renvoie la même valeur pour deux ou plusieurs enregistrements différents, ces enregistrements se retrouvent dans le même CI. Le RAP a comme fonction de pointer sur le premier enregistrement du CI qui lui-même pointe sur le deuxième enregistrement et ainsi de suite. Les enregistrements sont alors reliés par une chaîne de pointeur dont le pointeur initial est le RAP. Dans le cas où beaucoup d'enregistrements se retrouvent dans le même CI, le DBA peut spécifier l'emploi de plusieurs RAP afin de diminuer le temps de recherche. Plusieurs chaînes de pointeurs sont alors disponibles. C'est la fonction du paramètre *numrap*.

[4] La RAA est la partie d'une base de données HDAM dans laquelle sont hachés les segments racines. L'autre partie d'une telle base de données est appelée l'*Overflow Area*. Nous ne nous étendrons pas sur ce sujet. Toutefois, il faut estimer une bonne taille initiale à la RAA pour éviter un recours trop répétitif à l'*Overflow Area* qui signifierait que la BD arrive à saturation. Dans ce but, le paramètre *sizearea* indique la taille de la BD en terme de nombre de CI dans la RAA.

PASSWD: oblige l'*open* DL/1 à utiliser le nom de *dbname1* comme mot de passe VSAM à l'ouverture de n'importe quel ensemble de données de cette base. Cette option n'est donc valide que si la méthode d'accès de l'OS a été spécifiée comme étant VSAM. Cela permet d'interdire l'accès aux données par des programmes non-DL/1.

L'apport de cette instruction au schéma brut est qu'elle permet d'identifier une hiérarchie de types d'entités. Le mot clé NAME est utilisé pour donner un nom à cette hiérarchie.

En ce qui concerne les méthodes d'accès, elles nous permettent tout d'abord d'avoir une information technique importante relative au *design* physique. En fait, le choix d'une méthode d'accès DL/1 et OS est fait par le DBA pour des questions d'efficience d'accès. D'autre part, elles nous renseignent une quantité importante d'informations. Premièrement, elles limitent la syntaxe des instructions contenues dans la description de la BD (comme évoqué plus haut). Deuxièmement, elles nous indiquent la présence de clés d'accès. Une description complète des informations fournies par la connaissance de la méthode d'accès est présentée dans l'Annexe B. Les indications sur les clés d'accès sont les suivantes:

- HISAM, SHISAM: Présence d'un champ de séquence unique sur le type segment racine servant de clé d'accès;

- HIDAM: En ce qui concerne les bases de données en accès HIDAM, elles comprennent en fait deux DBD. La première concerne la BD proprement dite et la seconde la BD comprenant l'indexation. Nous les appellerons respectivement la BD HIDAM et la BD index. Nous sommes donc informés de la présence d'une autre base de données servant d'index. Par le fait de l'accès par un index, la présence d'un champ de séquence unique sur le type segment racine est également indubitable;

- GSAM: Ce type d'accès est utilisé uniquement pour les facilités de gestion de récupération d'IMS appliquées à un fichier séquentiel. La BD ne possède pas de hiérarchie et les instructions SEGM et FIELD ne sont pas présentes. Par conséquent, une BD utilisant l'accès GSAM n'apporte strictement aucune information conceptuelle. L'apport au schéma est donc nul;

- INDEX: Ce paramètre indique que la base de données est une BD servant d'index. Cette BD contient soit un ou plusieurs index secondaire(s), soit l'index d'une base de données en accès HIDAM. Dans le deuxième cas, elle nous indique la présence d'une base de données HIDAM à laquelle elle est reliée. Les informations pertinentes se trouvant dans la BD HIDAM, la BD index ne donne pas naissance à une hiérarchie de type d'entités. En ce qui concerne le premier cas, nous pouvons obtenir des renseignements sur une ou plusieurs clé(s) d'accès;

- DEDB: Présence d'un champ de séquence unique sur le type segment racine;

- MSDB: Pour les BD non reliées-terminal, présence d'un champ de séquence unique sur le type segment racine.

---

[5] Le reste de l'enregistrement est alors stocké dans l'*Overflow Area*. Ceci a pour but de ne pas encombrer les CI avec des segments dépendants trop longs ou ayant trop d'occurrences. Ainsi, dans le cas de grands enregistrements, les données dépendantes n'encombrent pas les CI. Cela permet à d'autres enregistrements de trouver place dans le CI correspondant à leur fonction de hachage.

Le mot clé VERSION donne la version de la DBD analysée. C'est donc une information sémantique portant sur la hiérarchie de types d'entités. Le mot clé RMNAME et ses paramètres nous renseignent sur la méthode de hachage utilisée. Le mot clé PASSWD donne une information de sécurité technique. Ces deux derniers éléments sont des éléments de conception technique importants. Le concept de hiérarchie n'étant pas modélisé dans notre environnement de référence, les informations sont alors attachées à la description technique et sémantique du type d'entité correspondant au type segment racine de la BD.

En ce qui concerne le schéma dans lequel se trouve la hiérarchie, il convient d'en dire quelques mots. Lors de l'analyse d'une déclaration de BD, si aucun schéma n'est présent dans notre environnement de travail, alors il faut en créer un. Si un schéma était déjà présent, alors la hiérarchie de types d'entités est incorporée à ce schéma.

3.2.1.2. L'instruction DATASET

Cette instruction identifie le support physique sur lequel les types segments et leurs champs sont stockés. Elle indique également plusieurs réglages du stockage physique. Suivant la méthode d'accès, il peut y avoir un ou plusieurs DATASET par DBD. Le premier DATASET désigne alors le groupe de données primaire et les autres des groupes de données secondaires. Les types segments d'une même BD physique peuvent donc se retrouver sur plusieurs supports différents. Les types segments stockés ensemble doivent suivre l'ordre hiérarchique et sont précédés par l'instruction DATASET. Les liens hiérarchiques liant les types segments se trouvant sur différents supports sont conservés. Le format est le suivant (Figure 22):

```
[Label]
DATASET
                DD1=ddname
                [,DD2=ddname2]
                [,DEVICE=st_dev_type]
                [,MODEL=model_number]
                [,OVFLW=ddname3]
                [,BLOCK=(blk_fact1[,blkfact2])]
                [,SIZE=(size1[,size2])]
                [,RECORD=(reclen1,reclen2)]
                [,SCAN=cyl_num]
                [,FRSPC=(free_block_freq,%free_space)]
                [,SEARCHA={0 1 2}]
                [,REL=({NO
                        TERM[,fldnm]
                        FIXED[,fldnm]
                        DYNAMIC[,fldnm]})]
```

Figure 22 : Syntaxe de l'instruction DATASET

Label: Dans certains cas, il se peut que l'on veuille rassembler des types segments sur le même support mais en prenant en compte un autre critère que celui de l'ordre hiérarchique (par ex. la taille du type segment ou sa fréquence d'accès). Il suffit alors de labeliser les différents DATASET. Les types segments compris dans des DATASET ayant le même *Label* sont alors rassemblés sur le même support et la description de ce dernier se fait uniquement dans le premier DATASET.
DD1: identifie le DATASET par un nom (*ddname*). Son utilisation par IMS dépend du type de BD auquel il se rattache.
DD2: donne le nom (*ddname2*) du DATASET en sortie requis pour les mises-à-jour des bases de données HSAM et SHSAM.
DEVICE: spécifie le type de support utilisé pour le stockage des données.

MODEL: spécifie le modèle du type de support.
OVFLW: donne le nom de la zone de débordement pour certaines BD.
BLOCK: indique soit les facteurs de blocage, soit la taille du bloc, ou soit la taille de l'intervalle de contrôle suivant le type de BD.
SIZE: spécifie la taille de l'intervalle de contrôle pour les ensembles de données VSAM et/ou la taille du bloc pour les ensembles de données OSAM.
RECORD: spécifie la taille logique des enregistrements du DATASET et de la zone de débordement.
SCAN: spécifie, pour les BD en accès direct, le nombre d'accès directs à effectuer lors de la recherche d'espace pour les insertions. Si la limite est atteinte, le segment est inséré à la fin de l'espace de stockage.
FRSPC: paramètrise l'espace laissé libre lors du chargement ou de la réorganisation des bases de données HDAM ou HIDAM. Le paramètre *free_block_freq* indique le nombre de blocs ou de CI à remplir avant de laisser un bloc ou un CI libre. Le deuxième paramètre, *%free_space*, spécifie le pourcentage minimum de chaque bloc ou CI qui doit être laissé libre.
SEARCHA: spécifie le type de recherche d'espace libre pour les BD en accès direct.
REL: spécifie si la BD MSDB est reliée_terminal ou non. Il n'y a pas de propriété de l'occurrence d'un type segment dans les MSDB non_reliées_terminal.

Cette déclaration définit donc le type de support sur lequel seront stockés les types segments s'y rattachant. La contribution au schéma se fait sous la forme d'une collection dans laquelle se retrouve les types d'entités provenant des types segments du DATASET. Les paramètres de stockage sont des paramètres physiques et se retrouvent par conséquent dans la description technique de la collection. Toutefois, une exception est à faire en ce qui concerne les BD MSDB, SHSAM et SHISAM. Celles-ci ne possédant qu'un seul type segment, il n'est pas pertinent d'en faire une collection. Les renseignements fournis par le DATASET sont alors intégrés dans la description technique du type d'entité correspondant à l'unique type segment. Pour les autres BD, même si elles ne possèdent réellement qu'un seul type segment, une collection est créée.

3.2.1.3. L'instruction AREA

Une BD DEDB utilise l'instruction AREA pour définir ses régions ("*area*") dans la BD. Cette instruction est en fait l'équivalent du DATASET pour les autres BD. De ce fait, elle suit la même règle: dans l'enchaînement des instructions, toutes les instructions AREA doivent se trouvées après l'instruction DBD et les instructions SEGM qui suivent font partie de cette région. Au moins une instruction AREA est requise et au plus 240 sont permises. Le format est le suivant (Figure 23):

```
AREA
        DD1=ddname1
        ,SIZE=size
        ,UOW=(num1,overfl1)
        ,ROOT=(num2,overfl2)
```

Figure 23 : Syntaxe de l'instruction AREA

DD1: donne le nom de l'*area*.
SIZE: spécifie la taille de l'intervalle de contrôle.
UOW: spécifie le nombre d'intervalles de contrôle dans une unité de travail (*num1*) et dans la zone de débordement de l'unité de travail (*overfl1*). Avec ces deux valeurs, nous pouvons aussi déduire le nombre de *Root Anchor Point*.

ROOT: donne les caractéristiques d'une *area* DEDB. Celles-ci sont l'espace total alloué à la partie adressable racine (*num2*) et l'espace réservé au débordement, le tout en terme d'unité de travail.

A l'instar de l'instruction DATASET, cette instruction permet d'identifier une collection dans laquelle se retrouve les types d'entités provenant des types segments attachés à l'instruction AREA. Les données techniques fournies par les mots-clés sont insérées dans la description technique de la collection.

3.2.1.4. L'instruction SEGM

Cette instruction est utilisée une fois pour chaque type segment à définir dans la description de la BD. Elle donne les renseignements concernant les types segments et son format est le suivant (Figure 24):

```
SEGM
                NAME=segname1
                [,PARENT={0
                          (segname2[,{SNGL
                                      DBLE}])}]
                ,BYTES=maxbytes[,minbytes]
                [,TYPE={DIR SEQ}]
                [,FREQ=freq]
                [,POINTER=[{HIER
                           HIERBWD
                           TWIN
                           TWINBWD
                           NOTWIN}]]
                [,RULES=({LAST FIRST HERE})]
                [,SSPTR=n]
                [,EXIT=((
                   [{*
                     exitname [,{KEY NOKEY}]
                              [,{NOPATH PATH}]
                              [,{DATA NODATA}]
                   }]
                  [,{CASCADE
                     (CASCADE [,{KEY NOKEY}]
                              [,{DATA NODATA}]
                              [,{NOPATH PATH}])
                     NOCASCADE
                   }]
                  [{LOG NOLOG}]),...)]
                [,COMPRTN={(routname,{DATA KEY},INIT)
                          (routname[{[,DATA],INIT
                                    ,,INIT}])}]
```

Figure 24 : Syntaxe de l'instruction SEGM

NAME: donne le nom du type segment. C'est par ce nom que le type segment est référencé par les programmes d'application et le DL/1. Chaque nom de type segment doit être unique dans l'environnement DL/1.

PARENT: spécifie le nom du type segment parent physique (*segname2*) de ce type segment. Le mot clé PARENT pour le type segment racine est soit omis soit indique la valeur 0. Le paramètre SNGL ou DBLE indique le type de pointeur utilisé par les segments parents pour accéder aux segments enfants. Le paramètre SNGL spécifie qu'un pointeur de type *Physical Child First* est placé dans le préfixe du segment parent et pointe vers le premier segment enfant. Le paramètre DBLE spécifie qu'un pointeur de type *Physical Child First* et un pointeur de type *Physical Child Last* sont placés dans le préfixe du segment parent et pointent respectivement sur le premier et le dernier segment enfant.

BYTES: donne la longueur en octets de la partie donnée du type segment (*maxbytes*). Pour définir des types segments en longueur variable, il suffit de spécifier les deux valeurs (*maxbytes* et *minbytes*) qui en donne alors la taille minimale et maximale.
TYPE: décrit le type des types segments dépendants, séquentiel (SEQ) ou direct (DIR), pour les BD DEDB.
FREQ: donne la fréquence moyenne des occurrences du type segment. Cela permet à DL/1 de déterminer les exigences d'espace de stockage.
POINTER: spécifie le type de la chaîne de pointeurs utilisée pour chaîner les segments. La signification des valeurs est indiquée ci-dessous et l'Annexe C nous montre l'utilisation correcte du paramètre suivant les différentes bases de données.

- **HIER**: chaîne de pointeurs en avant sur la séquence hiérarchique;
- **HIERBWD**: chaîne de pointeurs en avant et en arrière sur la séquence hiérarchique;
- **TWIN**: chaîne de pointeurs jumeaux en avant sur la séquence des occurrences;
- **TWINBWD**: chaîne de pointeurs jumeaux en avant et en arrière sur la séquence des occurrences;
- **NOTWIN**: seulement une occurrence permise pour ce type segment, donc pas de pointeur nécessaire.

RULES: spécifie où les segments sont insérés c'est-à-dire la séquence des jumeaux. Cette valeur est utilisée uniquement lorsque le type segment ne possède pas de champ de séquence unique. Notons aussi que la valeur par défaut pour les BD DEDB est HERE.
SSPTR: spécifie le nombre de pointeurs de sous-ensemble pour les BD DEDB pour garder plusieurs positions sur la BD.
EXIT: spécifie qu'une ou plusieurs routine(s) de capture de données est(sont) utilisée(s) et permette(nt) à des utilisateurs finals DB2 d'accéder à la BD. Les options qui suivent paramètrisent l'appel à cette routine. La ou les routine(s) s'applique(nt) au type segment et écrase(nt) les définitions de routines spécifiées dans l'instruction DBD.
COMPRTN: donne le nom d'une routine de compression des données fournie par l'utilisateur afin de réduire l'espace employé par le segment.

Cette déclaration est d'un apport capital pour le schéma. Chaque segment ainsi défini est transformé en un type d'entité dans le schéma. Le nom du type d'entité est donné par le mot clé NAME. Le mot clé PARENT définit le concept de type d'association *one-to-many* entre ce type d'entité jouant le rôle d'enfant et le type d'entité provenant du type segment *segname2* jouant le rôle de parent. La cardinalité du rôle joué par le type d'entité enfant est de type 1-1 et celle du rôle joué par le type d'entité parent est de type 0-N. Le type d'entité est ainsi inclus dans la hiérarchie des types d'entités. Deux éléments conceptuels fondamentaux sont par conséquent introduits dans le schéma.

Le paramètre SNGL ou DBLE du mot clé PARENT donne une description technique du type d'association. Le mot clé BYTES fournit la somme en octets des attributs du type d'entité. Cette valeur, comparée avec la somme des longueurs des champs, permet de détecter l'absence d'attributs. Le mot clé FREQ donne seulement une indication sur le nombre moyen d'occurrences du type d'entité enfant attachées à une occurrence du type d'entité parent. Il ne peut être interprété comme une limite de cardinalité du rôle joué par le type d'entité parent dans le type d'association *one-to-many*. Toutefois, il peut amener une suspicion sur cette cardinalité. Cette suspicion doit être vérifiée dans l'analyse du code DML et dans les données. Le mot clé POINTER renseigne une information technique du type d'entité à savoir le type d'enchaînement des segments. Cependant, pour la valeur NOTWIN, nous pouvons en déduire une

transformation du type d'association *one-to-many* en un type d'association *one-to-one*. La cardinalité du rôle joué par le type d'entité enfant reste 1-1 et la cardinalité du rôle joué par le type d'entité parent devient 0-1. Le mot clé RULES nous renseigne également sur l'enchaînement des segments et le mot clé COMPRTN fournit le nom d'une routine de compression des valeurs des entités. Ces deux dernières informations sont donc ajoutées à la description technique du type d'entité.

Afin d'illustrer l'apport des trois instructions analysées jusqu'à maintenant, nous allons voir le résultat d'une extraction sur un exemple. Cet exemple constitue une partie remodelée de l'étude de cas du chapitre 5 dont le code se trouve en Annexe D. Sur base de la source de la Figure 25, l'extraction nous donne le schéma de la Figure 26.

Pour commenter ce schéma, nous pouvons tout d'abord remarquer la présence d'une seule hiérarchie alors que nous avons deux descriptions de BD dans la source. Ceci est normal car les BD d'index ne donnent pas lieu à une hiérarchie. Ensuite, la hiérarchie représente bien l'ordre hiérarchique de déclarations des types segments. En effet, le rétro-ingénieur a pu remettre sa hiérarchie dans le bon ordre grâce à la numérotation des types d'associations que nous verrons plus tard. D'autre part, nous constatons également la présence de trois types d'associations *one-to-one*. Ces derniers proviennent du mot clé POINTER des trois types segments concernés possédant la valeur NOTWIN. Pour les autres types d'associations, les cardinalités sont bien celles attendues à savoir O-N et 1-1. Finalement, nous retrouvons les deux collections *ens1* et *ens2* provenant des deux instructions DATASET. La composition de ces collections n'étant pas représentée graphiquement, nous vous reportons pour ce sujet à l'étude de cas du chapitre 5.

```
DBD     NAME=(Personne)
                        ,ACCESS=(HDAM,VSAM)
                        ,RMNAME=(rand1,2,200,500) ,PASSWD=NO
                        ,VERSION=1
DATASET  DD1=ens1,BLOCK=50,SIZE=(,500),SCAN=5,FRSPC=(4,80)
SEGM     NAME=01Depart,PARENT=0,BYTES=53
SEGM     NAME=01Clasper,PARENT=(01Depart,DBLE),BYTES=48,FREQ=7 ,PTR=TWINBWD
SEGM     NAME=01Pers,PARENT=(01Clasper),BYTES=180,FREQ=25,POINTER=TWINBWD
SEGM     NAME=01Diplom,PARENT=(01Pers),BYTES=70,FREQ=5,POINTER=HIERBWD
SEGM     NAME=01Exper,PARENT=(01Pers),BYTES=75,FREQ=12,POINTER=HIERBWD
SEGM     NAME=01DesPer,PARENT=(01Pers),BYTES=200,FREQ=1,POINTER=NOTWIN
SEGM     NAME=01Projet,PARENT=(01Depart),BYTES=70
DATASET  DD1=ens2
SEGM     NAME=01Despjt,PARENT=(01Projet),BYTES=200,PTR=NOTWIN
SEGM     NAME=01Desdep,PARENT=(01Depart),BYTES=20,POINTER=NT
DBDGEN
FINISH
END


DBD     NAME=(Bdindx),ACCESS=INDEX
DATASET  DD1=ddindx,DEVICE=3350
SEGM     NAME=01idxus,BYTES=10
DBDGEN
FINISH
END
```

Figure 25 : Source IMS illustrant l'apport de DBD, DATASET et SEGM

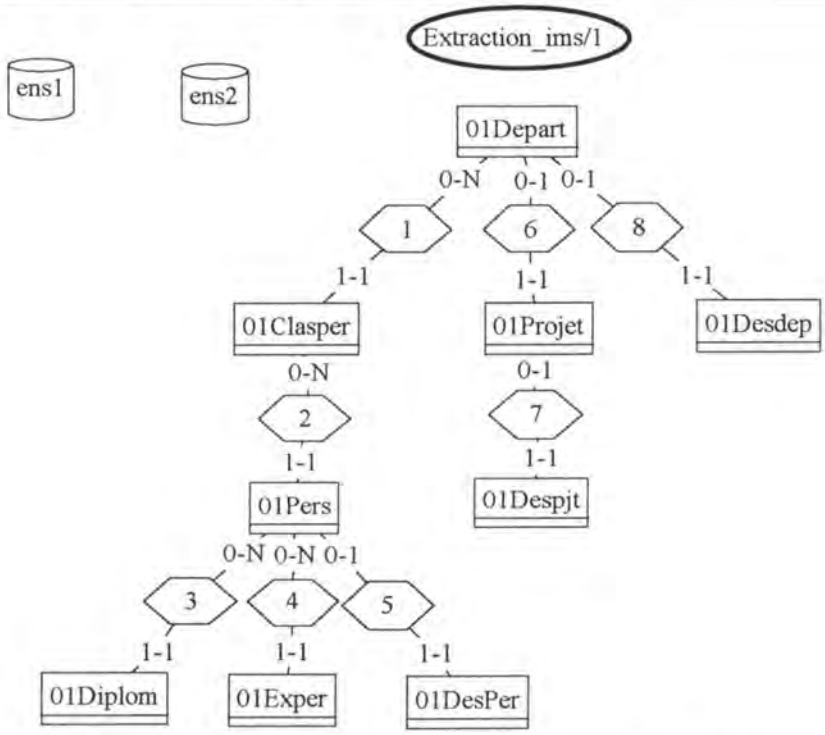

Figure 26 : Résultat de l'extraction de la Figure 25

3.2.1.5. L'instruction FIELD

Cette instruction est utilisée une fois pour chaque champ défini dans un type segment. Elle est obligatoire pour chaque champ référencé dans un PSB et pour chaque champ utilisé dans les SSA. Toutes les déclarations de champs suivent le type segment auquel elles se rapportent. Le format est le suivant (Figure 27):

```
FIELD
                NAME=(fldname[,SEQ[,{M U}]])
                ,BYTES=bytes
                ,START=startpos
                [,TYPE={X P C P H F}]
```

Figure 27 : Syntaxe de l'instruction FIELD

NAME: spécifie le nom du champ défini dans le type segment. Le nom du champ est unique dans le type segment auquel il appartient.
SEQ: indique que ce champ est un champ de séquence dans le type segment. La déclaration d'un champ de séquence doit suivre immédiatement la déclaration SEGM du type segment. Comme règle générale, un type segment ne peut avoir qu'un champ de séquence. Le suffixe U pris par défaut définit les occurrences du champ de séquence comme uniques dans la BD pour les types segments racines et uniques sous un type segment parent donné pour les types segments dépendants. Une exception autorise toutefois les valeurs multiples. Cette exception est matérialisée par l'adjonction du suffixe M à SEQ.
BYTES: spécifie la longueur en octets du champ.
START: spécifie la position de départ du champ dans le type segment. Cette position est calculée à chaque fois par rapport au début du type segment et le premier champ débute à la position 1.
TYPE: spécifie le type de donnée contenue dans le champ: **X**(hexadécimal), **P**(*packed decimal*), **C**(alphanumérique), **F**(mot binaire) et **H**(demi-mot binaire).

Cette déclaration nous amène à la notion d'attribut d'un type d'entité. Son nom nous est donné par le mot clé NAME. Le mot clé SEQ, utilisé sans le suffixe M, spécifie l'attribut comme identifiant et clé d'accès du type d'entité si le type segment est un type segment racine. Si le type segment est un type segment dépendant, alors l'identifiant du type d'entité est le champ de séquence plus le rôle joué par le type d'entité parent dans le type d'association le liant avec le type d'entité analysé. Le mot clé SEQ utilisé avec le suffixe M spécifie l'attribut comme une clé de tri qui est indiquée dans la description technique du type d'entité. Si nous trouvons sur un type segment racine plus d'un champ possédant le mot clé SEQ sans le suffixe M, alors le premier champ est l'identifiant primaire et les autres champs sont des identifiants secondaires. Pour les types segments dépendants, la présence de deux ou plusieurs autres champs avec le mot clé SEQ sans le suffixe M donne également lieu à la présence d'identifiants secondaires. Evidemment, ces identifiants sont également composés du rôle évoqué ci-dessus.

Le mot clé BYTES donne la longueur de l'attribut. Le TYPE fournit le type de l'attribut. Le mot clé START donne une information technique pour l'attribut à savoir sa position parmi les autres attributs du type d'entité.

Par ailleurs, la combinaison des mots clés BYTES et START pour plusieurs champs permet de retrouver le concept d'attribut décomposable en analysant la superposition des valeurs. Si une telle décomposition est trouvée, elle est modélisée dans le schéma. Dans certains cas, il se peut que la décomposition ne soit pas complète. Par exemple, un champ ADRESSE de 50 octets et ne possédant comme champ composant que RUE de 20 octets et NUMERO de 5 octets est un champ décomposable mais dont la décomposition est incomplète. Dans ce cas, l'incomplétude est renseignée dans la description sémantique de l'attribut décomposable.

En élargissant l'analyse, nous pouvons dire que cette composition libre des deux mots clés permet de définir plusieurs ensembles d'attributs et donc de définir plusieurs types d'entités dans un seul type segment. Nous avons alors deux groupes d'attributs définissant entièrement le type segment. Toutefois, une telle surdéfinition ne peut donner lieu à la création de deux types d'entités, même si nous remarquons deux groupes de champs pour l'ensemble du type segment. En effet, il ne nous est pas possible de savoir à quel groupe appartiennent les différents champs. Par exemple, si les champs des différents groupes sont mélangés et que deux champs commencent à la même position, comment savoir à quel groupe ils appartiennent? Nous laissons donc aux analyses ultérieures le soin de résoudre cette surdéfinition et les champs sont simplement modélisés comme des attributs atomiques.

Enfin, il nous reste le cas où un champ se pose sur l'intersection de deux autres champs sans les recouvrir complètement (Figure 28). Dans ce cas, nous ne procédons à aucune décomposition car il y a de fortes chances que cette situation soit là dans un autre but. A ce propos, il est à noter que ce cas est fréquent lors de la surdéfinition évoquée à l'instant. Par conséquent, ce champ est transformé en un attribut atomique.

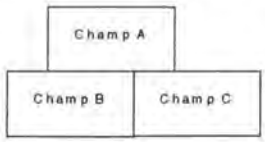

Figure 28 : Exemple de champ recouvrant une intersection de deux autres champs

Pour mieux comprendre l'extraction de l'instruction FIELD, nous allons procéder à une petite illustration. Pour ce faire, nous avons le code source de la Figure 29 et le résultat à la Figure 30.

En regardant le schéma résultat, nous faisons les constatations suivantes. L'attribut *numero* constitue l'identifiant du type d'entité *01Depart*. De plus, il est considéré également comme une clé d'accès. En ce qui concerne le type d'entité *01Clasper*, celui-ci possède comme identifiant la concaténation de l'attribut *ident* et du rôle *1.01Depart* ce qui est normal car c'est un type d'entité dépendant. Dans ce même type d'entité, nous remarquons également la décomposition de l'attribut *denom* en *nom* et *typec*. Enfin, l'attribut *bareme* qui se trouvait à l'intersection des attributs *fonction* et *niveau* ne donne lieu qu'à un attribut atomique normal.

```
DBD     NAME=Personne,ACCESS=(HDAM,VSAM)
DATASET  DD1=ens1
SEGM     NAME=01Depart,PARENT=0,BYTES=53
FIELD    NAME=(numero,SEQ,U),BYTES=3,START=1,TYPE=C
FIELD    NAME=(nom),BYTES=10,START=4,TYPE=C
FIELD    NAME=(chef),BYTES=20,START=14,TYPE=C
FIELD    NAME=(localisa), BYTES=20,START=34,   TYPE=C
SEGM     NAME=01Clasper,PARENT=01Depart,BYTES=123,FREQ=7
FIELD    NAME=(ident,SEQ,U),BYTES=10,START=1,TYPE=C
FIELD    NAME=denom,BYTES=40,START=11,TYPE=C
FIELD    NAME=nom,BYTES=20,START=11,TYPE=C
FIELD    NAME=typec,BYTES=20,START=31,TYPE=C
FIELD    NAME=(fonction),BYTES=25, START=51,TYPE=C
FIELD    NAME=(niveau),BYTES=3,START=77,TYPE=C
FIELD    NAME=bareme,BYTES=5,START=74,TYPE=C
DBDGEN
FINISH
END
```

Figure 29 : Source IMS illustrant l'apport de FIELD

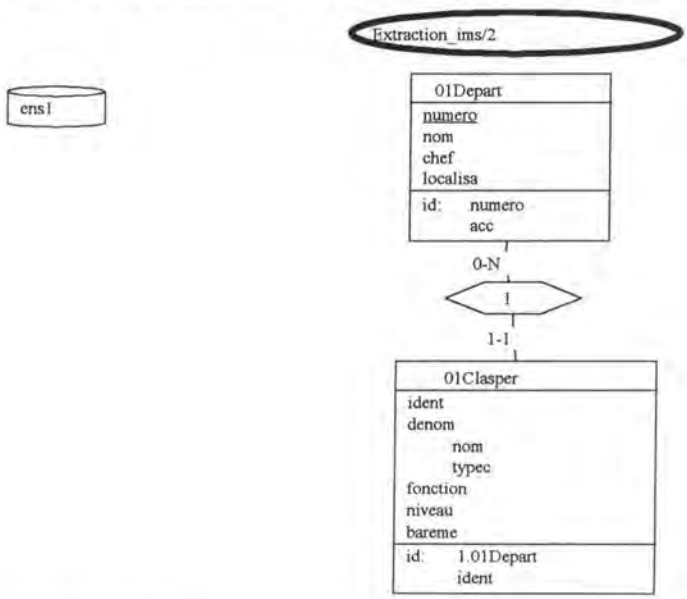

Figure 30 : Résultat de l'extraction de la Figure 29

3.2.1.6. L'instruction LCHILD

Dans le cadre de l'étude des bases de données physiques, nous nous limiterons ici à son utilisation pour référencer la base de données index primaire d'une DBD en accès

HIDAM. Pour ses autres utilisations, il faut se reporter aux points concernant les BD logiques et les index secondaires.

Comme nous l'avons vu précédemment, une base de données HIDAM est composée de deux bases de données, la BD HIDAM et la BD index. L'instruction LCHILD permet de faire le lien entre ces deux bases de données. De ce fait, elle est présente dans les deux BD et son format est le suivant (Figure 31):

```
LCHILD      NAME=(segname,dbname)
                  [,POINTER=INDX]
                  [,INDEX=fldname]
```

Figure 31 : Syntaxe de l'instruction LCHILD

Pour la base de données index, la signification des mots clés est la suivante:

NAME: *segname* est le nom du type segment racine de la BD HIDAM et *dbname* est le nom de cette BD HIDAM.
INDEX: *fldname* est le nom du champ de séquence du type segment racine de la BD HIDAM.
POINTER n'est pas employé ici.

Pour la base de données HIDAM, la signification des mots clés est la suivante:

NAME: *segname* est le nom de l'unique type segment de la BD index et *dbname* est le nom de cette BD index.
POINTER=INDX: indique que le lien est un lien d'index (et pas de type relation logique).
INDEX n'est pas employé ici.

Cette déclaration permet d'établir le lien entre la BD HIDAM et sa BD index. Le nom de la BD index et du type segment de cette BD sont ajoutés à titre indicatif dans la description technique du type d'entité provenant du type segment racine indexé de la BD HIDAM. Les informations concernant le support physique (DATASET) de la base de données index sont également ajoutées à cette description technique.

3.2.1.7. Apport de l'ordre dans les déclarations

L'ordre des déclarations en IMS est fondamental. Premièrement, l'ordre de succession des déclarations des types segments permet de retrouver l'ordre hiérarchique des types d'entités du schéma. Notre environnement de référence ne permettant pas de spécifier un ordre graphique, nous avons choisi de numéroter les types d'associations entre les types d'entités. Au fur et à mesure de l'extraction ordonnée des types segments, les types relations que les types segments ont avec leur type segment parent se voient transformés en types d'associations numérotés incrémentalement. De cette façon, le rétro-ingénieur peut remettre la hiérarchie dans l'ordre graphique correct en se basant sur cette numérotation.

Bien que n'étant pas aussi fondamental, l'ordre de déclaration des champs peut définir une priorité parmi les champs. Dès lors, il est important que cet ordre soit aussi respecté pour les attributs du type d'entité. Ceci se base sur l'hypothèse que l'ordre de déclaration des champs est le même que celui dérivant de l'analyse du positionnement du champ par le mot clé START. Dans le cas où cet ordre diffère, nous prendrons comme référence l'ordre fournit par la déclaration. Nous justifions ceci par le fait que le positionnement physique peut dépendre de besoins en matière de performance ou de

disponibilité de la BD. Dans ce cas, l'ordre de déclaration offre une vision plus conceptuelle et le positionnement une vision plus physique.

3.2.2. Les bases de données logiques

Pour l'implémentation d'une base de données logique, deux phases sont nécessaires:

- la déclaration d'un ou plusieurs types relation(s) logique(s). Pour ce faire, les instructions vues précédemment sont étendues pour permettre aux types relations logiques de s'intégrer aux DBD des BD physiques concernées;
- une description de BD logique se basant sur les définitions des types relations logiques. Cette description utilise les mêmes instructions que pour les DBD physiques.

Nous allons tout d'abord analyser la description d'une relation logique. Nous procéderons ensuite à l'analyse de la description d'une base de données logique.

3.2.2.1. La déclaration d'une relation logique

Par rapport à la description d'une base de données physique, les instructions suivantes restent inchangées: DBD, FIELD, DBDGEN, FINISH et END.

Les instructions SEGM et LCHILD sont étendues pour certains paramètres. Nous n'indiquons ici que les paramètres enrichis. Les autres paramètres non présents restent bien sûr valables.

Comme nous l'avons déjà vu, une relation logique implique trois ou quatre types segments. Nous allons premièrement décrire les modifications de l'instruction SEGM pour ces quatre types segments. Ensuite, l'instruction LCHILD sera analysée.

L'instruction SEGM pour l'enfant logique réel et ses parents

Le format de l'instruction SEGM pour l'enfant logique réel est le suivant (Figure 32):

```
SEGM
                    NAME=...
                    ,PARENT=(segname2[,{SNGL DBLE},]
                    [(segname3[,{VIRTUAL PHYSICAL}][,dbname2])])
                    ,BYTES=...
                    ,POINTER=[{HIER
                              HIERBWD
                              TWIN
                              TWINBWD
                              NOTWIN},{LTWIN
                                       LTWINBWD},]{LPARNT
                                                  CTR
                                                  PAIRED}
                    ,RULES=({L P V}{L P V B}{L P V},{LAST
                                                    FIRST
                                                    HERE})
```

Figure 32: SEGM pour l'enfant logique réel et ses parents

NAME: a la même signification que précédemment.
PARENT: *segname2* est le nom du type segment parent physique de ce type segment enfant logique. Le paramètre SNGL ou DBLE a la même signification qu'avant. *Segname3* est le nom du type segment parent logique de ce type segment enfant logique.

Le paramètre PHYSICAL indique le stockage de la clé concaténée du segment parent logique dans le préfixe du segment enfant logique. Le paramètre VIRTUAL indique que seules les données d'intersection sont stockées. *Dbname2* est le nom de la DBD dans laquelle se trouve le type segment parent logique. Ce nom peut être omis si le parent se trouve dans la même DBD.

BYTES: a la même signification que précédemment. Notons cependant que lorsque la clé concaténée est stockée, sa longueur doit être prise en compte et inclue ici.

POINTER: spécifie les différents types de pointeurs utilisés. En ce qui concerne la chaîne de pointeurs pour les segments physiques, la signification des valeurs est la même que précédemment. Les paramètres LTWIN et LTWINBWD nous renseignent sur le type de chaîne de pointeurs utilisé dans le cadre d'un type relation logique bidirectionnel virtuel. Ces pointeurs chaînent alors les segments enfants logiques réels. Le paramètre LTWIN spécifie que seul un pointeur vers l'avant est utilisé. Le paramètre LTWINBWD spécifie qu'un pointeur vers l'avant et un pointeur vers l'arrière sont utilisés.

Les paramètres LPARNT, CTR et PAIRED ne sont pas mutuellement exclusifs, c'est-à-dire que nous pouvons les retrouver indifféremment tous les trois. Ils sont alors séparés par des virgules. Le paramètre LPARNT indique qu'un pointeur vers le segment parent logique est inséré dans le préfixe du segment. Le paramètre CTR réserve un compteur dans le préfixe des segments parents logiques. Ce compteur renseigne sur le nombre de segments enfants logiques attachés à ce segment parent logique. Le paramètre PAIRED est indiqué pour les deux types segments enfants logiques réels d'un type relation bidirectionnel physique.

RULES: spécifie le type de chemin qui doit être utilisé pour insérer, effacer et remplacer un segment: P pour physique, L pour logique, V pour virtuel et B pour bidirectionnel virtuel. Ces paramètres sont spécifiés pour les types segments enfants logiques et pour leurs types segments parents (logique et physique). Les paramètres LAST, FIRST et HERE ont la même signification que précédemment.

L'instruction SEGM pour l'enfant logique virtuel

Le format de l'instruction SEGM pour l'enfant logique virtuel est le suivant (Figure 33):

```
SEGM
                    NAME=virtchild
                    ,PARENT=segname2
                    ,SOURCE=((segname3,DATA,dbname1))
                    ,POINTER=PAIRED
```

Figure 33: SEGM pour l'enfant logique virtuel

NAME: *virtchild* est le nom du type segment enfant logique virtuel. Ce dernier peut être suivi d'un champ de séquence qui contrôle la séquence de la chaîne des segments enfants logiques.

PARENT: *segname2* est le nom du type segment parent logique c'est-à-dire le type segment parent physique du type segment enfant logique virtuel.

SOURCE: *segname3* est le nom du type segment enfant logique réel et *dbname1* est le nom de la DBD dans laquelle se trouve ce type segment enfant logique réel. DATA indique que la clé et les données doivent être utilisées pour construire le segment.

POINTER=PAIRED: doit être inséré tel quel. Il définit ce type segment comme type segment enfant logique virtuel.

L'instruction LCHILD

" Pour chaque type segment enfant logique, une déclaration LCHILD doit être spécifiée immédiatement après la déclaration SEGM et/ou FIELD du type segment parent logique. "[IBM/IMS, 76]

Cette instruction fait donc le lien entre les types segments pour les types relations logiques parent-enfant. Elle est utilisée à chaque type relation logique dans laquelle le type segment est parent logique. Notons que l'enfant logique virtuel n'est pas réellement un enfant logique et ne possède donc pas de parent logique. Le format de cette instruction est le suivant (Figure 34):

```
LCHILD
                    NAME=(segname1,dbname)
                    [,POINTER={SNGL DBLE NONE}]
                    [,PAIR=segname2]
                    [,RULES={LAST FIRST HERE}]
```

Figure 34: LCHILD pour les types relations logiques

NAME: *segname1* est le nom du type segment enfant logique qui se trouve dans la base de données dont le nom est *dbname*. Ce dernier peut être omis si *segname1* se trouve dans la même DBD.
POINTER: réagit de la même façon que pour les pointeurs physiques. Le paramètre SNGL spécifie qu'un pointeur de type *Logical Child First* est placé dans le préfixe du segment parent logique et pointe vers le premier segment enfant logique. Le paramètre DBLE spécifie qu'un pointeur de type *Logical Child First* et un pointeur de type *Logical Child Last* sont placés dans le préfixe du segment parent logique et pointent respectivement sur le premier et le dernier segment enfant logique. Le paramètre NONE indique qu'aucun pointeur n'est réservé dans le préfixe du segment parent logique et donc que la relation logique n'est pas implémentée de ce côté.
PAIR: *segname2* est le nom du type segment enfant logique réel avec lequel le type segment est pairé.
RULES: est utilisé pour contrôler la séquence de la chaîne des segments logiques quand il n'y a pas de champ de séquence ou que le champ de séquence n'est pas unique. La signification des paramètres est la même que précédemment.

Pour l'apport au schéma, il faut tout d'abord identifier le type relation logique (unidirectionnel ou bidirectionnel) et ensuite identifier les types segments qui en font partie. La présence d'un type relation logique bidirectionnel nous est donnée par le paramètre de pairage du mot clé POINTER (PAIRED). Le mot clé PAIR des deux instructions LCHILD permet de déterminer les deux types segments pairés pour un type relation logique bidirectionnel physique. Pour un type relation logique bidirectionnel virtuel, il nous est renseigné par le mot clé SOURCE de l'instruction SEGM pour l'enfant logique virtuel. Ce même mot clé et le mot clé PAIR de l'instruction LCHILD établissent alors la correspondance entre les deux types segments.

La présence d'un type relation logique donne lieu à la création dans le schéma de deux types d'associations *one-to-many* dont il faut d'abord trouver les types d'entités jouant les rôles. Dans le cas d'un type relation logique unidirectionnel, un type d'entité est créé à partir du type segment enfant logique réel. Dans le cas d'un type relation logique bidirectionnel physique, un type d'entité est créé à partir des deux types segments enfants logiques réels. Le nom du type d'entité est la concaténation des noms des deux types segments. Ceci est justifié par le fait que la présence des deux types segments, qui ont une description identique, s'explique pour des raisons de chemin d'accès. Il n'y a donc pas lieu d'en faire deux types d'entités. Le même raisonnement est appliqué aux types relations logiques bidirectionnels virtuels. Un type d'entité est donc créé sur base du type segment enfant logique réel et du type segment enfant logique virtuel. Le nom du type d'entité est aussi la concaténation des deux noms des deux types segments. Les

attributs du type d'entité proviennent alors de la description du type segment enfant logique réel.

Pour l'instruction SEGM, les paramètres VIRTUAL/PHYSICAL, les paramètres du mot clé POINTER et ceux du mot clé RULES donnent des informations qui sont insérées dans la description technique du type d'entité créé. Le mot clé BYTES a la même signification que précédemment sauf qu'il faut lui retirer la longueur de la clé concaténée. Pour l'instruction LCHILD, les paramètres du mot clé POINTER et ceux du mot clé RULES donnent également des informations ajoutées à la description technique du type d'entité créé.

Pour en revenir à nos deux types d'associations, ceux-ci sont créés entre le type d'entité créé à l'instant et les types d'entités jouant les rôles de parents logiques et physiques. Les deux types d'entités parents sont indiqués par le mot clé PARENT de l'instruction SEGM des enfants logiques et par la présence de l'instruction LCHILD dans la déclaration des types segments parents logiques.

La cardinalité du rôle joué par chaque type d'entité parent est de 0-N et celle des deux rôles joués par le type d'entité enfant est de 1-1. En ce qui concerne le nom des deux types d'associations pour le bidirectionnel, ils sont suffixés respectivement par REL1 et REL2 et portent également un numéro incrémenté à chaque nouvelle relation logique. Le suffixe REL est là pour distinguer ces types d'associations qui proviennent de types relations logiques et les chiffres 1 et 2 sont là pour établir le lien entre les deux types d'association. L'unidirectionnel donne REL comme suffixe à son unique type d'association mais est également concerné par la numérotation.

Pour éclaircir cette extraction assez compliquée, il convient d'en donner une illustration. Pour se faire, il est préférable de séparer les types relations logiques analysés.

Nous commençons donc par le type relation logique unidirectionnel dont le code source se trouve à la Figure 35 et nous le passons à l'extracteur ce qui donne la Figure 36. Sur cette dernière, nous remarquons un type d'association dont le nom est *IREL*. Ce type d'association provient donc de notre type relation logique unidirectionnel dont nous avons détecter la présence par les deux parents attachés au type segment *SegB*. Notons aussi les cardinalités conformes de ce type d'association.

```
DBD     NAME=BD1,ACCESS=(HDAM,VSAM)
DATASET DD1=ens1
SEGM    NAME=SegA,PARENT=0,BYTES=53,RULES=PLV,FIRST
SEGM    NAME=SegB,PARENT=((SegA),(SegC,PHYSICAL,DB2)),BYTES=48,POINTER=LPARNT
      ,RULES=PLV,FIRST
DBDGEN
FINISH
END

DBD     NAME=BD2,ACCESS=HDAM
DATASET DD1=ens2
SEGM    NAME=SegC,PARENT=0,BYTES=28,RULES=PLV,FIRST
LCHILD  NAME=(SegB,DB1)
DBDGEN
FINISH
END
```

Figure 35 : Source IMS illustrant un type relation logique unidirectionnel

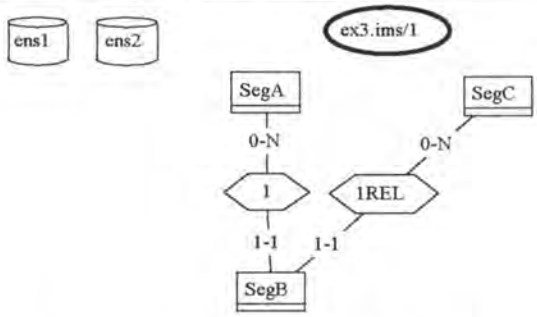

Figure 36 : Résultat de l'extraction de la Figure 35

Pour un type relation logique bidirectionnel physique, nous prenons comme code source la Figure 37 et nous avons comme résultat la Figure 38. Tout d'abord, le type relation logique bidirectionnel est identifié par les paramètres PAIRED des mots clés POINTER de l'instruction SEGM. Ensuite, les deux types segments pairés sont repérés par les valeurs du mot clé PAIR de l'instruction LCHILD. Il nous reste alors à modéliser ce type relation par le type d'entité *SegB/SegD* regroupant les deux types segments pairés *SegB* et *SegD*. Les types d'associations reliant ce type d'entité à ces parents portent alors les noms *1REL1* et *1REL2*.

```
DBD     NAME=BD1,ACCESS=(HDAM,VSAM)
DATASET DD1=ens1
SEGM    NAME=SegA,PARENT=0,BYTES=53,RULES=PLV,FIRST
LCHILD  NAME=(SegD,DB2),PAIR=SegB
SEGM
NAME=SegB,PARENT=((SegA),(SegC,PHYSICAL,DB2)),BYTES=48,POINTER=LPARNT,PAIRED
      ,RULES=PLV,FIRST
DBDGEN
FINISH
END

DBD     NAME=BD2,ACCESS=HDAM
DATASET DD1=ens2
SEGM    NAME=SegC,PARENT=0,BYTES=28,RULES=PLV,FIRST
LCHILD  NAME=(SegB,DB1),PAIR=SegD
SEGM
NAME=SegD,PARENT=((SegC),(SegA,PHYSICAL,DB1)),BYTES=48,POINTER=LPARNT,PAIRED
      ,RULES=PLV,FIRST
DBDGEN
FINISH
END
```

Figure 37 : Source IMS illustrant un type relation logique bidirectionnel physique

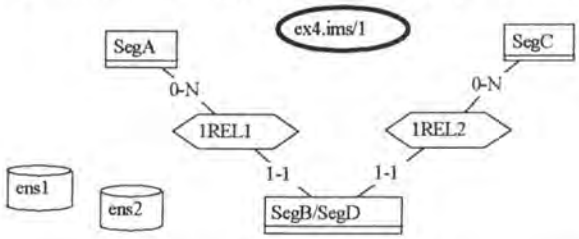

Figure 38 : Résultat de l'extraction de la Figure 37

Il ne nous reste plus que le type relation logique bidirectionnel virtuel dont le code se trouve à la Figure 39 et le résultat de l'extraction à la Figure 40. Ce type relation logique est repéré par le paramètre PAIRED du mot clé POINTER de l'instruction SEGM. Le caractère virtuel de ce type relation logique est indiqué par le mot clé SOURCE de l'instruction SEGM. Ce même mot clé permet de retrouver l'enfant logique

réel. Le type d'entité concaténé peut alors être créé avec comme nom *SegB/SegD*. Il suffit alors de rajouter les types d'associations *IREL1* et *IREL2* pour compléter la modélisation.

```
DBD     NAME=BD1,ACCESS=(HDAM,VSAM)
DATASET DD1=ens1
SEGM    NAME=SegA,PARENT=0,BYTES=53,RULES=LPV,FIRST
SEGM
NAME=SegB,PARENT=((SegA),(SegC,PHYSICAL,DB2)),BYTES=48,POINTER=LTWIN,LPARNT
      ,RULES=LPV,FIRST
DBDGEN
FINISH
END

DBD     NAME=BD2,ACCESS=HDAM
DATASET DD1=ens2
SEGM    NAME=SegC,PARENT=0,BYTES=28,RULES=LPV,FIRST
LCHILD  NAME=(SegB,DB1),POINTER=SNGL,PAIR=SegD
SEGM    NAME=SegD,PARENT=SegC,SOURCE=((SegB,DATA,DB1)),POINTER=PAIRED
DBDGEN
FINISH
END
```

Figure 39 : Source IMS illustrant un type relation logique bidirectionnel virtuel

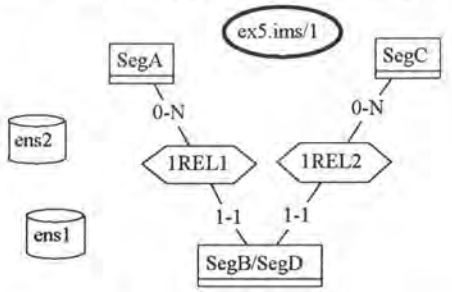

Figure 40 : Résultat de l'extraction de la Figure 39

3.2.2.2. La déclaration de la DBD logique

Comme nous l'avons vu précédemment, une base de données logique définit une nouvelle vue en se basant sur des bases de données physiques existantes reliées logiquement entre elles. Cette vue reste toujours une structure de données hiérarchique. Les instructions modifiées sont les suivantes: DBD, DATASET et SEGM. La déclaration des différents types segments dans une DBD logique doit se faire dans un ordre hiérarchique en suivant les règles vues au chapitre précédent. Les instructions DBDGEN, FINISH et END restent les mêmes et les instructions LCHILD et FIELD ne sont pas permises dans une DBD logique. Le format des instructions modifiées est le suivant (Figure 41):

```
DBD
                    NAME=dbname
                    ,ACCESS=LOGICAL
DATASET
                    LOGICAL
SEGM
                    NAME=segname1
                    [,PARENT={0 segname2}]
                    ,SOURCE=((segname3[,{DATA KEY}],dbname1)
                              [,(segname4[,{DATA KEY}],dbname2)])
```

DBDGEN
[FINISH]
END

Figure 41: Les différentes instructions et leur syntaxe pour une DBD logique

DBD:

NAME: *dbname* est le nom de cette DBD logique. Il est unique dans tout l'environnement.

ACCESS=LOGICAL: définit cette DBD comme une DBD logique.

DATASET LOGICAL: cette instruction doit être codée tel quel.

SEGM:

NAME: *segname1* spécifie le nom de ce type segment.

PARENT: *segname2* spécifie le nom du type segment parent de ce type segment. Ce type segment parent doit avoir été défini antérieurement. Le mot clé PARENT est omis ou possède la valeur 0 pour le type segment racine.

SOURCE: spécifie la ou les source(s) du type segment. Deux cas peuvent se présenter: les types segments concaténés et les types segments non concaténés.

Pour les types segments non concaténés, *segname3* spécifie le type segment source et *dbname1* la DBD physique dans laquelle se trouve ce type segment source. Le deuxième type segment source est alors omis.

Pour les types segments concaténés, *segname3* spécifie le type segment enfant logique comme défini dans la DBD physique. *Dbname1* donne le nom de la DBD physique dans laquelle le type segment *segname3* est défini. *Segname4* spécifie le type segment parent destination. *Dbname2* donne le nom de la DBD physique du type segment parent destination.

Les paramètres DATA et KEY nous indiquent si les données du segment sont placées ou non dans la zone d'entrée/sortie lors du *call*.

La déclaration de la DBD logique sert à indiquer dans quel sens sera parcouru le type relation logique. Par conséquent, elle n'apporte rien au schéma conceptuel mais indique un chemin par lequel les types d'entités seront accédés. Etant donné qu'un PCB est toujours défini sur la DBD logique et que ce PCB sert aussi à indiquer le chemin d'accès, l'apport au schéma sera donc fait lors de l'analyse de ce PCB. Toutefois, les noms des types segments repris dans le PCB sont ceux définis dans la DBD logique. Il faut donc garder trace de la correspondance entre les types segments de la DBD logique et les types segments de la DBD physique. Ceci est fait par l'analyse du mot clé SOURCE.

3.2.3. Les index secondaires

L'implémentation d'un index secondaire peut se décomposer en trois phases:

- la déclaration du type segment cible de l'index;
- la déclaration du type segment source de l'index;
- la déclaration de la DBD index secondaire.

Les deux premières déclarations se retrouvent dans la même DBD physique et la dernière fait l'objet d'une DBD particulière.

3.2.3.1. La déclaration du type segment cible

Pour cette déclaration, l'instruction LCHILD est étendue et une nouvelle instruction est ajoutée: XDFLD. Les autres instructions de la DBD restent les mêmes. Le type segment est reconnu comme type segment cible par l'ajout de l'instruction LCHILD suivie de l'instruction XDFLD. Ces deux instructions sont définies ci-après.

L'instruction LCHILD

Cette instruction fournit le lien avec la base de données index et son format est le suivant (Figure 42):

```
LCHILD
                NAME=(segname1,dbname)
                ,POINTER={INDX SYMB}
```

Figure 42: LCHILD pour le type segment cible

NAME: *segname1* est le nom du type segment pointeur défini dans la DBD index dont le nom est *dbname*.
POINTER: indique le type de pointeur utilisé (direct ou symbolique).

L'instruction XDFLD

Cette instruction spécifie les champs servant à l'indexation, le format est le suivant (Figure 43):

```
XDFLD
                NAME=fldname
                [,SEGMENT=segname]
                [,CONST=char]
                ,SRCH=list1
                [,SUBSEQ=list2]
                [,DDATA=list3]
                [,NULLVAL=value1]
                [EXTRTN=name1]
```

Figure 43: Description de l'instruction XDFLD pour le type segment cible

NAME: spécifie le nom du champ d'index secondaire. *Fldname* est un nom de champ normal qui peut être utilisé dans les SSA pour les appels aux index secondaires. Il doit être unique parmi tous les noms des champs du type segment indexé.
SEGMENT: *Segname* désigne le type segment source qui doit être un type segment de niveau égal ou inférieur au type segment cible. Si ce mot clé n'est pas présent, le type segment cible est considéré comme étant également le type segment source.
CONST: spécifie un caractère par lequel chaque segment pointeur de la BD index est identifié, dans le cas où une même BD index contient plusieurs types segments pointeurs.
SRCH: spécifie le ou les champ(s) du type segment index source qui doi(ven)t être utilisé(s) comme champ(s) de recherche de l'index secondaire. *List1* est une liste de un à cinq noms de champs définis dans le type segment source par l'instruction FIELD. Si deux ou plusieurs noms de champs sont pris en compte, alors ils doivent être séparés par des virgules et entourés par des parenthèses.
SUBSEQ: spécifie le ou les champ(s) du type segment index source qui doi(ven)t être utilisé(s) comme champ(s) de sous-séquence de l'index secondaire. *List2* obéit aux mêmes lois que *list1*.
DDATA: spécifie le ou les champ(s) du type segment index source qui doi(ven)t être utilisé(s) comme champ(s) de données dupliqués de l'index secondaire. *List3* obéit aux mêmes lois que *list1*.

NULLVAL: permet la suppression des segments pointeurs index quand les données du segment index source utilisées dans le champ de recherche du segment pointeur index contiennent la valeur spécifiée.
EXTRTN: spécifie le nom d'une routine utilisateur de maintenance de l'index qui supprime la création de segments pointeurs index sélectionnés.

L'apport au schéma de cette déclaration se fait sous la forme d'une clé d'accès pour le type d'entité provenant du type segment cible. Notre environnement de référence ne modélise pas les clés d'accès dont l'origine est autre que le type d'entité visé. Par conséquent, le cas où le type segment source est différent du type segment cible n'est pas représenté. Toutefois, la clé d'accès est mentionnée dans la description technique du type d'entité cible.

La clé d'accès, dont le nom est donné par le nom du champ d'index secondaire, est constituée des attributs provenant des champs de la *list1*. Les autres renseignements à savoir le nom du champ d'index, le nom du type segment pointeur index, le nom de la base de donnée et les autres options venant des différents mots clés sont ajoutés à la description technique de la clé d'accès.

3.2.3.2. La déclaration du type segment source

Le type segment est reconnu comme type segment source par l'instruction XDFLD codée précédemment dans le type segment cible. Il ne possède donc pas d'instruction particulière. Toutefois, il convient de remarquer la possible présence de champs appelés des champs_reliés_système. Ces nouveaux champs sont de deux types:

- un champ reprenant toute ou une partie de la clé concaténée d'un type segment source. Le nom de ce champ est long de 8 caractères et commence obligatoirement par les trois caractères "/CK". Ce champ est défini pour permettre l'utilisation de la clé concaténée d'un type segment source dans la sous-séquence ou les données dupliquées des types segments pointeurs. Il peut être repris dans la liste des champs de la sous-séquence ou des données dupliquées (cfr SUBSEQ et DDATA);
- un champ pour assurer l'unicité des valeurs de champ de séquence dans un index secondaire. La déclaration d'un tel champ est donc conseillée si des duplications de segments pointeurs peuvent survenir. Le nom de ce champ est long de 8 caractères et commence obligatoirement par les trois caractères "/SX". Ce champ peut être repris dans la liste des champs de la sous-séquence.

De par leur caractère technique, ces nouveaux champs ne donnent pas lieu à la création d'attribut pour le type d'entité. Ils sont uniquement ajoutés à titre indicatif dans la description technique du type d'entité.

Afin de mieux comprendre notre démarche, nous illustrons encore une fois l'apport de cette déclaration au schéma. Pour ce faire, nous nous basons sur le code de la Figure 44 et nous obtenons le schéma de la Figure 45. Sur ce schéma, nous constatons la présence de deux clés d'accès pour le type d'entité *01Clasper*. Ces deux clés d'accès proviennent des instructions XDFLD et LCHILD attachées au type segment qui est l'origine du type d'entité. Notons aussi l'absence logique de l'attribut /CKindpe qui ne doit pas être repris. En ce qui concerne les instructions XDFLD et LCHILD du type segment *01Pers*, elles ne donnent pas naissance à une clé d'accès pour le type d'entité

01Pers car les champs sources de l'indexation proviennent d'un autre type segment. Cette clé d'accès est cependant renseignée dans la description technique du type d'entité mais cela n'est pas représenté graphiquement.

```
DBD     NAME=Personne,ACCESS=(HDAM,VSAM),RMNAME=(rand1,2,200,500)
DATASET  DD1=ens1
SEGM    NAME=01Clasper,PARENT=0,BYTES=48,FREQ=7
FIELD   NAME=(nom,SEQ,U),BYTES=20,START=1,TYPE=C
FIELD   NAME=(fonction),BYTES=25, START=21,TYPE=C
FIELD   NAME=(niveau),BYTES=3,START=46,TYPE=X
FIELD   NAME=/CKindpe,BYTES=3,START=1
LCHILD  NAME=(01idxc1,Bdindx2),POINTER=INDX
XDFLD   NAME=01flind1,CONST=1,SRCH=(fonction,niveau),SUBSEQ=(fonction,niveau,/CKindpe),
                        DDATA=(fonction,niveau),NULLVAL=' ',EXTRTN=routind
LCHILD  NAME=(01idxc2,Bdindx2),POINTER=INDX
XDFLD   NAME=01flind2,CONST=2,SRCH=fonction,NULLVAL='$'
SEGM    NAME=01Pers,PARENT=(01Clasper),BYTES=40,FREQ=25,POINTER=TWINBWD
FIELD   NAME=(ident),SEQ,U,BYTES=40,START=1,TYPE=C
LCHILD  NAME=(01idxp1,Bdindx2),POINTER=INDX
XDFLD   NAME=01flind3,SEGMENT=01Diplom,CONST=3,SRCH=(denom,grade)
        ,SUBSEQ=(denom,grade),DDATA=(denom,grade),NULLVAL='$',EXTRTN=routind
SEGM    NAME=01Diplom,PARENT=(01Pers),BYTES=70,FREQ=5,POINTER=HIERBWD
                        ,RULES=(,FIRST)
FIELD   NAME=(denom),BYTES=60,START=1,TYPE=C
FIELD   NAME=(grade),SEQ,M,BYTES=10,START=61,TYPE=C
DBDGEN
FINISH
END

DBD     NAME=Bdindx2,ACCESS=INDEX
DATASET  DD1=ddindx2,DEVICE=3350
SEGM    NAME=01idxc1,PARENT=0,BYTES=28
LCHILD  NAME=(01Clasper,Personne),INDEX=01flind1
FIELD   NAME=(indx1,SEQ,U),BYTES=28,START=1
SEGM    NAME=01idxc2,PARENT=0,BYTES=25
LCHILD  NAME=(01Clasper,Personne),INDEX=01flind2
FIELD   NAME=(indx2,SEQ,U),BYTES=25,START=1
SEGM    NAME=01idxp1,PARENT=0,BYTES=70
LCHILD  NAME=(01Pers,Personne),INDEX=01flind3
FIELD   NAME=(indx3,SEQ,U),BYTES=70,START=1
DBDGEN
FINISH
END
```

Figure 44 : Source IMS illustrant les index secondaires

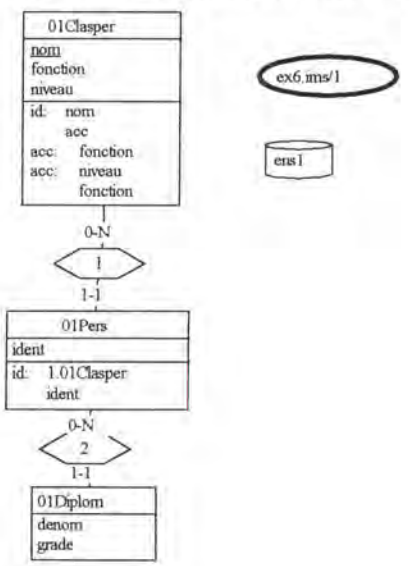

Figure 45 : Résultat de l'extraction de la Figure 44

3.2.3.3. La déclaration de la DBD index

Pour cette déclaration, toutes les instructions vues précédemment à l'exception de XDFLD sont utilisées. Le format et l'enchaînement de ces instructions sont représentés à la Figure 46. Leur enchaînement correspond à l'ordre dans lequel elles apparaissent. Les instructions SEGM, LCHILD et FIELD peuvent bien sûr apparaître plusieurs fois si la DBD contient plusieurs types segments pointeurs index.

```
DBD
                    NAME=(dbname1,...)
                    ,ACCESS=(INDEX[,VSAM]
                                  [,{PROT NOPROT}]
                                  [,DOSCOMP])
                    [,PASSWD={YES
                                NO}]
DATASET
                    DD1=ddname1
                    [,BLOCK=(blkfact1,blkfact2)]
                    [,SIZE=(size1,size2)]
                    [,RECORD=(reclen1,reclen2)]
SEGM
                    NAME=segname1
                    [,PARENT=0]
                    ,BYTES=bytes
                    [,FREQ=freq]
LCHILD
                    NAME=(segname1,dbname)
                    ,POINTER={SNGL SYMB}
                    ,INDEX=fldname
FIELD
                    NAME=(fldname1,SEQ,U)
                    ,BYTES=bytes
                    ,START=1
DBDGEN
[FINISH]
END
```

Figure 46: Les différentes instructions et leur syntaxe pour une DBD index secondaire

DBD:

NAME: *dbname1* est le nom de la ou des base(s) de données index secondaire(s).

ACCESS: INDEX identifie cette DBD comme une DBD index. VSAM est toujours utilisé pour les BD index secondaires. Le paramètre PROT renforce l'intégrité des données utilisées dans les index secondaires. Le paramètre DOSCOMP est indiqué quand la BD a été créée avec le DL1/DOS.

DATASET:

La signification des mots clés est la même que celle discutée pour les bases de données physiques.

SEGM:

NAME: *segname1* spécifie le nom du type segment pointeur.

PARENT: est optionnel. Il est codé de cette façon pour tous les types segments pointeurs si la BD en contient plusieurs.

BYTES et **FREQ** ont la même signification que précédemment.

LCHILD:

NAME: *segname1* spécifie le nom du type segment cible se trouvant dans la DBD dont le nom est *dbname*.

POINTER: indique le type de pointeurs utilisé (direct ou symbolique).

INDEX: *fldname* est le nom du champ indexé. Ce nom de champ doit être spécifié comme le nom d'un champ XDFLD du type segment cible.

FIELD:

Une seule instruction FIELD doit être codée après chaque instruction SEGM.

NAME: *fldname1* est le nom du champ. Il n'est pas utilisé par les programmes d'applications.

BYTES: spécifie la taille en octets du champ.

Les renseignements que nous pouvons glaner ici (le nom du type segment pointeur, le type de pointeurs utilisé et les informations fournies par les instructions DBD et DATASET) sont ajoutés à la description technique des différentes clés d'accès pour lesquelles cette BD a été créée. Cette BD ne donne donc pas naissance à de nouveaux types d'entités.

3.2.4. Les vues (PSB)

Un programme d'application doit toujours passer par une vue pour accéder à une DBD physique ou logique. Pour ce faire, il définit un PSB regroupant les différentes vues nécessaires à la bonne exécution du programme. Dans les PCB figurent les types segments et les champs sensibles au programme. L'enchaînement des instructions pour la définition d'un PSB est montré à la Figure 47. Les deux instructions SENSEG et SENFLD s'enchaînent de la même façon que leurs homologues SEGM et FIELD.

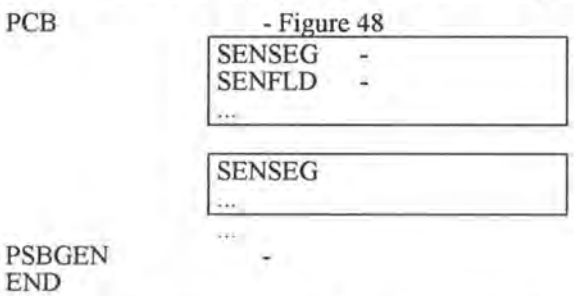

Figure 47 : Enchaînement des instructions DL/1 pour un PSB

3.2.4.1. L'instruction PCB

Cette instruction a plusieurs fonctions dont celle de définir une vue d'une base de données que le programme a l'intention d'utiliser. Le format dans le cas d'une vue est le suivant (Figure 48):

```
[label]
PCB
                    PCBTYPE=DB
                    ,{DBNAME NAME}=name
                    [,PCBNAME=pcbname]
                    [,PROCOPT={G[E][S][P][O][N][T][H]
                              I[E][P][H]
                              R[E][H]
                              D[E][P][H]
                              A[E][P][H]
                              L[S][P][N]}]
                    [,SB={NO COND}]
                    ,KEYLEN=value
                    [,POS={SINGLE MULTIPLE}]
                    [,PROCSEQ=indexdbname]
                    [,LIST={YES NO}]
```

Figure 48 : Syntaxe de l'instruction PCB

Label: sert à donner un nom au PCB. Il ne peut être utilisé si le mot clé PCBNAME est utilisé et inversement.
TYPE=DB: identifie ce PCB comme étant une vue sur une DBD.
DBNAME (NAME): spécifie le nom de la DBD sur laquelle porte la vue. Cette DBD peut être logique ou physique.
PCBNAME: sert à donner un nom au PCB. Il ne peut être utilisé si le paramètre *Label* est utilisé et inversement.
PROCOPT: spécifie les opérations permises sur les types segments sensibles déclarés dans ce PCB. Les paramètres peuvent apparaître dans n'importe quelle combinaison. Les opérations principales autorisées sont les suivantes: G(Get), I(Insert), R(Replace), D(Delete) ou A(toutes les opérations). Les autres paramètres ne seront pas évoqués.
SB: spécifie quel PCB sera bufférisé en utilisant le *sequential buffering*.
KEYLEN: spécifie la longueur en octets de la plus longue clé concaténée pour un chemin hiérarchique des types segments sensibles.
POS: spécifie si le positionnement est simple ou multiple.
PROCSEQ: spécifie le nom d'une DBD index secondaire utilisée pour accéder à la DBD spécifiée dans DBNAME. Si ce mot clé est présent, alors l'ordre de déclaration des types segments sensibles doit se conformer aux règles de restructuration pour les index secondaires.
LIST: spécifie si le PCB est inclus dans la liste PCB passée en entrée au programme.

3.2.4.2. L'instruction SENSEG

Cette instruction spécifie les types segments auxquels le programme est sensible. Son format est le suivant (Figure 49):

```
SENSEG
                NAME=name
                ,PARENT={0 name}
                ,PROCOPT={G
                          I[E][P]
                          R
                          D
                          A[E][P]
                          K}
                ,SSPTR=((n,{r u}),...)
                [,INDICES=list1]
```

Figure 49 : Syntaxe de l'instruction SENSEG

NAME: est le nom d'un type segment défini auparavant dans une DBD.
PARENT: est le nom du type segment parent de ce type segment. Si le type segment est un type segment racine, alors il faut coder PARENT=0.
PROCOPT: spécifie les opérations permises sur les occurrences de ce type segment. Ces opérations permises écrasent celles définies dans l'instruction PCB. Les paramètres peuvent apparaître dans n'importe quelle combinaison et leur signification est la même que précédemment. Notons cependant que le paramètre K spécifie ce type segment comme sensible uniquement sur la clé. Cela veut dire que le PA ne peut accéder aux données de ce type segment. Ce paramètre est donc là pour permettre à un PA qui n'a pas l'accès à ce type segment de quand même pouvoir accéder à ses types segments dépendants.
SSPTR: spécifie le nombre de sous-ensembles pointeurs et leur sensibilité.
INDICES: spécifie quelle(s) DBD index secondaire(s) contienne(nt) les champs de recherche qui sont utilisés pour les SSA d'un type segment indexé. *List1* peut contenir jusqu'à 32 noms de DBD index secondaires.

3.2.4.3. L'instruction SENFLD

Cette instruction spécifie les champs du type segment auxquels le programme est sensible. Si le type segment a le paramètre K pour le mot clé PROCOPT, alors les définitions de champs sensibles ne sont pas permises. Le format est le suivant (Figure 50):

```
SENFLD
                NAME=name
                ,START=startpos
                [,{REPLACE REPL}={YES NO}]
```

Figure 50 : Syntaxe de l'instruction SENFLD

NAME: est le nom d'un champ défini auparavant dans une DBD.
START: spécifie la position de départ de ce champ par rapport au début du segment dans la zone d'entrée/sortie de l'utilisateur.
REPLACE (REPL): spécifie si ce champ peut ou non être altéré dans un *call* de remplacement (Replace).

3.2.4.4. L'instruction PSBGEN

Cette instruction spécifie les caractéristiques du programme d'application. Son format est le suivant ():

```
PSBGEN
                PSBNAME=name
                [,LANG={COBOL PL/1 ASSEM PASCAL blank}]
                [,MAXQ={0 nr}]
                [,CMPAT={NO YES}]
                [,IOASIZE=value]
                [,SSASIZE=value]
                [,IOERPN={n (n,WTOR)}]
                [,OLIC={NO YES}]
                [,LOCKMAX={0 n}]
```

Figure 51 : Syntaxe de l'instruction PSBGEN

PSBNAME: donne le nom du PSB.
LANG: indique le langage dans lequel le programme d'application est écrit.
MAXQ: indique le nombre maximum de *calls* qui peuvent être émis entre deux points de synchronisation.
CMPAT: indique une certaine compatibilité de paramètres entre environnements IMS.
IOASIZE: spécifie la taille de la plus grande zone d'entrée/sortie utilisée par le PA.
SSASIZE: spécifie la longueur totale maximum de tous les SSA utilisés par le PA.
IOERPN: *n* est le code condition renvoyé par IMS à l'OS pour une terminaison normale.
OLIC: indique si l'utilisateur du PSB peut exécuter certains utilitaires IMS.
LOCKMAX: indique le nombre maximum de verrous qu'un PA peut avoir à un moment donné.

La contribution de ces vues au schéma se fait par la création de sous-schémas externes. En effet, nous pouvons considérer les programmes d'application comme les utilisateurs finals. De ce fait, les PCB attachés à ces PA sont donc les vues des utilisateurs finals. Ils donnent donc naissance à des sous-schémas externes qui montrent ainsi la vue conceptuelle de chacun des utilisateurs.

Les informations contenues dans un PCB n'apportent rien de nouveau sur les types d'entités et les types d'associations déjà présents dans le schéma. Les vues qui ressortent

des PCB sont surtout utiles pour montrer les chemins d'accès de la BD. Etant donné que notre environnement de référence ne modélise pas les chemins d'accès, ceux-ci sont donc représentés par les sous-schémas externes. Le sous-schéma externe est donc créé par recopie des types d'entités et des types d'associations du schéma. La recopie a lieu pour tous les types d'entités spécifiés par les types segments sensibles. Dans ces types d'entités ne figurent que les attributs spécifiés par les champs sensibles. Les types d'associations liant les types d'entités sont recréés entre les types d'entités recopiés. Pour respecter l'unicité des noms des types d'entités dans le schéma, l'environnement DB-MAIN se charge de suffixer les noms des types d'entités recopiés d'un chiffre. Toutefois, pour faciliter la lecture du schéma, nous ajoutons encore aux noms des types d'entités et des types d'associations recopiés le nom du PCB.

Pour être complet, les informations fournies par le mot clé PROCOPT sont ajoutées à la description technique de chaque type d'entité recopié. De même, le nom et les renseignements concernant le PSB sont ajoutés à la description sémantique du type d'entité racine du PCB.

A titre d'illustration, nous prenons comme exemple celui dont le code source se trouve à la Figure 52 et dont le schéma résultant se trouve à la Figure 53. De la hiérarchie se trouvant sur la gauche du schéma, l'extraction a déduit les deux sous-schémas correspondant aux deux vues détectées par les PCB. Notons dans ces deux sous-schémas la suffixation des noms des types d'entités et des types d'associations ainsi que l'absence des attributs non spécifiés comme sensibles par les instructions SENFLD. De même, le paramètre K du mot clé PROCOPT pour le type segment sensible *01Clasper* implique l'absence de tous les attributs du type d'entité *01Clasper*.

```
DBD     NAME=(Personne),ACCESS=(HDAM,VSAM),RMNAME=(rand1,2,200,500)
DATASET  DD1=ens1
SEGM     NAME=01Depart,PARENT=0,BYTES=53
FIELD    NAME=(numero,SEQ,U),BYTES=3,START=1,TYPE=C
FIELD    NAME=(nom),BYTES=10,START=4,TYPE=C
FIELD    NAME=(chef),BYTES=20,START=14,TYPE=C
FIELD    NAME=(localisa), BYTES=20,START=34,   TYPE=C
SEGM
NAME=01Clasper,PARENT=(01Depart,DBLE),BYTES=48,FREQ=7,POINTER=TWINBWD
FIELD    NAME=(nom,SEQ,U),BYTES=20,START=1,TYPE=C
FIELD    NAME=(fonction),BYTES=25, START=21,TYPE=C
FIELD    NAME=(niveau),BYTES=3,START=46,TYPE=X
SEGM     NAME=01Pers,PARENT=(01Clasper),BYTES=180,FREQ=25,POINTER=TWINBWD
FIELD    NAME=(ident),SEQ,U,BYTES=40,START=1,TYPE=C
FIELD    NAME=(nom),BYTES=20,START=1,TYPE=C
FIELD    NAME=(prenom),BYTES=20,START=21,TYPE=C
FIELD    NAME=(adresse),BYTES=91, START=41,TYPE=C
SEGM     NAME=01Diplom,PARENT=(01Pers),BYTES=70,FREQ=5,POINTER=HIERBWD
                        ,RULES=(,FIRST)
FIELD    NAME=(denom),BYTES=60,START=1,TYPE=C
FIELD    NAME=(grade),SEQ,M,BYTES=10,START=61,TYPE=C
SEGM     NAME=01Projet,PARENT=(01Depart),BYTES=70
FIELD    NAME=(numero,SEQ),BYTES=3,START=1,TYPE=C
FIELD    NAME=(nom),BYTES=20,START=4,TYPE=C
FIELD    NAME=(but),BYTES=30,START=24,TYPE=C
FIELD    NAME=(datefin),BYTES=10,START=54,TYPE=C
DBDGEN
FINISH
END


PCB      PCBTYPE=DB,DBNAME=Personne,PCBNAME=vue1,PROCOPT=GI,KEYLEN=63
SENSEG           NAME=01Depart,PARENT=0
SENFLD   NAME=numero,START=1
```

```
SENFLD      NAME=nom,START=4
SENFLD      NAME=chef,START=14
SENSEG      NAME=01Clasper,PARENT=01Depart,PROCOPT=K
SENSEG      NAME=01Pers,PARENT=01Clasper
SENFLD      NAME=nom,START=1
SENFLD      NAME=prenom,START=21
SENFLD      NAME=adresse,START=41
SENSEG      NAME=01Projet,PARENT=01Depart
SENFLD  NAME=numero,START=1
PCB     PCBTYPE=DB,DBNAME=Personne,PCBNAME=vue2,PROCOPT=G,KEYLEN=6
SENSEG      NAME=01Depart,PARENT=0
SENFLD  NAME=nom,START=1
SENSEG      NAME=01Projet,PARENT=01Depart,PROCOPT=GIR
SENFLD      NAME=nom,START=1
PSBGEN      PSBNAME=userview,LANG=COBOL,MAXQ=5
```

Figure 52 : Source IMS illustrant les vues

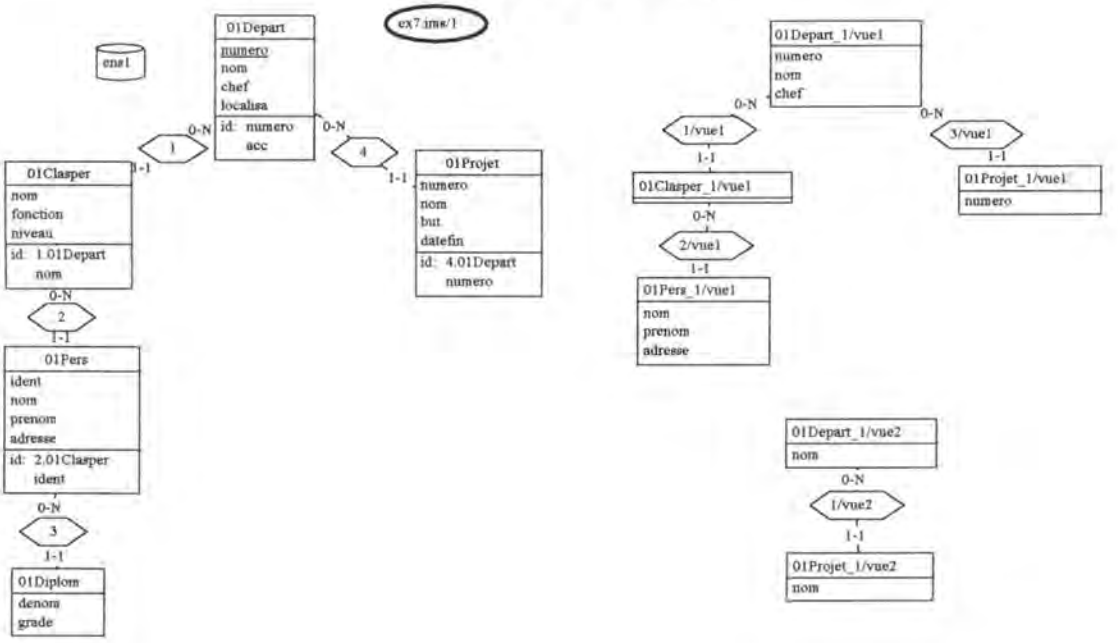

Figure 53 : Résultat de l'extraction de la Figure 52

3.2.5. L'extracteur

Dans cette analyse du code DDL, nous avons dressé toute une série d'apports au schéma conceptuel optimisé orienté IMS. Ces apports ont servi de spécifications pour un programme d'extraction que nous avons intégré à l'environnement DB-MAIN.

La phase d'analyse du code DDL peut donc se faire automatiquement grâce à ce programme. Pour nous résumer, nous allons reprendre les éléments principaux que nous avons apportés au schéma ainsi que leur provenance:

- schéma : DBD;
- collection : DATASET et AREA;
- type d'entité : SEGM;
- type d'association "physique" : SEGM;
- cardinalité supérieure du type d'association pour le rôle parent à 1 POINTER=NOTWIN dans SEGM;
- attribut d'un type d'entité : FIELD;

- identifiant d'un type d'entité : SEQ,U dans FIELD;
- clé de tri : SEQ,M dans FIELD;
- deux types d'associations "logiques" : LCHILD et SEGM;
- nouveau type d'entité associé aux deux types d'associations "logiques" : SEGM et LCHILD;
- clé d'accès pour un type d'entité : SRCH de XDFLD;
- sous-schéma externe : PCB, SENSEG et SENFLD.

3.3. Apport du DML et des programmes pour l'extraction

Pour le DDL, il était facile d'en faire l'analyse car l'entièreté du code était vouée à la description du schéma physique de la BD. Le cas du DML et des programmes est beaucoup plus compliqué car il mélange dans son code des spécifications relatives à la BD mais aussi et surtout des spécifications relatives aux programmes de l'application.

L'approche à adopter n'est donc plus la même et une analyse brute et aveugle du code n'est plus possible. Ici, il faut connaître à l'avance ce que l'on cherche. Ce n'est qu'avec cette hypothèse que nous ferons une rétro-ingénierie correcte.

Pour ce faire, cette section est principalement divisée en deux parties: les éléments conceptuels dont nous pouvons supposer l'existence par des indices fournis par le code DDL et les autres éléments dont l'existence n'est pas décelable dans le code DDL.

Nous allons également tenter de résoudre deux problèmes de la rétro-ingénierie évoqués auparavant à savoir la dissimulation de structure et les structures non-déclaratives. Ces deux problèmes seront donc évoqués tout au long de notre analyse.

Pour illustrer notre analyse, nous avons pris comme langage hôte le langage COBOL aux normes ANSI 85 dont on peut trouver une description dans [CLARINVAL, 91]. Le lecteur avisé pourra toujours transposer ces illustrations dans le langage hôte souhaité pour avoir une vue plus personnelle des choses.

Les informations supposées que nous allons exposer ici peuvent faire l'objet d'une recherche automatique dans l'atelier DB-MAIN. Ces éléments peuvent en effet être codés sous forme de patterns qui sont ensuite appliqués au code source. L'atelier renvoie alors le résultat de sa recherche. Il n'est dès lors plus indispensable de parcourir manuellement le code, ceci est fait par l'atelier.

Finalement, les modifications du schéma faites ultérieurement seront également faites sur les sous-schémas externes. Les sous-schémas externes représenteront ainsi toujours une vue conceptuelle correcte du schéma général.

3.3.1. Les éléments conceptuels à confirmer

Lors de l'analyse du code DDL, nous avons fait des suppositions sur des éléments conceptuels. Ces suppositions proviennent d'indices découverts dans le DDL. Il convient donc d'en établir d'abord la liste. Suite à cette liste, nous recherchons dans le code de l'application des confirmations de ces suppositions.

Il est évident que les éléments analysés peuvent très bien n'avoir fait l'objet d'aucune supposition. La phase de recherche de confirmations ne doit donc pas seulement être vue comme une recherche de confirmations mais aussi comme une recherche tout court. A ce titre elle doit donc toujours être exécutée même si le rétro-ingénieur n'a aucune supposition à confirmer.

3.3.1.1. Les suppositions

La cardinalité supérieure

Dans le DDL, le mot clé FREQ de l'instruction SEGM nous renseigne sur le nombre moyen d'occurrences du type segment enfant attachées à une occurrence du type segment parent. Si ce nombre moyen d'occurrences s'avère être une limite supérieure constante pour toutes les occurrences du type relation physique, alors nous avons un changement de cardinalité supérieure du rôle joué par le type d'entité parent dans le type d'association provenant de ce type relation physique. La cardinalité supérieure devient ce nombre moyen d'occurrence.

Si cette cardinalité supérieure est de 1, nous avons un changement du type d'association *one-to-many* en type d'association *one-to-one*. Un problème de structures non-déclaratives est ainsi résolu.

Les attributs manquants

En comparant la valeur du mot clé BYTES pour l'instruction SEGM à la somme des valeurs des mots clés BYTES des instructions FIELD de ce même type segment, nous pouvons remarquer une différence. Cette différence nous indique alors la présence d'attributs du type d'entité que le concepteur du DBD n'a pas cru bon de transformer en champ IMS.

Dans le cas où la différence est nulle, nous n'avons pas pour autant résolu le problème. En effet, un champ dont le nom est "FILLER" ou "REMPLISSAGE" ou tout autre synonyme peut également cacher toute une série de champs que le DBA n'a pas voulu déclarer.

Toutefois, ces champs manquants peuvent très bien avoir d'autres objectifs tels que l'anticipation de l'évolution de la BD ou encore la standardisation forcée des tailles des types segments. Toujours est-il que cela correspond à des suspicions qu'il faut vérifier.

Les attributs décomposables

Les attributs décomposables ne sont pas directement représentables en IMS. Ils sont repérables de deux façons.

Premièrement, le nom du champ peut nous renseigner sur son caractère décomposable. Par exemple, un nom de champ ADRESSE est fortement susceptible

d'être un attribut décomposable. De plus, si sa taille est assez grande, la suspicion est renforcée.

Deuxièmement, par le jeu des mots clés BYTES et START, nous pouvons recouper certains champs et ainsi découvrir une décomposition. Si la décomposition est complète, c'est-à-dire si la somme des longueurs des champs composants est égale à la longueur du champ décomposé, la modélisation de l'attribut décomposé peut se faire intégralement dans l'analyse du code DDL. Toutefois, si la décomposition n'est pas complète, alors il faut rechercher les champs composants manquants.

Les attributs répétitifs

Les attributs répétitifs sont repérables par leur nom et leur longueur excessive. Par exemple, un champ "TELEPHONES" d'une longueur de 50 octets ou un champ "ENFANTS" d'une longueur de 100 octets nous amène à suspecter la répétitivité d'un seul attribut de longueur moindre. De même, la succession d'un champ de même longueur ayant le même nom uniquement suffixé ou préfixé par un chiffre est un signe d'attribut répétitif.

Pour en terminer avec les attributs, nous pouvons dire que les attributs manquants sont un problème de dissimulation de structure et les attributs répétitifs et décomposables sont des structures non-déclaratives.

La surdéfinition de plusieurs types d'entités

En élargissant le cadre des attributs décomposables, nous arrivons à la surdéfinition de plusieurs types d'entités sur un seul type segment. Ceci constitue un problème de dissimulation de structures. La composition des mots clés BYTES et START permet donc d'implémenter deux ou plusieurs types d'entités sur un même type segment. Cette approche, bien que vivement déconseillée, est quand même utilisée par certains concepteurs. Nous devons donc essayer de la repérer. Pour cela, nous reprenons les deux moyens utilisés pour les attributs décomposables.

Premièrement, les noms et les longueurs des champs sont à même de dévoiler une telle surdéfinition. Par exemple, si nous trouvons dans un même type segment les noms de champs "ENTETE-COMM" et "LIGNE-COMM" et que ces deux champs ont comme longueur celle du type segment, alors nous avons repéré une surdéfinition. Cependant, il se peut que ces grands champs décomposables ne soit pas présents.

Deuxièmement, la combinaison des mots clés BYTES et START est primordiale. En effet, si l'ensemble de tous les champs peut se diviser en deux ou plusieurs groupes dans chacun desquels nous avons une définition complète du type segment, nous avons alors déterminé une surdéfinition. Reste alors à savoir à quelle définition appartient chaque champ.

Les identifiants et les clés étrangères

Quand un champ de séquence unique est défini sur un type segment dépendant, alors les occurrences de ce type segment sont uniques sous le même segment parent mais pas pour l'entièreté de la BD. Toutefois, il se peut que les valeurs de ce champ de séquence soient uniques pour l'ensemble de la BD.

Ayant repéré les identifiants, il serait utile de voir si ces derniers ne sont pas des clés étrangères dans d'autres types segments. Ces clés étrangères peuvent contenir le champ de séquence considéré comme identifiant mais aussi toute ou une partie de la clé

concaténée des types segments supérieurs. Il est généralement facile de les repérer car elles ont le même nom ou presque et la même longueur que l'identifiant supposé.

Ce dernier problème est une structure non-déclarative.

La redéfinition d'un type d'entité

La redéfinition d'un type d'entité sur plusieurs types segments est un phénomène fréquent en IMS induit par les contraintes de modélisation.

En effet, comme illustré dans les Figure 4 et Figure 5, lorsque nous avons plusieurs types d'associations *one-to-many* entre deux types d'entités, cela peut donner naissance à la redéfinition du même type d'entité sur plusieurs types segments enfants attachés au même type segment parent. D'autre part, il est possible de modéliser un type d'association *many-to-many* de plusieurs façons, l'une d'entre d'elles faisant appel à la redéfinition d'un type d'entité.

Ces types segments redéfinis sont identiques mais possèdent un nom différent. Il convient de retrouver ces redéfinitions du même type d'entité. Pour cela, nous avons deux moyens.

Premièrement, deux ou plusieurs noms de types segments ayant un préfixe ou un suffixe identique sont suspectés (ex: "VOIT-UTIL" et "VOIT-POSS").

Deuxièmement, la structure de définition des champs de ces types segments est normalement parfaitement identique. On peut cependant supposer des légères différences si le concepteur a voulu personnifier ces types segments. Par exemple, il est possible de trouver un champ de kilométrage pour le type segment portant sur la voiture utilisée et un champ de date d'achat pour le type segment portant sur la voiture possédée.

Ce problème est soit une structure non-déclarative s'il n'était pas possible de le modéliser autrement ou soit une dissimulation de structure si une autre modélisation était possible mais que cette solution a été choisie pour des raisons de facilité ou de performances.

3.3.1.2. La recherche des confirmations

Dans la plupart des langages, il existe une zone pour la définition des variables et une zone pour le programme. Nous séparons donc l'analyse du code en ces deux parties: la zone de définition des variables et la zone des codes programmes. Chacune des deux peut comporter des confirmations à nos suppositions et par la même résoudre nos problèmes.

La zone de définition des variables

Dans cette zone sont définies les variables employées par le programme. Lors d'un *call*, le segment est renvoyé dans une zone de réception du programme. Cette zone d'entrée/sortie est par conséquent une variable. Il peut évidemment y avoir plusieurs variables d'entrée/sortie différentes pour le même type segment. Elles sont alors utilisées dans différents *calls*. En COBOL, les variables sont définies dans la zone DATA DIVISION.

L'analyse des définitions des variables d'entrée/sortie permet de confirmer les suppositions faites sur les attributs manquants, décomposables et répétitifs ainsi que sur la surdéfinition d'un type segment. En effet, la variable d'entrée/sortie peut avoir une définition plus fine que celle des champs du type segment.

En premier lieu, il faut rechercher quelles sont les variables d'entrée/sortie qui reçoivent le type segment visé. Pour cela, il faut prendre note des noms de variables d'entrée/sortie des *calls* renvoyant le segment. Nous avons alors la liste des variables d'entrée/sortie pour chaque type segment. D'autre part, les variables d'entrée/sortie ne sont pas les seules à nous intéresser. En effet, si une partie ou toute une variable d'entrée/sortie est transférée dans une autre variable de la zone de travail, alors cette nouvelle variable doit faire également partie de la liste des variables à analyser. Pour savoir si un tel transfert a lieu, il faut analyser le programme se trouvant dans la PROCEDURE DIVISION et rechercher si des instructions MOVE portent sur les variables d'entrée/sortie. La variable destination du MOVE est alors ajoutée à la liste.

Ensuite, il faut prendre garde aux champs du type segment qui sont accessibles par le programme. En effet, seuls les champs sensibles sont transférés aux programmes. Dès lors, il faut comparer la variable de réception avec la structure du type segment mais en retirant de cette structure les champs qui ne sont pas sensibles.

Troisièmement, il faut analyser la structuration de la variable d'entrée/sortie c'est-à-dire ses variables composantes. En termes COBOL, nous regardons quels sont les différents niveaux (01,02,03,...) et leur contenu. Ensuite, nous construisons une vue linéaire de la variable d'entrée/sortie en mettant à plat ces niveaux. Nous établissons alors les positions de départ des variables composantes en se servant de la longueur additionnée de ces variables. La structure de définition de la variable est dans ce cas identique à celle du type segment diminué des champs non sensibles. La comparaison des deux définitions est alors facile et permet de rechercher nos confirmations. Cette opération doit bien sûr être réalisée pour chaque variable d'entrée/sortie de chaque type segment.

Pour les attributs manquants, il n'est pas possible de les retrouver car, comme nous l'avons dit, ils ne sont pas transférés lors du *call*. Pour les champs de remplissage (ex: "FILLER"), il faut regarder si la variable composante se trouvant à l'endroit du champ de remplissage ne possède pas un autre nom de variable ou une décomposition plus fine en plusieurs autres variables. Pour les attributs décomposables, nous regardons d'abord si les champs supposés décomposables (ex: "ADRESSE") possède une décomposition dans la variable d'entrée/sortie. Ensuite, pour le cas d'une décomposition incomplète, nous regardons si les éléments manquants se trouvent dans la variable d'entrée/sortie. En ce qui concerne les attributs répétitifs, la présence du mot clé OCCURS confirme la répétitivité de l'attribut. Il se peut aussi que certains attributs non supposés répétitifs le soient par la découverte de ce mot clé. A défaut du mot clé OCCURS, le remplacement ou la décomposition d'un champ supposé répétitif (ex: TELEPHONES ou ENFANTS) par la répétition d'un nom sur un même niveau inférieur et suffixé ou préfixé par un chiffre (ex: TEL1, TEL2 ou 1ENF, 2ENF) peut confirmer ce champ comme répétitif.

La confirmation de la surdéfinition d'un type segment peut aussi être trouvée avec les variables d'entrée/sortie. En effet, il est possible en COBOL de redéfinir une variable par le biais du mot clé REDEFINES. Si ce dernier porte sur la variable d'entrée/sortie toute entière, la surdéfinition est ainsi prouvée. Il est ensuite aisé de savoir quel champ rentre dans quelle définition en regardant les deux définitions de la variable. Dans le cas où le mot clé REDEFINES ne porte que sur une partie de la variable, il se peut que certains champs soient communs et d'autres différents. Si nous prenons l'exemple de la commande avec son entête et ses lignes, le champ du numéro de commande sera commun aux deux définitions mais les autres champs seront différents. Nous prenons donc comme règle de dénoncer une surdéfinition d'un type segment à chaque emploi du mot clé REDEFINES dans une définition de variable d'entrée/sortie. D'un autre côté,

l'absence de ce mot clé n'exclut pas la présence d'une surdéfinition. En effet, celle-ci peut être matérialisée par l'emploi pour un même type segment de plusieurs variables d'entrée/sortie ayant une définition différente. Il est donc utile de comparer les définitions de toutes les variables d'entrée/sortie d'un même type segment afin d'en déduire ou non une surdéfinition. Dans le cas où une surdéfinition est découverte, celle-ci ne donne pas lieu à un éclatement du type d'entité. Nous laissons cette responsabilité à la phase de conceptualisation des structures de données. Toutefois, nous matérialisons cette surdéfinition par des regroupements des attributs du type d'entité. Le type d'entité possède alors deux ou plusieurs (suivant le nombre de surdéfinitions) attribut(s) décomposable(s) reprenant tous les autres attributs de leur définition. Par conséquent, le rétro-ingénieur sait qu'il a devant lui une surdéfinition et prendra alors la décision lors de la conceptualisation des structures de données.

Le problème de la redéfinition d'un type d'entité se partage entre les deux zones analysées. Pour ce qui est de cette zone, elle permet d'avoir une confirmation sur la structure quasi identique des définitions et sur les noms ambigus mais ne permet pas d'établir clairement la redéfinition. Elle peut aussi clarifier quels sont les champs communs et les champs individuels.

La zone des codes programmes

Dans cette zone figurent les programmes. En COBOL, cette zone est appelée la PROCEDURE DIVISION. Parmi les programmes se trouvent bien sûr les appels à IMS. Cette zone est la plus dure à analyser car les renseignements que nous y cherchons sont noyés dans le reste du code. Toutefois, nous allons quand même essayer d'une part de savoir **où** chercher et d'autre part de savoir **quoi** chercher.

Essayons tout d'abord de savoir où chercher. Comme nous l'avons déjà vu, les programmes interagissent avec la BD par l'intermédiaire de *calls*. Ces *calls* sont le seul moyen de consulter ou de modifier la BD. Dès lors, les codes programmes susceptibles de renfermer des informations conceptuelles doivent forcément se trouver aux alentours de ces *calls*. Le rétro-ingénieur peut donc faire une sélection des modules, procédures ou fonctions renfermant ces appels à IMS. Cette sélection peut par exemple être exhaustive ou se référer à un certain pourcentage d'appels. D'autre part, le nom de ces modules, procédures ou fonctions peut s'avérer très utile. Par exemple, un module nommé STATISTIQUES_BD ou encore VERIF_CONTRAINTE_BD suppose qu'il contienne les informations que nous cherchons. Enfin, le plus simple est de se renseigner, si c'est possible, sur la façon dont l'application a été conçue. Ces zones de recherche ne sont bien sûr pas limitatives et chacun est libre de chercher où il veut.

Signalons aussi que les routines de captures de données attachées aux types segments ne sont là que pour les utilisateurs DB2 et sont généralement écrites en assembleur. Elles ne sont donc pas des endroits de recherche.

Maintenant que nous savons où chercher, il est important de savoir quoi chercher. Dans la zone de définition des variables, nous avons déjà confirmé les suppositions concernant les attributs et la surdéfinition de types segments. Il nous reste par conséquent quatre suppositions à confirmer: les identifiants, les clés étrangères, la cardinalité supérieure et la redéfinition d'un type d'entité.

En ce qui concerne les identifiants et les clés étrangères, l'apport des données est d'une aide beaucoup plus grande. Toutefois, dans certains programmes de vérification de l'état de la BD, il est possible de trouver du code examinant l'unicité des valeurs du champ suspecté et listant les valeurs de champs non uniques comme des erreurs. Nous

avons alors confirmation de l'unicité et donc de l'identifiant. De même, lors des insertions et suppressions des segments, si l'identifiant supposé est recopié ou effacé dans le champ supposé comme clé étrangère, alors cette dernière est vérifiée.

Quant à la cardinalité supérieure, elle peut également être confirmée par la présence de code spécialement conçu pour le besoin de la vérification de cette contrainte. Cependant, dans le code de manipulation normal, nous pouvons aussi retrouver les procédures d'insertion ou de consultation des segments enfants. Si ces insertions et consultations sont entourées d'une boucle ayant comme limite le nombre suspecté, cela correspond aussi à une confirmation de notre limite de cardinalité supérieure.

Pour confirmer la redéfinition d'un type d'entité, il convient d'analyser les insertions et les suppressions des segments concernés. Si à chaque insertion ou suppression d'un segment, les autres segments suspectés comme redéfinis sont également insérés ou supprimés, alors la redéfinition est établie. Nous savons alors que la BD contient les mêmes données dans plusieurs types segments. Comme pour la surdéfinition, nous ne prenons pas ici la responsabilité de modifier le schéma. La redéfinition est indiquée dans la description sémantique des types d'entités impliqués. Le rétro-ingénieur prendra sa décision lors de la l'intégration des schémas.

3.3.2. Les autres éléments conceptuels

Nous nous trouvons ici dans le cas où aucun indice ne nous permet de suspecter la présence de ces éléments conceptuels. Leur recherche en est rendue encore plus difficile. De plus, ces éléments conceptuels à rechercher dépendent du modèle conceptuel utilisé. Il est donc difficile d'en établir une liste exhaustive. Nous nous contenterons ici d'en citer les principaux.

3.3.2.1. Les contraintes de coexistante, d'au-moins-un et d'exclusion

Ces contraintes s'appliquent aux éléments facultatifs d'un type d'entité à savoir les attributs facultatifs et les rôles de cardinalité minimale à 0. La contrainte de coexistence indique une coexistence entre les éléments facultatifs à savoir que si l'un de ces éléments est présent, alors les autres éléments visés par la contrainte sont également présents. La contrainte d'au-moins-un indique qu'au moins un élément facultatif doit être présent. Et la contrainte d'exclusion indique que si un élément est présent, alors les autres sont exclus.

Ces contraintes sont implémentées dans le code programme aux abords des *calls* concernant les insertions et les suppressions. Il est donc facile d'en détecter la présence en regardant les enchaînements d'insertions ou suppressions pour un type segment. Chaque contrainte est codée d'une manière différente. Nous entendons par enchaînement des insertions la façon dont réagit le programme lors de l'insertion d'un segment. L'enchaînement doit donc être vérifié pour tous les *calls* concernant des insertions sur le type segment visé. L'enchaînement des suppressions est analysé de la même façon.

3.3.2.2. La cardinalité inférieure

Dans le code DDL, il n'est pas possible de spécifier le nombre minimum qu'un segment parent doit avoir comme segments enfants. Dès lors, la cardinalité inférieure du type d'entité parent doit être spécifiée dans les programmes.

Le code associé aux insertions et suppressions est encore une fois essentiel. La cardinalité inférieure se remarque par l'enchaînement des insertions et suppressions pour les segments enfants. Par exemple, si un segment enfant est toujours inséré lors de l'insertion d'un segment parent, une cardinalité inférieure de 1 pour le rôle du type d'entité parent dans le type d'association est établie.

3.3.2.3. Les dépendances fonctionnelles

Les dépendances fonctionnelles sont surtout trouvées par l'analyse des données. Toutefois, il est possible d'en découvrir dans le code programme. Le code programme les concernant se trouve alors attaché aux traitements des variables recevant les champs. Il est quand même très difficile d'établir une dépendance fonctionnelle car celle-ci doit être certifiée pour toutes les manipulations des champs en question et donc l'analyse doit porter sur tout le programme.

3.4. Apport des données pour l'extraction

Une analyse du contenu de la base de données peut s'avérer très utile pour confirmer des éléments conceptuels ou en découvrir de nouveaux. Pour cela, il faut coder des programmes chargés d'analyser ce contenu et ayant comme but la confirmation ou la découverte d'éléments conceptuels. Ces programmes ne doivent bien sûr altérer ni la BD ni les performances du système.

D'autre part, pour mener notre analyse dans les meilleures conditions, il faut que le contenu de la BD soit conséquent. En effet, pour des recherches comme celles des cardinalités, il faut avoir une BD assez importante pour pouvoir émettre des hypothèses sûres. Plus la BD sera grande, plus l'élément trouvé sera garanti.

D'une manière générale, nous pouvons dire que tous les éléments mis en évidence dans l'analyse du DML peuvent être reconfirmés ou découverts ici. Toutefois, certaines informations sont ici plus "faciles" à trouver que d'autres. Cette facilité s'explique par la possibilité de mettre en oeuvre un programme de recherche. Dans le cas où un tel programme n'est pas réalisable, seule la connaissance de la signification des données contenues dans la BD est salutaire.

Remarquons aussi que cette phase permet de résoudre les problèmes de dissimulation de structure, de structures non-déclaratives mais aussi de pertes de spécifications.

3.4.1. Les confirmations programmables

3.4.1.1. Les identifiants et les clés étrangères

Les identifiants sont les éléments les plus faciles à détecter. Il suffit de lire la BD en établissant une liste parallèle dans laquelle sont insérées les valeurs du ou des champs. Si, à la fin de la lecture, la liste ne contient que des valeurs uniques, alors ce(s) champ(s) peu(ven)t correspondre à un identifiant.

De même, il est possible de concevoir une procédure vérifiant la présence d'une clé étrangère en comparant la liste de ses valeurs avec la liste des valeurs de l'identifiant. Si la première est inclue dans la seconde, alors la clé étrangère est vérifiée.

3.4.1.2. Les dépendances fonctionnelles

Une analyse des correspondances de valeurs entre deux ou plusieurs champs est aisément faisable. Nous pouvons procéder de la même façon que pour les identifiants. Nous construisons une liste avec la valeur du premier champ et la valeur du deuxième champ correspondante. Si cette correspondance est toujours la même, la dépendance fonctionnelle est établie. Si la correspondance est toujours la même pour un groupe de valeur, alors nous pouvons établir une dépendance fonctionnelle multivaluée.

3.4.1.3. Les cardinalités inférieures et supérieures et les contraintes de coexistence, d'au-moins-un et d'exclusion

Les cardinalités sont facilement analysables par le biais des instructions DL/1. En effet, une suite de *calls* de type Get Next Within Parent montre le nombre de segments enfants de chaque segment parent. Nous prenons ensuite les valeurs minimales et maximales pour l'ensemble de la BD et nous obtenons ainsi nos cardinalités. Il faut quand même signaler que les cardinalités ne peuvent être que supposées. En effet, rien n'interdit que dans le futur, un segment parent ait plus de segments enfants que notre nombre maximal actuel. De même, rien n'interdit qu'un segment parent ait moins de segments enfants que notre nombre minimal actuel. Toutefois, la cardinalité inférieure est plus suspectable et elle est utile quand nous recherchons une confirmation d'une cardinalité minimale de 1.

Dans le même ordre d'idée, la vérification des contraintes de coexistence, d'au-moins-un et d'exclusion peut suivre le même procédé à savoir une suite de *calls* de type Get Next Within Parent. Ces *calls* sont alors dirigés de telle façon que nous puissions avoir une confirmation de ces contraintes.

3.4.1.4. La redéfinition d'un type d'entité

Si les valeurs des champs des segments suspectés sont identiques, alors il est fort probable que nous soyons en présence d'une redéfinition. Ceci est surtout valable si nous avons déjà détecté des identifiants pour ces types segments et que ces identifiants possèdent les mêmes valeurs.

3.4.1.5. L'attribut facultatif

Nous introduisons ici un nouvel élément qui n'était pas décelable lors de l'analyse du DML. Dans l'analyse des données, il nous est possible de découvrir l'existence d'un attribut facultatif par la mise à blanc ou à une valeur hors cadre (*high-value*) du champ correspondant. Un petit programme peut dès lors facilement parcourir la BD à la recherche de ces champs non remplis.

3.4.2. Les confirmations sur base de la signification du contenu

Comme nous l'avons dit, ces confirmations sont difficiles. Elles supposent de savoir reconnaître la nature des données de la BD car l'écriture d'un programme n'est pas facile ou pas possible. Ces confirmations portent sur la surdéfinition et sur les attributs.

En fait, savoir détecter la décomposition d'un champ en analysant uniquement les données demande de connaître exactement ce que veulent dire des données. Il en est de même pour les attributs manquants et les attributs répétitifs. Pour les surdéfinitions, le

problème reste le même et il faut savoir reconnaître à quelle définition appartient tel segment.

3.5. Intégration des schémas

Des différentes sources analysées, nous avons ressorti plusieurs schémas. Il serait donc maintenant utile d'intégrer ces différents schémas.

En ce qui concerne les sous-schémas externes dérivés des vues IMS (PSB), ceux-ci n'apportent pas d'informations supplémentaires sur le schéma général. Cependant, dans le cas où nous avons analyser plusieurs DBD, les schémas correspondant peuvent avoir eu des liens autres que les relations logiques. Il serait dès lors intéressant d'examiner ces schémas afin de voir s'ils n'ont pas d'éléments en commun. A ce titre, la recherche de redéfinitions effectuée plus haut nous est d'un apport sérieux.

Ces redéfinitions sont donc traitées et les types d'entités sont regroupés sous un seul type d'entité. Les types d'associations attachés aux différents types d'entités redéfinis sont maintenant attachés au nouveau type d'entité de regroupement. Les attributs qui n'étaient pas présents dans l'ensemble des types d'entités redéfinis deviennent maintenant des attributs facultatifs.

Le rétro-ingénieur peut effectuer cette intégration comme il lui semble le plus facile. Nous conseillons toutefois de prendre la hiérarchie la plus importante comme référence. Nous entendons par la que les nouveaux types d'entités provenant des redéfinitions sont ceux appartenant à cette hiérarchie de référence. Le schéma est alors axé autour de cette hiérarchie et est ainsi plus lisible.

Ayant relié les différents schémas, nous obtenons alors le schéma optimisé orienté IMS. Ce schéma est donc le résultat de la phase d'extraction des structures de données. Nous avons ainsi fait la rétro-ingénierie du processus de conception physique.

Chapitre 4 : La conceptualisation des structures de données

4.1. Introduction

Dans le chapitre précédant, nous avons procédé à l'extraction du schéma optimisé orienté IMS. Ayant obtenu ce schéma, il faut maintenant le libérer de ses optimisations et de ses structures IMS. Le but de ce chapitre est donc d'obtenir un schéma conceptuel normalisé. Ce dernier n'aura alors plus rien à voir avec la plateforme IMS sur laquelle il était implémenté.

Pour arriver à notre fin, nous avons deux sous-processus à savoir la conceptualisation de base et la normalisation conceptuelle. La difficulté de cette phase repose principalement sur le premier sous-processus. Comme le dit [HAINAUT, 93a]: *la conceptualisation de base est un processus critique qui requiert une connaissance avancée et une expérience dans la technologie et la conception de BD, en même temps que de très fortes connaissances du domaine.* La difficulté tient dans la multitude de possibilités pour modéliser un même élément conceptuel. A ce titre, IMS se différencie fortement d'autres modèles comme le modèle relationnel dans lequel les possibilités sont moindres. En effet, le modèle relationnel transforme les types d'entités en tables et les types d'associations en clés étrangères ce qui est assez direct. Par contre, le modèle hiérarchique et IMS en particulier offre par ses hiérarchies de relations physiques et ses types relations logiques de nombreux moyens pour modéliser les types d'associations, ceci sans tenir compte des clés étrangères et autres possibilités permises dans les programmes. Ces différentes possibilités requièrent d'avoir une expérience approfondie du modèle ce qui n'est pas notre cas. Nous allons donc tenter de le mener à bien avec nos connaissances.

Pour être le plus clair possible, nous allons dans un premier temps expliquer le processus de conception logique. Suite à cela, nous pourrons en faire la rétro-ingénierie dans la section sur la conceptualisation de base. Il nous restera enfin la normalisation conceptuelle et la stratégie.

4.2. La conception logique

Comme nous l'avons dit dans l'introduction, la conception logique IMS est très riche en possibilités de modélisation. Il convient donc d'attaquer le problème prudemment.

Pour cela, nous allons d'abord poser le problème à savoir quelles sont les règles que doit suivre un schéma conforme IMS. Ensuite, nous exposerons une méthode générale de transformation d'un schéma Entité-Association en un schéma conforme IMS que nous avons retirée de la littérature. Etant donné que cette méthode ne prend évidemment pas en compte tous les cas possibles, nous poursuivrons alors l'étude par une analyse la plus complète possible de chaque élément conceptuel.

4.2.1. Le schéma conforme IMS

Sur base des caractéristiques du modèle hiérarchique énoncées dans le chapitre 2, nous établissons un corpus de règles pour l'obtention d'un schéma conforme IMS.

Les règles 1,2,3 et 4 concernent les hiérarchies de relations physiques. Grâce à la modélisation des types relations logiques, nous pouvons établir les règles 5,6,7 et 8. Les autres règles viennent ensuite compléter le corpus.

1. un schéma conforme IMS est composé d'une ou plusieurs hiérarchie(s) de types d'entités;
2. une hiérarchie de types d'entités est composée d'un type d'entité racine et de zéro, un ou plusieurs sous-arbre(s) attaché(s) à ce type d'entité racine par un type d'association binaire dont les cardinalités sont de 1-1 pour le sous-arbre et de 0-N pour le type d'entité racine;
3. récursivement, un sous-arbre est composé d'un type d'entité racine et de zéro, un ou plusieurs sous-arbre(s) attaché(s) à ce type d'entité racine par un type d'association binaire dont les cardinalités sont de 1-1 pour le sous-arbre et de 0-N pour le type d'entité racine;
4. pour un type d'entité d'une hiérarchie, les types d'entités avec lesquels il possède un type d'association dans lequel la cardinalité de son rôle est de 0-N sont appelés ses types d'entités enfants physiques et, lui, est appelé le type d'entité parent physique;
5. pour tous les types d'entités du schéma sauf les types d'entités racines des hiérarchies, nous pouvons avoir **un** type d'association binaire le reliant à un type d'entité se

trouvant ailleurs dans le schéma et dont les cardinalités sont de 1-1 pour ce type d'entité appelé type d'entité enfant logique et de 0-N pour l'autre type d'entité appelé type d'entité parent logique;

6. tous les types d'entités du schéma peuvent avoir zéro, un ou plusieurs types d'associations binaires avec des types d'entités enfants logiques se trouvant ailleurs dans le schéma sauf avec les types d'entités racines des hiérarchies; les cardinalités sont de 0-N pour les types d'entités parents logiques et de 1-1 pour les types d'entités enfants logiques;
7. un type d'entité ne peut être à la fois enfant logique et parent logique;
8. un type d'entité enfant logique ne peut avoir des types d'entités enfants physiques qui soient aussi des types d'entités enfants logiques;
9. il ne peut y avoir de types d'associations ternaires, n-aires et récursifs;
10. il ne peut y avoir de types d'associations *one-to-one* et *many-to-many*;
11. il est possible de représenter un identifiant pour les types d'entités racines qui est alors également une clé d'accès; pour les autres types d'entités, il est possible de représenter un identifiant dont les composants sont un attribut et le rôle joué par son type d'entité parent dans le type d'association le reliant à ce parent; cet identifiant constitue également une clé de tri;
12. les attributs répétitifs, facultatifs, et décomposables n'existent pas;
13. les clés étrangères et les contraintes référentielles n'existent pas sauf dans le cas des identifiants composés vus à la règle 11;
14. les dépendances fonctionnelles n'existent pas;
15. les types d'associations et les rôles ne possèdent pas de noms.

Ce corpus de règles permet donc de vérifier si un schéma est conforme au modèle hiérarchique IMS. Nous pouvons maintenant nous attacher à rechercher les transformations qui vont nous permettre d'obtenir un tel schéma conforme sur base d'un schéma conceptuel quelconque.

4.2.2. Méthode générale de transformation d'un schéma Entité-Association en un schéma conforme IMS

Nous exposons ici une méthode de transformation d'un schéma Entité-Association en un schéma conforme IMS. Cette méthode reste bien sûr générale et fort théorique. Cependant, elle a le mérite de donner une marche à suivre et de mettre en évidence les transformations les plus naturelles à appliquer. Cette méthode est tirée de [BATINI, 92] et nous l'exposons ci-dessous.

Cette méthode, ou procédure suivant les termes de l'auteur, génère d'abord un schéma réseau et ensuite crée des duplications de types d'entités pour générer les hiérarchies. Ensuite, les possibilités offertes par les types relations logiques sont utilisées pour connecter ses hiérarchies. A noter aussi que la vue de l'auteur est de créer un schéma IMS et non un schéma Entité_Association conforme. Nous avons donc du transformer un peu la méthode. En voici les étapes:

Etape 1 : Pour chaque type d'association *one-to-one* R entre les types d'entités E1 et E2, faire l'une des deux possibilités ci-dessous:

- Créer un type d'association parent-enfant dans une direction entre E1 et E2; transférer les attributs de R dans E1 et/ou E2; le type d'entité qui est normalement accédé en premier devrait être choisi comme parent.
- Combiner les deux types d'entités E1 et E2 dans un seul type d'entité et représenter les attributs de R dans le type d'entité résultant.

Etape 2 : Pour chaque type d'association *many-to-many* R entre les types d'entités E1 et E2, créer un type d'entité additionnel E3 et transférer les attributs de R dans E3. Créer alors deux types d'associations *one-to-many*, un de E1 à E3 et l'autre de E2 à E3.

Etape 3 : Pour chaque type d'association n-aire R entre les types d'entités E1, E2, ..., En, créer un type d'entité Er pour représenter le type d'association. Transférer les attributs de R dans Er. Définir alors n types d'associations parent-enfant des types d'entités E1, E2, ..., En vers le type d'entité Er.

A ce point, le schéma est devenu un schéma représentant une structure réseau avec des types d'entités ayant plusieurs parents.

Etape 4 : Cette étape concerne les types d'entités ayant plusieurs parents. Si un type d'entité E1 possède deux (ou plusieurs) types d'entités E2 et E3 parents, et que ces types d'entités parents n'ont pas de parents, alors nous pouvons inverser les types d'associations et ainsi c'est E1 qui devient le type d'entité parent. Si la hiérarchie résultante est raisonnable et significative, alors l'étape est effectuée.

Nous émettons quelques doutes sur la faisabilité de cette étape 4 car elle inverse les types d'associations *one-to-many*.

Etape 5 : Cette étape s'applique également aux types d'entités ayant plusieurs parents. Deux cas sont distingués à savoir le cas où il n'y a que deux parents et le cas où il y a plus de deux parents.

- Cas de deux parents. Considérons le type d'entité enfant E1 et les types d'entités parents E2 et E3. Nous définissons E1 sous le parent avec lequel il est le plus relié. Ce choix peut être vue par exemple en fonction du nombre d'accès. Le lien avec le parent restant est alors instancié par un type d'association.

Ce dernier cas est indiqué ici pour rester cohérent avec la méthode de l'auteur qui voit ici une transformation en structure IMS et non en un schéma conceptuel conforme. De ce fait, il ne prévoit pas un deuxième type d'association mais un type relation logique IMS. Il est évident que cette transformation ne s'applique pas à notre cas car nous voulons seulement arriver à un schéma conforme ce qui est le cas si un type d'entité possède deux types d'entités parents.

- Cas de plus de deux parents. Nous définissons le type d'entité enfant E1 sous un type d'entité parent choisi parmi tous les parents. Ensuite, nous créons des copies secondaires de ce type d'entité E1 et nous les plaçons comme types d'entités enfants de ce type d'entité E1. Chacune de ces copies est alors reliée aux types d'entités parents restants par un type d'association (qui est vu par l'auteur comme une relation logique).

Etape 6 : Pour le cas où le système hiérarchique ne permet pas les types relations logiques, il est possible de procéder de la façon suivante pour les types d'entités ayant plusieurs parents:

- Pour chaque type d'entité Ex du schéma avec m (m>1) types d'entités parents, créer m copies du type d'entité Ex et redéfinir les types d'associations de telle façon qu'ils pointent maintenant sur chacune des copies du type d'entité Ex.

L'auteur indique également qu'il est possible de combiner les étapes 5 et 6 pour autant que les contraintes de modélisation soient respectées et que la signification originelle du schéma conceptuel soit préservée.

Etape 7 : Le schéma résultant possède maintenant une ou plusieurs hiérarchie(s) interconnectée(s). Si une seule hiérarchie est désirée, un type d'entité fictif est créé et des types d'associations sont créés entre ce type d'entité fictif et les types d'entités racines des hiérarchies. Le type d'entité devient alors racine et le schéma ne possède alors qu'une seule hiérarchie.

Cette méthode donne un avant goût de la difficulté à traduire des éléments conceptuels en un schéma conforme IMS. Malheureusement, elle reste trop générale et surtout elle ne prend pas en compte tous les cas possibles. De plus, certains problèmes comme la représentation des attributs répétitifs, décomposables et facultatifs ne sont pas exposés. Il faut donc palier à ces défauts et ceci est fait au point suivant.

4.2.3. Les transformations possibles des éléments conceptuels

Nous allons à présent voir chaque élément conceptuel en particulier et examiner toutes les possibilités de modélisation. Il faut pour cela garder à l'esprit tous les éléments de modélisation d'IMS mais aussi les possibilités de modélisation dans le code programme.

La liste de ces modélisations n'est pas exhaustive d'un point de vue général car elle se réfère uniquement aux possibilités offertes par le modèle hiérarchique. Pour une étude plus complète, on peut consulter [HAINAUT, 93b].

4.2.3.1. L'attribut répétitif

Il est possible de modéliser l'attribut répétitif dans la hiérarchie des types d'entités. En effet, rien n'est plus facile pour modéliser un attribut répétitif que d'en faire un type d'entité dépendant. Cet attribut répétitif est donc représenté par instance. La cardinalité du type d'association étant de 0-N, nous perdons comme information la répétitivité de l'attribut et son éventuel caractère obligatoire. Ces informations perdues doivent alors être codées dans les programmes. Cette transformation est illustrée à la Figure 54.

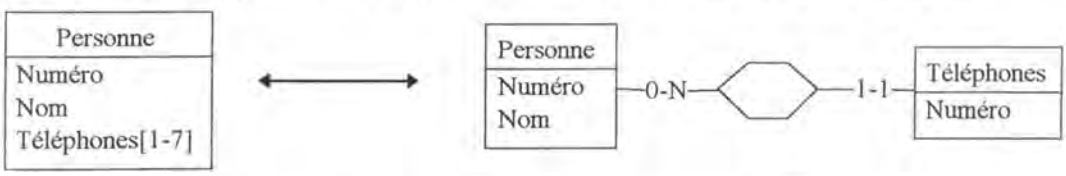

Figure 54 : Modélisation de l'attribut répétitif

4.3.2.2. L'attribut facultatif

Il est également possible de modéliser l'attribut facultatif sous la forme d'un type d'entité dépendant étant donné qu'en IMS, une occurrence d'un type d'entité parent ne possède pas obligatoirement une occurrence d'un type d'entité enfant. Le problème est que la répétitivité passe alors à N au lieu de 1. Encore une fois, les programmes ou d'autres moyens permettent de résoudre ce problème. La transformation est illustrée à la Figure 55.

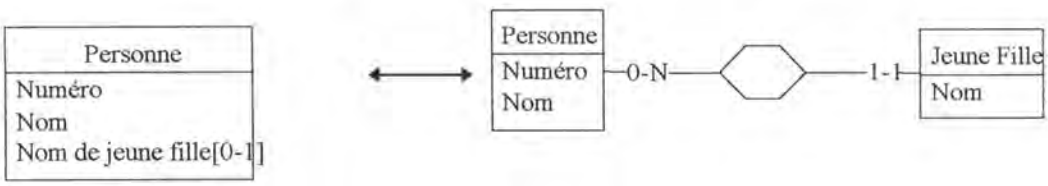

Figure 55 : Modélisation de l'attribut facultatif

4.3.2.3. L'attribut décomposable

Au même titre que les deux points précédents, les attributs décomposables peuvent être représentés par un type d'entité dépendant. Le nom du type d'entité dépendant provient du nom de l'attribut décomposé et les attributs de ce type d'entité proviennent des attributs composants. La transformation est illustrée à la Figure 56.

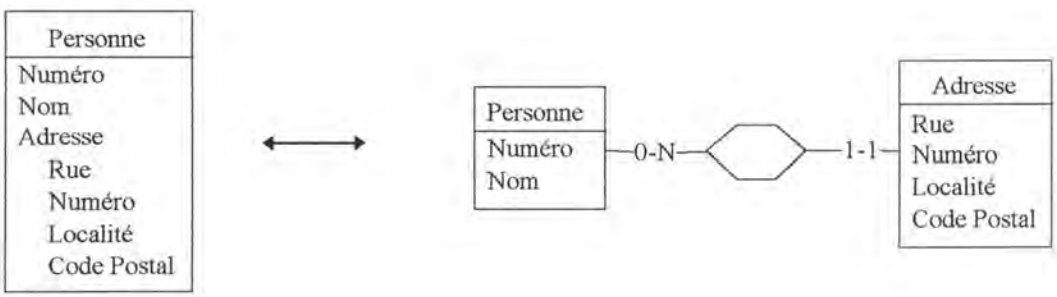

Figure 56 : Modélisation de l'attribut décomposable

4.3.2.4. Le type d'association many-to-many

Nous abordons ici un point crucial dans la détraduction. En effet, un type d'association *many-to-many* peut être modélisé de nombreuses façons. Nous allons donc en établir la liste.

Tout d'abord, il est possible de le supprimer. Le lien entre les deux types d'entités est alors représenté par deux clés étrangères que l'on rassemble dans un autre type d'entité auquel on rajoute aussi les attributs du type d'association. Ces clés étrangères représentent un identifiant dans les deux types d'entités. Il faut bien sûr prendre en charge la mise à jour de ces contraintes référentielles dans les programmes. Cette façon de modéliser est représentée à la Figure 57. Le grand problème de cette modélisation est l'identifiant. En effet, si un des types d'entités impliqués est un type d'entité dépendant, alors son identifiant est composé d'un attribut du type d'entité mais aussi de l'identifiant de son type d'entité parent. Il y a un problème pour copier l'identifiant comme clé étrangère. La solution consiste à trouver facilement le type d'entité correspondant sans passer par la hiérarchie, ce qui est résolu par la définition d'un index secondaire sur cet identifiant.

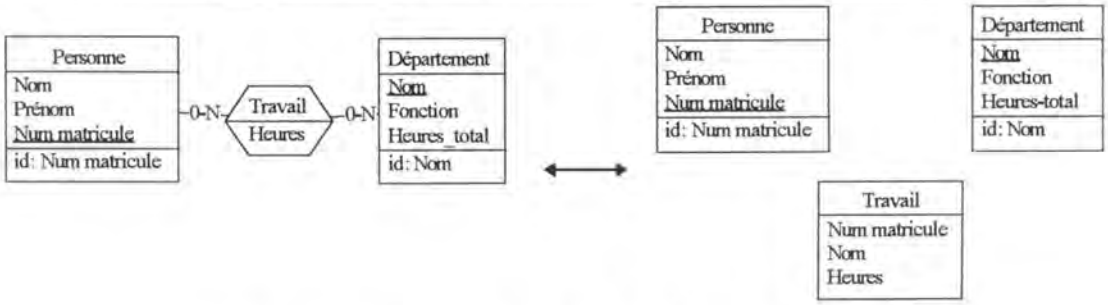

Figure 57 : Modélisation par des clés étrangères

Deuxièmement, un autre moyen consiste à concaténer les deux types d'entités en un seul. La charge de travail est alors reportée sur les programmes. Nous ne représentons pas cette modélisation ici car elle nous parait par trop fantasque.

Pour en revenir à des modélisations plus simples, nous pouvons également modéliser le type d'association *many-to-many* par le biais d'un type d'association parent-enfant (Figure 58). Nous avons alors un type d'association *one-to-many*. Si le type d'association *many-to-many* n'est accessible que dans un sens, cette solution convient bien. Si le type d'association doit être accessible dans les deux sens, alors il est encore possible d'établir un index secondaire sur le type d'entité enfant. Nous avons alors l'accès dans les deux sens. Les attributs du type d'association sont copiés dans le type d'entité enfant et ainsi n'apparaissent que si le type d'association existe. Pour représenter des *Personnes* qui ne seraient attachées à aucun *Département*, nous pouvons créer une occurrence fictive du type d'entité *Département* à laquelle seraient attachées ces personnes.

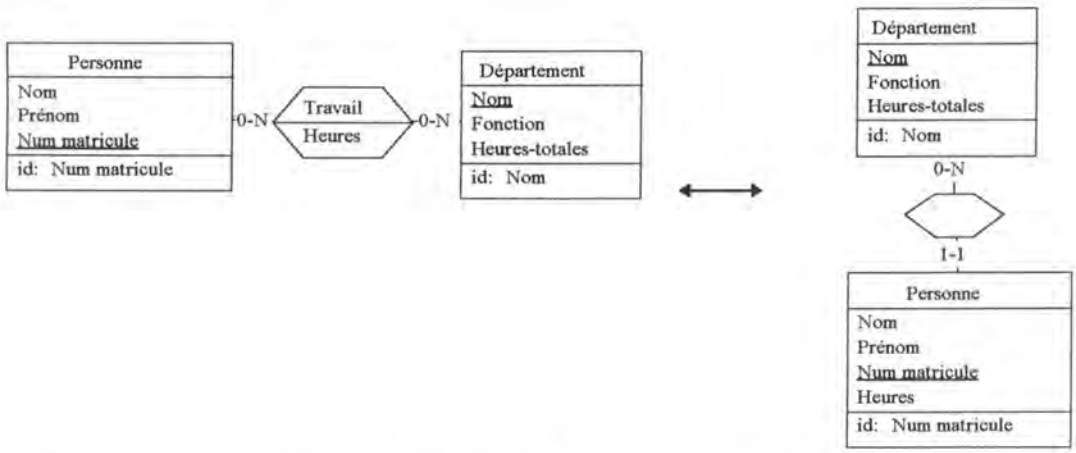

Figure 58 : Modélisation par un type d'association *one-to-many*

Une autre façon de faire pour obtenir l'accès dans les deux sens est de définir deux hiérarchies contenant deux types d'associations parent-enfant mais ayant chacune un des deux types d'entités comme racine (Figure 59). Nous pouvons alors accéder le type d'association *many-to-many* dans les deux sens. Le prix à payer est l'obligation d'effectuer une redéfinition de deux types d'entités et d'assurer l'intégrité pour ces deux redéfinitions. Il est encore possible bien évidemment d'avoir un index secondaire sur les types d'entités enfants.

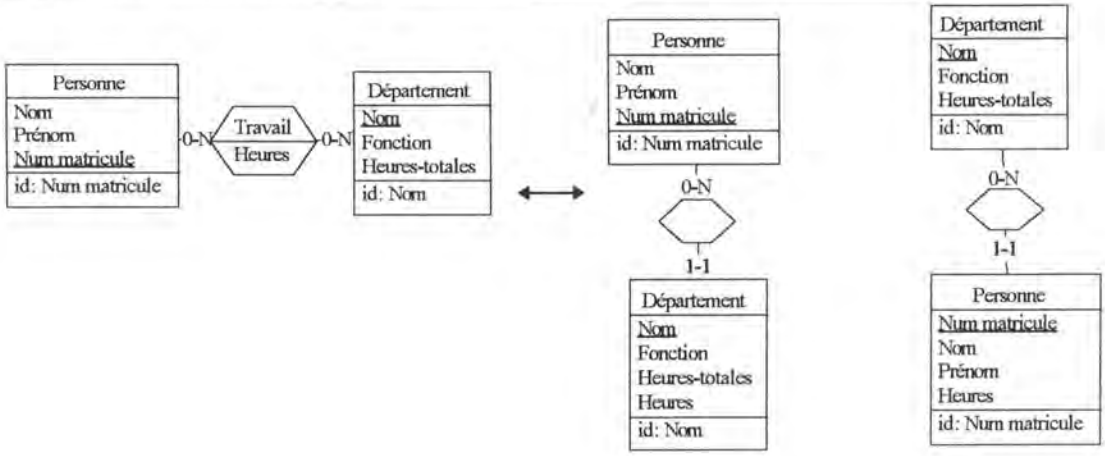

Figure 59 : Modélisation par deux types d'associations *one-to-many*

En utilisant les facilités offertes par les types relations logiques, il est aussi possible de modéliser notre type d'association *many-to-many*. Encore une fois, selon que l'on veut l'accès dans un sens ou dans les deux, nous pouvons employer les types relations logiques unidirectionnels ou bidirectionnels. Ces deux façons de modéliser ne se représentent dans le schéma logique que d'une seule façon à savoir un nouveau type d'entité pour le type d'association qui est relié aux deux types d'entités (Figure 60).

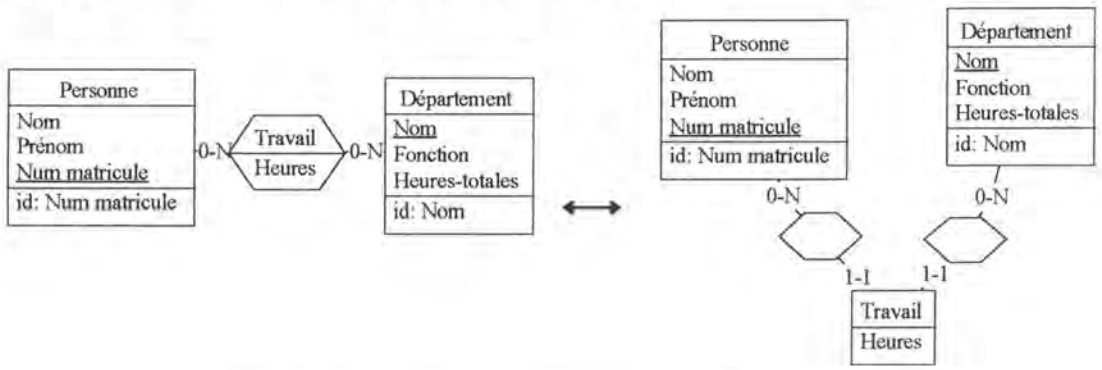

Figure 60 : Modélisation par un type d'entité

Notons enfin qu'une combinaison des différentes modélisations vues ici est toujours possible. Par exemple, on peut combiner des types d'associations *one-to-many* avec des clés étrangères. Cette remarque montre combien les possibilités de modélisation sont grandes dans le contexte hiérarchique.

4.3.2.5. Le type d'association one-to-one

Ce type d'association peut être remplacé par un nouveau type d'association *one-to-many*. Les attributs du type d'association *one-to-one* sont représentés dans le type d'entité enfant du nouveau type d'association. Le maintien du type d'association *one-to-one* se fait par les procédures. Toutefois, IMS permet dans un cas particulier de modéliser un type d'association *one-to-one*. Etant donné que ce cas est particulier et est surtout là pour des raisons techniques (pointeurs) et non conceptuelles, nous considérons qu'il n'est pas possible de le modéliser. La modélisation possible est illustrée à la Figure 61.

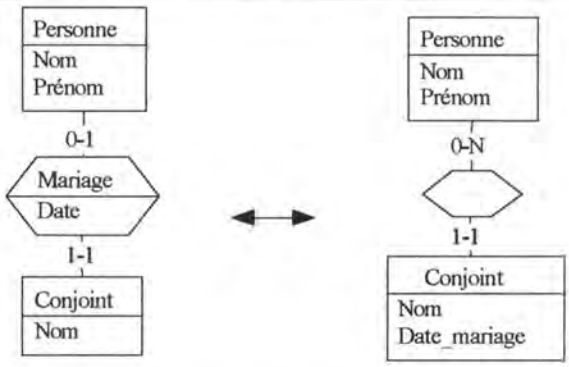

Figure 61 : Modélisation par un type d'association *one-to-many*

Une autre modélisation a été évoquée dans la méthode générale. Cette modélisation propose de réunir dans un seul type d'entité les deux types d'entités reliés ainsi que les attributs du type d'association. Ce cas est illustré à la Figure 62.

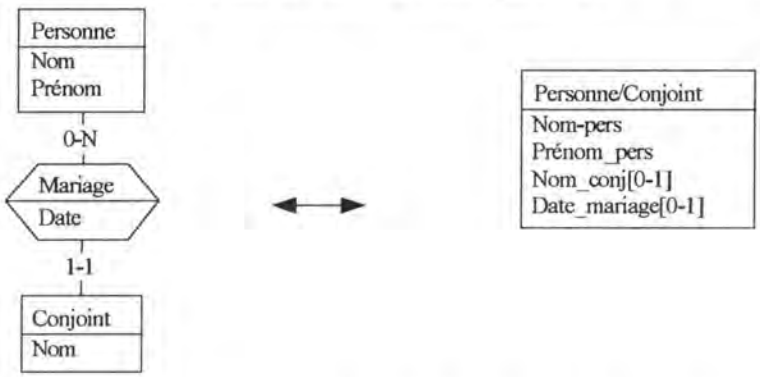

Figure 62 : Modélisation par un seul type d'entité

4.3.2.6. Le type d'association récursif

Une première façon de modéliser un type d'association récursif est de représenter les rôles par deux types d'entités (Figure 63). Ces derniers sont ensuite reliés au type d'entité par deux types d'associations *one-to-many*. Dans l'exemple, pour retrouver le chef d'une personne, il faut consulter le type d'entité *chef* de la personne et voir par le deuxième type d'association la personne qui est ce chef. Ces nouveaux types d'entités sont en fait utilisés comme des pointeurs. La perte d'information sur les cardinalités (le rôle *a comme chef* de 0-1 devient 0-N) doit être prise en compte par les programmes.

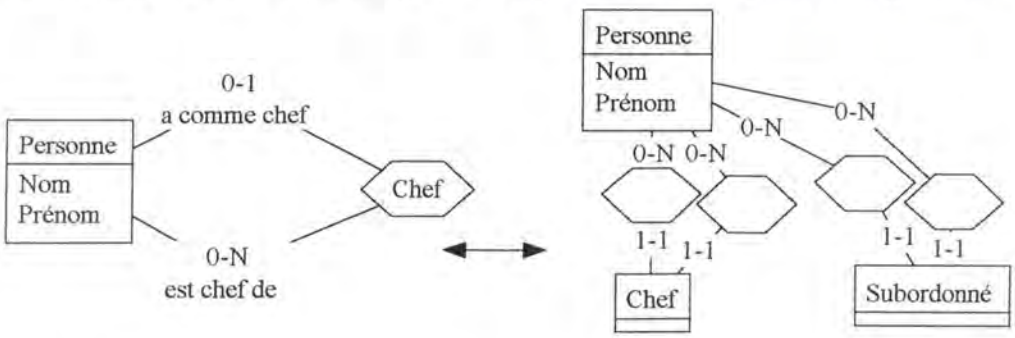

Figure 63 : Modélisation par deux types d'entités pointeurs

Si on désire accéder directement aux informations du chef ou des subordonnés, nous pouvons recopier les attributs désirés dans les nouveaux types d'entités. Il n'est alors plus indispensable de garder les deuxièmes types d'associations car ces types d'entités perdent leur caractère de pointeurs. Nous créons donc deux types d'entités pour les deux rôles. Ces types d'entités sont rattachés au type d'entité par deux types d'associations (Figure 64). Pour retrouver le chef d'une personne, il suffit alors de consulter le type d'entité *Chef* et pour retrouver les subordonnés de consulter le type d'entité *Subordonné*. Pour la perte au niveau des cardinalités, il faut se référer aux programmes.

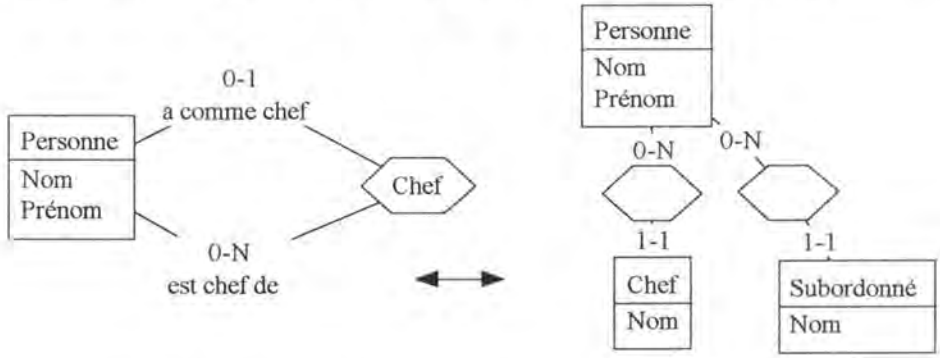

Figure 64 : Modélisation par deux types d'entités

D'autre part, nous pouvons voir le type d'entité *Chef* non pas comme le nom du chef de la personne mais comme un indicateur sur la personne qui montre si cette personne est ou on un chef. De ce fait, il est possible de créer un type d'entité enfant pour un rôle et de rattacher l'autre rôle comme un type d'entité de ce type d'entité enfant (Figure 65). Dans cet exemple, les deux types d'entités *Chef* et *Subordonné* sont créés et le type d'entité *Subordonné* est rattaché au type d'entité *Chef*. Ainsi, pour retrouver les subordonnés, nous sommes obligés de passer par le type d'entité *Chef* ce qui est normal car une personne n'a de subordonnés que si elle est chef. Une fois sur le subordonné, nous consultons le type d'entité *Personne* auquel il est relié.

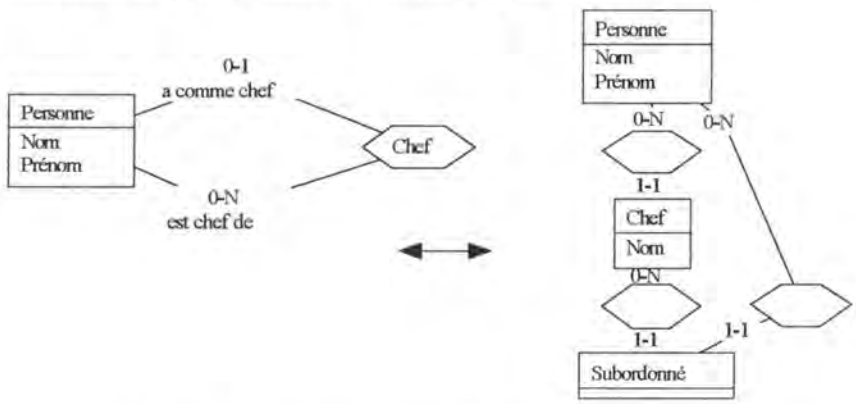

Figure 65 : Modélisation par suite de types d'entités

Enfin, un type d'association récursif peut aussi être modélisé au moyen de clés étrangères dans le type d'entité. Le type d'entité évite ainsi d'avoir un ou deux type(s) d'entité(s) enfant(s) pointeur(s). Le problème se trouve dans le maintien de ces clés étrangères par les programmes. De plus, nous sommes alors limités dans les cardinalités par le nombre de clés étrangères que le concepteur veut bien définir.

Enfin, nous pouvons combiner à loisir les différentes possibilités offertes suivant nos desiderata. Le concepteur prend alors en compte des facteurs techniques tels que les accès ou le volume maximum de la BD.

4.3.2.7. Le type d'association n-aire

Cette modélisation constitue un problème épineux en IMS. En effet, même par les types relations logiques, il n'est pas possible de modéliser efficacement ce type d'association.

Le moyen le plus simple est celui exposé dans les étapes de la méthode générale. Les programmes doivent alors supporter la charge que représente le maintien de ces redéfinitions. Cette modélisation est illustrée à la Figure 66.

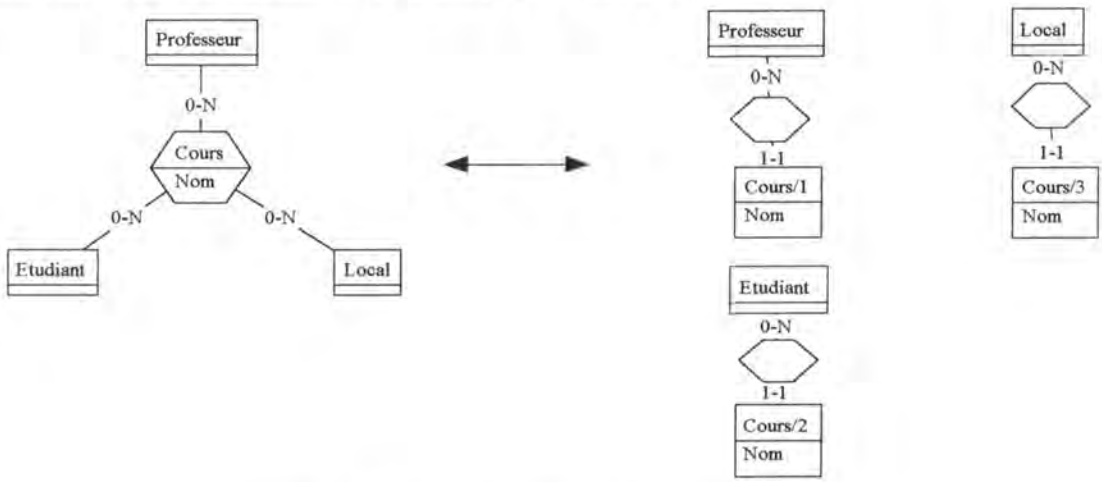

Figure 66 : Modélisation par redéfinition

Pour alléger la tâche des programmes, nous pouvons relier un type d'entité redéfini à deux types d'entités parents. Nous divisons alors par deux la charge de travail pour le maintien de la redéfinition par les programmes. Sur base de la Figure 66, nous procédons à cette modélisation ce qui donne la Figure 67. Bien sûr, les programmes doivent encore assurer le maintien pour les redéfinitions restantes.

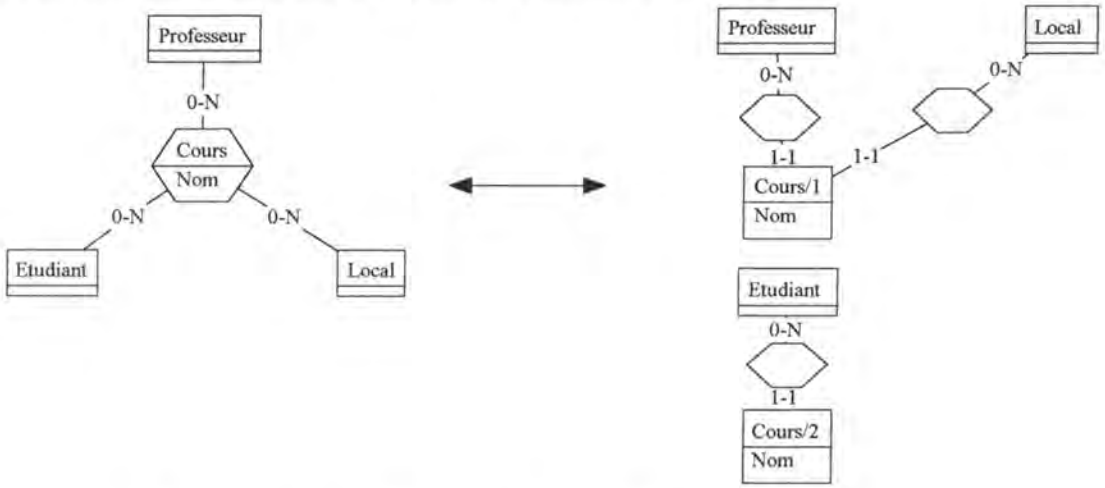

Figure 67 : Modélisation par redéfinition allégée

Enfin, il est possible de créer un nouveau type d'entité reprenant en plus des attributs du type d'association les identifiants des types d'entités impliqués dans le type

d'association n-aire. Ces identifiants constituent alors des clés étrangères dont il faut assurer l'intégrité avec le type d'entité source. Bien sûr, la combinaison des possibilités reste de mise et est laissée en choix au concepteur.

Pour terminer, notons que dans les deux premiers cas, nous avons d'abord du transformer le type d'association n-aire en un type d'entité relié aux types d'entités jouant les rôles dans le type d'association. Nous avons alors le cas du point suivant à savoir un type d'entité avec plus de deux parents. Nous avons quand même cru bon de montrer ici deux modélisations possibles.

4.3.2.8. Un type d'entité avec plus de deux parents

Comme nous venons de le dire, les deux modélisations portant sur le type d'association n-aire sont également valables ici. Cependant, à des fins d'illustration de la méthode générale et plus particulièrement de la deuxième phase de l'étape 5, nous proposons ici un autre exemple se trouvant à la Figure 68. Les programmes ont pour mission de gérer la maintenance de ces redéfinitions.

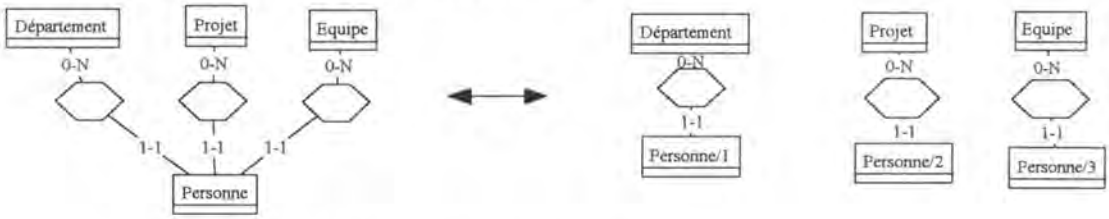

Figure 68 : Modélisation par redéfinition

Il est ici possible d'alléger la charge des programmes. En effet, comme illustré à la Figure 69, si nous attachons les copies du type d'entité à ce même type d'entité, les programmes ont alors facile à maintenir les redéfinitions car ces dernières sont rattachées au même type d'entité. Cette facilité est bien sûr également possible pour la modélisation des types d'associations n-aires.

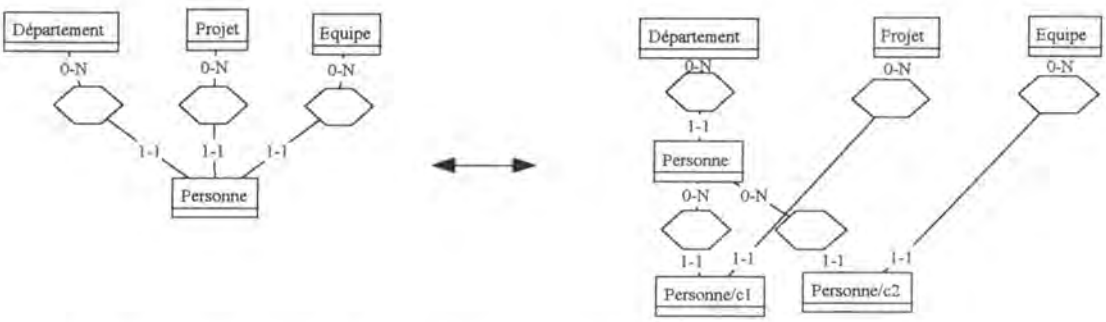

Figure 69 : Modélisation avec plusieurs types d'associations

Enfin, le procédé de la clé étrangère est encore une fois d'une grande utilité. Il suffirait dans notre exemple de mettre l'identifiant du type d'entité *Projet* et du type d'entité *Equipe* dans le type d'entité *Personne* qui serait toujours rattaché au type d'entité *Département*. Les programmes auraient alors le devoir de gérer la maintenance de ces clés étrangères mais la solution évite la redondance des redéfinitions.

Encore une fois, la combinaison des différentes possibilités et notamment des clés étrangères avec les types d'associations *one-to-many* est possible. Ces combinaisons laissent un grand choix au concepteur qui peut optimiser à loisir son schéma.

Cependant, ces combinaisons sont très mal vues par le rétro-ingénieur car elles peuvent lui cacher des modélisations ou encore lui faire déduire de mauvaises modélisations. Il faut donc être très prudent dans la phase de conceptualisation de base de la section suivante.

4.3. La conceptualisation de base

4.3.1. La phase préliminaire

Nous avons regroupé ici les tâches à accomplir en préliminaire aux deux sous-processus principaux. Ces tâches ont pour objectifs d'éclaircir le schéma et de le débarrasser de ses éléments techniques.

Tout d'abord, les noms des types d'entités et des attributs doivent subir un petit nettoyage. En effet, il est commun aux concepteurs de coder ces noms de façon à ce qu'ils puissent directement établir les caractéristiques des éléments. Dans une étude de cas, nous avons ainsi pu remarquer que tous les noms des types segments et des champs d'une BD commençaient par les deux mêmes lettres servant à l'identification du projet concerné par la BD. Il convient donc de supprimer tous ces préfixes d'ordres techniques et de donner un nom plus explicite aux éléments du schéma.

Deuxièmement, il faut donner un nom significatif aux rôles et aux types d'associations du schéma. En effet, la structure hiérarchique ne permettant qu'un type relation physique et un type relation logique entre un type segment parent et un type segment enfant, les noms de rôles et de types d'associations n'ont plus de raison d'être et n'existent pas. Pour éclaircir le schéma, il faut donc retrouver ces noms de rôles. Malheureusement, nous n'avons pas de sources nous permettant de savoir clairement ces noms. Il faut par conséquent se baser sur les deux types d'entités reliés et sur son bon sens.

Ensuite, les collections et les clés d'accès sont supprimées car elles n'apportent que des informations techniques liées à la conception physique. Effectivement, peu importe de savoir au niveau conceptuel sur quel support sont stockés les types d'entités et par quel moyen technique ils sont accédés.

De même, les descriptions techniques des éléments du schéma sont également enlevées car elles ne sont aussi là que pour donner des caractéristiques techniques des éléments. Les descriptions sémantiques sont gardées car nous les considérons du niveau conceptuel.

Enfin, dans le cas de l'utilisation d'un type d'entité fictif ayant comme but la présence d'une seule et unique hiérarchie, il faut éliminer ce type d'entité. Toutefois, il faut réaliser

cette opération avec la plus grande prudence et faire toutes les vérifications possibles pour être sûr du caractère fictif de ce type d'entité.

4.3.2. La détraduction

Comme déjà évoqué, la détraduction essaye de retrouver les structures conceptuelles desquelles le développeur a dérivé les structures logiques du schéma source. Il convient donc de connaître les différentes façons de modéliser logiquement un même élément conceptuel ce qui a été fait à la section précédante. Il nous reste donc ici à indiquer les moyens de détecter ces transformations afin de retrouver nos éléments conceptuels.

4.3.2.1. L'attribut répétitif

Comme nous l'avons vu, l'attribut répétitif n'est pas représentable en IMS. Toutefois, il a été indiqué qu'il était possible de le découvrir par l'analyse du DML. Dans la conception logique, nous avons également remarqué que nous pouvions modéliser cet attribut dans la hiérarchie des types d'entités.

Pour cette dernière modélisation, il peut notamment être repéré par le fait que le type d'entité ne possède qu'un seul attribut. Cependant, dans le cas d'un attribut répétitif décomposable, le type d'entité possède plusieurs attributs.

Nous pouvons donc transformer ce type d'entité en attribut répétitif du type d'entité parent. La répétitivité correspond à la cardinalité supérieure que jouait son type d'entité parent dans le type d'association.

4.3.2.2. L'attribut facultatif

De même, l'attribut facultatif n'est pas non plus représentable en IMS mais a également pu être repéré par l'analyse des données. Cependant, il est aussi possible de le modéliser sous la forme d'un type d'entité dépendant.

Cet attribut est repérable par le fait qu'il ne possède généralement qu'un seul attribut sauf dans le cas d'un attribut décomposable facultatif.

Nous transformons ce type d'entité en un attribut facultatif du type d'entité parent sous la condition que la cardinalité inférieure que jouait son type d'entité parent dans le type d'association est restée de 0. Nous entendons par là que l'analyse du DML et des données n'est pas venue mettre cette cardinalité à 1.

4.3.2.3. L'attribut décomposable

Au même titre que les deux points précédents, les attributs décomposables ne sont pas directement représentables en IMS mais peuvent avoir été détectés dans le DML. Le procédé d'en faire un type d'entité dépendant est aussi faisable et il nous importe donc de le transformer.

Ce type d'entité se voit donc devenir un attribut décomposable de son type d'entité parent. Le nom de l'attribut est donné par le nom du type d'entité et les attributs composants sont ceux du type d'entité transformé.

4.3.2.4. Le type d'association many-to-many

Nous avons donc vu dans la précédante section les nombreuses possibilités de modélisation de ce type d'association. Nous devons maintenant les transformer. Nous nous trouvons devant cinq cas: un type d'entité avec deux clés étrangères, un type d'entité concaténé, les types d'associations provenant des types relations logiques, les deux hiérarchies avec redéfinitions et le type d'association provenant du type relation physique parent-enfant.

Pour les clés étrangères, il est possible de les repérer dans le code DML et dans les données. Les clés sont supprimées et le type d'association *many-to-many* est créé entre les deux types d'entités. Les attributs qui ne sont pas des clés étrangères deviennent des attributs du type d'association.

Lors d'une concaténation, il est plus difficile de la détecter. En cas de découverte, le type d'entité est alors éclaté en deux types d'entités avec un type d'association *many-to-many* entre les deux. Les autres types d'associations du type d'entité concaténé sont reliés si possible aux nouveaux types d'entités suivant leur nature. Si cela n'est pas possible, alors ils sont reliés aux deux types d'entités. Cette structure sera retravaillée lors de la normalisation conceptuelle.

Pour les types d'associations provenant de types relations logiques, nous transformons le type d'entité issu du type segment pointeur en un type d'association. Nous pouvons le faire car les cardinalités des rôles joués par ce type d'entité dans les deux types d'associations *one-to-many* sont de 1-1. Si ce type d'entité possède des attributs, ceux-ci deviennent les attributs du type d'association *many-to-many*. Les cardinalités des rôles sont de 0-N.

Pour les deux hiérarchies avec redéfinition, nous rassemblons en un seul type d'entité les types d'entités redéfinis. Les types d'associations qui étaient attachés aux deux types d'entités redéfinis sont désormais attachés au nouveau type d'entité. Nous avons alors deux types d'entités avec deux types d'associations *one-to-many* entre les deux, chacun ayant une partie *one* et une partie *many*. Ces deux types d'associations sont alors transformés en un seul type d'association *many-to-many*. La cardinalité des rôles est alors de 1-N du fait des cardinalités de 1-1 des types d'associations *one-to-many*. Toutefois, nous permettons une cardinalité de 0-N car la procédure de maintien de l'intégrité peut permettre cette cardinalité.

Enfin, dans le cas où nous avons un simple type relation physique parent-enfant, le type d'association *one-to-many* est transformé en type d'association *many-to-many*. La cardinalité du rôle joué par le type d'entité qui était type segment enfant est de 1-N. Ceci est du à la cardinalité de 1-1 du rôle joué dans le type d'association *one-to-many*. Toutefois, si le type d'entité parent possède une entité supposée fictive à laquelle sont rattachées des entités du type d'entité enfant, nous pouvons alors supposer une cardinalité de 0-N plutôt que 1-N.

4.3.2.5. Le type d'association one-to-one

Ce type d'association peut déjà avoir été découvert par les analyses précédantes. Comme ces dernières ainsi que la conception logique le montraient, ce type d'association est représenté par un type d'association *one-to-many*.

Ce type d'association *one-to-many* est donc transformé en type d'association *one-to-one*. La cardinalité supérieure du rôle joué par le type d'entité parent devient alors 0-1. La cardinalité du rôle joué par le type d'entité anciennement enfant reste de 1-1.

Dans le cas de la concaténation des deux types d'entités en un seul, le cas est considéré dans le cadre de la déoptimisation. En résumé, nous éclatons le type d'entité en deux et nous relions les deux types d'entité par un type d'association *one-to-one*. Les cardinalités nous sont indiquées par l'analyse des données et des programmes.

4.3.2.6. Le type d'association récursif

La recherche d'un type d'association récursif est rendue difficile par les nombreuses possibilités de modélisation offertes au concepteur. Premièrement, nous avons la modélisation par des types d'associations provenant de types relations logiques. Cette modélisation nous incite donc à contrôler tous les types d'associations provenant de types relations logiques. Nous pouvons alors repérer une transformation d'un type d'association récursif et ainsi le remettre dans sa modélisation initiale.

Ensuite, il faut également contrôler les types d'entités enfants car ceux-ci peuvent également être utilisés comme implémentation. Le nom de ces types d'entités enfants peut aider mais c'est surtout le comportement du type d'entité analysé lors du DML qui est important.

D'autre part, la présence d'un cycle comme celui de la Figure 65 est susceptible de renfermer un type d'association récursif. Il est donc important d'y prêter une grande attention. Dans les trois cas que nous venons d'énoncer, les cardinalités des deux rôles sont de 0-N. Toutefois, l'analyse des données et des programmes peut venir changer cette cardinalité.

Enfin, par la présence de clés étrangères, il est possible de détecter un type d'association récursif. Les cardinalités inférieures sont toujours de 0 et les cardinalités supérieures nous sont données par le nombre de clés étrangères pour chaque rôle.

4.3.2.7. Le type d'association n-aire

La conception logique a mis en évidence deux façons de modéliser les types d'associations n-aires. Dans le premier cas prenant en compte la redéfinition d'un type d'entité, il faut analyser les types d'entités supposés redéfinis et voir si cela ne cache pas un type d'association n-aire. A ce titre, la redéfinition dans laquelle les types d'entités redéfinis sont reliés au même type d'entité permet de faire une recherche plus facile.

Le deuxième cas où nous utilisons des types d'associations provenant de types relations logiques est également suspecté et fait l'objet d'une analyse. Dans ces deux derniers cas, les types d'entités redéfinis sont rassemblés en un seul type d'entité ce qui nous ramène au cas d'un type d'entité avec plus de deux parents. Ce cas est alors transformé en un type d'association n-aire.

Enfin, un type d'entité regroupant plusieurs clés étrangères est aussi source d'un type d'association n-aire. Lorsque nous avons repéré ce type d'entité, nous le transformons en un type d'association n-aire tout en préservant ses types d'associations avec les autres types d'entités. Les attributs qui ne sont pas des clés étrangères sont maintenus dans le type d'association. Les cardinalités de ce dernier nous sont données par l'analyse du DML et des données.

4.3.2.8. Un type d'entité avec plus de deux parents

Pour retrouver cette situation, nous sommes confrontés à la redéfinition et ses différentes facettes ainsi qu'aux clés étrangères. Nous allons appliquer le même procédé que pour le type d'association n-aire. Une fois les éléments repérés, nous appliquons les deux phases suivantes.

En ce qui concerne la redéfinition, un seul type d'entité reprend les types d'entités redéfinis et les types d'associations que ces derniers avaient sont rattachés au nouveau type d'entité. Les cardinalités sont bien sûr de 1-1 pour le type d'entité nouvellement créé et de 0-N pour ses types d'entités parents.

En repérant un clé étrangère, nous la transformons en un type d'association *one-to-many* avec le type d'entité dont elle est l'identifiant et nous retrouvons alors une structure dans laquelle un type d'entité possède plus de deux parents.

4.3.3. La déoptimisation

Le SGBD IMS amène souvent le concepteur de la base de données à faire des choix d'optimisation. Nous allons dans cette partie éclairer ces choix. Pour ce faire, nous analyserons les différentes familles d'optimisation à savoir la dénormalisation, la redondance structurelle et la restructuration.

4.3.3.1. La dénormalisation

La technique de dénormalisation la plus courante consiste à rassembler deux types d'entités reliés par un type d'association *one-to-one* en un seul type d'entité, afin de réduire les accès d'entrée/sortie. Cette structure est repérable par le fait que le déterminant d'une dépendance fonctionnelle n'est pas l'identifiant du type d'entité. Un autre moyen que nous avons mis en évidence est la surdéfinition. Cette dernière établit clairement l'existence d'une telle structure au contraire du premier moyen qui ne fait que le supposer.

Si nous établissons l'existence de ce type d'optimisation, alors nous éclatons le type d'entité en deux types d'entités reliés par un type d'association *one-to-one*. Les types d'associations reliés au type d'entité initial sont rattachés aux deux nouveaux types d'entités. Cette structure sera retravaillée dans le cadre de la normalisation conceptuelle.

Un autre technique de dénormalisation consiste à représenter un type d'association *one-to-many* par une clé étrangère plutôt que par une relation physique ou logique. L'existence de cette clé ayant été vérifiée, elle est supprimée et remplacée par un type d'association *one-to-many* entre les deux types d'entités concernés.

4.3.3.2. La redondance structurelle

La redondance structurelle consiste à ajouter des éléments au schéma qui sont calculables à partir d'autres éléments. Nous pouvons en ressortir trois en IMS.

Tout d'abord, la plus classique est la redondance d'attributs où nous trouvons un même attribut dupliqué dans plusieurs types d'entités. Cette redondance d'attributs est repérable par l'analyse du DML et du DDL. Cette redondance est pratique en IMS car elle permet d'éviter le parcours de toute une hiérarchie pour retrouver une information. D'un autre côté, elle implique, outre la redondance, l'obligation de coder dans les

programmes le maintien des valeurs de l'attribut. Si nous trouvons une telle redondance, l'attribut dupliqué est tout simplement supprimé dans notre schéma. Il est cependant noté dans la description sémantique du type d'entité où il a été supprimé.

Ensuite, la deuxième redondance structurelle apparaît par la duplication de types d'entités. En effet, pour les mêmes raisons que les attributs, il est utile en IMS de dupliquer des types d'entités dans plusieurs hiérarchies. Nous avons alors le problème de la redéfinition qui nécessite du code programme et est par conséquent repérable. Cette redéfinition amène à la suppression du type d'entité dupliqué. Cette suppression doit faire l'objet de toutes les attentions car il ne faut pas que celle-ci altère le schéma. Le ou les type(s) d'entité(s) supprimé(s) doi(ven)t donc être choisi(s) judicieusement. De plus, si le type d'entité supprimé jouait des rôles, alors ces derniers doivent être reliés au type d'entité restant.

4.3.3.3. La restructuration

A la différence des deux précédantes optimisations, celles-ci n'entraînent pas de redondance. Elles sont généralement divisées en quatre parties.

Tout d'abord , nous avons le partitionnement horizontal qui consiste à couper un type d'entité en deux de telle façon que la population du type d'entité initial soit partitionnée dans les nouveaux types d'entités. Les grands types segments dépendants n'étant pas bien vus en IMS, cette restructuration est souvent employée. Elle peut être repérée par une redéfinition dans les zones de données mais pas dans les codes programmes car alors elle passerait du côté des redondances structurelles. Dans le cas d'une découverte d'une telle restructuration, les types d'entités éclatés sont rassemblés dans le même type d'entité. Les rôles que jouaient ces types d'entités sont rattachés au nouveau type d'entité.

Ensuite, il y a le rassemblement horizontal qui est l'inverse du partitionnement horizontal c'est-à-dire que deux types d'entités différents sont rassemblés en un seul. Cette technique est repérable par la surdéfinition. Il est alors facile d'éclater ce type d'entité en plusieurs types d'entités indépendants. Les types d'associations attachés au type d'entité initial sont rattachés aux types d'entités éclatés.

Dans le même esprit, nous avons le partitionnement vertical et le rassemblement vertical. Le partitionnement vertical est courant en IMS à cause de la structure hiérarchique. En effet, le stockage des types segments est tel que le concepteur doit juger correctement la taille des différents types segments. De plus, les types segments de taille trop grande ne sont pas appréciés par IMS, surtout si ces types segments sont racines car alors ils entraînent la lecture de plusieurs blocs pour avoir le type segment dépendant désiré. Il n'est dès lors par rare de voir un type d'entité se transformer en un type segment et plusieurs types segments enfants de ce type segment. Ces types segments enfants sont alors placés le plus à droite possible dans l'ordre hiérarchique pour éviter qu'ils n'encombrent le bloc de stockage initial. Il est donc probable que notre hiérarchie possède des types segments dépendants qui sont en fait des attributs du type d'entité parent. Ces types d'entités dépendants deviennent donc des attributs du type d'entité. Le rassemblement vertical est beaucoup plus rare en IMS. De plus, nous pouvons le comparer à la dénormalisation. De ce fait, il n'est pas évoqué ici.

4.3.4. Les difficultés de la conceptualisation de base

Pour conclure cette phase, nous voulons insister encore une fois sur les difficultés inhérentes à une rétro-ingénierie d'une base de données IMS. En effet, nous avons vu que les possibilités offertes au concepteur pour modéliser sa BD sont innombrables. De plus, elle mettent toutes en présence les mêmes concepts à savoir les clés étrangères, les types d'associations provenant de types relations logiques et les hiérarchies.

Dès lors, le travail du rétro-ingénieur est rendu très difficile car il doit détecter les différents éléments conceptuels sans les confondre. Cette phase apparaît donc comme la phase principale dans la conceptualisation des structures de données.

Nous ne pouvons donc que trop conseiller au rétro-ingénieur de se munir d'une stratégie élaborée ou à défaut de réaliser cette phase en totale collaboration avec un concepteur de l'entreprise habitué à ces différentes transformations.

4.4. La normalisation conceptuelle

A ce stade, nous avons déjà un schéma conceptuel très élaboré et proche du schéma initial. Toutefois, il est encore possible au rétro-ingénieur de le raffiner en prenant en compte des facteurs tels que la minimalité ou l'expressivité.

Nous ne nous étendrons pas sur ces possibles transformations du schéma actuel. Effectivement, ces transformations ne sont plus en rapport direct avec IMS. Elles n'ont comme but que celui donné par le rétro-ingénieur. Par exemple, ce dernier peut mettre le schéma sous une forme normale quelconque ou encore le transformer afin qu'il devienne conforme à la méthode de conception de l'entreprise. On peut consulter à ce sujet les études des techniques transformationnelles appliquées à la rétro-ingénierie et notamment [HAINAUT, 93b].

Il nous importe cependant de revenir à un point que nous avions laissé en suspens lors de la conceptualisation de base. Ce point concerne l'éclatement d'un type d'entité en plusieurs types d'entités. Peu importe la source de cet éclatement, le problème qui survient est de savoir à quel type d'entité rattacher les types d'associations anciennement liés à l'unique type d'entité. Maintenant que nous avons un schéma conceptuel correct, il est possible d'effectuer ce rattachement de manière optimale. Nous entendons par là de ne plus rattacher les types d'associations aux deux nouveaux types d'entités mais de les rattacher à un seul des deux suivant la signification du type d'association. Il faut pour cela avoir une vue du schéma claire ce qui implique de faire cette opération maintenant. De plus, les deux types d'entités éclatés peuvent être reliés par une relation IS-A ou un type d'association *one-to-many* ou *many-to-many*. Ceci peut être illustré par l'exemple du type segment contenant les entêtes et les lignes de commande. Ainsi, l'éclatement du type d'entité ne devrait pas donner naissance à deux types d'entités séparés mais à un type d'association *one-to-many* du type d'entité *Entête* vers le type d'entité *Lignes*.

Enfin, signalons que nous n'avons pas pris en compte les relations IS-A dans la conceptualisation de base. Ces relations peuvent être conceptualisées à ce stade. Comme nous l'avons dit dans le paragraphe précédent, nous pouvons suspecter une relation IS-A sur les types d'entités ayant fait l'objet d'une surdéfinition ou d'une redéfinition. Pour le reste, elles peuvent être déduites normalement des différentes situations évoquées dans la littérature.

4.5. La stratégie

La stratégie de conception lors de l'ingénierie d'une BD IMS est d'une grande importance.

En effet, plusieurs points font que le concepteur doit faire face à des choix importants. Le premier point réside dans le choix du type d'entité qui fera office de type segment racine. Ce choix conditionne ensuite tout le reste de la conception car les autres types d'entités sont ajoutés à la hiérarchie et doivent donc se soumettre aux règles de cette hiérarchie. Un autre point réside dans la modélisation choisie pour transformer les éléments conceptuels non-représentables en IMS. Les possibilités offertes sont forts multiples et il est également possible d'établir une stratégie suivie pour toute la conception.

Le travail du rétro-ingénieur IMS n'est donc pas facile car il doit deviner quels choix a fait le concepteur. Toutefois, la culture d'entreprise ou des codes de conduite sont d'une grande aide et peuvent résoudre ce problème. Nous conseillons donc fortement au rétro-ingénieur d'avoir des entretiens avec les concepteurs initiaux s'ils sont encore disponibles. Si ce n'est pas le cas, un entretien avec le responsable informatique ou le responsable des données et bases de données (*Data Management* et *Data Base Management*) peut se révéler très utile.

Dans le cas où le rétro-ingénieur ne peut retrouver la moindre information de stratégie, il est encore possible de comparer différentes applications et d'en retirer les similitudes de développement.

Dans le cas extrême où aucune stratégie n'est trouvée ou déduite, alors il y a de fortes chances que le schéma final, bien que correct, ne ressemble en aucune façon au schéma de la conception initiale.

Chapitre 5 : Etude de cas

5.1. Introduction

Dans ce chapitre, nous allons voir par une étude de cas comment se déroule une rétro-ingénierie complète sur base des résultats de nos études des chapitres précédants.

Cette étude de cas se décompose donc en deux phases à savoir l'extraction des structures de données et la conceptualisation de ces structures. Dans ces deux phases principales, nous retrouvons également tous les sous-processus analysés précédemment.

Pour des raisons de place, nous avons placé en annexes une grande partie des sources et des figures illustrant notre explication. Ceci est voulu afin de ne pas encombrer inutilement le texte explicatif.

5.2. L'extraction des structures de données

Premièrement, nous allons effectuer l'extraction du code DDL. Pour ce faire, nous avons à notre disposition le code source DDL de l'annexe D. Ce code source provient du fichier EDC.IMS qui donne alors son nom au schéma.

En passant le code source à l'extracteur de DB-MAIN, nous obtenons les fenêtres suivantes dans l'environnement DB-MAIN:

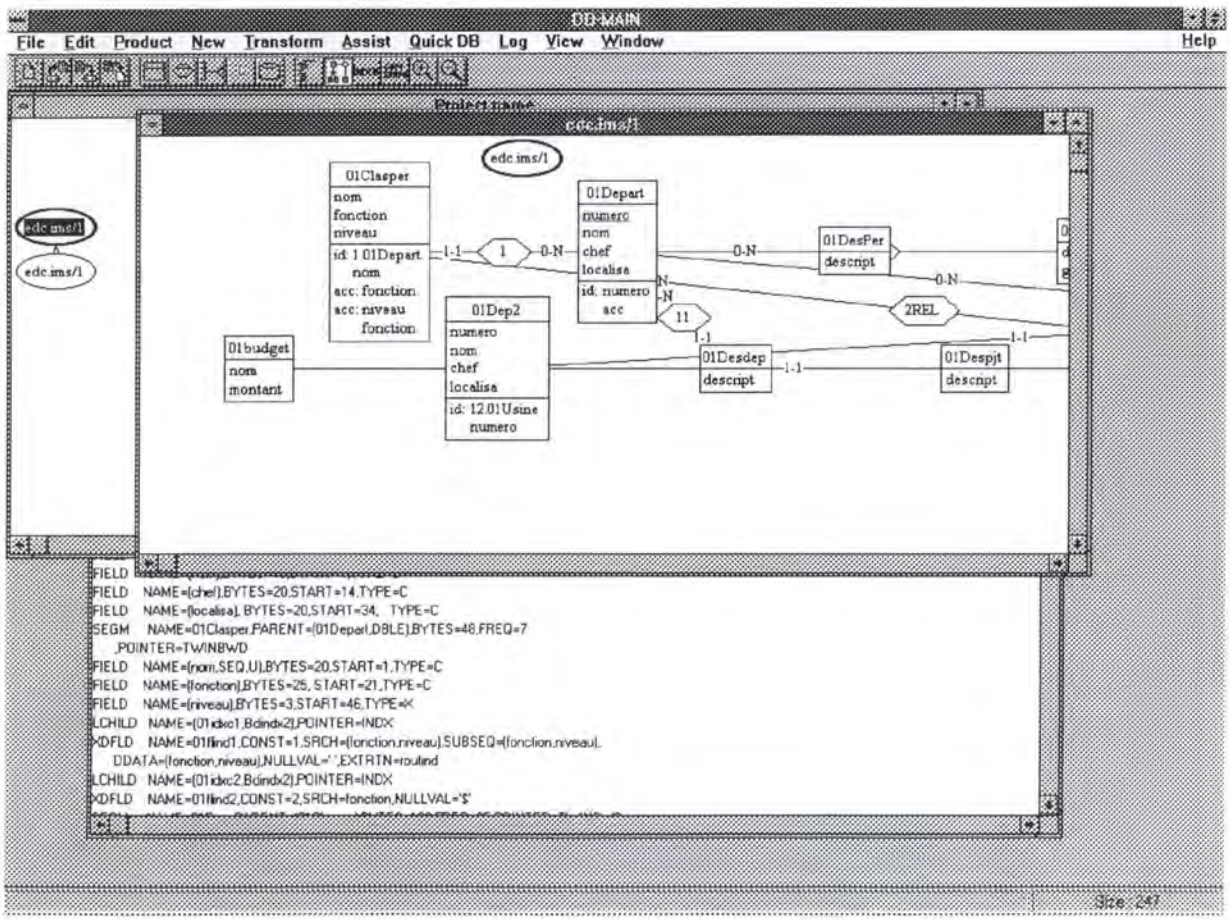

Figure 70 : Ecran DB-MAIN après extraction

Dans cet écran, nous remarquons que la fenêtre projet contient deux produits à savoir le code source et le schéma conceptuel issu de l'extraction. Nous pouvons

également noter que ce schéma est en désordre graphique d'un point de vue IMS. Il faut dès lors le remettre dans un ordre graphique respectant l'ordre hiérarchique. Ceci peut être réalisé grâce aux numérotations des types d'associations provenant des relations physiques et des types d'associations provenant des relations logiques. Nous obtenons alors le schéma suivant:

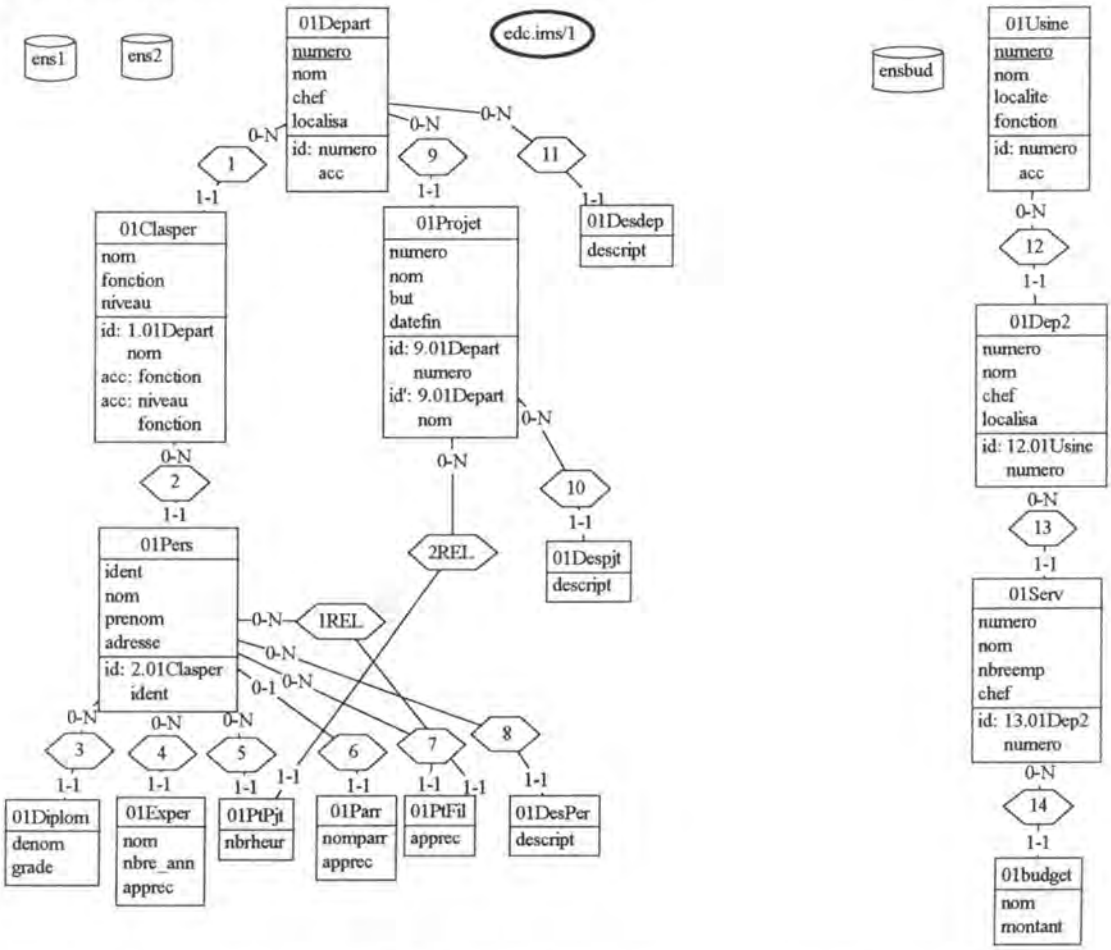

Figure 71 : Schéma résultant de l'extraction dans un ordre graphique IMS

Il y a beaucoup de commentaires à faire sur ce schéma. Tout d'abord, nous remarquons deux hiérarchies alors que nous avions six bases de données définies. C'est tout à fait normal car il n'y a que deux bases de données physiques, les autres étant des bases de données index et logiques. Deuxièmement, sur ce schéma ne figurent pas les descriptions techniques et sémantiques. Pour illustration, nous fournissons respectivement aux Figure 78 et Figure 79 la description technique et sémantique du type d'entité 01Depart. Nous pouvons d'ailleurs remarquer dans la description technique les renseignements concernant la base de données et le type segment. De même, nous constatons dans la description sémantique le nom de la version de la base de données que nous prenons comme version pour la hiérarchie.

Ensuite, nous remarquons la présence des deux collections attachées à la première hiérarchie et de la collection attachée à la deuxième hiérarchie. A titre d'information, nous indiquons à la Figure 80 le contenu de la collection *ensbud* tiré d'une boîte de dialogue de DB-MAIN.

Nous allons maintenant nous attaquer à la première hiérarchie. En ce qui concerne les identifiants, le premier type d'entité *01Depart* contient un identifiant qui est aussi clé

d'accès. Cela provient de la ligne 9 du code source où le champ *numero* est défini avec le mot clé SEQ,U. Comme ce type d'entité est le type d'entité racine, il est normal que cet identifiant soit aussi une clé d'accès. Pour les lignes 15, 26 et 57, les identifiants sont composés du champ devenu attribut et du rôle du type d'entité parent. Ces identifiants se retrouvent dans les types d'entités *01Clasper*, *01Pers* et *01Projet*. Le mot clé SEQ de la ligne 58 donne lieu à un identifiant secondaire du type d'entité *01Projet* et est aussi composé de l'attribut et du rôle du type d'entité parent. Il nous reste alors la ligne 40 où nous avons un champ avec le mot clé SEQ,M. Cela donne naissance à une clé de tri qui est indiquée dans la description technique du type d'entité *01Exper* que l'on retrouve à la Figure 81.

Au niveau des cardinalités, nous pouvons constater une exception au tandem 1-1, 0-N qui se trouve au rôle joué par le type d'entité *01Pers* dans le type d'association *6*. En effet, cette cardinalité n'est pas de 0-N mais de 0-1. Cela provient de la ligne 46 où le type segment *01Parr* est défini et où le mot clé POINTER a comme valeur NOTWIN. Comme nous l'avons vu, cette situation se traduit par une cardinalité de 0-1.

Pour ce qui est de la deuxième hiérarchie, il n'y rien de spécial à dire. Regardons tout de même les identifiants des types d'entités dépendants et du type d'entité racine qui sont bien comme nous le voulons.

En ce qui concerne l'extraction des vues données par les PCB, nous prenons comme code celui exposé à l'annexe E. Le résulta de l'extraction est le suivant (Figure 72):

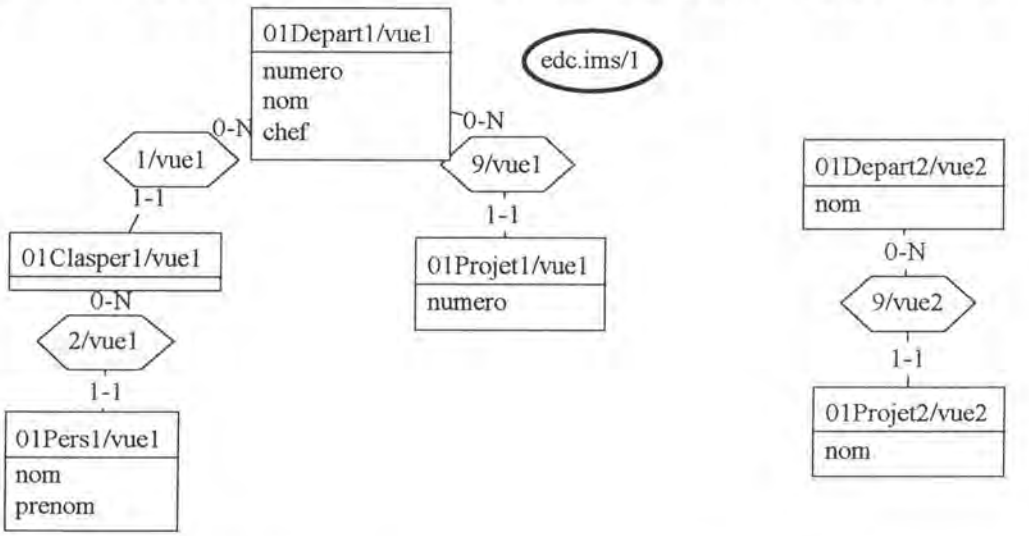

Figure 72 : Sous-schémas externes provenant des PCB

Dans ces sous-schémas externes, nous pouvons remarquer la suffixation des noms des types d'entités et des types d'association par le nom du PCB. De plus, nous pouvons voir que seuls les attributs provenant de champs sensibles ont été recopiés. Les identifiants et les clés d'accès ont aussi disparus pour avoir une vision utilisateur du schéma.

Nous pouvons maintenant passer à l'analyse du code DML et des programmes. Dans le programme de l'annexe G, nous constatons aux lignes 56 à 58 un transfert du contenu du champ *ADRESSE* du type d'entité *01Pers* vers la variable WRK-ADRESSE de la ligne 26 qui, elle, se décompose en plusieurs autres variables. Nous avons donc détecté une décomposition d'un attribut du type d'entité. Cette décomposition peut donc

s'intégrer au schéma. Ainsi, l'attribut ADRESSE se décompose en RUE, NUMERO, CODE POSTAL et LOCALITE.

Aux lignes 59 à 61, nous remarquons que l'insertion d'un segment *01Exper* suit l'insertion d'un segment *01Pers*. Si d'autres insertions de segments *01Pers* n'ont pas lieu autre part, alors nous pouvons déduire une cardinalité inférieure de 1 pour le rôle joué par le type d'entité *01Pers* dans le type d'association *4*. De même, aux lignes 70 à 72, nous constatons le même phénomène à savoir l'insertion d'un segment *01Dep2* suite à l'insertion d'un segment *01Usine*. De ce fait, nous déduisons également une cardinalité inférieure de 1 pour le rôle joué par le type d'entité *01Usine* dans le type d'association *12*. Il reste encore les lectures des lignes 62 à 69 dans lesquelles nous remarquons une lecture d'au plus 5 segments *01Diplom* pour un segment *01Pers*. Nous déduisons alors une cardinalité supérieure de 5 pour le rôle joué par le type d'entité *01Pers* dans le type d'association *3*. Enfin, les lignes 73 à 76 nous renseignent sur l'existence d'une redéfinition pour le type d'entité *01Depart* vers le type d'entité *01Dep2*.

Nous espérons que ces quelques exemples de codes programmes renfermant des informations ont permis de mieux comprendre l'analyse du DML. Pour être vraiment complet, il aurait fallu replacer ces lignes de code dans des situations réelles mais cela n'apportait rien de plus à nos déductions.

En ce qui concerne l'analyse des données, nous ne pouvons en donner un exemple étant donné que celui-ci devrait être conséquent et prendrait alors trop de place ici. Toutefois, prenons comme hypothèse que cette analyse nous donne les informations suivantes. Premièrement, les occurrences du type d'entité *01Clasper* ne dépasse jamais le nombre de 7 sous un même département. Nous mettons donc à 7 la cardinalité supérieure du rôle joué par le type d'entité *01Depart* dans le type d'association *1*. De même, nous remarquons dans les données l'apparition d'un même segment *01Projet* sous différents segments *01Depart*. Nous en déduisons un type d'association *many-to-many* et mettons donc à N la cardinalité supérieure du rôle joué par le type d'entité *01Projet* dans le type d'association *9*. De la même façon, nous constatons qu'il n'y a jamais qu'un et un seul segment *01Desdep* sous un segment *01Depart*. Nous changeons alors les cardinalités 0-N du rôle joué par le type d'entité *01Depart* dans le type d'association *11* par les valeurs 1-1 et nous obtenons un type d'association *one-to-one*. Enfin, nous remarquons que les segments *01Budget* sont toujours d'un nombre inférieur à 10 en dessous d'un segment *01Serv*. Par conséquent, nous mettons à 10 la cardinalité du rôle joué par le type d'entité *01Serv* dans le type d'association *14*. Nous en avons alors terminé avec l'analyse des données.

Ayant réalisé l'analyse du DML et des programmes ainsi que des données. Nous pouvons procéder à l'intégration de schéma. Etant donné que nous avons repéré une redéfinition du type d'entité *01Depart* sous le type d'entité *01Dep2*, nous rassemblons ces deux types d'entités en un seul. Nous prenons comme type d'entité de référence le type d'entité *01Depart*. Nous motivons ce choix car la hiérarchie à laquelle il appartient est la plus riche. De plus, le nom de l'autre type d'entité contient le chiffre 2 ce qui lui confère le statut supposé de copie. Ces deux faits nous font donc choisir le type d'entité de la première hiérarchie. Nous avons alors le schéma résultant de la Figure 73 qui constitue le schéma optimisé orienté IMS et qui clôture la phase d'extraction des structures de données.

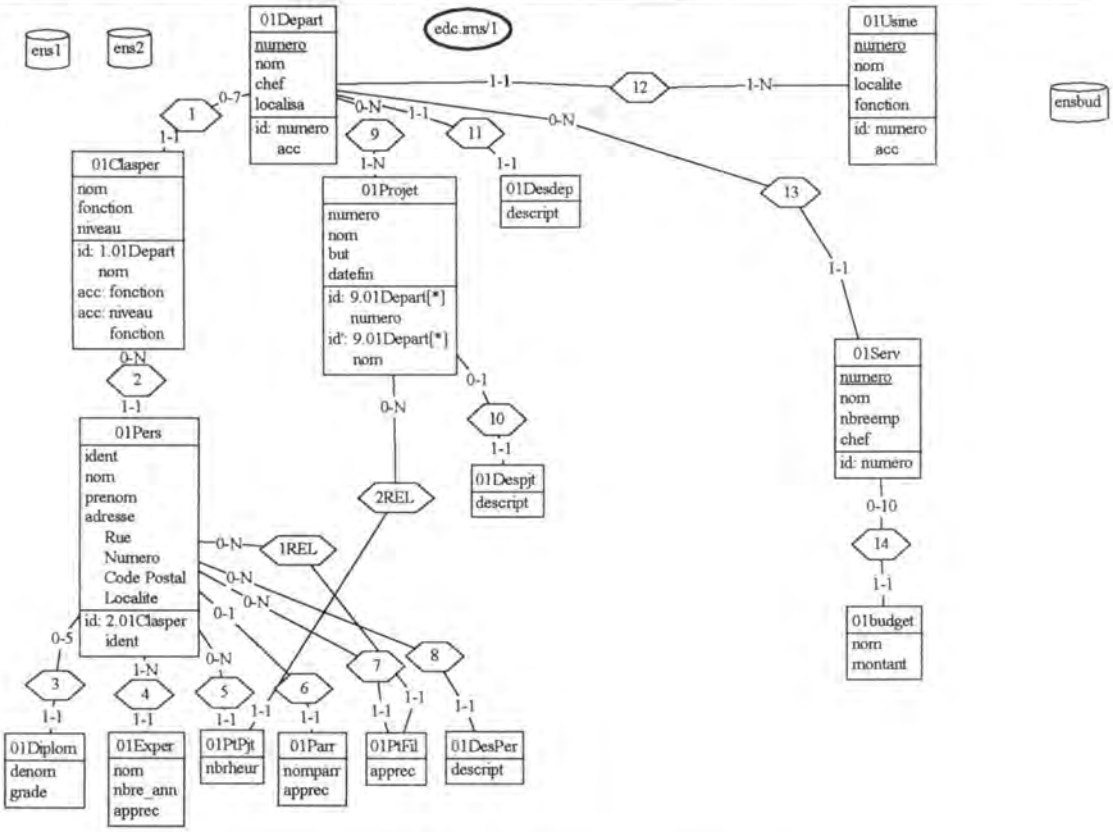

Figure 73 : Schéma optimisé orienté IMS

5.3. La conceptualisation des structures de données

Ayant reçu le schéma optimisé orienté IMS de la phase précédante, nous pouvons maintenant commencer la phase de conceptualisation des structures de données.

Premièrement, nous devons réaliser la phase préliminaire. Celle-ci nous donne le schéma de la Figure 74 comme résultat.

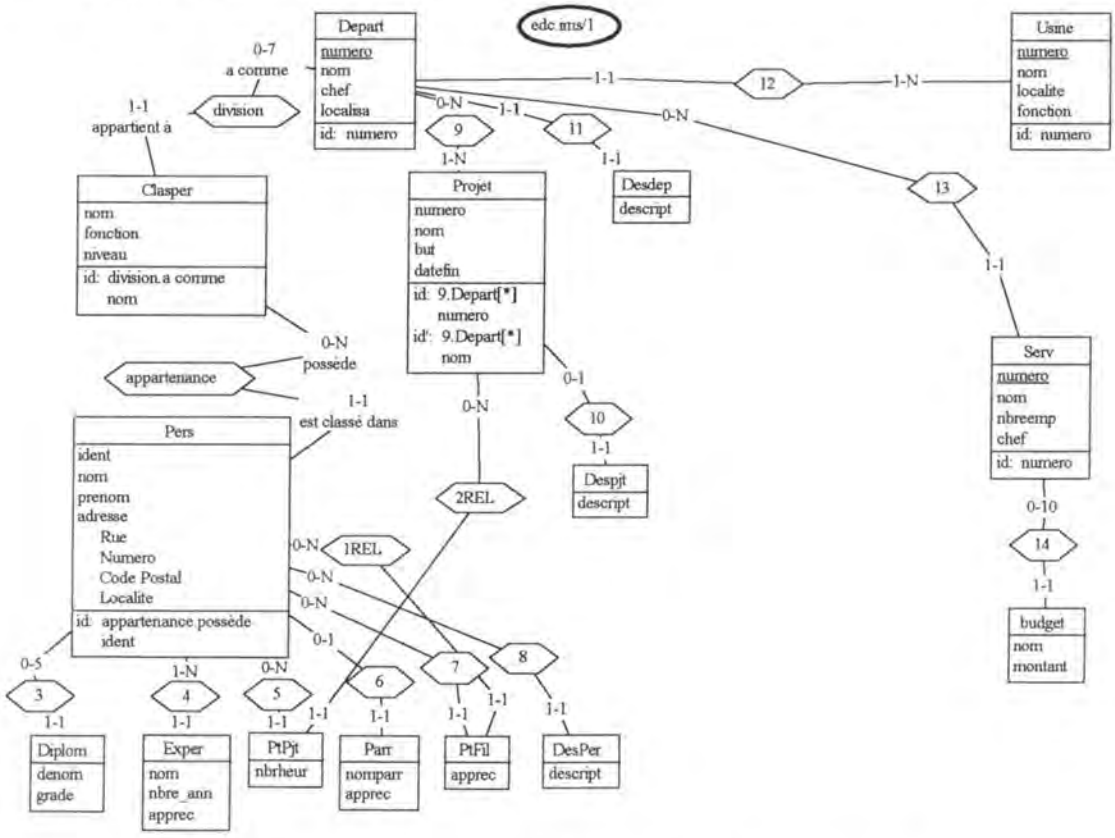

Figure 74 : Schéma conceptuel après la phase préliminaire

Durant cette phase préliminaire, nous avons d'abord supprimé les deux chiffres précédants les types d'entités. En effet, il apparaît évident que ces chiffres sont là pour

identifier les types d'entités d'un projet ou d'un autre concept de développement. Ils ne sont donc pas utiles ici. Ensuite, nous avons commencé à donner des noms aux types d'associations et aux rôles. Toutefois, pour ne pas trop charger le schéma et le rendre illisible, nous n'avons donné des noms que pour les types d'associations et les rôles entre *Depart* et *Clasper* et entre *Clasper* et *Pers*. Pour le reste des noms, nous laissons au lecteur le soin de ce travail. Il nous reste alors à nous occuper des clés d'accès, des collections et des descriptions techniques. Pour automatiser le processus, nous avons fait appel à une transformation globale de l'environnement DB-MAIN dont la boîte de dialogue se trouve à la Figure 82.

Nous pouvons maintenant passer à la phase de détraduction.

Dans le schéma, nous remarquons déjà que les types d'entité *DesPer* et *DesPjt* ne possède qu'un seul attribut et sont reliés à leur type d'entité parent par une relation *one-to-one* et *one-to-many*. Il est donc évident que le type d'entité *DesPjt* est un attribut facultatif de son type d'entité parent. Nous considérons également le type d'entité *DesPers* comme un attribut facultatif même s'il possède un type d'association *one-to-many* car il n'est pas impossible que nos recherches n'est pas réussies à trouver une suspection sur ce type d'entité. De par leur nom, nous pouvons supposer que ces attributs facultatifs correspondent à une description du type d'entité parent. Par conséquent, nous les supprimons et nous créons deux attributs facultatifs dont le nom est *Description*.

Le type d'entité *Diplom* est, quant à lui, susceptible d'être un attribut facultatif décomposable répétitif car la cardinalité du rôle joué par son parent est de 5 et qu'il est composé de deux attributs. De même, il est transformé en un attribut facultatif décomposable répétitif du type d'entité *Pers* et dont la répétitivité est de 5.

En analysant les types d'entités *PtFil* et *Parr*, nous déduisons un type d'association récursif du type d'entité *Pers*. Le premier rôle instancié par le type d'entité *Parr* est le rôle de Parrain dont la cardinalité est de 0-1. Le deuxième rôle est instancié par le type d'entité *PtFil* servant de pointeur. En effet, un type d'association provenant d'une relation logique est aussi attaché à ce type d'entité. Nous créons donc notre type d'association récursif dont le nom est *Parrainage* et dont les deux rôles sont *parraine* et *a comme parrain*. Le type d'entité *Parr* contenait l'attribut *nomparr* qui est une recopie de l'attribut *nom* du type d'entité *Pers*. Il ne figure donc pas comme attribut du type d'association récursif. Par contre, l'attribut *apprec* apparaissait dans les deux types d'entités. Il est donc considéré comme un attribut du type d'association récursif.

Nous avons encore le type d'entité *PtPjt* qui possède aussi un type d'association provenant d'une relation logique. Nous supposons que cette modélisation cache un type d'association *many-to-many*. De ce fait, nous le créons entre le type d'entité *Projet* et le type d'entité *Pers*. L'attribut qui appartenait au type d'entité *PtPjt* est considéré comme un attribut du type d'association *many-to-many*. Nous avons choisi comme nom au type d'association un nom qui nous semble significatif.

Finalement, il reste le type d'entité *Desdep* avec son type d'association *one-to-one* le reliant au type d'entité *Depart*. Nous supposons que ce type d'entité *Desdep* représente un attribut du type d'entité *Depart* et que le concepteur en a fait un type d'entité séparé pour des raisons d'optimisation. Nous le transformons donc en un attribut et nous lui donnons le nom *Description* du fait de son nom *Desdep*.

Nous avons donc en résultat le schéma de la Figure 75 suivante.

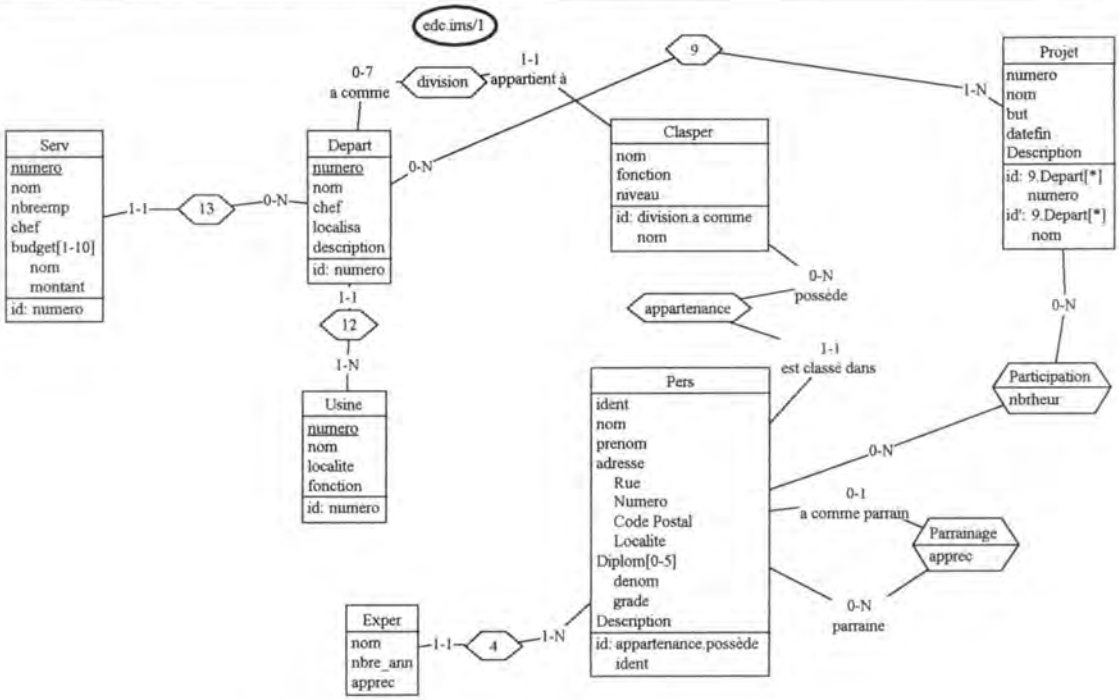

Figure 75 : Schéma conceptuel de base

Les différentes transformations opérées ici ont été très facile à réaliser grâce à l'atelier DB-MAIN et ses puissants outils transformationnels. Il a suffit de sélectionner les éléments à transformer et de choisir ensuite le résultat de la transformation. Le reste à savoir la suppression de l'ancien élément et son remplacement par le nouveau se fait automatiquement par l'atelier. Cette phase est donc réalisée assez rapidement.

Ce schéma conceptuel de base correspond en fait à notre schéma final. En effet, la normalisation conceptuelle ne nous apporte rien car nous n'avons pas de critère de modélisation imposé. Il est bien sûr encore possible de changer les noms pour avoir un schéma plus significatif mais nous ne le prenons pas en compte ici.

Conclusion

Notre but dans ce mémoire était d'étudier le processus de rétro-ingénierie des bases de données IMS et DATACOM/DB. Le charge de travail fournie sur IMS a cependant été plus grande que prévue. Nous avons donc été dans l'impossibilité d'étudier complètement la rétro-ingénierie des bases de données DATACOM/DB. Cependant, une partie du travail concernant l'analyse syntaxique du langage a pu être faite, ce qui pourra constituer un départ pour un travail futur.

En ce qui concerne IMS, nous nous sommes heurtés à beaucoup de problèmes dans notre étude. Dès la phase d'extraction du code DDL, il s'est avéré que notre documentation sur la syntaxe était incorrecte et incomplète. Nous avons alors pris comme documentation celle fournie par IBM dans la dernière version d'IMS. Malheureusement, cette documentation, bien que meilleure, comportait encore des incohérences. Il a donc fallu un certain temps pour étudier cette phase et nous ne sommes pas encore certain de la justesse de la syntaxe analysée. Toutefois, si des incohérences persistent, elles ne sont plus très nombreuses et ne concernent pas des éléments importants de l'extraction.

Pour continuer avec la documentation, la littérature traitant d'IMS est abondante mais ce n'est pas pour autant un avantage. En effet, nous avons pu constater tout au long de l'étude des contradictions entre auteurs. Cela nous a fort perturbés car il était alors difficile de savoir qui avait raison. En approfondissant les recherches, il était quand même possible de trouver la vérité mais cela était fort consommateur d'énergie.

Au niveau de la complexité du SGBD IMS, elle est sans comparaison avec les autres systèmes que nous avons eu l'occasion d'analyser jusqu'à maintenant. En effet, en voulant permettre au concepteur de paramètrer tous les facteurs techniques d'une BD, le système IMS offre des instructions de définition de la BD très volumineuses ce qui en rend difficile la compréhension et l'analyse.

De plus, le système était au début prévu pour implémenter uniquement des hiérarchies de types relations physiques. Ensuite, afin de palier aux inconvénients que cela engendrait pour la conception et l'accès aux données, IBM a introduit les types relations logiques et les index secondaires. Malheureusement, ces nouvelles possibilités engendrent également un amas de facteurs techniques qu'il faut prendre en compte. En

outre, les possibilités de modélisation s'en trouvent agrandies sans pour autant faciliter la tâche du concepteur. Il est d'ailleurs conseillé dans certaines littératures de n'employer les types relations logiques que lorsque cela est vraiment nécessaire. Une preuve de cette complexité dans la modélisation se trouve dans le corpus de règles pour l'obtention d'un schéma conforme IMS.

Au niveau d'un type segment, la possibilité de décrire anarchiquement ses champs permet toute une série de modélisations qu'il faut également prendre en compte et dont la plus difficile concerne la surdéfinition de types segments.

Toutes ces remarques nous amènent à voir la rétro-ingénierie des bases de données IMS comme un processus plus complexe que les autres rétro-ingénieries. Plus précisément, les sous-processus d'analyse du DML et de la conceptualisation de base sont parmi les plus compliqués car ils doivent prendre en compte tous les cas possibles.

Le fruit de cette étude est donc un mémoire dans lequel nous avons essayé d'expliquer nos recherches et les résultats de ces recherches. Il est intéressant de noter que nous avons tenté de faire un tour le plus complet possible des différentes possibilités. Les phases d'analyse du code DML et des programmes ainsi que la phase de conceptualisation de base ont été les plus difficiles mais aussi les plus enrichissantes. Finalement, nous pensons avoir fait une analyse assez complète du problème. Nous avons surtout adopté une vue pratique ce qui donne lieu à un résultat plus vraisemblable que celui que l'on peut trouver dans la littérature.

Personnellement, nous en avons retiré beaucoup autant au niveau de la connaissance du système IMS que dans les techniques de conception. En effet, la complexité des modélisations nous a fait prendre conscience des différentes possibilités et des combinaisons de celles-ci. Nous connaissons maintenant les différentes astuces à employer pour modéliser une structure non-directement modélisable.

En ce qui concerne l'atelier DB-MAIN qui nous a servi de référence, nous avons quelques remarques à faire. Premièrement, nous avons été surpris par la vitesse de son évolution. En effet, sur le temps que nous avons passé à l'utiliser, il s'est continuellement bonifié. Toutefois, il recèle encore certaines lacunes.

La première remarque concerne la modélisation technique. En effet, les types permis d'attributs ne sont pas nombreux. En IMS, nous avons comme type du décimal packé (*packed decimal*), des mots et demi-mots ou encore de l'hexadécimal. Nous avons trouvé regrettable de ne pas pouvoir définir ces types dans l'atelier. D'un autre côté, nous ne pouvons représenter les chemins d'accès de nos hiérarchies. C'est d'autant plus désolant que c'est un élément essentiel de la modélisation technique.

Ensuite, l'emplacement graphique des éléments ne donne pas entière satisfaction dans le cas d'IMS. Effectivement, une extraction d'une base de données de plusieurs centaines de types d'entités et de types d'associations donne un schéma incompréhensible. Ceci est du à l'emplacement arbitraire qui est donné aux éléments créés. Il serait souhaitable pour la modélisation IMS de définir des règles d'emplacements graphiques permettant aisément de définir nos hiérarchies directement dans un ordre graphique hiérarchique. Ainsi, le concepteur IMS trouverait après l'extraction le schéma dans le bon ordre et ne serait pas obligé de remettre l'ordre hiérarchique en se basant sur les numéros des types d'associations. Ce point est sans doute le plus gênant.

D'autre part, nous aurions aimé voir les descriptions techniques et sémantiques dans les vues standards ou étendues. L'extracteur insérant assez bien d'informations dans ces

descriptions, il serait intéressant de les voir toutes en une fois plutôt que de passer par chaque élément individuel.

Par contre les outils transformationnels proposés sont très puissants et nous avons eu grand plaisir à les utiliser. Nous trouvons cependant que l'approche de l'outil reste trop théorique et que l'amélioration des fonctions de génération automatique du code DDL le bonifierait certainement. Ces générations automatiques constituent à notre avis l'avenir de cet atelier et un moyen sûr d'étendre son utilisation au plus grand nombre.

Pour conclure sur l'outil, nous pensons que les inconvénients sont connus de l'équipe de recherche et que la rapidité d'évolution de l'outil aura vite fait de les éliminer.

En guise de conclusion finale, nous voulons encore souligner l'enrichissement personnel que nous avons eu à faire ce mémoire. Nous espérons également qu'il aura apporté de précieux enseignements et qu'il aura permis de faire avancer la recherche en matière de rétro-ingénierie.

Liste des abréviations

ANSI	:	American National Standard Institute
BD	:	Base de Données
BIKIT	:	Babbage Institute for Knowledge and Information Technology
CASE	:	Computer Aided Software Engineering
CI	:	Control Interval
COBOL	:	COmon Business Oriented Language
DBA	:	Data Base Administrator
DBD	:	Data Base Description
DDL	:	Data Definition Language
DEDB	:	Data Entry Data Base
DL/1	:	Data Language One
DML	:	Data Manipulation Language
DMS	:	Data Management System
DSC	:	Data Structure Conceptualization
DSE	:	Data Structure Extraction
ESDS	:	Entry Sequenced Data Set
FUNDP	:	Facultés Universitaires Notre-Dame de la Paix
GHN	:	Get Hold Next
GHNP	:	Get Hold Next within Parent
GHU	:	Get Hold Unique
GN	:	Get Next
GNP	:	Get Next within Parent

GSAM	:	Generalized Sequential Access Method
GU	:	Get Unique
HDAM	:	Hierarchical Direct Access Method
HIDAM	:	Hierarchical Indexed Direct Access Method
HISAM	:	Hierarchical Indexed Sequential Access Method
HSAM	:	Hierarchical Sequential Access Method
IBM	:	International Business Machines
IMS	:	Information Management System
KSDS	:	Key Sequenced Data Set
MSDB	:	Main Storage Data Base
NASA	:	National Aeronautical and Space Agency
OS	:	Operating System
OSAM	:	Overflow Sequential Access Method
PA	:	Programme d'Application
PCB	:	Program Communication Block
PL/1	:	Program Language One
PSB	:	Program Specification Block
RAA	:	Root Adressable Area
RAP	:	Root Anchor Point
RI	:	Rétro-Ingénierie
SGBD	:	Système de Gestion de Bases de Données
SHISAM	:	Simple Hierarchical Indexed Sequential Access Method
SHSAM	:	Simple Hierarchical Sequential Access Method
SQL	:	Structured Query Language
SSA	:	Segment Search Argument
VSAM	:	Virtual Sequential Access Method

Bibliographie

BARNETT, 74 : A.J. BARNETT, J.A. LIGHTFOOT, *Information Management System (IMS) A User's Experience with Evolutionary Development*, in Proceedings of the Share Working Conference on Data Base Management Systems, Montreal, Canada, July 23-27/1973, North-Holland, 1974

BATINI, 92 : C. BATINI, S. CERI, S. B. NAVATHE, *Conceptual Database Design, An Entity-Relationship Approach*, Chapter 14 : Logical Design for the Hierarchical Model p. 377-409, Benjamin/Cummings, 1992

CASANOVA, 83 : M. CASANOVA, J. AMAREL de SA, *Designing Entity Relationship Schemas for Conventional Information Systems*, in Proc. of Entity-Relationship Approach, pp. 265-278, 1983

CASANOVA, 84 : M. CASANOVA, J. AMAREL de SA, *Mapping Uninterpreted Schemes into Entity-Relationship Diagrams : Two Applications to Conceptual Schema Design*, in IBM J. Res. & Develop., Vol. 28, No 1, January, 1984

CLARINVAL, 91 : A. CLARINVAL, *Comprendre et Connaître le COBOL 85*, Presses Universitaires de Namur, 1991

COHEN, 78 : L. J. COHEN, *DataBase Management Systems*, Chapter 4.1 : IMS/VS p. 4-1 à 4-54, Editions QED Information Sciences, 1978.

DATE, 90 : C.J. DATE, *An Introduction to Database Systems*, Appendix B : A Hierarchic System : IMS p. 753-789, Volume I, Fifth Edition, Addison-Wesley, 1990

DAVIS, 77 : B. DAVIS, *The Selection of DataBase Software*, Chapter 6: IMS (DL/1) p. 135-223, National Computing Centre, 1977

DAVIS, 85 : K.H. DAVIS, K.A. ADARSH, *A Methodology for Translating a Conventional File System into an Entity-Relationship Model*, in Proc. of Entity-Relationship Approach, October, 1985

DAVIS, 88 : K.H. DAVIS, A.K. ARORA, *Converting a Relational Database Model to an Entity-Relationship Model*, in Proc. of Entity-Relationship Approach: a bridge to the user, 1988

ELMASRI, 89 : R. ELMASRI, S. B. NAVATHE, *Fundamentals of Database Systems*, Chapter 10 : The Hierarchical Data Model p. 253-279, Chapter 23.2 : A hierarchical System - IMS p. 683-704, Benjamin/Cummings, 1989

ELMASRI, 94 : R. ELMASRI, S. B. NAVATHE, *Fundamentals of Database Systems*, Second Edition, Chapter 11 : The Hierarchical Data Model and the IMS System p. 343-389, Benjamin/Cummings, 1994

FONKAM, 92 : M.M. FONKAM, W.A. GRAY, *An Approach to Eliciting the Semantics of Relational Databases*, in Proc. of 4th Int. Conf. on Advance Information Systems Engineering- CAISE'92, pp. 463-480, May, LNCS, Springer-Verlag, 1992

GALASCI, 89 : nom composé pour (H. BRIAND, J.B. CRAMPES, C. DUCATEAU, Y. HEBRAIL, D. HERIN-AIME, J. KOULOUMDJIAN, R. SABATIER) avec la participation de G. MERCKY et G. HEBRAIL, *Conception de bases de données, du schéma conceptuel aux schémas physiques*, Chapitre 5 : Schéma logique hiérarchique p. 63-93, Chapitre 12 : Schéma physique hiérarchique p. 185-209, Dunod Informatique, 1989

GELLER, 89 : J. R. GELLER, *IMS Administration, Programming and Data Base Design*, John Wiley & Sons, 1989

HAINAUT, 93a : J-L. HAINAUT, *Database Reverse Engineering A systematic Approach*, Cours postgrade en informatique, 1993

HAINAUT, 93b : J-L. HAINAUT, *Transformation Techniques for Database Reverse Engineering*, Cours postgrade en informatique, 1993

HAINAUT, 94 : J-L. HAINAUT, V. ENGLEBERT, J. HENRARD, J-M.HICK, D. ROLAND, *Database Evolution: the DB-MAIN Approach*, 1994

HAINAUT, 95 : J-L. HAINAUT, V. ENGLEBERT, J. HENRARD, J-M.HICK, D. ROLAND, *Requirements for Information System Reverse Engineering Support*, 1995

HENNEBERT : H. HENNEBERT, *IMS/360*, rapport pour l'Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix.

IBM/IMS, 76 : IBM World Trade Systems Center, *IMS/VS-DB PRIMER*, Editions IBM Palo Alto, First Edition, 1976.

IBM/IMS, 93 : IBM, *IMS/ESA Version 4, Utilities Reference: System*, IBM, First Edition, April 1993.

NAVATHE, 88 : S.B. NAVATHE, A. AWONG, *Abstracting Relational and Hierarchical Data with a Semantic Data Model*, in Proc. of Entity-Relationship Approach: a Bridge to the User, 1988

NILSSON, 85 : E.G. NILSSON, *The Translation of COBOL Data Structure to an Entity-Rel-type Conceptual Schema*, in Proc. of Entity-Relationship Approach, October, 1985

SPRING, 90 : F.N. SPRINGSTEEL, C. KOU, *Reverse Data Engineering of E-R Designed Relational Schemas*, in Proc. of Databases, Parallel Architectures and their Applications, March, 1990

THALMAN, 84 : N. MAGNENAT-THALMAN, D. THALMAN, *Gestion de fichiers et bases de données*, Chapitre 8 : Les modèles hiérarchisé et réseau p. 185-210, Chapitre 10 : IMS p. 251-282, Editions ESKA, deuxième édition, 1984.

WINANS, 90 : J. WINANS, K.H. DAVIS, *Software Reverse Engineering from a Currently Existing IMS Database to an Entity-Relationship Model*, in Proc. of Entity-Relationship Approach: the Core of Conceptual Modeling, pp. 345-360, October, 1990

Annexes

Annexe A

Dans cette annexe, nous fournissons une description complète des syntaxes précises des instructions exigées par les différentes méthodes d'accès. Ces descriptions sont tirées de [IBM/IMS, 93].

A.1. Base de données HSAM

```
DBD
                NAME=dbname1
                ,ACCESS={HSAM
                         SHSAM}
DATASET
                DD1=ddname1
                ,DD2=ddname2
                [,BLOCK=(blk_fact1,blkfact2)]
                [,RECORD=(reclen1,reclen2)]
SEGM
                NAME=segname1
                ,PARENT={0 segname2}
                ,BYTES=maxbytes
                [,FREQ=freq]
FIELD
                NAME=(fldname1[,SEQ,{M U}])
                ,BYTES=bytes
                ,START=startpos
                [,TYPE={X P C}]
DBDGEN
[FINISH]
END
```

A.2. Base de données HISAM

```
DBD
                NAME=(dbname1)
                ,ACCESS={(HISAM [,VSAM])
                         (SHISAM [,VSAM])}
                [,PASSWD={YES
                          NO}]
                [,EXIT=((
                   [{*
                     exitname [,{KEY NOKEY}]
                              [,{NOPATH PATH}]
                              [,{DATA NODATA}]
                   }]
                  [,{CASCADE
```

```
                        (CASCADE [,{KEY NOKEY}]
                                 [,{DATA NODATA}]
                                 [,{NOPATH PATH}])
                        NOCASCADE
                      }]
                    [{LOG NOLOG}]),...)]
                  [,VERSION='n']
DATASET
                  DD1=ddname1
                  ,OVFLW=ddname3
                  [,BLOCK=(blk_fact1,blkfact2)]
                  [,SIZE=(size1,size2)]
                  [,RECORD=(reclen1,reclen2)]
SEGM
                  NAME=segname1
                  ,PARENT={0
                           (segname2[lpsegname[,{VIRTUAL
                           PHYSICAL}][,dbname1)])}
                  ,BYTES=maxbytes[,minbytes]
                  [,FREQ=freq]
                  [,POINTER=,{LPARNT
                              CTR
                              PAIRED}]
                  [,RULES=([{L P V}{L P V B}{L P V},]
                           {LAST FIRST HERE})]
                  [,SOURCE=(segname,{DATA KEY},dbname)]
                  [,EXIT=((
                      exitname [,{KEY NOKEY}]
                               [,{NOPATH PATH}]
                               [,{DATA NODATA}]
                      }]
                    [,{CASCADE
                      (CASCADE [,{KEY NOKEY}]
                               [,{DATA NODATA}]
                               [,{NOPATH PATH}])
                      NOCASCADE
                    }]
                  [{LOG NOLOG}]),...)]
                  [,COMPRTN={(routname,{DATA KEY},INIT)
                            (routname[{[,DATA],INIT
                                      ,,INIT}])}]
LCHILD
                  NAME=(segname1,dbname)
                  [,POINTER=[{DBLE NONE SYMB}]]
                  [,PAIR=segname2]
                  [,RULES={LAST FIRST HERE}]
FIELD
                  NAME={(fldname1[,SEQ,{M U}])
                        syst_rel_fldname}
                  ,BYTES=bytes
                  ,START=startpos
                  [,TYPE={X P C}]
XDFLD
                  NAME=fldname
                  [,SEGMENT=segname]
                  [,CONST=char]
                  ,SRCH=list1
                  [,SUBSEQ=list2]
                  [,DDATA=list3]
                  [,NULLVAL=value1]
                  [EXTRTN=name1]
DBDGEN
[FINISH]
END
```

A.3. Base de données HDAM

```
DBD
                NAME=(dbname1)
                ,ACCESS=(HDAM [,{OSAM VSAM}])
                RMNAME=(random[,numrap
                              ,sizeblock,sizeraa])
                [,PASSWD={YES
                          NO}]
                [,EXIT=((
                   [{*
                     exitname [,{KEY NOKEY}]
                                [,{NOPATH PATH}]
                                [,{DATA NODATA}]
                     }]
                   [,{CASCADE
                      (CASCADE [,{KEY NOKEY}]
                                [,{DATA NODATA}]
                                [,{NOPATH PATH}])
                      NOCASCADE
                     }]
                   [{LOG NOLOG}]),...)]
                [,VERSI0N='n']
DATASET
                DD1=ddname1
                [,BLOCK=blk_fact1]
                [,SIZE=(,size1)]
                [,SCAN=cyl_num]
                [,FRSPC=(free_block_freq,%free_space)]
                [,SEARCHA={0 1 2}]
SEGM
                NAME=segname1
                ,PARENT={0
                         (segname2[,{SNGL
                                     DBLE}]
                                  [(lpsegname[,{VIRTUAL
                                                PHYSICAL}][,dbname1])])
                                                }
                ,BYTES=maxbytes[,minbytes]
                [,FREQ=freq]
                [,POINTER={HIER
                           HIERBWD
                           TWIN
                           TWINBWD
                           NOTWIN}
                           ,{LTWIN
                           LTWINBWD}
                           ,{LPARNT
                           CTR
                           PAIRED}]
                [,RULES=([{L P V}{L P V B}{L P V},]
                         {LAST FIRST HERE})]
                [,SOURCE=(segname,{DATA KEY},dbname)]
                [,EXIT=((
                   [{*
                     exitname [,{KEY NOKEY}]
                                [,{NOPATH PATH}]
                                [,{DATA NODATA}]
                     }]
                   [,{CASCADE
                      (CASCADE [,{KEY NOKEY}]
                                [,{DATA NODATA}]
                                [,{NOPATH PATH}])
                      NOCASCADE
                     }]
                   [{LOG NOLOG}]),...)]
                [,COMPRTN={(routname,{DATA KEY},INIT)
                          (routname[{[,DATA],INIT
                                    ,,INIT}])}]
LCHILD
                NAME=(segname1,dbname)
```

```
                    [,POINTER=[{DBLE NONE INDX SYMB}]]
                    [,PAIR=segname2]
                    [,RULES={LAST FIRST HERE}]
FIELD
                    NAME={(fldname1[,SEQ,{M U}])
                          syst_rel_fldname}
                    ,BYTES=bytes
                    ,START=startpos
                    [,TYPE={X P C}]
XDFLD
                    NAME=fldname
                    [,SEGMENT=segname]
                    [,CONST=char]
                    ,SRCH=list1
                    [,SUBSEQ=list2]
                    [,DDATA=list3]
                    [,NULLVAL=value1]
                    [EXTRTN=name1]
DBDGEN
[FINISH]
END
```

A.4. Base de données HIDAM

```
DBD
                    NAME=dbname1
                    ,ACCESS=(HIDAM [,{OSAM VSAM}])
                    [,PASSWD={YES
                              NO}]
                    [,EXIT=((
                        [{*
                          exitname [,{KEY NOKEY}]
                                   [,{NOPATH PATH}]
                                   [,{DATA NODATA}]
                          }]
                        [,{CASCADE
                           (CASCADE [,{KEY NOKEY}]
                                    [,{DATA NODATA}]
                                    [,{NOPATH PATH}])
                           NOCASCADE
                          }]
                        [{LOG NOLOG}]),...)]
                    [,VERSI0N='n']
DATASET
                    DD1=ddname1
                    [,BLOCK=blk_fact1]
                    [,SIZE=(,size1)]
                    [,SCAN=cyl_num]
                    [,FRSPC=(free_block_freq,%free_space)]
                    [,SEARCHA={0 1 2}]
SEGM
                    NAME=segname1
                    ,PARENT={0
                             (segname2[,{SNGL
                                         DBLE}]
                                      [(lpsegname[,{VIRTUAL
                                                    PHYSICAL}][,dbname1])])
                                                    }
                    ,BYTES=maxbytes[,minbytes]
                    [,FREQ=freq]
                    [,POINTER={HIER
                               HIERBWD
                               TWIN
                               TWINBWD
                               NOTWIN}
                               ,{LTWIN
                               LTWINBWD}
                               ,{LPARNT
                               CTR
```

```
                                PAIRED}]
                    [,RULES=([{L P V}{L P V B}{L P V}]
                                ,{LAST FIRST HERE})]
                    [,SOURCE=(segname,{DATA KEY},dbname)]
                    [,EXIT=((
                        [{*
                          exitname [,{KEY NOKEY}]
                                   [,{NOPATH PATH}]
                                   [,{DATA NODATA}]
                        }]
                        [,{CASCADE
                          (CASCADE [,{KEY NOKEY}]
                                   [,{DATA NODATA}]
                                   [,{NOPATH PATH}])
                          NOCASCADE
                        }]
                        [{LOG NOLOG}]),...)]
                    [,COMPRTN={(routname,{DATA KEY},INIT)
                                (routname[{[,DATA],INIT
                                          ,,INIT}])}]
LCHILD
                    NAME=(segname1,dbname)
                    [,POINTER=[{DBLE NONE INDX SYMB}]]
                    [,PAIR=segname2]
                    [,RULES={LAST FIRST HERE}]
FIELD
                    NAME={(fldname1[,SEQ,{M U}])
                          syst_rel_fldname}
                    ,BYTES=bytes
                    ,START=startpos
                    [,TYPE={X P C}]
XDFLD
                    NAME=fldname
                    [,SEGMENT=segname]
                    [,CONST=char]
                    ,SRCH=list1
                    [,SUBSEQ=list2]
                    [,DDATA=list3]
                    [,NULLVAL=value1]
                    [EXTRTN=name1]
DBDGEN
[FINISH]
END
```

A.5. Base de données MSDB

```
DBD
                    NAME=dbname1
                    ,ACCESS=MSDB
DATASET
                    ,REL={{NO
                          TERM[,fldnm]
                          FIXED[,fldnm]
                          DYNAMIC[,fldnm]})
SEGM
                    NAME=segname1
                    ,BYTES=maxbytes
FIELD
                    NAME=(fldname1[,SEQ,U])
                    ,BYTES=bytes
                    ,START=startpos
                    [,TYPE={X P C H F}]
DBDGEN
[FINISH]
END
```

A.6. Base de données DEDB

```
DBD
                NAME=(dbname1)
                ,ACCESS=DEDB
                RMNAME=random
                [,EXIT=((
                   [{*
                     exitname [,{KEY NOKEY}]
                                [,{NOPATH PATH}]
                                [,{DATA NODATA}]
                     }]
                   [,{CASCADE
                     (CASCADE [,{KEY NOKEY}]
                                [,{DATA NODATA}]
                                [,{NOPATH PATH}])
                     NOCASCADE
                     }]
                   [{LOG NOLOG}]),...)]
                [,VERSION='n']
                [,PASSWD={YES
                          NO}]
AREA
                DD1=ddname1
                ,SIZE=size1
                ,UOW=(num1,overfl1)
                ,ROOT=(num2,overfl2)
SEGM
                NAME=segname1
                ,PARENT={0
                        (segname2[,{SNGL
                                    DBLE}])}
                ,BYTES=maxbytes,minbytes
                ,TYPE={DIR SEQ}
                [,RULES={HERE LAST FIRST}]
                [,SSPTR=n]
                [,EXIT=((
                   [{*
                     exitname [,{KEY NOKEY}]
                                [,{NOPATH PATH}]
                                [,{DATA NODATA}]
                     }]
                   [,{CASCADE
                     (CASCADE [,{KEY NOKEY}]
                                [,{DATA NODATA}]
                                [,{NOPATH PATH}])
                     NOCASCADE
                     }]
                   [{LOG NOLOG}]),...)]
                [,COMPRTN={(routname,{DATA KEY},INIT)
                          (routname[{[,DATA],INIT
                                    ,,INIT}])}]
FIELD
                NAME=(fldname1[,SEQ,U])
                ,BYTES=bytes
                ,START=startpos
                [,TYPE={X P C}]
DBDGEN
[FINISH]
END
```

A.7. Base de données INDEX

```
DBD
                NAME=(dbname1,...)
                ,ACCESS=(INDEX[,VSAM]
                                [,{PROT NOPROT}]
                                [,DOSCOMP])
                [,PASSWD={YES
```

```
                                    NO}]
DATASET
                    DD1=ddname1
                    [,BLOCK=(blkfact1,blkfact2)]
                    [,SIZE=(size1,size2)]
                    [,RECORD=(reclen1,reclen2)]
SEGM
                    NAME=segname1
                    [,PARENT=0]
                    ,BYTES=bytes
                    [,FREQ=freq]
LCHILD
                    NAME=(segname1,dbname)
                    [,POINTER={SNGL SYMB}]
                    ,INDEX=fldname
FIELD
                    NAME=(fldname1[,SEQ,U])
                    ,BYTES=bytes
                    ,START=startpos
                    [,TYPE={X P C}]
DBDGEN
[FINISH]
END
```

A.8. Base de données logique

```
DBD
                    NAME=dbname1
                    ,ACCESS=LOGICAL
DATASET
                    LOGICAL
SEGM
                    NAME=segname1
                    ,PARENT={0 segname2}
                    ,BYTES=maxbytes
                    [,FREQ=freq]
DBDGEN
[FINISH]
END
```

Annexe B

Dans cette annexe, nous indiquons une description complète des informations fournies par la connaissance de la méthode d'accès. Ces informations sont tirées de [IBM/IMS, 93].

B.1. Base de données HSAM

Pour une base de données HSAM, on spécifie:

- un DATASET;
- le *ddname* d'un DATASET en input est utilisé quand une application retire des données de la BD;
- le *ddname* d'un DATASET en input est utilisé pendant le chargement de la BD;
- de 1 à 255 type(s) segment(s) pour la BD;
- de 0 à 255 champ(s) dans chaque type segment, avec un maximum de 1000 champs dans la BD;
- Optionnellement, on peut définir une BD SHSAM qui contient seulement un type segment de longueur fixe. Ainsi défini, aucun préfixe n'est mis dans les occurrences du type segment.

Et on ne peut pas spécifier:

- l'utilisation de pointeurs hiérarchiques ou enfants physiques/jumeaux physiques entre segments dans la BD;
- l'utilisation de relation logique ou d'index entre segments.

B.2. Base de données HISAM

Pour une base de données HISAM, on spécifie:

- de 1 à 10 DATASET;
- le *ddname* d'un VSAM Key Sequence Data Set (KSDS) et d'un VSAM Entry Sequenced Data Set (ESDS). HISAM/VSAM supporte seulement un DATASET;
- Optionnellement, on peut définir une BD SHISAM qui contient seulement un type segment de longueur fixe. Ainsi défini, aucun préfixe n'est mis dans les occurrences

du type segment. La longueur de l'enregistrement logique spécifié pour une BD SHISAM doit être le même que la longueur du segment spécifié;
- au moins un type segment pour chaque DATASET, et un maximum de 255 types segments pour la BD;
- de 0 à 255 champ(s) pour chaque type segment, et un maximum de 1000 pour la BD, un d'entre eux doit être un champ de séquence unique dans le segment racine pour indexer les occurrences du segment racine;
- des relations logiques (optionnel) utilisant les options de pointeur symbolique quand un segment dans une BD HISAM pointe vers un autre segment dans une BD HISAM, et les options de pointeur direct ou symbolique quand un segment dans une BD HISAM pointe vers un segment dans une BD HDAM ou HIDAM;
- des routines "*Segment Edit/Compression exit*", qui sont optionnelles, pour permettre à des routines de l'utilisateur de manipuler chaque occurrence d'un type segment de ou vers un stockage auxiliaire;
- une routine "*Data Capture exit*", qui est optionnelle, pour permettre à des utilisateurs finals DB2 de mettre à jour les données IMS. La routine de sortie peut aussi être utilisée en SHISAM.

Et on ne peut spécifier l'utilisation de pointeurs hiérarchiques ou enfants physiques/jumeaux physiques entre segments dans une BD HISAM.

B.3. Base de données HDAM

Pour une base de données HDAM, on spécifie:

- le nom du module de hachage fourni par l'utilisateur pour le placement des occurrences du segment racine;
- de 1 à 10 DATASET;
- combien d'espace libre doit être distribué dans chaque groupe de données;
- le *ddname* d'un ensemble de données OSAM ou ESDS pour chaque groupe de données défini;
- au moins un type segment pour chaque DATASET, et un maximum de 255 types segments pour la BD;
- des routines "*Segment Edit/Compression exit*", qui sont optionnelles, pour permettre à des routines de l'utilisateur de manipuler chaque occurrence d'un type segment de ou vers un stockage auxiliaire;
- l'utilisation de pointeurs hiérarchiques ou enfants physiques/jumeaux physiques entre segments dans la BD;
- des relations logiques, optionnelles, entre segments utilisant les options de pointeurs symboliques et/ou d'adresse directe;
- de 0 à 255 champ(s) pour chaque type segment, et un maximum de 1000 pour la BD;
- un maximum de 32 relations d'index secondaires, optionnelles, par type segment et un maximum de 1000 pour la BD;
- une routine "*Data Capture exit*", qui est optionnelle, pour permettre à des utilisateurs finals DB2 de mettre à jour les données IMS.

B.4. Base de données HIDAM

Pour une base de données HDAM, on spécifie:

- de 1 à 10 DATASET;
- combien d'espace libre doit être distribué dans chaque groupe de données;
- le *ddname* d'un ensemble de données OSAM ou ESDS pour chaque groupe de données défini;
- au moins un type segment pour chaque DATASET, et un maximum de 255 types segments pour la BD;
- des routines "*Segment Edit/Compression exit*", qui sont optionnelles, pour permettre à des routines de l'utilisateur de manipuler chaque occurrence d'un type segment de ou vers un stockage auxiliaire;
- un maximum de 32 relations d'index secondaires, optionnelles, par type segment et un maximum de 1000 pour la BD;
- l'utilisation de pointeurs hiérarchiques ou enfants physiques/jumeaux physiques entre segments dans la BD;
- des relations logiques, optionnelles, entre segments utilisant les options de pointeurs symboliques et/ou d'adresse directe;
- de 0 à 255 champ(s) pour chaque type segment, et un maximum de 1000 pour la BD, un d'entre eux doit être un champ de séquence unique dans le segment racine pour indexer les occurrences du segment racine;
- une routine "*Data Capture exit*", qui est optionnelle, pour permettre à des utilisateurs finals DB2 de mettre à jour les données IMS.

B.5. Base de données MSDB

Pour une base de données HDAM, on doit spécifier:

- un nom de BD;
- un DATASET;
- un type segment pour la BD;
- de 0 à 255 champs dans la BD.

Et on ne peut spécifier:

- une relation logique ou d'index entre types segments;
- des champs utilisés dans des index secondaires.

B.6. Base de données DEDB

Pour une base de données HDAM, on doit spécifier:

- un nom de BD;
- de 1 à 240 parties dans la BD;
- de 1 à 127 type(s) segment(s) pour la BD;
- de 0 à 255 champ(s) pour chaque type segment, et un maximum de 1000 pour la BD, un d'entre eux doit être un champ de séquence unique dans le segment racine pour indexer les occurrences du segment racine;
- le *ddname* ou le nom d'une partie utilisé pour décrire une partie;
- une routine "*Data Capture exit*", qui est optionnelle, pour permettre à des utilisateurs finals DB2 de mettre à jour les données IMS.

On peut optionnellement spécifier jusqu'à 8 pointeurs de sous-ensembles pour chaque type enfant du parent.

Et on ne peut spécifier:

- une relation logique ou d'index entre types segments;
- des champs utilisés dans des index secondaires.

B.7. Base de données INDEX

La génération d'une DBD index primaire d'une BD HIDAM crée une BD index composée d'un type segment qui indexe les occurrences du type segment racine de la BD HIDAM. Un segment index contient:

- la clé du champ de séquence de l'occurrence du segment racine qu'il indexe;
- dans son préfixe, un pointeur direct d'adresse vers l'occurrence segment racine.

Pour une BD index primaire d'une BD HIDAM, on doit spécifier:

- un nom de BD;
- un DATASET, on doit spécifier le *ddname* d'un ensemble de données OSAM ou le *ddname* d'un KSDS;
- un type segment;
- la relation index requise entre la BD index primaire et le type segment racine de la BD HIDAM;
- un champ dans le type segment.

Et on ne peut spécifier des champs utilisés dans des index secondaires.

La génération d'une DBD index secondaire crée une BD index secondaire possédant de 1 à 16 types segments pointeurs index. Ceux-ci sont utilisés pour indexer les types segments cibles en HISAM, HDAM, ou HIDAM.

Pour une BD index secondaire, on doit spécifier:

- un nom de BD;
- un DATASET. Si toutes les clés de segments pointeurs index sont uniques, on doit spécifier le *ddname* d'un KSDS. Si des clés de segments pointeurs index ne sont pas uniques, on doit spécifier les *ddnames* d'un KSDS et d'un ESDS. Un index secondaire doit utiliser VSAM;
- de 1 à 16 type(s) segment(s);
- de 1 à 16 relation(s) d'index secondaire(s);
- de 1 à 1000 champ(s) pour chaque type segment.

B.8. Base de données logique

La génération d'une BD logique crée une BD logique possédant des types segments logiques. Un type segment logique est un type segment défini dans une BD logique qui représente un type segment ou la concaténation de deux types segments définis dans une ou plusieurs BD.

Pour une BD logique, on doit spécifier:

- un nom de BD;
- un DATASET;
- de 1 à 255 types segments. Chaque type segment défini le nom d'un type segment logique, et le nom du ou des type(s) segment(s) dans les BD physiques qui doivent être traités lors d'un *call* pour former le type segment logique.

Les relations logiques utilisées pour créer une BD logique doivent avoir été définies dans une ou plusieurs BD physique(s). Tous les champs requis pour les segments dans une BD logique doivent avoir été définis dans des BD physiques.

B.9. Restrictions sur les instructions

Nous allons ici établir un tableau dans lequel nous pouvons voir le nombre de fois qu'une instruction apparaît dans la déclaration de la BD suivant la méthode d'accès utilisée. Ce tableau est représenté ci-dessous à la Figure 76.

Instruction	HSAM	HISAM	HDAM	HIDAM	MSDB	DEDB	INDEX	Logique
PRINT	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1
DBD	1	1	1	1	1	1	1	1
Dataset	1	1-10	1-10	1-10	1	0	1	1
AREA	0	0	0	0	0	1-240	0	0
SEGM	1-255	1-255	1-255	1-255	1	1-127	1[6]	1-255
LCHILD	0	0-255	0-255	0-255	0	0	1[6]	0
FIELD[7]	0-1000	1-1000	0-1000	1-1000	0-255	1-1000	1[8]	0
XDFLD[7]	0	0-1000	0-1000	0-1000	0	0	0	0
DBDGEN	1	1	1	1	1	1	1	1
FINISH	1	1	1	1	1	1	1	1
END	1	1	1	1	1	1	1	1

Figure 76 : Fréquence des instructions pour les différentes BD

[6] Un maximum de 16 pour les BD index secondaires.

[7] Le maximum d'instructions combinées FIELD et XDFLD par DBD est de 1000.

[8] Un maximum de 1000 pour les BD index secondaires.

Annexe C

Cette annexe nous montre l'utilisation correcte des paramètres du mot clé POINTER suivant les différentes bases de données. Ce tableau est tiré de [IBM/IMS, 93].

Paramètre	GSAM MSDB DEDB	HSAM SHSAM SHISAM	HISAM	HDAM	HIDAM	INDEX
HIER	INVALID	VALID	IGN	VALID	VALID	IGN
HIERBWD	INVALID	INVALID	IGN	VALID	VALID	IGN
TWIN	INVALID	INVALID	IGN	VALID	VALID	IGN
TWINBWD	INVALID	INVALID	IGN	VALID	VALID	IGN
CTR	INVALID	INVALID	VALID	VALID	VALID	IGN
LTWIN	INVALID	INVALID	IGN	VALID[9]	VALID[9]	IGN
LTWINBWD	INVALID	INVALID	IGN	VALID[9]	VALID[9]	IGN
LPARNT	INVALID	INVALID	VALID[10]	VALID[11]	VALID[11]	IGN
PAIRED	INVALID	INVALID	VALID[12]	VALID[13]	VALID[13]	IGN

Figure 77 : Utilisation correcte des paramètres du mot clé POINTER

INVALID : ce paramètre ne peut être spécifié; IGN : ce paramètre peut être spécifié mais est ignoré; VALID : ce paramètre est valide.

---

[9] Utilisé quand un segment enfant logique participe dans une relation logique. Ceci devrait être spécifié si le segment existe dans une BD HDAM ou HIDAM et le parent logique est relié à l'enfant logique avec des adresses directes.

[10] Peut être utilisé quand un segment enfant logique est défini dans une BD HISAM et le parent logique est défini dans une BD HDAM ou HIDAM.

[11] Peut être utilisé quand un segment enfant logique est défini dans une BD HDAM ou HIDAM et le parent logique dans une BD HDAM et HIDAM.

[12] Peut être utilisé quand un segment enfant logique est défini dans une BD HISAM et le parent logique est défini dans une BD HDAM, HISAM ou HIDAM, et la relation logique est bidirectionelle.

[13] Utilisé quand une relation logique bidirectionelle est défini avec deux segments enfants logiques, les deux étant physiquement présents ou dans l'instruction SEGM pour l'enfant logique virtuel.

Annexe D

Cette annexe contient le code source DDL servant de base pour l'extraction de notre étude de cas.

```
1       DBD     NAME=(Personne)
2               ,ACCESS=(HDAM,VSAM)
3               ,RMNAME=(rand1,2,200,500) ,PASSWD=NO
4               ,EXIT=((sortie1,NOKEY,NODATA,NOCASCADE))
5               ,VERSION=1
6       DATASET  DD1=ens1,BLOCK=50,SIZE=(,500),SCAN=5
7               ,FRSPC=(4,80)
8       SEGM     NAME=01Depart,PARENT=0,BYTES=53
9       FIELD    NAME=(numero,SEQ,U),BYTES=3,START=1,TYPE=C
10      FIELD    NAME=(nom),BYTES=10,START=4,TYPE=C
11      FIELD    NAME=(chef),BYTES=20,START=14,TYPE=C
12      FIELD    NAME=(localisa), BYTES=20,START=34,   TYPE=C
13      SEGM     NAME=01Clasper,PARENT=(01Depart,DBLE),BYTES=48,FREQ=7
14              ,POINTER=TWINBWD
15      FIELD    NAME=(nom,SEQ,U),BYTES=20,START=1,TYPE=C
16      FIELD    NAME=(fonction),BYTES=25, START=21,TYPE=C
17      FIELD    NAME=(niveau),BYTES=3,START=46,TYPE=X
18      LCHILD   NAME=(01idxc1,Bdindx2),POINTER=INDX
19      XDFLD    NAME=01flind1,CONST=1,SRCH=(fonction,niveau),SUBSEQ=(fonction,niveau),
20              DDATA=(fonction,niveau),NULLVAL=' ',EXTRTN=routind
21      LCHILD   NAME=(01idxc2,Bdindx2),POINTER=INDX
22      XDFLD    NAME=01flind2,CONST=2,SRCH=fonction,NULLVAL='$'
23      SEGM   NAME=01Pers,PARENT=(01Clasper),BYTES=180,FREQ=25,
24              POINTER=TWINBWD
25      LCHILD   NAME=(01PtFil,)
26      FIELD    NAME=(ident),SEQ,U,BYTES=5,START=1,TYPE=C
27      FIELD    NAME=(nom),BYTES=15,START=6,TYPE=C
28      FIELD    NAME=(prenom),BYTES=20,START=21,TYPE=C
29      FIELD    NAME=(adresse),BYTES=91, START=41,TYPE=C
30      LCHILD   NAME=(01idxp1,Bdindx2),POINTER=INDX
31      XDFLD  NAME=01flind3,SEGMENT=01Diplom,CONST=3,SRCH=(denom,grade)
32              ,SUBSEQ=(denom,grade),DDATA=(denom,grade),NULLVAL='$',EXTRTN=routind
33      SEGM     NAME=01Diplom,PARENT=(01Pers),BYTES=70,FREQ=5,POINTER=HIERBWD
34              ,RULES=(,FIRST)
35      FIELD    NAME=(denom),BYTES=60,START=1,TYPE=C
36      FIELD    NAME=(grade),SEQ,M,BYTES=10,START=61,TYPE=C
37      SEGM     NAME=01Exper,PARENT=(01Pers),BYTES=75,FREQ=12,POINTER=HIERBWD
38              ,RULES=(,FIRST)
39      FIELD    NAME=(nom),BYTES=20,START=1,TYPE=C
40      FIELD    NAME=(nbre_ann,SEQ,M),BYTES=5,START=21,TYPE=P
41      FIELD    NAME=( apprec),BYTES=50,START=26,TYPE=C
42      SEGM     NAME=01PtPjt,PARENT=(01Pers,DBLE,(01Projet,VIRTUAL,Personne))
43              ,BYTES=5,POINTER=TWINBWD,LTWINBWD,LPARNT
44              ,RULES=(P,P,P,LAST)
```

```
45      FIELD   NAME=(nbrheur),BYTES=5,START=1,TYPE=C
46      SEGM    NAME=01Parr,PARENT=(01Pers),BYTES=30,POINTER=NOTWIN
47      FIELD   NAME=nomparr,BYTES=20,START=1,TYPE=C
48      FIELD   NAME=(apprec),BYTES=10,START=21,TYPE=C
49      SEGM    NAME=01PtFil,PARENT=(01Pers,(01Pers,VIRTUAL,))
50              ,BYTES=10,FREQ=5,POINTER=TWIN,LTWIN,LPARNT
51              ,RULES=(L,L,L,LAST)
52      FIELD   NAME=(apprec),BYTES=10,START=1,TYPE=C
53      SEGM    NAME=01DesPer,PARENT=(01Pers),BYTES=200,FREQ=1
54      FIELD   NAME=(descript),BYTES=200,START=1,TYPE=C
55      SEGM    NAME=01Projet,PARENT=(01Depart),BYTES=70
56      LCHILD  NAME=(01PtPjt,Personne),POINTER=DBLE,RULES=,LAST
57      FIELD   NAME=(numero,SEQ),BYTES=3,START=1,TYPE=C
58      FIELD   NAME=(nom,SEQ),BYTES=20,START=4,TYPE=C
59      FIELD   NAME=(but),BYTES=30,START=24,TYPE=C
60      FIELD   NAME=(datefin),BYTES=10,START=54,TYPE=C
61      DATASET DD1=ens2
62      SEGM    NAME=01Despjt,PARENT=(01Projet),BYTES=200
63              ,COMPRTN=(routcomp,KEY,INIT)
64      FIELD   NAME=(descript),BYTES=200,START=1,TYPE=C
65      SEGM    NAME=01Desdep,PARENT=(01Depart),BYTES=200
66              ,COMPRTN=(routcomp,KEY,INIT)
67      FIELD   NAME=(descript),BYTES=200,START=1,TYPE=C
68      DBDGEN
69      FINISH
70      END

71      DBD     NAME=Bdindx2,ACCESS=INDEX
72      DATASET DD1=ddindx2,DEVICE=3350
73      SEGM    NAME=01idxc1,PARENT=0,BYTES=28
74      LCHILD  NAME=(01Clasper,Personne),INDEX=01flind1
75      FIELD   NAME=(indx1,SEQ,U),BYTES=28,START=1
76      SEGM    NAME=01idxc2,PARENT=0,BYTES=25
77      LCHILD  NAME=(01Clasper,Personne),INDEX=01flind2
78      FIELD   NAME=(indx2,SEQ,U),BYTES=25,START=1
79      SEGM    NAME=01idxp1,PARENT=0,BYTES=70
80      LCHILD  NAME=(01Pers,Personne),INDEX=01flind3
81      FIELD   NAME=(indx3,SEQ,U),BYTES=70,START=1
82      DBDGEN
83      FINISH
84      END

85      DBD     NAME=(BudUsine)
86              ,ACCESS=(HIDAM,VSAM)
87              ,EXIT=((sortie2,NOKEY,NODATA,NOCASCADE))
88              ,VERSION=1
89      DATASET DD1=ensbud,BLOCK=50,SIZE=(,500),SCAN=5
90      SEGM    NAME=01Usine,PARENT=0,BYTES=110
91      LCHILD  NAME=(01idxus,Bdindx),POINTER=INDX
92      FIELD   NAME=(numero,SEQ,U),BYTES=10,START=1,TYPE=C
93      FIELD   NAME=(nom),BYTES=20,START=11,TYPE=C
94      FIELD   NAME=(localite),BYTES=30,START=31,TYPE=C
95      FIELD   NAME=(fonction),BYTES=50,START=61,TYPE=C
96      SEGM    NAME=01Dep2,PARENT=(01Usine),BYTES=53,POINTER=HIERBWD
97      FIELD   NAME=(numero,SEQ,U),BYTES=3,START=1,TYPE=C
98      FIELD   NAME=(nom),BYTES=10,START=4,TYPE=C
99      FIELD   NAME=(chef),BYTES=20,START=14,TYPE=C
100     FIELD   NAME=(localisa), BYTES=20,START=34,  TYPE=C
101     SEGM    NAME=01Serv,PARENT=(01Dep2),BYTES=49 , POINTER=HIERBWD
102     FIELD   NAME= (numero,SEQ),BYTES=4,START=1,TYPE=C
103     FIELD   NAME=(nom),BYTES=20,START=5,TYPE=C
104     FIELD   NAME=(nbreemp),BYTES=5,START=25,TYPE=P
105     FIELD   NAME=(chef),BYTES=20,START=30,TYPE=C
106     SEGM    NAME=01budget,PARENT=(01Serv),BYTES=30,POINTER=HIERBWD
107     FIELD   NAME=(nom),BYTES=20,START=1,TYPE=C
108     FIELD   NAME=(montant),BYTES=10,START=21,TYPE=P
109     DBDGEN
110     FINISH
111     END
```

```
112     DBD     NAME=(Bdindx),ACCESS=INDEX
123     DATASET  DD1=ddindx,DEVICE=3350
124     SEGM    NAME=01idxus,BYTES=10
125     LCHILD  NAME=(01Usine,BudUsine),INDEX=numero
126     FIELD   NAME=(numindx,SEQ,U),BYTES=10,START=1
127     DBDGEN
128     FINISH
129     END

130     DBD    NAME=(Bdlog1),ACCESS=LOGICAL
131     DATASET LOGICAL
132     SEGM   NAME=01LDept,PARENT=0,SOURCE=((01Depart,DATA,Personne))
133     SEGM   NAME=01LClaPr,PARENT=01LDept,SOURCE=((01Clasper,KEY,Personne))
134     SEGM   NAME=01LPers,PARENT=01LClaPr,SOURCE=((01Pers,DATA,Personne))
135     SEGM   NAME=01LFille,PARENT=01LPers,SOURCE=((01PtFil,DATA,Personne)
136       ,(01Pers,DATA,Personne))
137     SEGM   NAME=01LDesP,PARENT=01LFille,SOURCE=((01DesPer,DATA,Personne))
138     DBDGEN
139     FINISH
140     END

141     DBD    NAME=(Bdlog2),ACCESS=LOGICAL
142     DATASET LOGICAL
143     SEGM   NAME=01LDept,PARENT=0,SOURCE=((01Depart,DATA,Personne))
144     SEGM   NAME=01LClaPr,PARENT=01LDept,SOURCE=((01Clasper,KEY,Personne))
145     SEGM   NAME=01LPers,PARENT=01LClaPr,SOURCE=((01Pers,DATA,Personne))
146     SEGM   NAME=01LPjt,PARENT=01LPers,SOURCE=((01PtPjt,DATA,Personne)
147       ,(01Projet,DATA,Personne))
148     SEGM   NAME=01LDespj,PARENT=01LPjt,SOURCE=((01Despjt,DATA,Personne))
149     DBDGEN
150     FINISH
151     END
```

Annexe E

Cette annexe contient le code source DDL sur les PCB.

```
PCB     PCBTYPE=DB,DBNAME=Personne,PCBNAME=vue1,PROCOPT=GI,KEYLEN=63
SENSEG          NAME=01Depart,PARENT=0
SENFLD          NAME=numero,START=1
SENFLD          NAME=nom,START=4
SENFLD          NAME=chef,START=14
SENSEG          NAME=01Clasper,PARENT=01Depart,PROCOPT=K
SENSEG          NAME=01Pers,PARENT=01Clasper
SENFLD          NAME=nom,START=1
SENFLD          NAME=prenom,START=21
SENFLD          NAME=adresse,START=41
SENSEG          NAME=01Projet,PARENT=01Depart
SENFLD          NAME=numero,START=1
PCB     PCBTYPE=DB,DBNAME=Personne,PCBNAME=vue2,PROCOPT=G,KEYLEN=6
SENSEG          NAME=01Depart,PARENT=0
SENFLD          NAME=nom,START=1
SENSEG          NAME=01Projet,PARENT=01Depart,PROCOPT=GIR
SENFLD          NAME=nom,START=1
PSBGEN          PSBNAME=userview,LANG=COBOL,MAXQ=5
```

Annexe F

Cette annexe contient les figures illustrant l'étude de cas du chapitre 5.

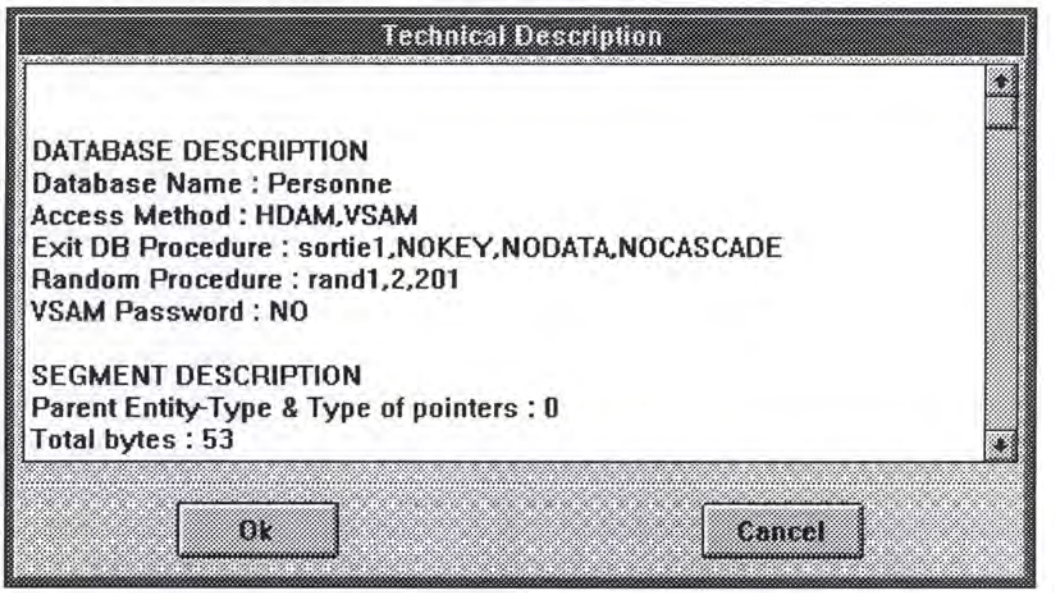

Figure 78 : Description technique du type d'entité *01depart*

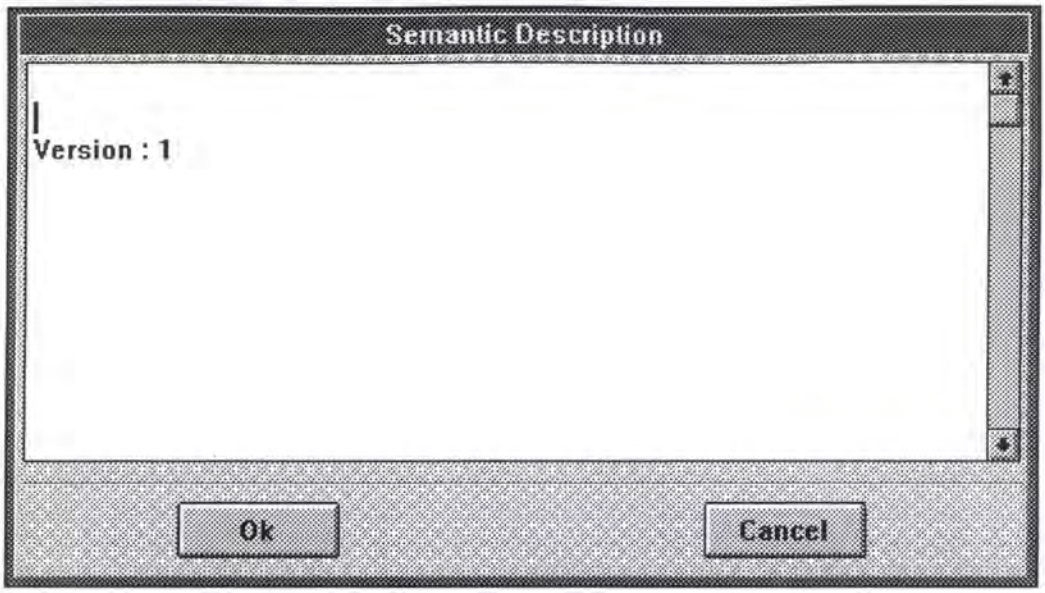

Figure 79 : Description sémantique du type d'entité *01Depart*

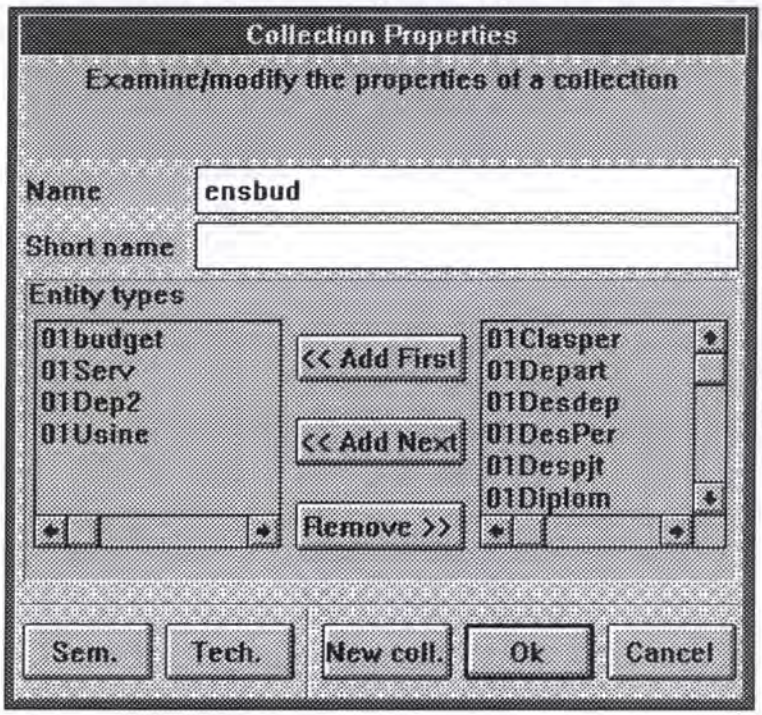

Figure 80 : Boîte de dialogue sur la collection *ensbud*

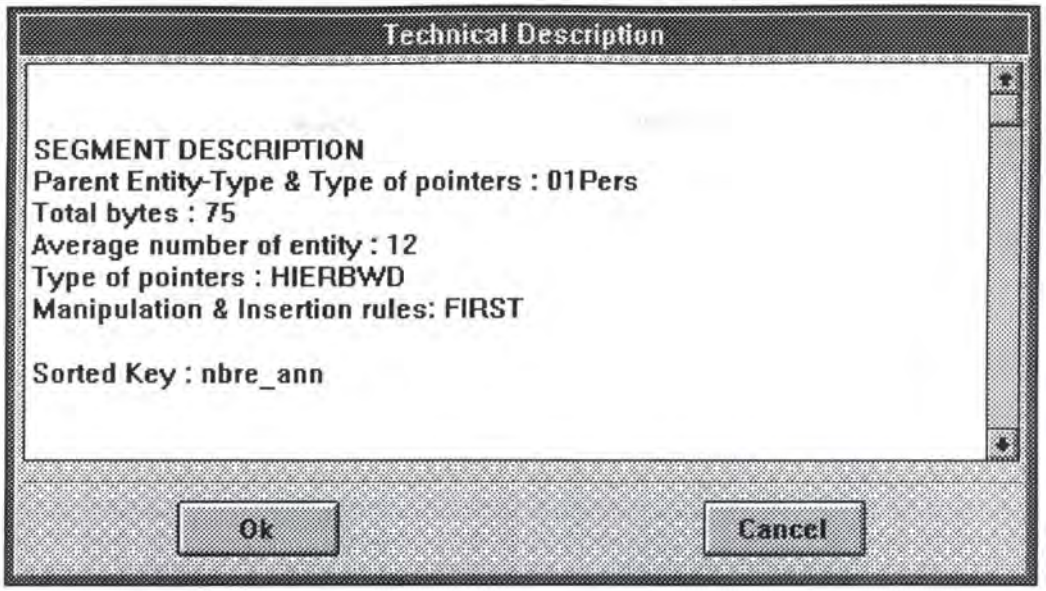

Figure 81 : Description technique du type d'entité *01Exper*

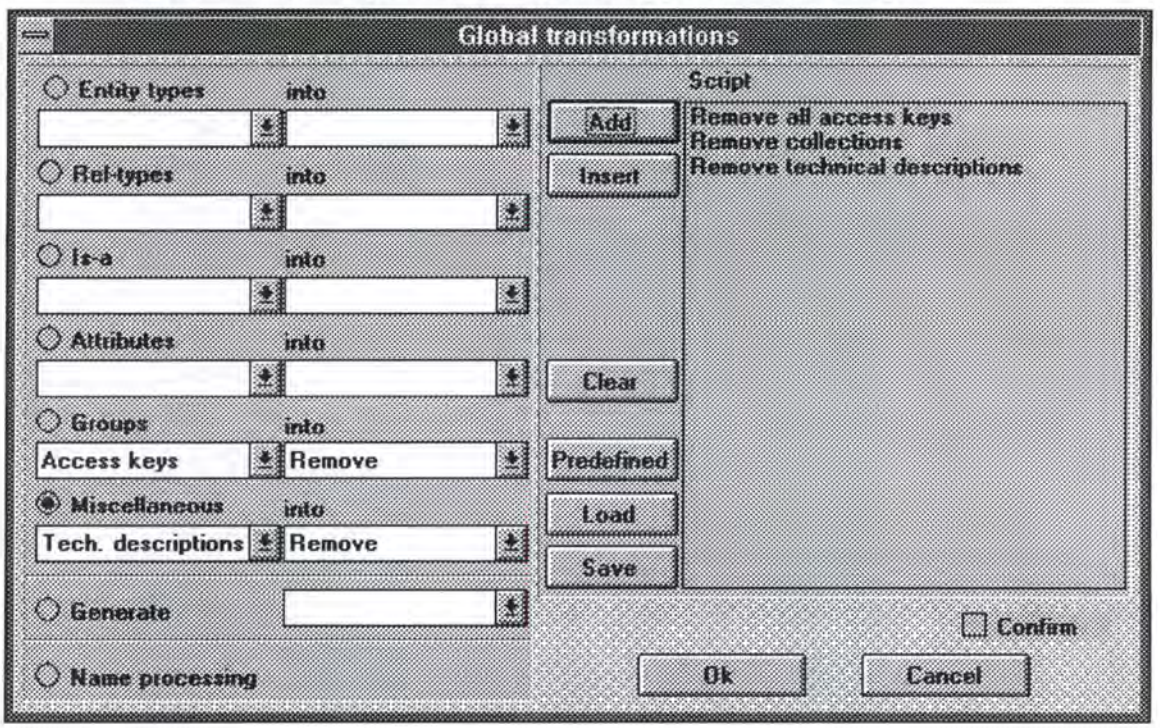

Figure 82 : Boîte de dialogue pour une transformation globale

Annexe G

Cette annexe contient le code COBOL des programmes manipulant la base de données définie dans l'annexe D.

```
1       IDENTIFICATION DIVISION
2       PROGRAM-ID.  DEPART.
3       ENVIRONMENT DIVISION.
4       ...
5       DATA DIVISION.
6       WORKING-STRORAGE SECTION.
7       01 GU          PIC XXXX VALUE 'GU  '.
8       01 GHU         PIC XXXX VALUE 'GHU '.
9       01 GN          PIC XXXX VALUE 'GN  '.
10      01 GHN         PIC XXXX VALUE 'GHN'.
11      01 GNP         PIC XXXX VALUE 'GNP'.
12      01 GHNP        PIC XXXX VALUE 'GHNP'.
13      01 REPL        PIC XXXX VALUE 'REPL'.
14      01 ISRT        PIC XXXX VALUE 'ISRT'.
15      01 DLET        PIC XXXX VALUE 'DLET'.

16      01 IO-PCB.
17              02 FILLER        PIC X(10).
18              02 IO-STAT-CODE      PIC XX.
19              02 FILLER        PIC X(20).

20      01 SSA         PIC X(30).

21      01 PERS.
22              02 ID            PIC X(5).
23              02 NOM           PIC X(15).
24              02 PRENOM        PIC X(20).
25              02 ADRESSE       PIC X(91).
26      01 WRK-ADRESSE.
27              02 RUE           PIC X(50).
28              02 NUMERO        PIC X(5).
29              02 CODE-POS      PIC X(6).
30              02 LOCALITE      PIC X(30).
31      01 DIPLOM.
32              02 DENOM         PIC X(60).
33              02 GRADE         PIC X(10).
34      01 EXPER.
35              02 NOM           PIC X(20).
36              02 NBANN         PIC X(5).
37              02 APPREC        PIC X(50).
38      01 USINE.
39              02 NUMERO        PIC X(10).
40              02 NOM           PIC X(20).
41              02 LOCALITE      PIC X(30).
```

```
42              02 FONCTION  PIC X(50).
43      01 DEP2.
44              02 NUMERO    PIC X(3).
45              02 NOM       PIC X(10).
46              02 CHEF      PIC X(20).
47              02 LOCAL     PIC X(20).
48      01 I     PIC 99.

49      01 DEPART.
50              02 NUMERO    PIC X(3).
51              02 NOM       PIC X(10).
52              02 CHEF      PIC X(20).
53              02 LOCAL     PIC X(20).

        ...

54      PROCEDURE DIVISION
55      DEBUT.

        .....
56      *** lecture d'un segment Personne
57              CALL 'CBLTDLI' USING GU, IO-PCB, PERS, SSA.
58              MOVE ADRESSE OF PERS TO WRK-ADRESSE.

59      *** insertion d'un segment Personne et de son segment Experience
60              CALL 'CBLTDLI' USING ISRT, IO-PCB, PERS.
61              CALL 'CBLTDLI' USING ISRT, IO-PCB, EXPER.



62      *** lecture des segment Diplome d'un segment Personne
63              CALL 'CBLTDLI' USING GU, IO-PCB, PERS,SSA.
64              CALL 'CBLTDLI' USING GU, IO-PCB,DIPLOM.
65              PERFORM VARYING I FROM 2 BY 1 UNTIL i>5 OR DENOM="
66                      CALL 'CBLTDLI' USING GNP, IO-PCB,DIPLOM
67                      MOVE .....
68                      .....
69              END-PERFORM.

70      *** insertion d'un segment Usine et d'un segment Dep associe
71              CALL 'CBLTDLI' USING ISRT, IO-PCB, USINE.
72              CALL 'CBLTDLI' USING ISRT, IO-PCB, DEP.

73      *** gestion de la redefinition
74              CALL 'CBLTDLI' USING GU, IO-PCB,DEPART,SSA.
75              MOVE DEPART TO DEP.
76              CALL 'CBLTDLI' USING ISRT, IO-PCB, DEP.

        .......

77      STOP RUN.
```