THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Contribution à la rétro-ingénierie des bases de données IMS

Grenier, Christophe; Jacquemin, Laurent

Award date:
1997

Facultés Universitaires Notre-Dame de la Paix
INSTITUT D'INFORMATIQUE
rue Grandgagnage, 21 5000 NAMUR
Tel 081/72.49.83. fax 081/72.49.67.

Année académique 1996-1997

Contribution à la rétro-ingénierie des bases de données IMS

Par

Christophe GRENIER

Laurent JACQUEMIN

Promoteur: *Jean-Luc Hainaut*

Mémoire présenté en vue de l'obtention du grade de
Maître en Informatique

Remerciements

Nous tenons à exprimer toute notre reconnaissance à notre promoteur Jean-Luc Hainaut, pour ses nombreux conseils et l'attention qu'il nous a portée durant la rédaction de ce mémoire.

Nous tenons également à remercier Jean Henrard, Vincent Englebert et toute l'équipe DB-MAIN pour les nombreux conseils pratiques qu'ils nous ont fournis, et pour leur grande disponibilité.

Nous tenons encore à exprimer notre profonde gratitude à tout le personnel informatique de chez d'Ieteren et spécialement à Messieurs Pierre-François Declercq, Pascal Cousin, Georges Moulu, Gerlando Taibi pour les nombreux conseils et l'aide qu'ils nous ont fournis durant notre stage.

Nous remercions également toutes les personnes qui ont participé de près ou de loin à l'élaboration de ce mémoire.

Nous n'oublions pas les nombreuses personnes (professeurs, assistants, parents et amis) qui nous ont soutenu durant nos études.

Table des matières

PARTIE 1: INTRODUCTION

CHAPITRE 1: Introduction

1.1. Le contexte général du mémoire

La rétro-ingénierie des bases de données consiste à retrouver les spécifications fonctionnelles et techniques d'une application au travers de sources différentes[1]. Ces sources sont principalement:

- la définition de la base de données,
- les programmes existants,
- les données,
- la documentation existante,
- la culture de l'entreprise.

C'est par une analyse minutieuse de ces sources que le rétro-ingénieur obtiendra, à la fin du processus de rétro-ingénierie, un schéma conceptuel *plus ou moins proche* du schéma initial qui avait donné naissance à l'application. En effet, le schéma obtenu à la fin du processus peut différer fortement du schéma initial. Ceci peut s'expliquer par le fait que la rétro-ingénierie exploite une gamme très diverse de documentations où l'information utile est noyée dans toute une série d'autres informations. Ainsi, le SGBD IMS ne permettant pas de déclarer explicitement les clés étrangères dans le code de définition des structures de données, le rétro-ingénieur doit faire une analyse détaillée des programmes pour découvrir l'existence d'une telle contrainte. Le talent du rétro-ingénieur et son expérience de l'application sont aussi des facteurs importants.

La rétro-ingénierie est encore un domaine relativement délaissé par les chercheurs, peut-être du fait que ce processus est plus difficile et moins formel que le processus de conception. Néanmoins, depuis quelques années, le projet DB-MAIN mené à l'institut a permis une avancée considérable dans ce domaine. La participation de nombreuses entreprises privées dans ce projet montre le réel intérêt des organisations pour ce domaine relativement nouveau. Cet intérêt est dû au fait que certaines entreprises sont confrontées à des problèmes lorsque, par exemple, elles veulent migrer d'un SGBD à un autre. Elles éprouvent des problèmes si le schéma conceptuel a disparu, si la documentation n'a pas été tenue à jour, ou même si elle est inexistante. C'est dans des cas pareils qu'un outil comme DB-MAIN peut aider une équipe à réaliser la rétro-ingénierie d'une application. Un atelier comme DB-MAIN peut aussi être utile pour régler les problèmes de l'an 2000.

La but de ce mémoire est donc de présenter une méthodologie de rétro-ingénierie de bases de données IMS, un système relativement ancien mais encore fort répandu. Nous compléterons le

[1] Le processus de rétro-ingénierie d'une base de données peut aussi être défini comme l'inverse du processus de conception d'une base de données.

travail de PH. RICHARD [**RICHARD, 95**], qui s'était attaché à l'analyse du code DDL d'IMS. Le but de ce mémoire est également de développer ou d'affiner l'outil DB-MAIN, par la réalisation d'applications utiles pour la rétro-ingénierie des bases de données IMS.

Pour mener à bien ce projet, nous avons effectué un stage au Data Processing Center de la société d'Ieteren (importateur VW, Audi, Porsche, Seat et Skoda), utilisatrice du système IMS. Nous avons pu étudier une application gérant l'ensemble des pièces de rechange et des accessoires des différentes marques d'automobiles citées ci-dessus. Cette étude nous fut très utile car nous avons pu, en analysant ces applications 'grandeur nature', relever certaines astuces utilisées dans le développement de l'application. C'est lors de ce stage que nous nous sommes rendu compte de la difficulté d'effectuer la rétro-ingénierie d'une base de données IMS. Contrairement à des systèmes plus récents, IMS ne permet pas de déclarer explicitement certaines structures ou contraintes comme les clés étrangères, les valeurs nulles, ou d'autres. De plus, en raison de la structure hiérarchique (en arbre) d'IMS, le concepteur d'une base de données doit faire preuve d'astuces pour représenter certaines constructions conceptuelles non conformes à IMS. Toute la difficulté est de retrouver ces astuces.

1.2. L'outil DB-MAIN

Lors de cette introduction, il nous semble utile de décrire brièvement l'outil DB-MAIN, pour ceux qui ne le connaîtraient pas encore. DB-MAIN a constitué notre environnement de référence lors de la rédaction du mémoire.

DB-MAIN est un outil graphique, programmable, basé sur les transformations de schémas, dédié à l'ingénierie des bases de données.
Les principaux objectifs de cet outil sont d'aider l'utilisateur lors des processus de:

- conception des bases de données,
- rétro-ingénierie des bases de données,
- ré-ingénierie des bases de données,
- maintenance et évolution des bases de données.

L'outil offre trois niveaux de transformations:

- Les transformations élémentaires (T)
  Appliquer la transformation T à l'objet courant O.
- Les transformations globales (P,T) à travers un assistant spécifique.
  Appliquer une transformation T à l'ensemble des objets qui remplissent la condition P.
- Les transformations orientées modèle (M)
  Appliquer les transformations nécessaires pour que le schéma courant satisfasse le modèle M.

Un des buts de DB-MAIN est de supporter le processus de rétro-ingénierie. Pour ce faire, il inclut des analyseurs de texte sophistiqués.

Une autre caractéristique importante de l'outil est qu'il est programmable. Son langage de programmation s'appelle VOYAGER 2. Il permet à l'utilisateur de créer lui-même tous les outils dont il aurait besoin. Les applications que nous avons développées dans le cadre de ce mémoire ont été réalisées en VOYAGER 2.

Enfin, le modèle de représentation des structures de données utilisé dans DB-MAIN est le modèle entité-association, décrit dans **[BODART, 93]**.

1.3. Structure du mémoire

La Figure 1-1 montre la structure du mémoire.

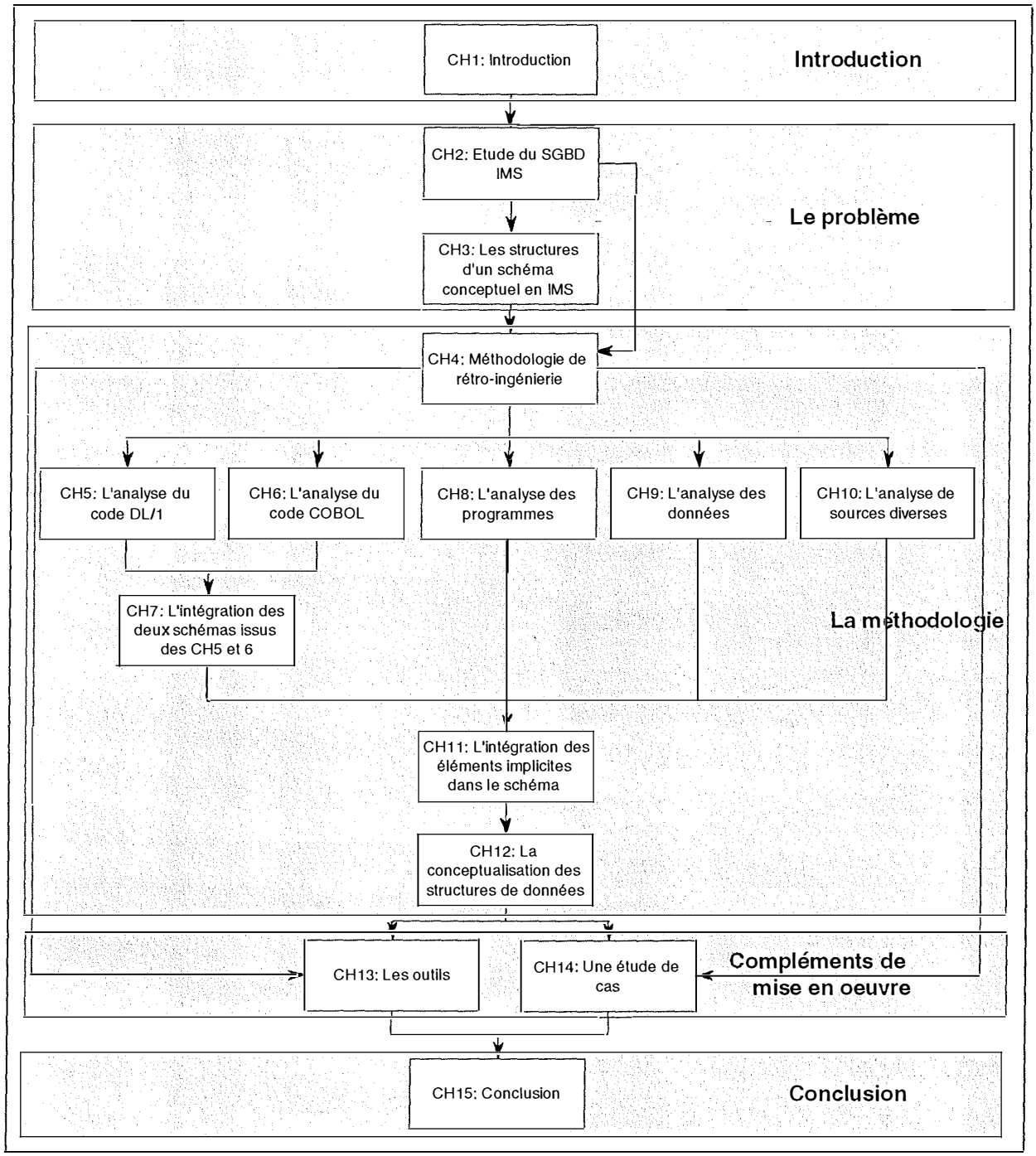


Figure 1-1: Structure du mémoire

Nous pouvons remarquer sur cette Figure 1-1 que le mémoire est divisé en cinq parties (cadres grisés).

Après la partie introductive, nous commencerons par une partie consacrée au problème, dans laquelle le chapitre deux décrira le modèle hiérarchique, et le SGBD IMS en particulier. Le chapitre trois sera consacré à la représentation des structures d'un schéma conceptuel dans un schéma conforme à IMS. Nous y exposerons une série de transformations de structures afin qu'elles soient compatibles à IMS.

La partie trois constituera le coeur de ce mémoire, puisqu'il traitera de la méthodologie de rétro-ingénierie de bases de données IMS. Le chapitre quatre présentera cette méthodologie. Les chapitres cinq à douze développeront en détail chaque étape de la méthodologie présentée au chapitre quatre.

La partie quatre est intitulée *compléments de mise en oeuvre*. Dans cette partie, nous présenterons d'une part les outils existants (et notamment ceux développés dans le cadre de ce mémoire) qui permettent de faciliter le travail du rétro-ingénieur (chapitre treize). D'autre part, le chapitre quatorze sera consacré à une étude de cas. Elle reprendra un exemple de processus complet de rétro-ingénierie, sur base du travail effectué chez d'Ieteren. Toutes les différentes étapes seront éclairées d'exemples.

La partie cinq conclura ce mémoire.

PARTIE 2: LE PROBLEME

CHAPITRE 2: L'étude du modèle hiérarchique et du SGBD IMS

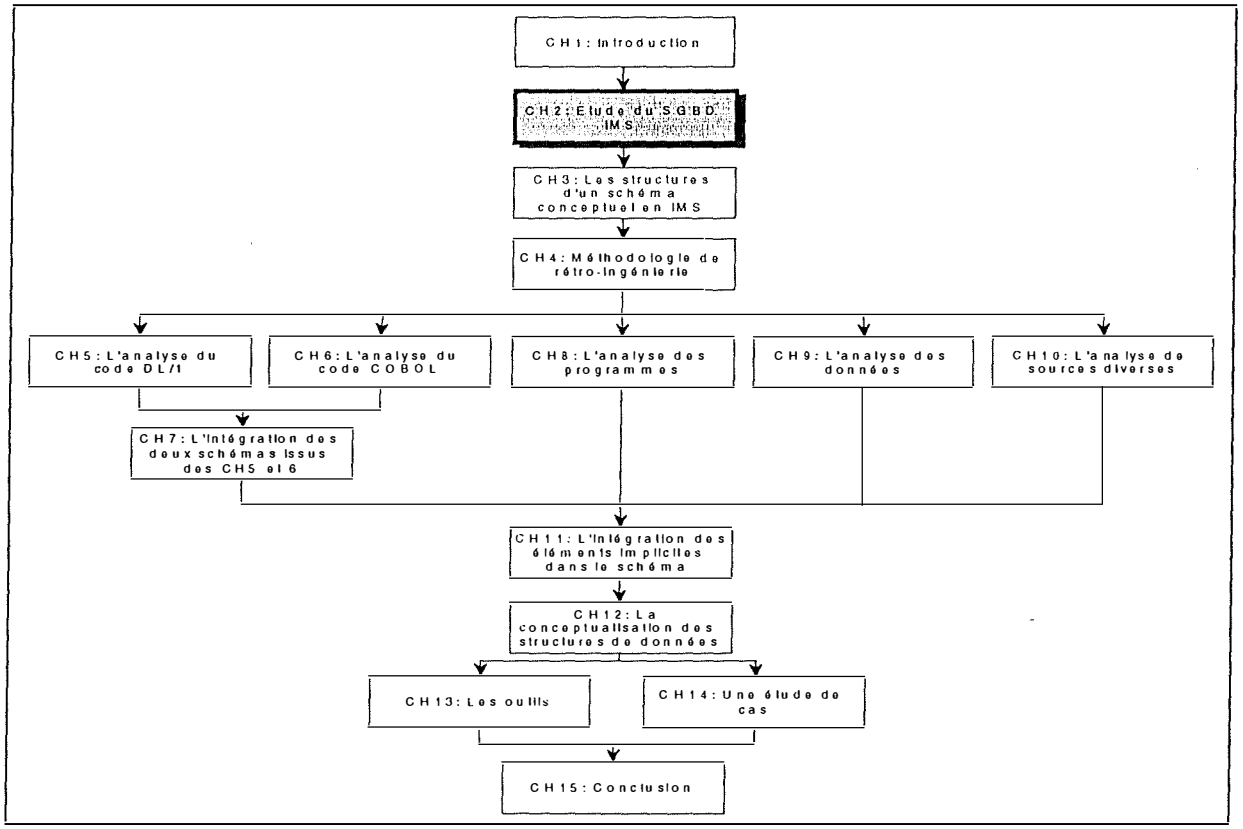


Figure 2-1: Le chapitre 2

2.1. Introduction

Avant toute chose, nous nous devons de préciser que ce chapitre deux est en grande partie issu du mémoire de Ph. RICHARD **[RICHARD, 95]**.

Information Management System (IMS) est un système de gestion de bases de données (SGBD) hiérarchique apparu à la fin des années 60. La première version apparue en 1968 est le fruit d'un projet conjoint entre IBM et North American Rockwell.

IMS est un SGBD à langage hôte, c'est-à-dire qu'il ne fournit pas son propre langage de programmation pour le développement des applications. Les langages hôtes les plus souvent utilisés avec IMS sont le COBOL, le PL/1, le FORTRAN et même l'assembleur. Des modules d'interface sont fournis pour que ces langages puissent faire appel au Data Manipulation Language (DML) d'IMS, c'est-à-dire pour permettre aux programmes d'avoir accès aux données. Enfin, notons que le langage de description des bases de données d'IMS (DDL-Data Description Language) est le DL/1.

Dans ce chapitre deux, nous allons tout d'abord voir quels sont les concepts de base du modèle hiérarchique, de façon générale, en utilisant cependant la terminologie IMS. Nous nous intéresserons ensuite à l'architecture d'IMS, à son DDL (Data Description Language) et à son DML (Data Manipulation Language). Nous introduirons aussi la notion de schéma logique. Nous terminerons ce chapitre par une brève conclusion.

2.2. Le modèle hiérarchique

2.2.1. Les concepts de base du modèle hiérarchique

Le modèle hiérarchique repose sur trois concepts essentiels: le type de segment, le type de relation physique parent-enfant et la hiérarchie.

Le type de segment est un ensemble de segments qui regroupent le même type d'informations élémentaires. Un type de segment contient donc une ou plusieurs occurrence(s): le(s) segments. Un type de 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 de relation physique parent-enfant est un type d'association binaire *one-to-many* entre un type de segment parent physique et un type de segment enfant physique. Une occurrence d'un type de relation physique consiste en une occurrence du type de segment parent physique et une ou plusieurs occurrence(s) du type de segment enfant physique.

Une base de données hiérarchique contient un certain nombre de hiérarchies. ***Une hiérarchie*** consiste en un ensemble ordonné de types de segments et de types de relations physiques arrangés de façon à former un arbre. Cette hiérarchie de types de 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 de segment racine et un ensemble de zéro, un ou plusieurs sous-arbre(s). Ces sous-arbres sont eux-mêmes constitués d'un type de segment racine et d'un ensemble de zéro, un ou plusieurs sous-arbres, et ainsi de suite. Un arbre possède donc plusieurs niveaux. Le premier niveau est celui du type de segment racine. Les sous-arbres du type de segment racine sont du deuxième niveau et ainsi de suite. Le nombre de niveaux correspond donc à la profondeur de l'arbre.

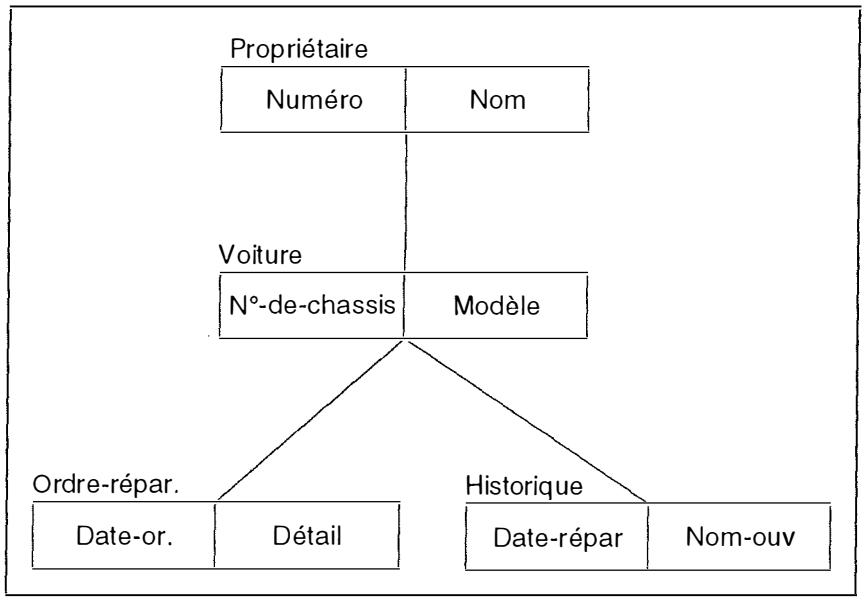


Figure 2-2: Exemple de hiérarchie de types de segments

Dans l'exemple de la Figure 2-2, nous remarquons que chaque *Propriétaire* possède un *Numéro*, un *Nom* et zéro, une ou plusieurs *Voiture(s).* Chaque *Voiture* possède un *N°-de-chassis*, un nom de *Modèle*, zéro, un ou plusieurs Ordre(s) de réparation (*Ordre-répar*) et zéro, un ou plusieurs *Historique(s).*
Dans cet exemple, l'arbre a comme racine le type de segment *Propriétaire* et possède un seul sous-arbre ayant comme racine le type de segment *Voiture.* Le sous-arbre *Voiture* possède deux sous-arbres *Ordre-répar* et *Historique.* L'ordre dans lequel sont placés les sous-arbres est important. Dans l'exemple, le sous-arbre *Ordre-répar* précède le sous-arbre *Historique.* En effet, une hiérarchie se lit de haut en bas et de gauche à droite.
Cet arbre a trois niveaux:

Niveau 1: *Propriétaire,*
Niveau 2: *Voiture* et
Niveau 3: *Ordre-répar* et *Historique.*

La BD contient quatre types de segments. Le type de segment *Propriétaire* est le type de segment racine et les trois autres sont des types de segment dépendants. La BD contient aussi trois types de relations physique parent-enfant qui sont (*Propriétaire*, *Voiture*), (*Voiture*, *Ordre-répar*) et (*Voiture*, *Historique*). Enfin, nous pouvons remarquer que *Ordre-répar* et *Historique* sont des types de segments feuilles.

Nous allons maintenant nous tourner vers les occurrences. Chaque arbre d'occurrence consiste en un arbre dont la racine est une occurrence du type de segment racine, et possédant zéro, une ou plusieurs occurrence(s) des types de segments immédiatement dépendants du type de segment racine.

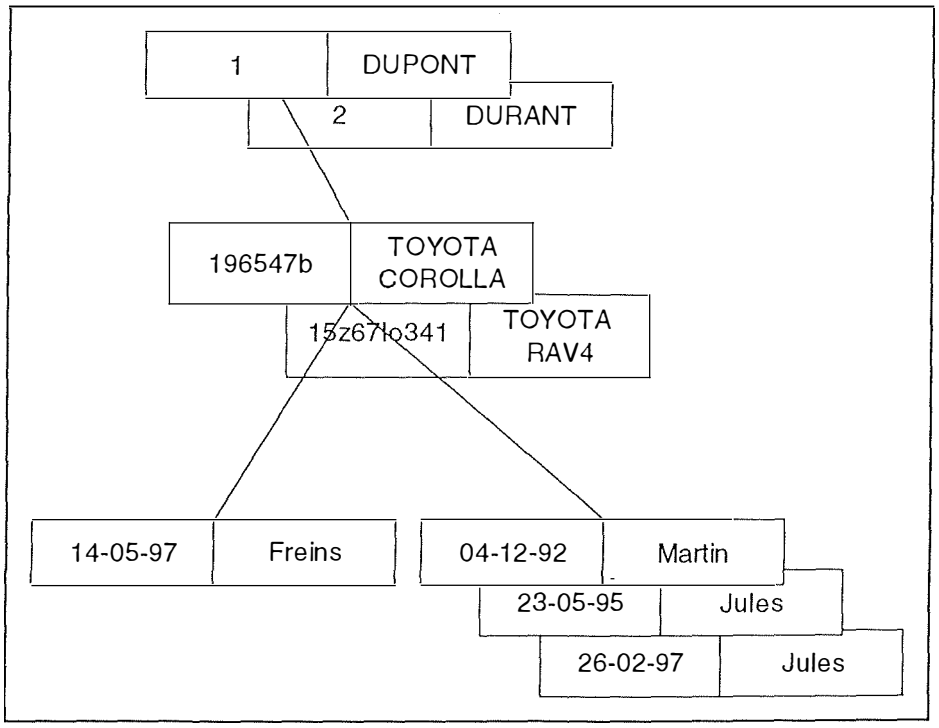


Figure 2-3: Arbre d'occurrence pour la hiérarchie de la Figure 2-2

Dans l'exemple de la Figure 2-3, issu de celui de la Figure 2-2, on s'aperçoit que le sieur Dupont possède deux voitures, dont une est en réparation pour cause de freins défectueux. Nous découvrons aussi l'historique des réparations de la voiture.
En IMS, un arbre d'occurrence est appelé enregistrement physique (physical record).

Nous terminerons par donner la définition du concept de **jumeaux**, qui est un concept propre aux arbres d'occurrences. Deux segments (occurrences d'un même type de segment) sont dits jumeaux lorsqu'ils ont comme parent la même occurrence du type de segment parent.

2.2.2. Les caractéristiques du modèle

[BATINI, 92] a mis en évidence trois propriétés que doit avoir un schéma hiérarchique:

⇒ Chaque type de segment, excepté le type de segment racine, participe, en tant que type de segment enfant, à exactement un et un seul type de relation physique.

⇒ Un type de segment peut participer en tant que type de segment parent dans aucun, un ou plusieurs type(s) de relation(s) physique(s).

⇒ Si un type de segment participe en tant que parent dans plus d'un type de relation physique, alors ses types de segments enfants sont ordonnés. L'ordre est montré, par convention graphique, de gauche à droite dans un schéma hiérarchique.

Ces trois propriétés nous disent en clair que chaque type de segment, sauf la racine, possède un et un seul parent mais peut avoir zéro, un ou plusieurs enfant(s). Cela signifie qu'il est, à ce stade, impossible de trouver des types d'associations *many-to-many*. Nous en reparlerons au paragraphe 2.2.3.

Les deux caractéristiques suivantes ont été énoncées par **[GALACSI, 89]**.
La *première caractéristique* est que plusieurs types de relations entre deux types de segments sont représentés par une redondance de types de segments. La Figure 2-4 montre une situation où on a deux types d'associations entre *Personne* et *Voiture*. La Figure 2-5 est la structure hiérarchique correspondante.

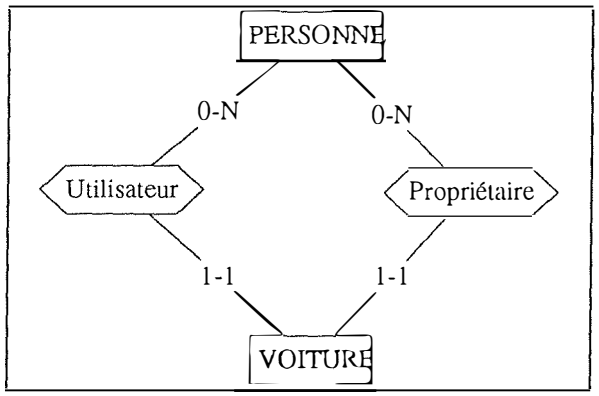


Figure 2-4: Schéma conceptuel avec deux types d'associations entre deux types d'entités

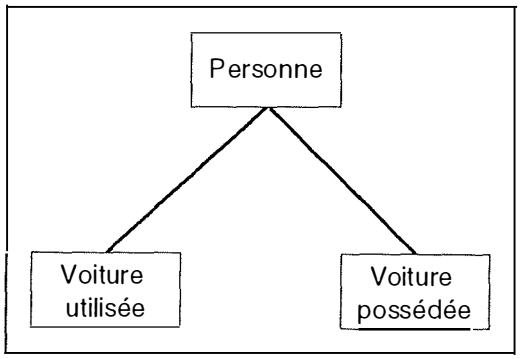


Figure 2-5: Représentation redondante de la Figure 2-4 dans un modèle hiérarchique

La *deuxième caractéristique* concerne la manipulation de la base de données. En effet, pour rechercher l'information, nous sommes obligés de passer par le type de segment racine et de parcourir le chemin de l'arbre menant à l'information voulue.

En partant de ce constat, **[GALACSI, 89]** dégage quelques inconvénients:

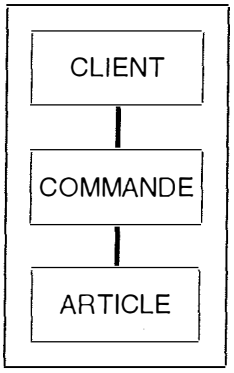


Figure 2-6: Structure hiérarchique

⇒ L'ajout d'un segment ne peut s'effectuer que si les segments supérieurs dont il dépend existent dans l'arbre. Dans l'exemple de la Figure 2-6, on ne peut introduire un article sans devoir créer un client et une commande correspondants.

⇒ 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 le vouloir, l'article qui était rattaché à sa commande.

⇒ La modification d'un segment entraîne la recherche de tous les segments qui contiennent l'information recherchée. Ainsi, dans l'exemple de la Figure 2-5, si la couleur d'une voiture change, il faut la modifier dans les deux types de segments.

⇒ Il est facile de voyager en descendant dans l'arbre mais partir des feuilles est plus difficile.

Néanmoins, ces inconvénients sont aussi des avantages en terme 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 va de même pour la suppression.

2.2.3. La relation logique parent-enfant

Nous avons vu au point 2.2.1. qu'un type de segment dépendant ne pouvait avoir qu'un et un seul parent. Le modèle hiérarchique permet cependant une entorse à cette structure rigide. Cette exception s'appelle le type de relation logique parent-enfant et permet d'établir, entre

deux types de segments appartenant à la même arborescence ou à deux arborescences différentes, des relations hiérarchiques parent-enfant.
Les deux types de segments concernés se voient attribuer les qualificatifs de type de segment parent logique et type de segment enfant logique.

Nous pouvons illustrer cette situation par le schéma conceptuel de la Figure 2-7, qui montre que le type d'entité C a deux "parents" (A et B). Cette situation, en IMS, pourrait être résolue par la situation REDONDANTE de la Figure 2-8. Dans ce cas, nous avons une redondance des types de segments qui se retrouvent dans les deux BD. De plus, nous avons une perte de sémantique car on perd le lien formé par le type d'entité C. Dans ce cas, ce sera au programmeur de tenir compte de cette perte au niveau des programmes d'application.

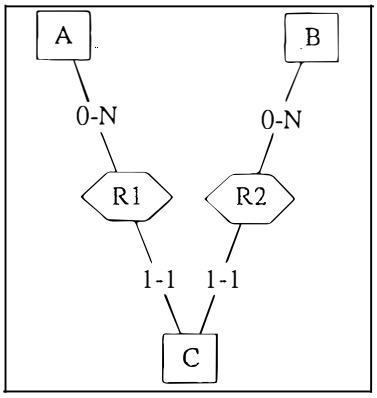


Figure 2-7: Schéma conceptuel où le type d'entité C a deux parents

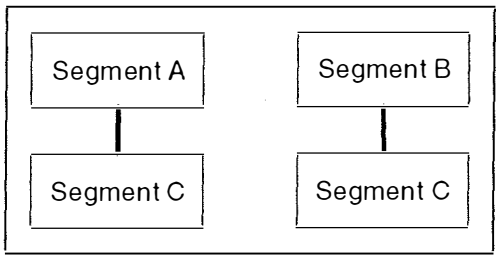


Figure 2-8: Solution redondante de la Figure 2-7 dans le modèle hiérarchique

La solution de la Figure 2-9, avec une relation logique, permet d'éviter cette redondance. Le trait discontinu entre les types de segments B et C représente la relation logique. Le type de segment B est le parent logique, et le type de segment C l'enfant logique. Ces relations logiques permettent donc de contourner la règle du "un seul parent pour un type de segment dépendant".

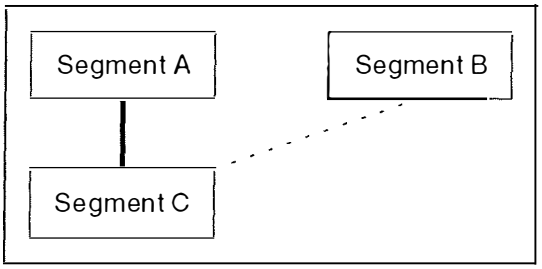


Figure 2-9: Solution non redondante de la Figure 2-7 (relation logique)

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

- 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 la base.
- Un enfant logique ne peut avoir qu'un parent logique et qu'un parent physique.
- Un parent logique peut être la racine d'une BD ou se trouver à n'importe quel niveau de hiérarchie dans sa BD.
- Un parent logique peut avoir un ou plusieurs enfant(s) logique(s).
- Un type de segment ne peut être à la fois enfant logique et parent logique.
- Un enfant logique ne peut avoir des enfants physiques qui sont aussi enfants logiques d'un autre type de segment.

La définition d'une relation logique nous permet de résoudre de façon non redondante le problème de la Figure 2-4 (deux types d'associations entre deux types d'entités).

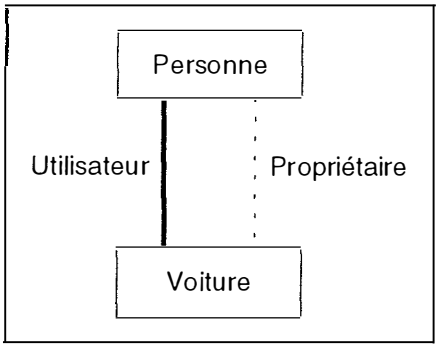


Figure 2-10: Résolution de la Figure 2-4 par un type de relation logique

Le grand avantage de ce type de relation logique est qu'il permet de résoudre le problème de la modélisation des types d'association *many-to-many*. Ainsi, si on suppose l'existence d'un type d'association [N-N] entre un type d'entité EMPLOYE et un type d'entité DEPARTEMENT, une solution non redondante est présentée à la Figure 2-11.

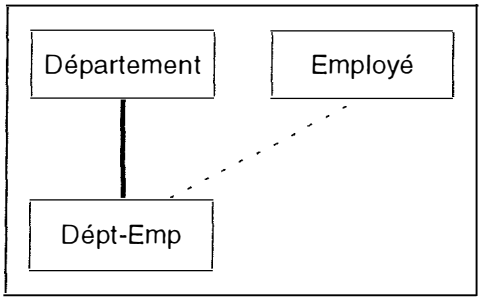


Figure 2-11: Type de relation logique unidirectionnel

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 de segment formant le pointeur logique, ici *Dépt-Emp*. Ces données sont couramment appelées des **données d'intersection**.

La notion de type de relation logique va de paire avec celle de BD logique. **[DATE, 90]** a défini une BD logique: "Une BD logique, comme une BD physique, consiste en un

arrangement hiérarchique de types de segments. Cependant, les types de segments en question, quoique accessibles par cette BD logique, appartiennent réellement à une ou plusieurs BD physique(s)."
Pour ce faire, les BD logiques implémentent la notion de type de relation logique vue auparavant. Dans la construction d'une BD logique, on prend comme type de segment racine le type de segment racine d'une BD physique. On prend comme types de segments dépendants n'importe quel type de segment enfant physique. Si le type de segment enfant physique sélectionné est aussi un type de segment enfant logique, on peut le concaténer avec son type de segment parent logique. La relation est ainsi établie. A partir de la Figure 2-11, on peut établir la BD logique de la Figure 2-12.

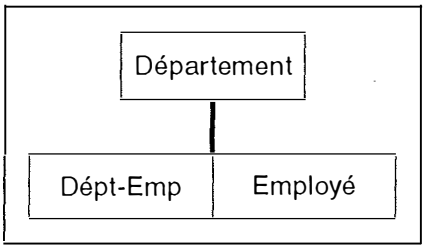


Figure 2-12: BD logique basée sur la Figure 2-11

Nous avons ainsi modélisé un type d'association [N-N].

Il y a différents types de relations logiques. Le premier est le **type de relation logique unidirectionnel**. La Figure 2-11 montre un tel type. Il est unidirectionnel parce qu'il n'est accessible que dans le sens *Département-Employé.* Dans l'exemple de la Figure 2-11, le pointeur logique aurait pu tout aussi bien être l'enfant physique d'*Employé* et l'enfant logique de *Département.* L'accès aurait alors du se faire dans le sens *Employé-Département.* Dans le cas où on veut parcourir la relation dans les deux sens, il faut déclarer deux types de segments pointeurs, comme à la Figure 2-13. Les types de segments *Dept-Emp* et *Emp-Dept* sont appelés des types de segments pairés. Nous pouvons déclarer deux BD logiques qui permettront l'accès dans les deux sens. On appelle ce type de relation logique un **type de relation logique bidirectionnel**.

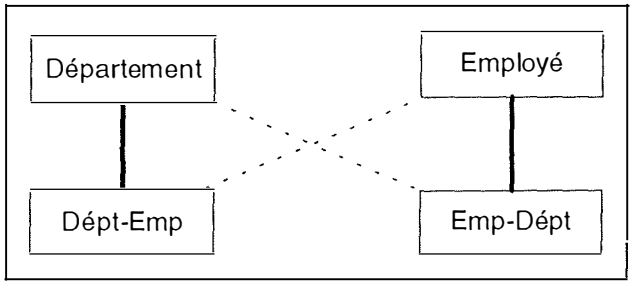


Figure 2-13: Type de relation logique bidirectionnel

A partir de la Figure 2-13, nous pouvons construire deux BD logiques, qui permettront l'accès dans les deux sens (Figure 2-14).

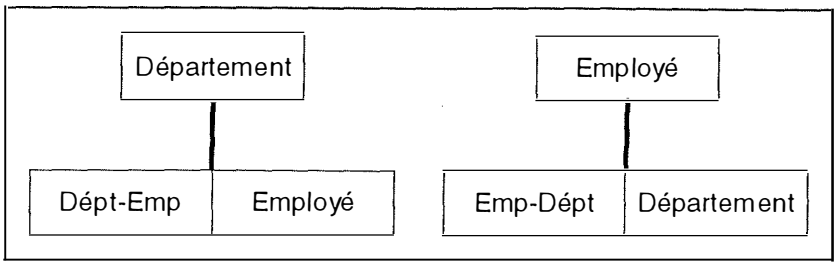


Figure 2-14: BD logique basée sur la Figure 2-13

Ce type de relation bidirectionnel se décompose en deux catégories, suivant la manière dont les deux types de segments pairés sont stockés.
La première solution consiste à les déclarer tous les deux comme types de segments enfants logiques avec la même description. Dans ce cas, ils sont tous les deux réellement stockés et sont appelés des types de segments enfants logiques REELS. IMS se charge, lors d'une insertion d'un des deux segments, d'insérer également l'autre. Ce type de relation reçoit dans ce cas le nom de type de relation logique **bidirectionnel physique**.
L'autre solution consiste à déclarer un des deux types de segments comme type de segment enfant logique réel et l'autre comme type de segment enfant logique virtuel. Les données d'intersection se trouvent dans l'enfant logique réel. L'autre type de segment n'existe pas et est donc matérialisé par une suite de pointeurs logiques sur le type de segment enfant logique réel. On appellera ce type de relation logique un type de relation logique **bidirectionnel virtuel**.

2.3. Le SGBD IMS

2.3.1. Architecture d'un environnement IMS

L'architecture d'un environnement IMS est décrite à la Figure 2-15.

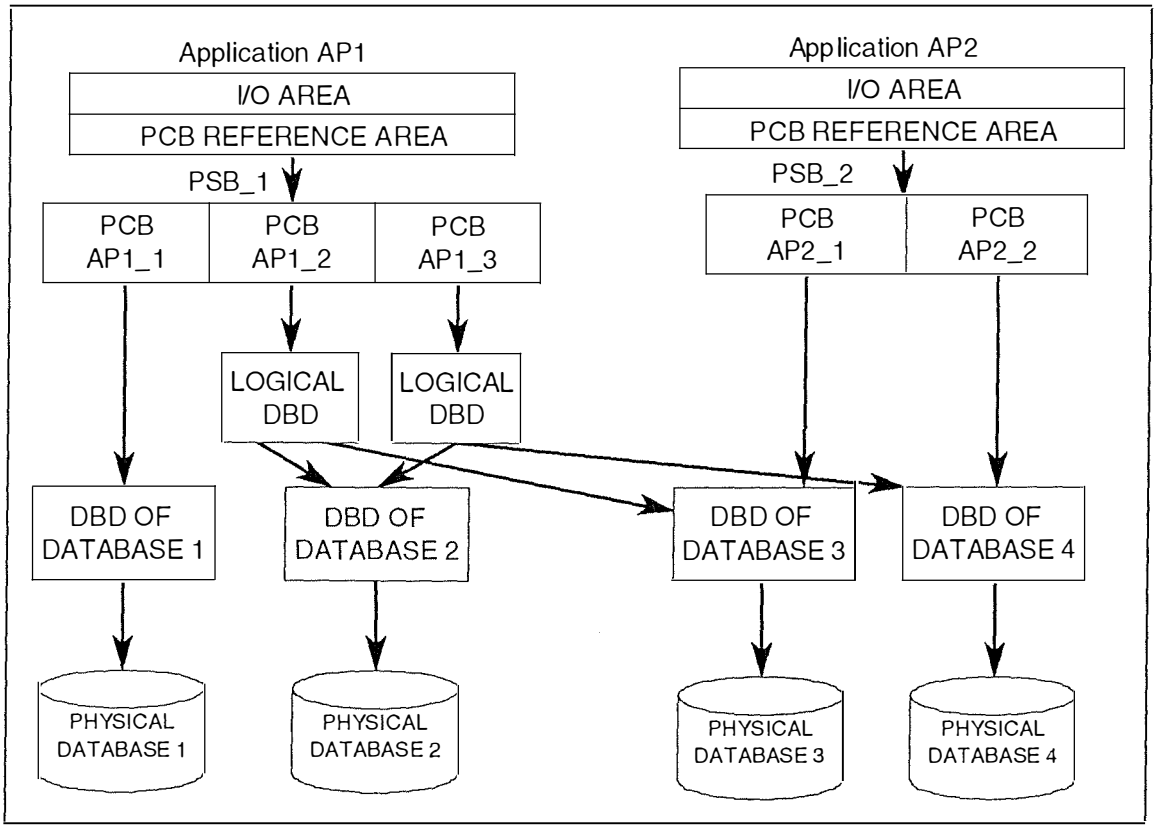


Figure 2-15: Exemple d'architecture d'un environnement IMS

Avant de décrire plus avant les composants de cet environnement, il faut noter que le DDL (Data Description Language)[1] d'IMS, appelé DL/1, est composé, des *Data Base Description (DBD)* physiques, des *Data Base Description* logiques et des *Program Communication Block (PCB)*. Les programmes d'application (couche supérieure) font appel aux instructions du DML (Data Manipulation Language)[2].
Dans les points suivants, nous décrivons les éléments de la Figure 2-15 en partant du bas de celle-ci.

[1] Pour rappel, le DDL d'un système de gestion de bases de données (SGBD) est un langage dont le but est de décrire la structure des bases de données. C'est donc dans ce langage descriptif que l'on trouvera la description des types de segments, des champs, des identifiants,... Le paragraphe 2.3 abordera brièvement le DDL d'IMS.

[2] Pour rappel, Le DML est un ensemble d'instructions qui manipulent les bases de données. Les programmes d'application écrits en langage hôte (COBOL,...) font appel aux instructions du DML par le biais d'un CALL. Le paragraphe 2.4 abordera brièvement le DML.

2.3.2. La Data Base Description (DBD)

Une BD peut être composée de plusieurs BD physiques (quatre dans la Figure 2-15). Ces BD physiques sont définies à la construction par le DBA (DataBase Administrator) et leurs définitions sont consignées, en même temps que les méthodes d'accès, dans les Data Base Description (DBD).
Chaque DBD (écrite en DL/1, le DDL d'IMS) correspond à une et une seule BD physique. 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 de relations logiques.

2.3.3. Le Program Communication Block (PCB)

En IMS, il existe deux types de vues: les Program Communication Block (PCB) et les DBD logiques. Commençons par les PCB. Un PCB correspond à une vue particulière, "simplifiée" d'une DBD. Il peut exister plusieurs PCB pour une même DBD. Cette vue ne peut reposer que sur une seule DBD physique. **[ELMASRI, 89]** reprend les règles concernant un PCB. Un PCB est une sous-hiérarchie d'une DBD qui observe ces quatre règles:

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

Les types de segments et les champs repris dans un PCB sont dits SENSIBLES.

2.3.4. La Logical Data Base Description

Les DBD logiques constituent le deuxième type de vue en IMS. Elle est définie via une DBD dont on spécifie l'accès comme logique. Contrairement aux PCB, une base de données logique est définie sur une ou plusieurs BD physique(s) contenant des types de relations logiques, comme on l'a dit au point 2.3.2.
[DATE, 90] a détecté les différences entre les deux types de vues que sont les DBD logiques et les PCB:

- ⇒ Une base de donnée logique peut être définie sur une ou plusieurs DBD physiques, contrairement aux PCB.
- ⇒ Une vue définie sur 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. Donc, il faut voir les BD logiques comme de vraies structures physiques bien que les types de segments appartiennent à des BD physiques.

Sur la Figure 2-15, les DBD logiques sont rattachées à un PCB. On doit donc définir un PCB sur une DBD logique. 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. De plus, tout programme doit, pour pouvoir accéder à une DBD, le faire au moyen d'un PCB.

2.3.5. Le Program Specification Block(PSB)

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.

2.3.6. 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 bias du CALL. Le paragraphe 2.5 décrit en détails l'instruction d'appel "CALL", qui permet l'accès à la base de données par le programme d'application.

2.3.7. 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. On peut ainsi placer un index sur un type de segment quelconque et ainsi 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 de segment sur base de la valeur de n'importe quel(s) champ(s) de ce type de segment.
- Un index qui donne l'accès à un type de segment donné sur base de la valeur de n'importe quel(s) champ(s) d'un autre type de segment d'un niveau inférieur.

Les index secondaires sur des types de segments dépendants amènent ce qu'on appelle la restructuration de la hiérarchie. **[DATE, 90]** a décrit cette caractéristique:
- Le type de segment indexé devient le type de segment racine.
- Les types de segments ancêtres de ce type de segment deviennent les types de segments dépendants les plus à gauche dans la hiérarchie, et en ordre inverse. Cela veut dire que le type de segment parent du type de segment indexé devient son type de segment enfant le plus à gauche. On ajoute ensuite à ce dernier comme type de segment enfant son type de segment parent et ainsi de suite.
- Les types de segments dépendants du type de segment indexé apparaissent comme avant, excepté qu'ils se trouvent à la droite des types de segments introduits par la règle précédente.
- Aucun autre type de segment n'est inclus.

La Figure 2-16 montre un exemple de restructuration due à l'existence d'un index secondaire visant le type de segment Voiture issu de la Figure 2-2.

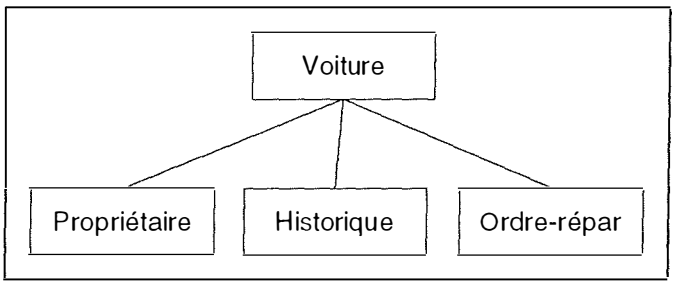


Figure 2-16: Un exemple de restructuration due à un index secondaire

2.4. Le Data Description Language (DDL)

Comme nous l'avons déjà dit, le DDL d'IMS est le DL/1. C'est donc le langage dans lequel on décrit les bases de données, ainsi que les éléments qui les composent. Le chapitre cinq abordera en détail la syntaxe du DL/1 et les apports de celui-ci au schéma que l'on peut déduire.

On peut différencier les apports possibles du DL/1 au schéma en quatre catégories:

1. Les apports des déclarations des DBD physiques.
2. Les apports des déclarations des DBD logiques.
3. Les apports des déclarations d'index secondaires.
4. Les apports des déclarations des vues (PCB).

Il est évident que le DL/1 respecte toutes les caractéristiques du modèle hiérarchique présenté dans ce chapitre. Cependant, il permet quelques entorses au modèle, comme par exemple le fait que l'on peut définir des relations *one-to-one*. Nous en reparlerons en détail dans le chapitre cinq.

Le problème avec le DL/1 est qu'il ne permet pas d'exprimer certaines structures ou contraintes comme les clés étrangères, les contraintes de coexistence, les champs décomposables, facultatifs ou multivalués. Comme nous le verrons, cela ne facilite pas le travail du rétro-ingénieur qui devra faire des recherches dans les programmes, les données et toute autre source valable pour trouver trace de ces structures ou contraintes dites non-déclaratives.

2.5. Le Data Manipulation Language (DML)

Comme nous l'avons déjà dit, les programmes d'application font appel aux instructions du DML par le biais d'un CALL. Le DML, par ce CALL, se chargera alors d'exécuter les manipulations de données demandées par le programme. La syntaxe de l'instruction CALL peut varier selon le langage hôte utilisé (COBOL,PL/1,...). Les accès à la base de données peuvent être directs mais, dans beaucoup de cas, ils s'effectuent au moyen de procédures spécifiques. Ces procédures servent d'interface entre le programme et la base de données elle-même. C'est pourquoi nous avons décidé de ne pas donner une syntaxe précise de l'accès à la base de données. Nous avons repris une instruction générale munie des paramètres principaux pouvant exister:

CALL BD GU Z-PCB RES A-SSA munie de 4 paramètres:

- Le paramètre **GU** représente la fonction que l'on désire effectuer sur la base de données. Les opérations de DML permises par le DL/1 sont les suivantes:
  - *GU* (Get Unique) renvoie le premier élément correspondant à la recherche;
  - *GN* (Get Next) renvoie le segment suivant la position courante dans la BD;
  - *GNP* (Get Next Within Parent) renvoie le segment enfant du même parent suivant la position courante;
  - *GHU* (Get Hold Unique), *GHN* (Get Hold Next) et *GHNP* (Get Hold Next Within Parent) agissent comme les trois précédentes 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.
- Le paramètre **Z-PCB** est une zone définie pour chaque PCB, qui contient des informations sur le déroulement de la requête.
- Le paramètre **RES** est une variable destinée à recevoir le résultat de la requête. C'est la zone d'entrée-sortie du programme.
- Le paramètre **A-SSA** est un SSA (Segment Search Arguments). Il peut y en avoir plusieurs. Un SSA est une variable du langage hôte qui représente un sous-ensemble[3] des champs d'un type de segment donné de la base de données. Par exemple, lors d'une insertion (CALL INSERT...), c'est dans un SSA que nous placerons les valeurs à insérer dans la nouvelle occurrence du segment. De même, lors d'une recherche, c'est dans le SSA que nous placerons les valeurs à rechercher.

[3] Un SSA peut également reprendre l'ensemble des champs d'un type de segment.

2.6. Le schéma logique IMS

Un schéma logique est un schéma normalement conforme à un modèle particulier (modèle relationnel, modèle hiérarchique,...) et qui constitue la vue qu'a le programmeur des structures de données.

Pour [**GALACSI, 89**], le schéma logique consiste à enrichir le schéma conceptuel en intégrant l'usage des données. On y décrit la structure des données perçues par le programmeur.
Nous exprimerons le schéma logique IMS dans le modèle entité-association. En effet, pour [**HAINAUT, 96a**], le modèle entité-association peut représenter aussi des structures logiques. Les éléments du modèle entité-association sont interprétés en fonction du SGBD utilisé, ici IMS. Ainsi, un type d'entité logique représentera un type de segment, un attribut logique représentera un champ, un type d'association logique représentera un type de relation parent-enfant, et ainsi de suite.

Comme nous le verrons au chapitre quatre, le schéma logique de notre méthodologie sera enrichi par rapport au modèle hiérarchique. En effet, il intégrera aussi quelques constructions non conformes à IMS, comme les clés étrangères, les attributs décomposables, facultatifs ou répétitifs.

2.7. Conclusion

Ce chapitre nous a permis de faire connaissance avec le modèle hiérarchique, et le SGBD IMS en particulier. Maintenant que nous connaissons les principales caractéristiques d'un schéma IMS, et avant de rentrer dans le vif du sujet à partir du chapitre quatre, nous allons, dans le chapitre trois, voir comment passer d'un schéma conceptuel à un schéma conforme à IMS.

CHAPITRE 3: La représentation des structures d'un schéma conceptuel en IMS

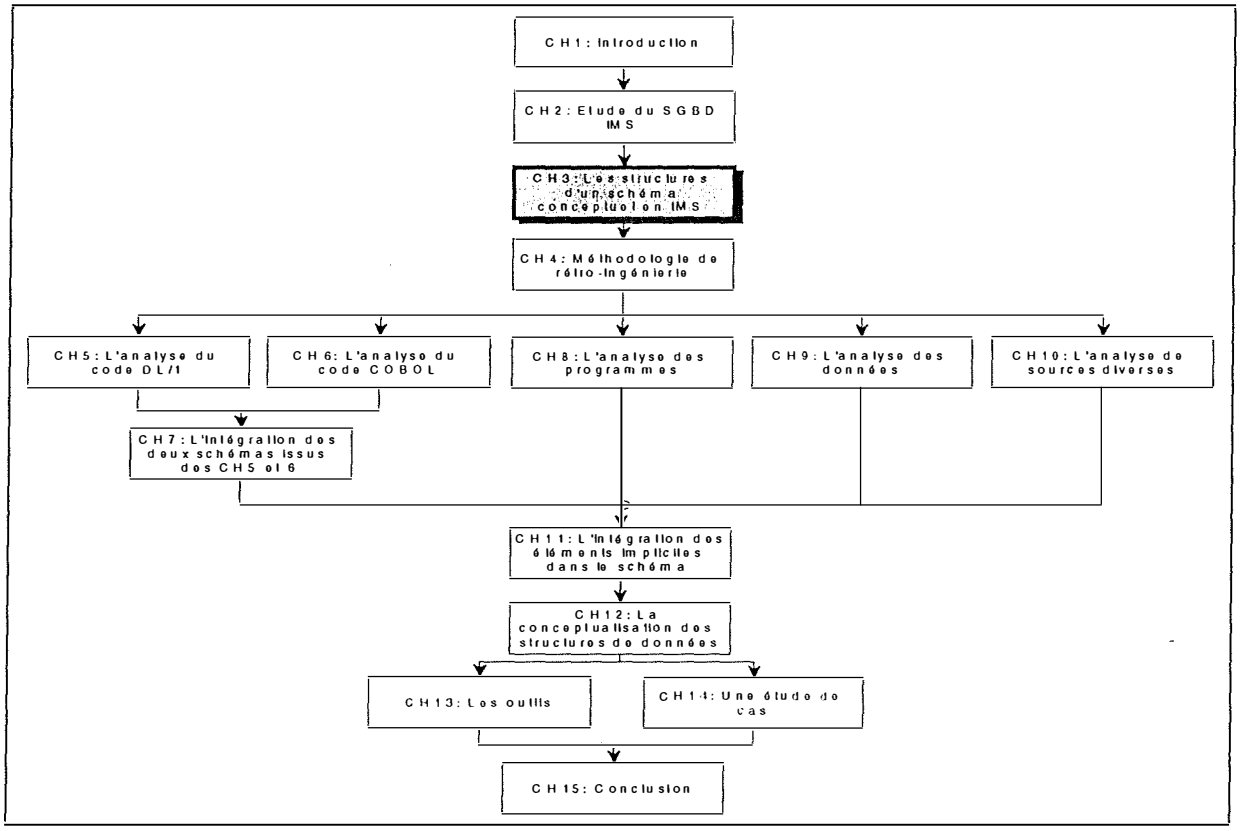


Figure 3-1: Le chapitre 3

3.1. Introduction

Le but de ce chapitre trois est de montrer quelques transformations qui permettent de modifier les éléments d'un schéma conceptuel non conformes à IMS en des constructions équivalentes conformes à IMS.

Qu'est-ce qu'une transformation?
[HAINAUT, 95b] décrit une transformation comme « un opérateur *T* qui remplace une construction source C dans un schéma *S* par une construction *C'*, ce qui donne le schéma *S'*. C'est la cible de la construction source *C* par *T*: *C'=T(C)* (*structural mapping*)».
Cette approche ne tient compte que de l'aspect syntaxique de la transformation, en spécifiant une relation inter-schéma *T*. Pour spécifier complètement une transformation, il convient aussi de spécifier une relation inter-instance *t* (la sémantique de la transformation). Ainsi, on a *c'=t(c)* (*instance mapping*). La Figure 3-2 décrit cette situation.

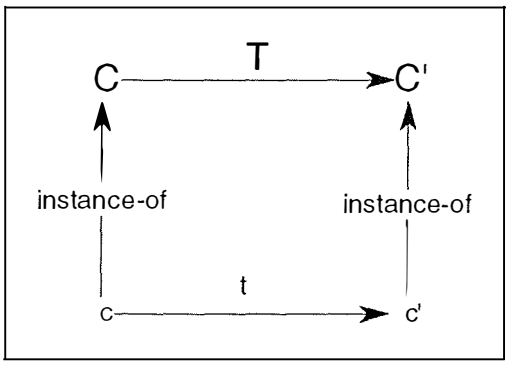


Figure 3-2: Principes de transformations

Une transformation sera donc formée du couple Σ=<T,t>.

Qu'est-ce qu'une transformation réversible?
Une transformation réversible est une transformation qui préserve la sémantique. **[HAINAUT, 95b]** distingue deux types de réversibilité:

1. La réversibilité simple
La transformation Σ1=<T1,t1> est **réversible** si et seulement si il existe une transformation Σ2=<T2,t2> telle que T2(T1(C))=C et t2(t1(c))=c. Autrement dit, Σ2 est l'inverse de Σ1mais le contraire est faux.

2. La réversibilité symétrique
La transformation Σ1=<T1,t1> est **symétriquement réversible** si et seulement si il existe une transformation Σ2=<T2,t2> telle que:
[T2(T1(C))=C] et [t2(t1(c))=c]
et [T1(T2(C'))=C'] et [t1(t2(c'))=c']

Autrement dit, Σ1 est réversible et son inverse Σ2 l'est aussi.

Avant de commencer à parler de rétro-ingénierie à partir du chapitre quatre, il nous a semblé utile d'aborder la conception logique, et plus particulièrement la transformation des structures conceptuelles non conformes à IMS par des structures qui le sont, sans perte de sémantique.

Donc, ce chapitre trois va présenter une série de transformations symétriquement réversibles qui permettent d'exprimer des structures conceptuelles non conformes à IMS de façon détournée, sans perte de sémantique. Ces transformations pourront être utilisées pour la détraduction du schéma IMS. Mais avant de décrire ces transformations, nous allons dans un premier point spécifier quelles règles un schéma IMS doit suivre, en fonction de ce que nous avons vu au chapitre deux, consacré au SGBD IMS.

3.2. Le schéma conforme IMS

Nous présentons ici un ensemble de propriétés qu'un schéma devrait posséder pour être conforme à IMS. Il ne représente rien d'autre qu'une synthèse du paragraphe 2.2 du chapitre deux qui décrivait les caractéristiques du modèle hiérarchique. Ces règles sont tirées de **[RICHARD, 95]**.

Voici les règles qu'un schéma doit respecter pour être conforme à IMS:

1. Un schéma conforme à 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 [0-N] ou [0-1] 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 [0-N] ou [0-1] pour le type d'entité racine.
4. Il y a deux classes de types d'associations: les types d'associations physiques et les types d'associations logiques; tous deux sont binaires, non récursifs.
5. Les types d'associations physiques concernent un type d'entité parent, dont le rôle aura la cardinalité [0-N] ou [0-1] et un type d'entité enfant, dont le rôle aura la cardinalité [1-1]. Un parent physique peut avoir zéro, un ou plusieurs enfant(s). Un enfant n'a qu'un seul parent physique.
6. Les types d'associations logiques concernent un type d'entité parent logique, dont le rôle aura la cardinalité [0-N] et un type d'entité enfant logique, dont le rôle aura la cardinalité [1-1]. Un type d'entité racine ne peut être enfant logique. Un enfant logique n'a qu'un seul parent logique.
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. Les attributs sont monovalués,
    atomiques,
    et obligatoires.
11. Les attributs peuvent se chevaucher.
12. Les clés étrangères et les contraintes référentielles n'existent pas.

Ces règles permettent de voir si un schéma est conforme au modèle hiérarchique IMS. Nous pouvons maintenant nous attacher à rechercher les transformations qui vont permettre de passer d'un schéma conceptuel à un schéma conforme à IMS, et vice versa.

3.3. Les transformations de base

Remarque: Dans ce paragraphe 3.3, lorsque nous illustrerons les transformations, la partie gauche des figures représentera le schéma conceptuel et la partie droite le pendant conforme à IMS.

L'ensemble de transformations qui suit n'est pas exhaustif, car il s'attache uniquement aux problèmes de représentation de certains éléments conceptuels non conformes à IMS. Pour une étude plus complète des transformations, on peut consulter **[HAINAUT, 95b]**, **[HAINAUT, 95c]** ou **[HAINAUT, 95d]**. Le lecteur peut aussi tester ces transformations dans l'atelier DB-MAIN.
Les principales conséquences des règles décrites dans le point 3.2 sont les suivantes.

3.3.1. La représentation des attributs décomposables

A. Transformer un attribut décomposable en un type d'entité (représentation par les instances)

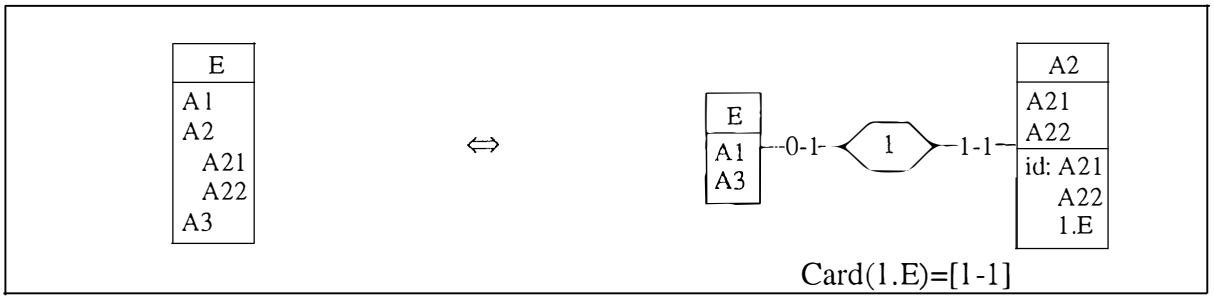


Figure 3-3: Modélisation d'un attribut décomposable par un type d'entité (instances)

On voit à la Figure 3-3, l'attribut décomposable peut être modélisé au moyen d'un type d'entité. Le nom du type d'entité ainsi créé correspond au nom de l'attribut décomposable et ses champs portent les mêmes noms que les champs qui composent l'attribut décomposable. Dans le schéma de droite, le caractère obligatoire du champ décomposable n'apparaît pas. Autrement dit, la cardinalité du rôle *1.E* est à [0-1] pour avoir un schéma conforme à IMS. Mais pour avoir un schéma de droite équivalent à celui de gauche, il faudra maintenir la cardinalité [1-1] dans les programmes d'application, d'où la contrainte Card.

B. Transformer un attribut décomposable en un type d'entité (représentation par valeur)

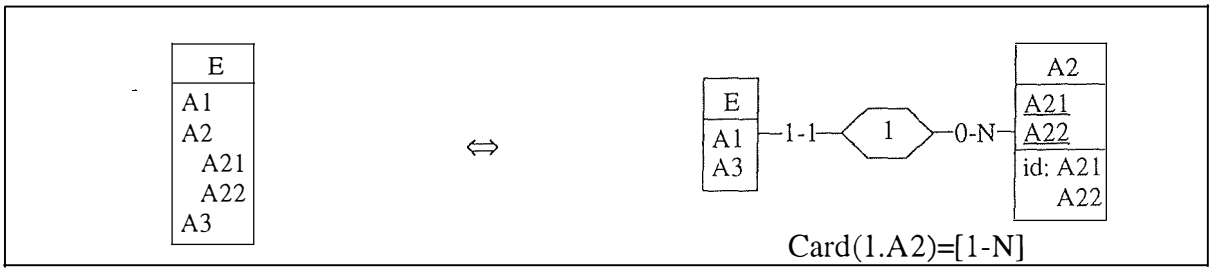


Figure 3-4: Modélisation d'un attribut décomposable par un type d'entité (valeur)

Cette transformation par valeur est tout à fait valable. Il faudra gérer dans les programmes une contrainte concernant la cardinalité minimale 1 dans [1-N], car cette cardinalité n'est pas représentable directement dans le SGBD IMS..

C. Transformer un attribut décomposable en une liste d'attributs (désagrégation)

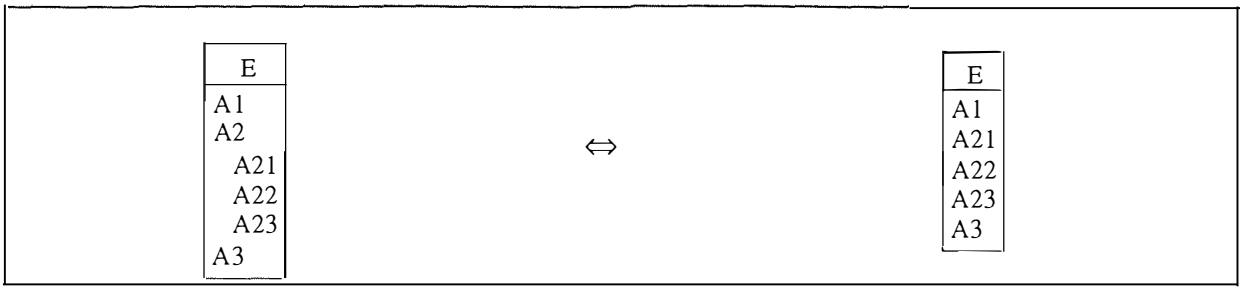


Figure 3-5: Modélisation d'un attribut décomposable par une liste d'attributs

La Figure 3-5 montre que l'attribut décomposable *A2*, non représentable en IMS, est remplacé par ses composants. Il s'agit d'une transformation symétriquement réversible.

3.3.2. La représentation des attributs multivalués

A. Transformer un attribut multivalué en un type d'entité

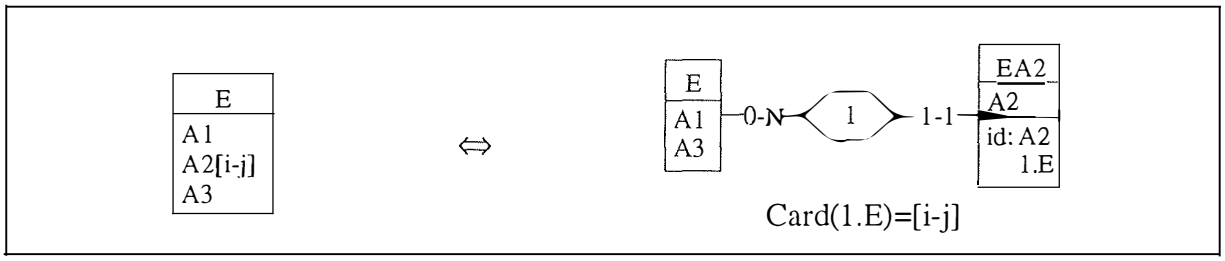


Figure 3-6: Modélisation d'un attribut multivalué par un type d'entité

L'attribut multivalué n'existe pas dans la hiérarchie IMS. La première modélisation possible d'un attribut multivalué est de le transformer en type d'entité. Il est important de remarquer que cette transformation nécessite l'expression de contraintes dans les programmes. En effet, dans le schéma de droite de la Figure 3-6, nous sommes obligés de mettre la cardinalité de *1.E* à [0-N] pour être conformes à IMS. Or, en faisant cela, nous perdons la cardinalité exacte [i-j]. Ces problèmes peuvent donc être réglés par des contraintes écrites dans les programmes (voir la contrainte Card dans la partie droite de la Figure 3-6).

B. Transformer un attribut multivalué en une série d'attributs similaires (instanciation)

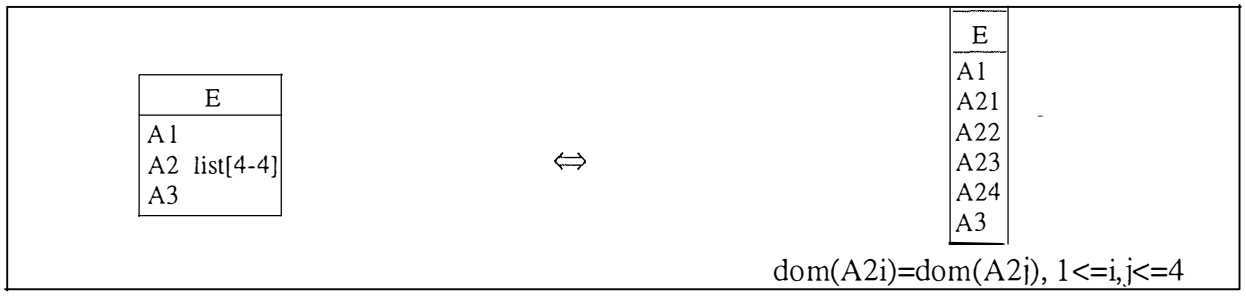


Figure 3-7: Modélisation d'un attribut multivalué en une série d'attributs similaires

La Figure 3-7 montre que l'attribut *A2* est remplacé par une série d'attributs *A21*, *A22*, *A23* et *A24*.

C. Transformer un attribut multivalué en un attribut long (concaténation)

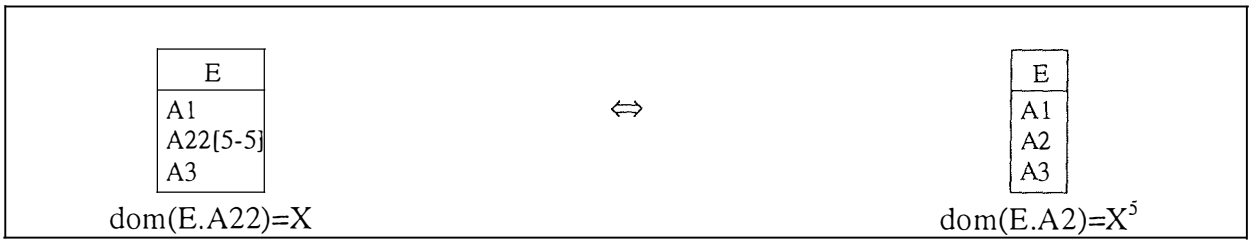


Figure 3-8: Modélisation d'un attribut multivalué en un attribut long

La Figure 3-8 montre que l'attribut multivalué *A22* est remplacé par un attribut atomique *A2* de longueur importante (voir la contrainte *dom*). Chaque valeur de *A2* est la concaténation des valeurs de l'attribut *A22*.

3.3.3. La représentation des attributs facultatifs

A. Transformer un type d'attribut facultatif par un type d'entité

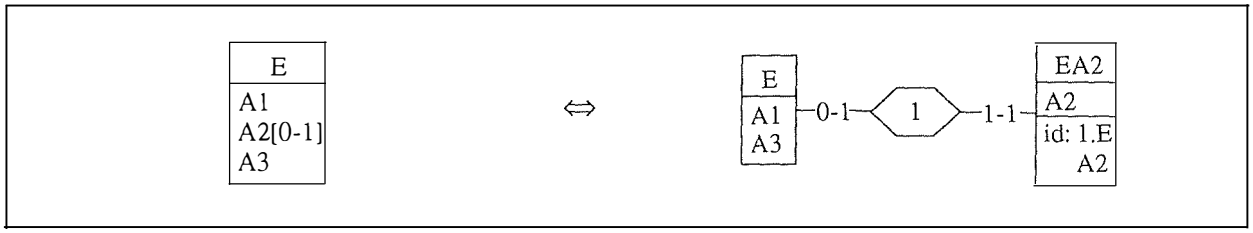


Figure 3-9: Modélisation d'un attribut facultatif par un type d'entité

L'attribut facultatif peut lui aussi être remplacé par un type d'entité.

B. Les valeurs nulles

Une autre possibilité pour représenter un attribut facultatif est l'introduction dans le domaine de l'attribut concerné d'une valeur nulle. Cette valeur nulle permet donc de modéliser l'attribut facultatif en un attribut obligatoire.

3.3.4. La représentation des types d'associations many-to-many

Le type d'association *many-to-many*, interdit en IMS, peut être modélisé de diverses manières.

A. Transformer un type d'association *many-to-many* en un type d'entité

Une première manière de représenter une relation *many-to-many* est de remplacer le type d'association par un type d'entité qui est relié aux deux premiers types d'entités par des relations *one-to-many*, une physique et une logique (Figure 3-10).

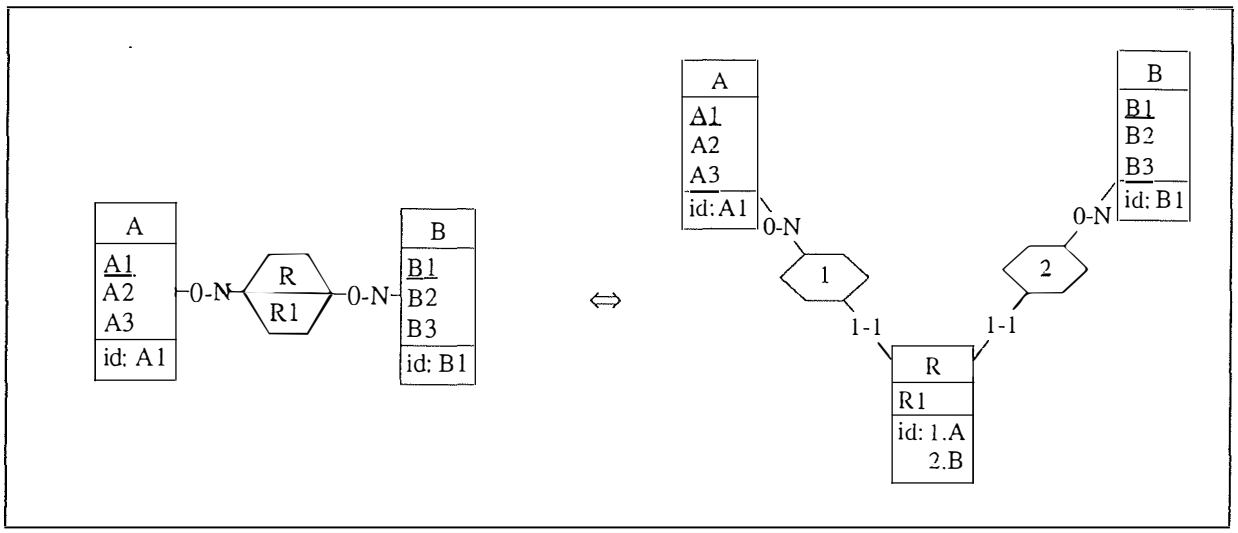


Figure 3-10: Modélisation d'un type d'association *many-to-many* par un type d'entité

B. Transformer un type d'association *many-to-many* en une clé étrangère

Variante 1: **[HAINAUT, 95a]**

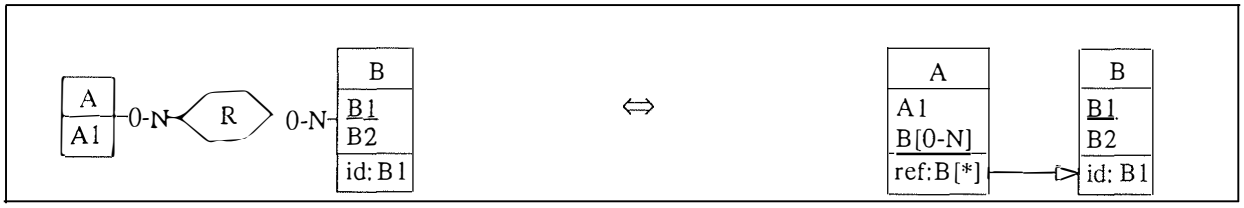


Figure 3-11: Modélisation d'un type d'association *many-to-many* par une clé étrangère (variante 1)

Cette première variante transforme un type d'association binaire *many-to-many* en une clé étrangère associée au type d'entité A. Evidemment, comme les clés étrangères ne peuvent être déclarée dans le DL/1 d'IMS, il faudra veiller à gérer ces contraintes dans les programmes d'application.

De plus, la partie droite du schéma de la Figure 3-11 n'est pas encore entièrement conforme à IMS (attribut multivalué *B* dans *A*). On devra donc encore transformer ce schéma, par exemple en appliquant la transformation vue à la Figure 3-6.

Variante 2:

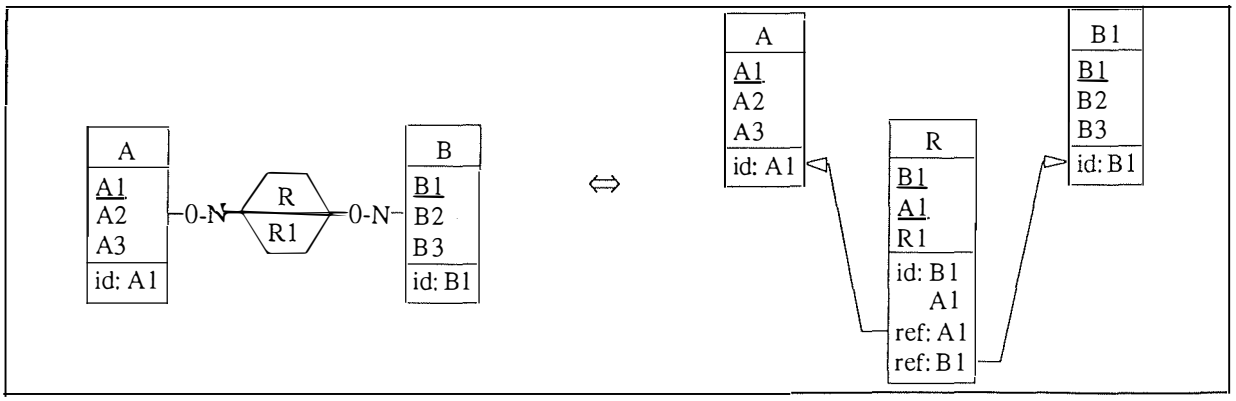


Figure 3-12: Modélisation d'un type d'association *many-to-many* par des clés étrangères (variante 2)

Il est aussi possible de représenter un type d'association *many-to-many* dans un schéma IMS en le remplaçant par un nouveau type d'entité qui sert de lien entre les deux types d'entités initiaux.
La transformation de la Figure 3-12 en inclut en fait deux:

1. On transforme le type d'association *R* en type d'entité. Il s'agit en fait de la transformation de la Figure 3-10, déjà conforme à IMS.
2. On transforme les deux types d'associations *one-to-many* en clés étrangères.

Pour cela, le nouveau type d'entité doit posséder les identifiants des deux types d'entités (qui deviennent des clés étrangères et des identifiants dans le type d'entité créé), ainsi que les attributs du type d'association (s'ils existent).
Comme on l'a déjà dit, les clés étrangères ne peuvent pas être exprimées directement en IMS, comme c'est le cas dans le modèle relationnel. On devra donc les exprimer dans les programmes d'application.

3.3.5. La représentation des types d'associations one-to-one

A. Transformer un type d'association *one-to-one* en un type d'association *one-to-many*

Il est possible de modéliser un type d'association *one-to-one* à l'aide d'un type d'association *one-to-many*. Les attributs éventuels du type d'association sont représentés dans le type d'entité enfant; ainsi ils n'existent que si l'enfant existe. Il est important de remarquer que la cardinalité [0-1] doit être maintenue dans les programmes.

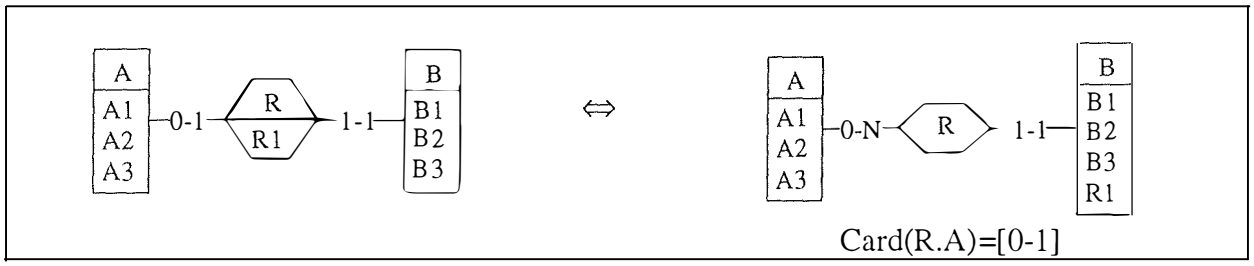


Figure 3-13: Modélisation d'un type d'association *one-to-one* par un type d'association *one-to-many*

Néanmoins, comme on l'a déjà dit, le SGBD IMS permet de déclarer des types de relations *one-to-one*. Etant donné que c'est un cas particulier, nous avons décidé de présenter une transformation possible, même si la situation de gauche de la Figure 3-13 est possible en IMS. Pour plus de détail sur cette possibilité, veuillez consulter le chapitre cinq.

B. Transformer un type d'association *one-to-one* en un type d'entité

Il est possible de modéliser le type d'association *one-to-one* à l'aide d'un seul type d'entité. Il faut cependant remarquer qu'il doit y avoir une contrainte de coexistence entre les attributs de B et les attributs du type d'association *one-to-one*. Cette contrainte fait que le schéma de droite de la Figure 3-14 doit encore être nettoyé de ses attributs facultatifs, non conformes à IMS. La contrainte de coexistence devra évidemment être exprimée dans les programmes.

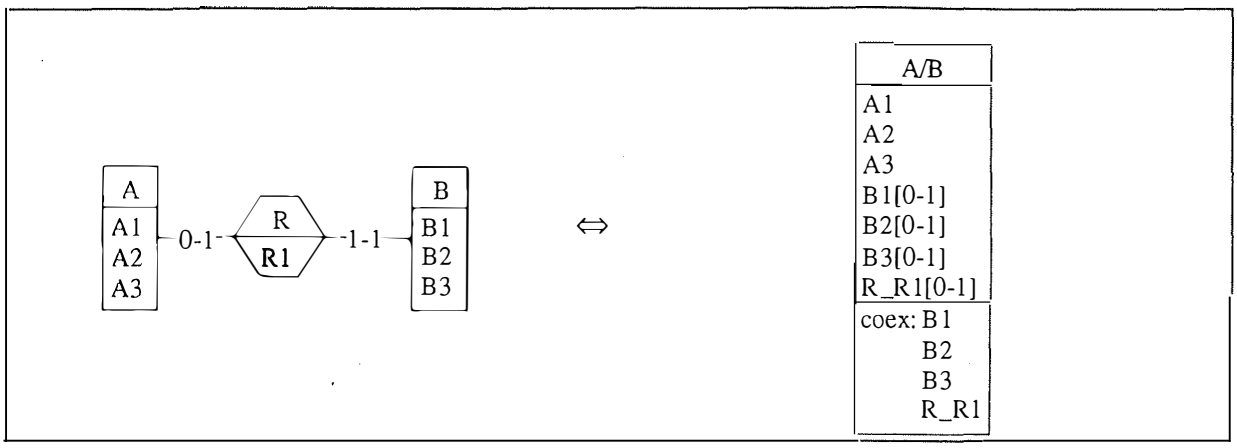


Figure 3-14 : Modélisation d'un type d'association *one-to-one* par un type d'entité

C. Transformer un type d'association *one-to-one* en une clé étrangère **[HAINAUT, 95d]**

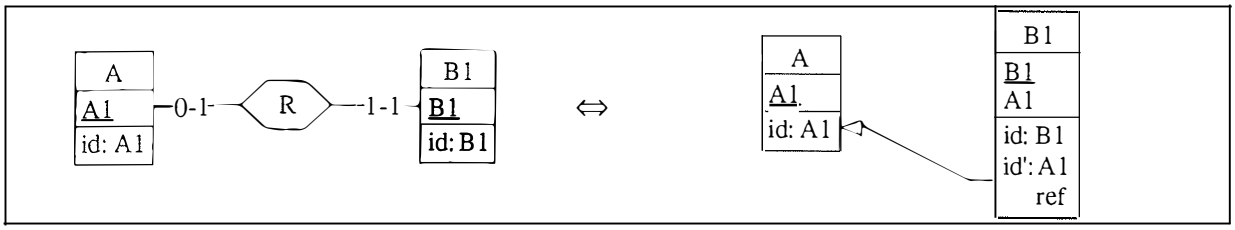


Figure 3-15: Modélisation d'un type d'association *one-to-one* par une clé étrangère

Cette transformation modélise un type d'association binaire *one-to-one* en une clé étrangère associée au type d'entité *B*. La clé étrangère ne pouvant pas être exprimée dans le SGBD IMS, il faudra l'exprimer dans les programmes.

3.3.6. La représentation des types d'associations récursifs

A. Transformer un type d'association récursif en un type d'entité

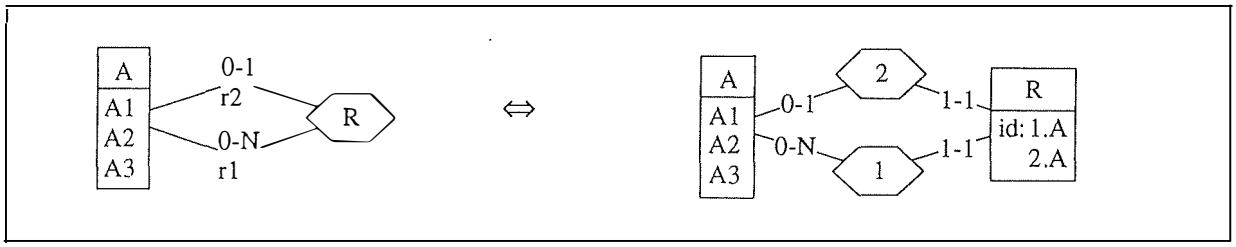


Figure 3-16 : Modélisation d'un type d'association récursif par un type d'entité

La transformation de la Figure 3-16 consiste à remplacer le type d'association récursif R en un type d'entité du même nom. Nous obtenons alors une situation conforme à IMS si nous considérons, par exemple, le type d'association *1* comme un type de relation parent-enfant logique et le type d'association *2* comme un type de relation parent-enfant physique.

B. Transformer un type d'association récursif en un attribut (clé étrangère)

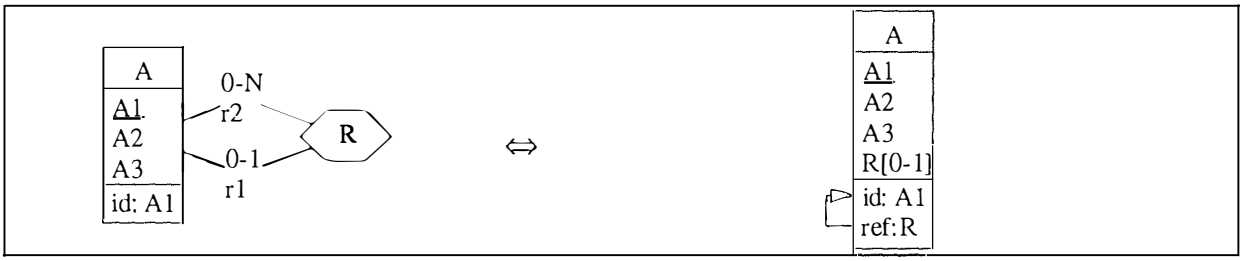


Figure 3-17: Modélisation d'un type d'association récursif par un attribut

La Figure 3-17 montre la transformation du type d'association récursif *R* en un attribut du type d'entité *A* auquel il est rattaché. La partie droite du schéma n'est pas encore totalement conforme à IMS. En effet, l'attribut *R* est facultatif, ce qu'il est impossible de représenter en IMS. Il faut donc appliquer la transformation vue à la Figure 3-9. On peut aussi remarquer la présence d'une clé étrangère de *R* vers *A1*: celle-ci devra être gérée dans les programmes d'application.

3.3.7. La représentation des types d'associations n-aires

A. Transformer un type d'association n-aire par décomposition

Variante 1: Un des rôles du type d'association n-aire a la cardinalité [1-1]

Dans ce cas, la transformation du type d'association n-aire de la Figure 3-18 en un type d'entité donnera deux types d'associations *one-to-many* et un *one-to-one*. Il est alors possible de réunir les types d'entités *R* et *A*. Cela nous donne alors la partie droite de la Figure 3-18. Cette transformation est conforme à IMS grâce à l'existence des relations logiques.

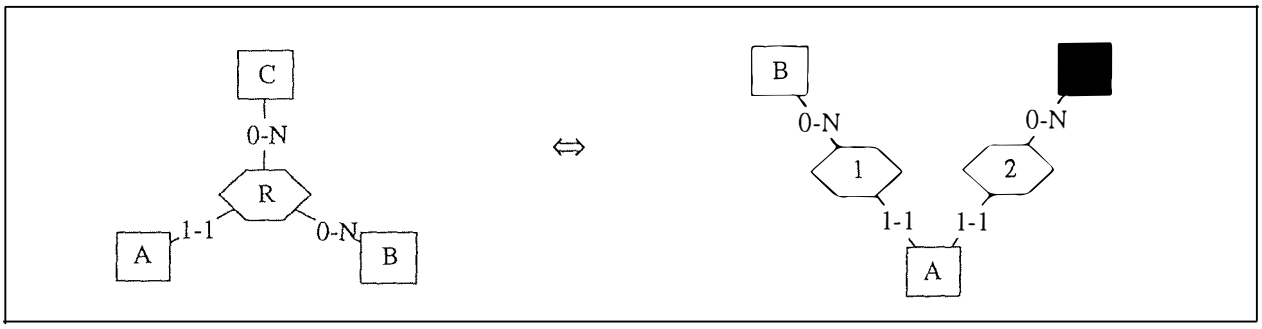


Figure 3-18: Modélisation d'un type d'association n-aire par décomposition (variante 1)

Variante 2: Un des rôles du type d'association n-aire a la cardinalité [0-1]

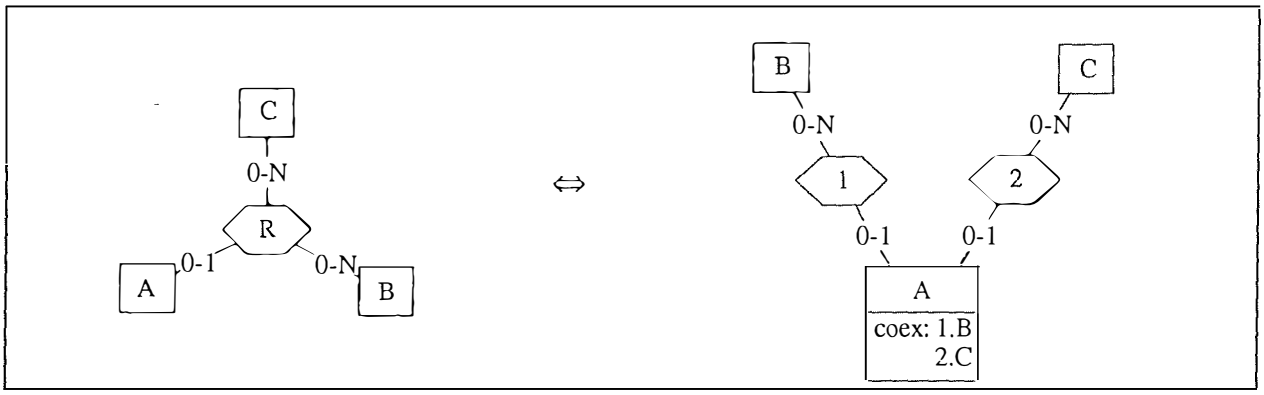


Figure 3-19: Modélisation d'un type d'association n-aire par décomposition (variante 2)

La transformation de la Figure 3-19 s'effectue en deux étapes. On transforme d'abord le type d'association n-aire en type d'entité, ce qui donne deux types d'associations *one-to-many* et un *one-to-one ([0-1]-[1-1])*. Cela nous permet ensuite de réunir les types d'entités *R* et *A* et ainsi d'obtenir le schéma de droite de la Figure 3-19. Ce schéma n'est pas encore conforme au SGBD IMS. Il faut encore transformer les deux types d'associations en les transformant en types d'entités.
La Figure 3-20 ci-dessous donne le résultat de la transformation des types d'associations de la partie droite de la Figure 3-19 en types d'entités. On obtient alors un schéma conforme à IMS.

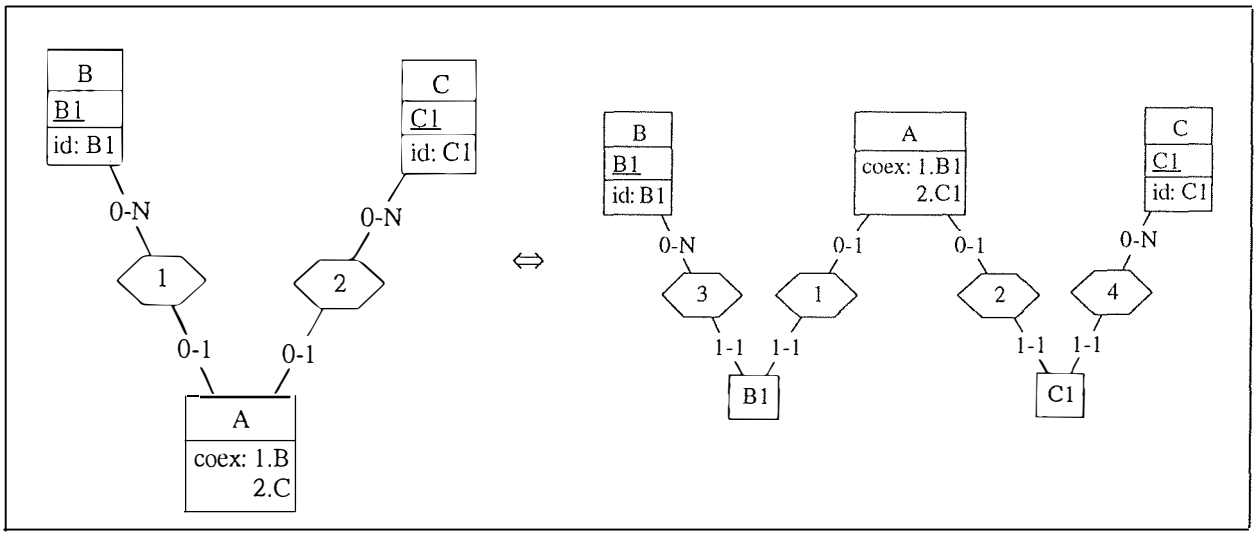


Figure 3-20: Modélisation de types d'associations en clés étrangères

Variante 3: tous les rôles ont la cardinalité [0-N]

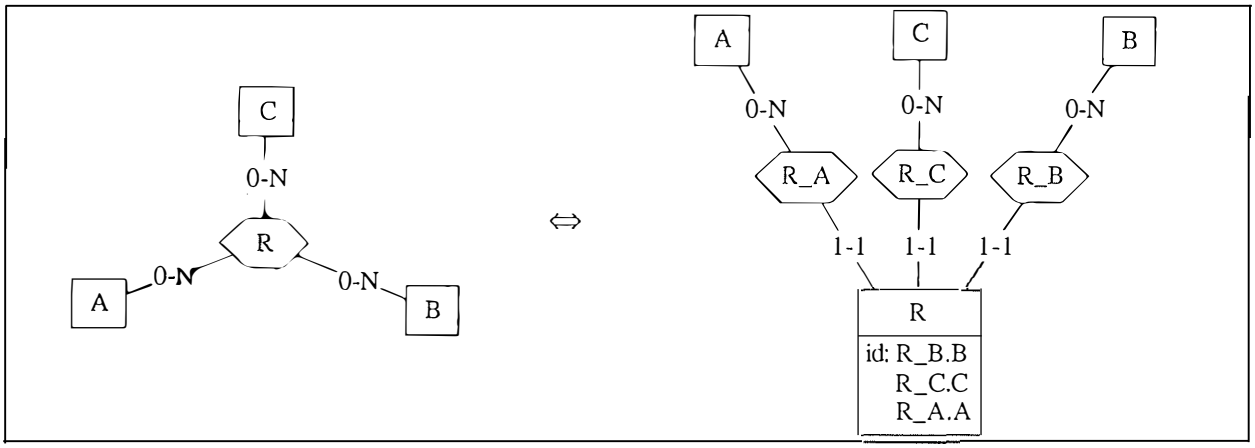


Figure 3-21: Modélisation d'un type d'association n-aire par décomposition (variante 3)

La transformation présentée à la Figure 3-21 n'est bien sûr pas encore conforme à IMS. Le type d'entité R a trois parents. Il suffira d'appliquer la transformation de la Figure 3-22 ou encore celle de la Figure 3-23 ci-après.

3.3.8. La représentation des types d'entités avec plus de deux parents

A. Modéliser par la transformation d'un des types d'associations en type d'entité

On choisit aléatoirement un des types d'associations de la partie gauche de la Figure 3-22 et on le transforme en type d'entité. On obtient alors le schéma de la partie droite de la figure, qui n'est pas encore conforme à IMS en raison de la présence d'une association *one-to-one*. Il

suffit alors de transformer ce type d'association *one-to-one* en *one-to-many*, comme vu précédemment.

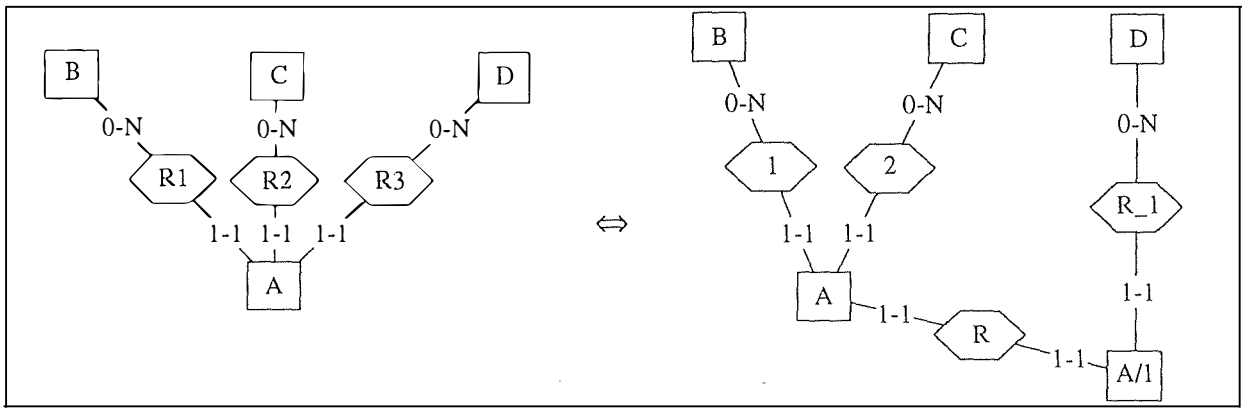


Figure 3-22: Modélisation des types d'entités avec plus de deux parents par la transformation d'un des types d'associations en type d'entité

B. Modéliser au moyen de clés étrangères

Il est aussi possible de modéliser cette situation en remplaçant plusieurs relations parent-enfant par des clés étrangères. Il faudra évidemment gérer les contraintes référentielles au niveau des programmes. On peut aussi se contenter d'une seule clé étrangère. Nous serions alors obligés d'implémenter un type de relation logique pour être conforme au SGBD IMS.

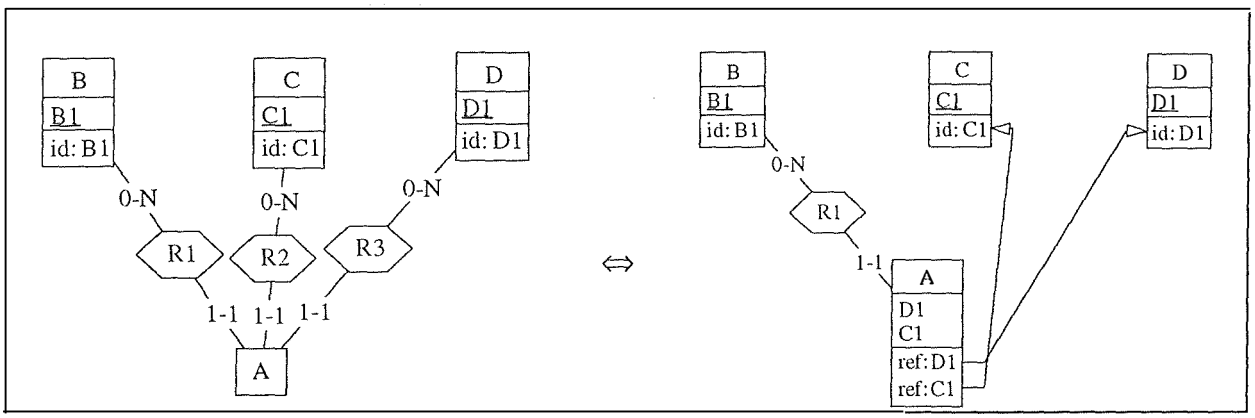


Figure 3-23: Modélisation des types d'entités avec plus de deux parents par des clés étrangères

3.4. Conclusion

Ce chapitre trois nous a permis de considérer un ensemble de transformations symétriquement réversibles pour représenter des constructions conceptuelles en vue de les rendre conforme à IMS. Ces transformations nous serons très utiles dans le cadre de la phase de conceptualisation des structures de données du processus de rétro-ingénierie. Il est temps maintenant de rentrer dans le vif du sujet, à savoir la rétro-ingénierie.

PARTIE 3: LA METHODOLOGIE

CHAPITRE 4: Une Méthodologie de rétro-ingénierie

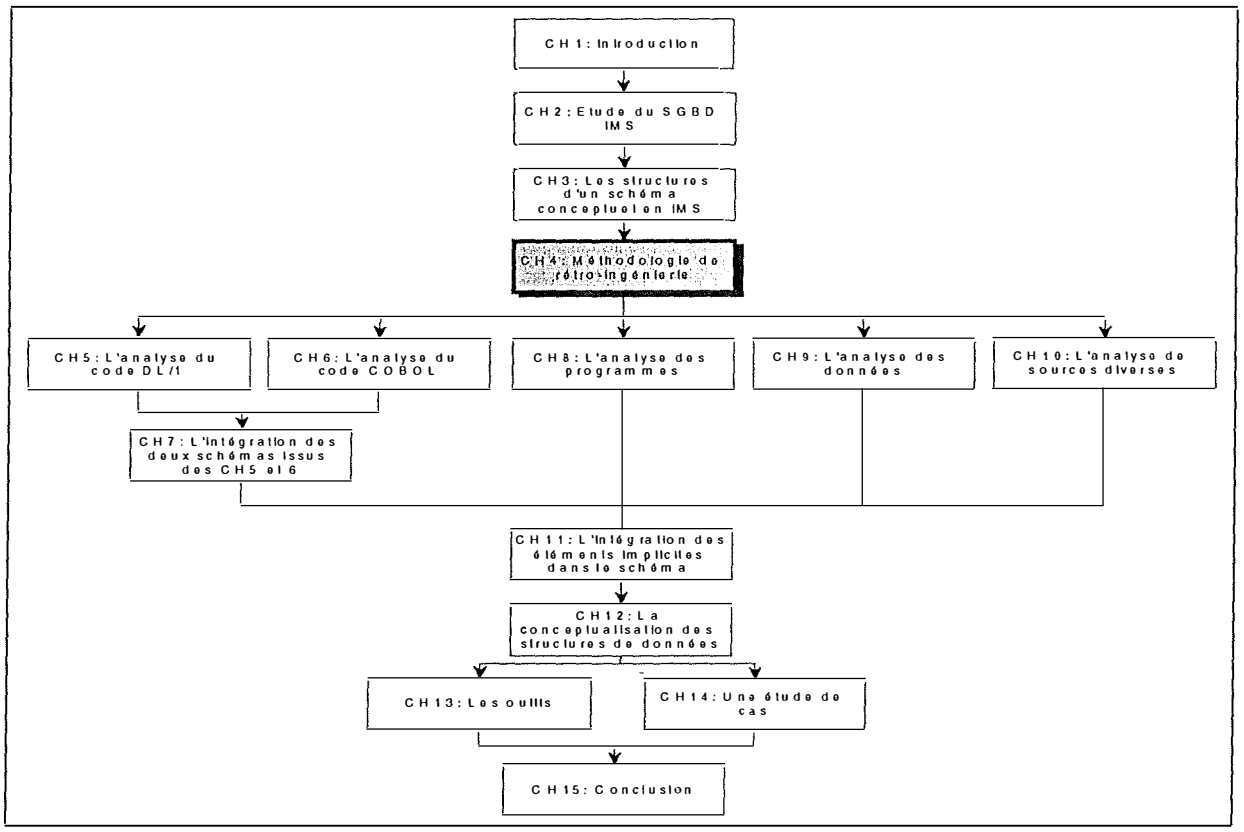


Figure 4-1: Le chapitre 4

4.1. Introduction

Le but de ce chapitre quatre est de décrire une méthodologie de rétro-ingénierie de bases de données IMS. La rétro-ingénierie n'est certainement pas un domaine propice à la définition de méthodes strictes et inflexibles, à suivre quoi qu'il advienne. Les deux méthodes présentées dans ce chapitre, aussi bien la méthode générale que celle adaptée à IMS, sont donc très flexibles. L'aspect séquentiel inhérent à la présentation de celles-ci doit être relativisé. En effet, il ne sera pas rare d'effectuer des retours en arrière, des recherches en parallèles pour trouver, par exemple, des structures implicites.

Dans ce chapitre, nous commencerons par décrire la méthode générale de rétro-ingénierie proposée par **[HAINAUT, 95a]**. Dans un deuxième point, nous présenterons la méthodologie générale adaptée au cas d'IMS.

4.2. La méthodologie générale

Ce paragraphe 4.2 aborde la méthodologie générale d'un processus de rétro-ingénierie proposée par **[HAINAUT, 95a]**. La Figure 4-2 présente cette méthode générale.

Comme on peut le voir, il y a quatre schémas représentés dans la Figure 4-2:

- Le schéma logique enrichi conforme au SGBD,
- Le schéma de travail,
- Le schéma conceptuel de base et
- Le schéma conceptuel normalisé.

Ces quatre schémas peuvent être mis en relation avec les modèles physique, logique et conceptuel que nous allons décrire brièvement.

⇒ *Le modèle physique*

Le modèle physique est le modèle adapté à un SGBD particulier, ici IMS. Ce modèle inclut des caractéristiques physiques: fichier, index, noms réels, type et longueur physique réels. Ainsi, dans un schéma physique IMS, la notion de *collection* peut être considérée comme une abstraction des ensembles de données (DATASET) d'IMS. La notion de clé d'accès, elle, spécifie le mécanisme d'accès direct d'IMS, à savoir les index.

⇒ *Le modèle logique*

Le modèle logique est la description des structures de données perçues par le programmeur. Les éléments du modèle entité-association sont interprétés en fonction du SGBD utilisé, ici IMS. Ainsi, un type d'entité logique représentera un type de segment, un attribut logique représentera un champ, un type d'association représentera un type de relation parent-enfant physique ou logique, et ainsi de suite.

De nouvelles constructions apparaissent aussi à ce niveau, comme les clés étrangères, les attributs multivalués et décomposables. Ce modèle ne contient aucune caractéristique physique.

⇒ *Le modèle conceptuel*

Pour **[BODART, 93]**, le modèle conceptuel « a pour but d'exprimer la sémantique à l'exclusion des problèmes de représentation physique des données (codage interne), d'accès aux données et d'organisation de stockage ».

Le *processus de rétro-ingénierie* se divise en deux parties, l'extraction des structures de données et la conceptualisation des structures de données.

L'*extraction des structures de données* a pour but de dégager toute l'information possible à partir du schéma DMS-DDL, du schéma physique, des données et des programmes. L'intégration de toute cette information donne un schéma logique enrichi conforme au SGBD.

La *conceptualisation des structures de données* a pour but de nettoyer le schéma logique enrichi conforme au SGBD de toutes ses caractéristiques physiques, de le détraduire et de le désoptimiser. Lorsque cela est terminé, il convient de donner au schéma des caractéristiques

conceptuelles acceptables afin que le schéma final représente au mieux la vue conceptuelle qu'avait le concepteur avant de commencer son travail.

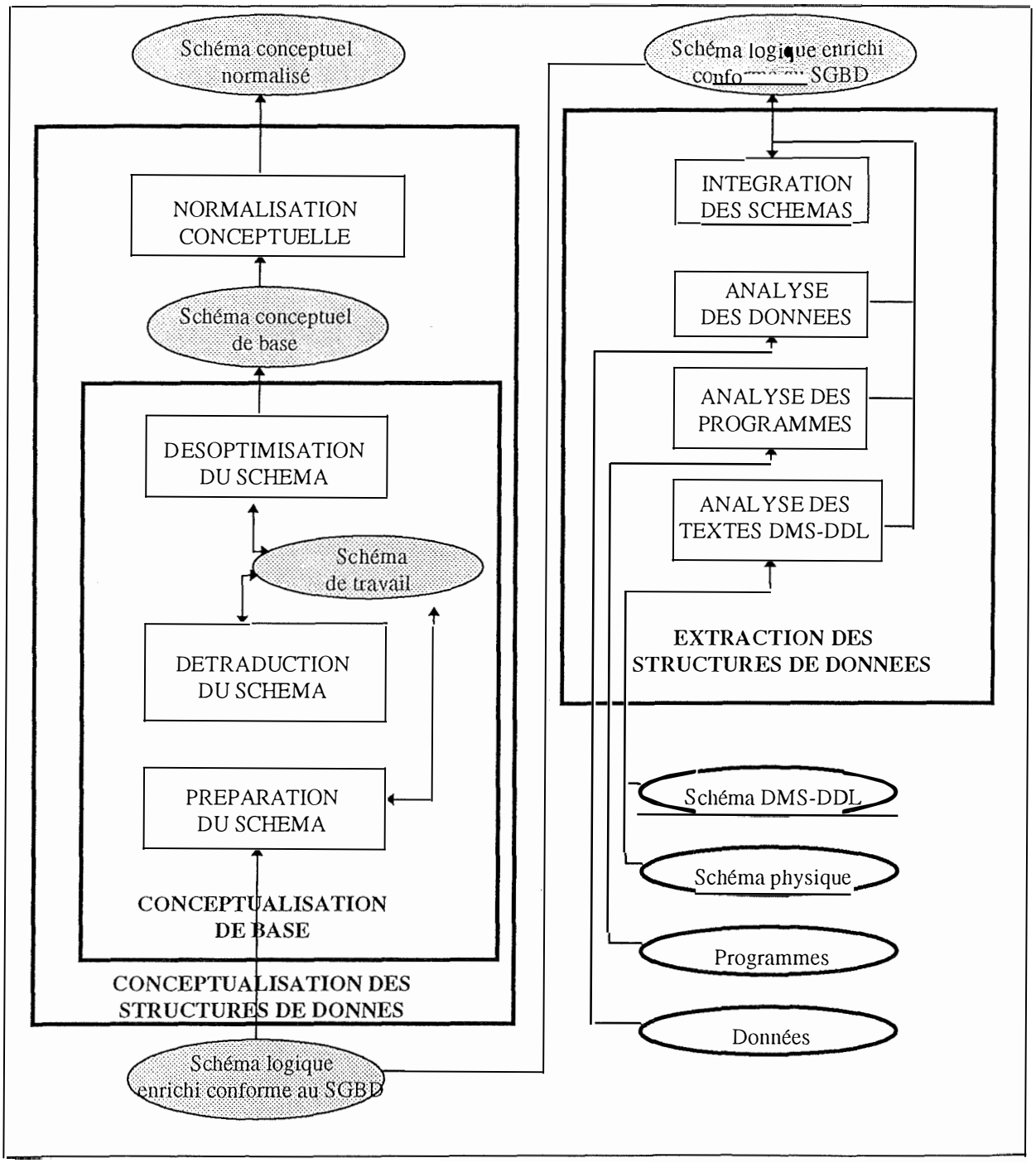


Figure 4-2: La méthodologie générale de rétro-ingénierie

Nous allons maintenant nous intéresser de plus près à ces deux phases importantes du processus, que sont l'extraction des structures de données et la conceptualisation des structures de données (entourées de cadres en gras dans la Figure 4-2).

4.2.1. L'extraction des structures de données

Le but de l'extraction des structures de données est d'obtenir le schéma logique enrichi conforme au SGBD cible. La base de départ du processus d'extraction est constituée du schéma DMS-DLL, du schéma physique, des programmes et des données.
Cinq problèmes ont été mis en évidence par **[HAINAUT, 96b]** dans cette phase:

* La dissimulation de structures *(structure hiding)*

  La dissimulation de structures s'applique à des structures de données qui peuvent être implémentées dans le SGBD utilisé **mais** qui ne sont pas implémentées telles quelles. Elles sont remplacées par des contraintes ou des structures plus générales et cela dans un souci de simplicité, de généricité ou d'efficacité.

  Par exemple, alors qu'il est possible de déclarer un type de relation *one-to-one* ([0-1]-[1-1]) en IMS, le concepteur déclare un type de relation *one-to-many* à la place et écrit une contrainte dans les programmes pour maintenir la cardinalité [0-1].

* Les propriétés génériques

  Certains systèmes de gestion de bases de données offrent un langage de programmation qui permet d'exprimer librement une large variété de contraintes sur les données. Il devient donc malaisé de trouver 'facilement' ces contraintes, puisque chaque programmeur est susceptible de le faire différemment.

  Par exemple, dans un SGBD offrant cette fonctionnalité, les contraintes d'intégrité référentielle pourraient être encodées de diverses façons, et leur découverte pourrait donc s'avérer plus complexe par rapport à des SGBD où une clé étrangère ne peut être exprimée que d'une seule façon.

* Les structures non-déclaratives

  Les structures non-déclaratives sont des contraintes ou des structures qui ne peuvent pas être implémentées dans le SGBD utilisé. Ces contraintes ou structures sont dès lors représentées et vérifiées dans les programmes d'application.

  Par exemple, déclarer un champ décomposable est impossible en IMS. On peut néanmoins découvrir le caractère décomposable de ce champ déclaré comme atomique dans les programmes, par l'assignation du contenu de ce champ dans une variable qui, elle, est décomposable.

  Un autre exemple est que dans le cas d'une base de données IMS, il est impossible d'implémenter une relation *many-to-many*. Une relation *one-to-many* peut donc s'avérer être une relation *many-to-many* par une analyse des programmes.

  Enfin, la déclaration d'une clé étrangère étant également impossible, elle devra se faire par une procédure au niveau des programmes d'application.

* Les propriétés environnementales

  Dans certaines situations, l'environnement du système garantit que les données présentes dans la base de données satisfont à une propriété donnée. C'est pourquoi les concepteurs trouvent inutile de traduire cette propriété dans les structures de données ou les programmes. Le problème vient donc du fait qu'on ne pourra pas retrouver cette propriété par une analyse des programmes.

* La perte de spécifications

  La perte de spécification a lieu lorsque des contraintes ou des structures ne sont pas implémentées dans le SGBD ni dans les programmes, intentionnellement ou non. Elles peuvent être retrouvées dans la culture d'entreprise ou dans les structures organisationnelles.

  Par exemple, le fait qu'une commande ne peut être passée le dimanche, n'est ni mentionné, ni contrôlé dans les programmes. Le fait que l'entreprise est fermée le dimanche impose cette contrainte.

Ces cinq problèmes mettent en évidence la difficulté de retrouver **toute** l'information à partir des différentes sources. Cependant, une grande majorité des informations peut être retrouvée par les analyses qui constituent la phase d'extraction des structures de données et qui sont décrites ci-dessous.

L'analyse des textes DMS-DDL (Data Management System - Data Definition Language) et du schéma physique est réalisée en premier lieu car elle apporte le plus grand nombre d'informations. Elle consiste à analyser les codes de définition des bases de données. De cette analyse, on peut obtenir un premier squelette du schéma. Les autres sources d'information apportent proportionnellement moins d'informations. Elles permettent cependant de confirmer ou d'affiner le premier schéma obtenu.

L'analyse des programmes est également importante, surtout dans le cas de SGBD assez anciens qui ne permettent pas de déclarer grand-chose explicitement. Cette analyse est également beaucoup plus complexe. En effet, l'information recherchée peut se trouver partout dans les programmes. De plus, le nombre de programmes d'application peut être très imposant. Les apports principaux de cette phase sont le détail et la confirmation de certaines structures et/ou contraintes découvertes lors de la phase précédente, ainsi que la découverte de nouvelles structures.

L'analyse des données consiste à examiner le contenu de la base de données pour découvrir de nouvelles structures et/ou de nouvelles contraintes. En effet, certaines de ces structures ne seront découvertes que lors de cette analyse. Lors de cette phase, il est également possible de confirmer ou d'infirmer la présence de structures découvertes auparavant.

L'intégration des schémas a pour but d'intégrer dans un schéma sans pertes d'informations les différents éléments découverts dans les phases précédentes. Cette phase d'intégration des schémas a pour résultat le *schéma logique enrichi conforme au SGBD.*

4.2.2. La conceptualisation des structures de données

Cette phase du processus se base sur le résultat de l'extraction des structures de données, à savoir le *schéma logique enrichi conforme au SGBD.* Le résultat doit être un *schéma conceptuel normalisé.* Pour ce faire, il convient de détecter et de transformer les structures non conceptuelles, les redondances, les structures résultant d'une optimisation et les structures propres au SGBD cible. Ce processus se base donc fortement sur les techniques de transformations des schémas. Cette phase se décompose en deux sous-phases[1]:

[1]Les définitions suivantes sont tirées de **[RICHARD, 95]**.

La conceptualisation de base, qui a pour but principal d'extraire les concepts sémantiques sous-jacents au schéma source. Elle se divise en trois phases: la préparation, la détraduction et la désoptimisation du schéma. Elle produit comme résultat un schéma conceptuel de base (brut). Les trois phases peuvent être appliquées en parallèle. Le schéma intermédiaire porte donc le nom de schéma de travail, puisque ce schéma est susceptible

La préparation du schéma consiste à nettoyer les différents noms appartenant au schéma et qui pourraient être influencés par des éléments physiques.

La détraduction du schéma consiste à retrouver les structures conceptuelles desquelles le concepteur a dérivé les structures logiques.

La désoptimisation du schéma a pour but de retrouver les structures provenant d'une optimisation et à les éliminer si cela est nécessaire.

La normalisation conceptuelle améliore le *schéma conceptuel de base* pour lui donner les qualités d'un *schéma conceptuel normalisé.* Ces qualités sont l'expressivité, la généricité, la simplicité, la minimalité, la lisibilité et l'extensibilité.

Remarque: RECHERCHE DE LA STRATEGIE DE CONCEPTION

La phase de recherche de la stratégie de conception consiste à retrouver la stratégie de conception que le concepteur avait adoptée. Si cette recherche ne s'effectue pas, nous obtiendrons un schéma conceptuel normalisé mais qui risque d'être très éloigné du schéma conceptuel du concepteur. Cette recherche n'apparaît pas telle quelle dans le schéma de la Figure 4-2, car elle doit être faite à tous les niveaux de la méthode, dès qu'on est confronté à un choix de conceptualisation.

4.3. La méthodologie adaptée à IMS

Ce paragraphe 4.3 est vraiment le centre de ce mémoire. Nous allons y proposer une méthodologie de rétro-ingénierie adaptée à IMS. Cette méthode se basera sur la méthode générale vue au point 4.2., mais avec quelques changements.
Ce paragraphe permettra d'avoir une vue d'ensemble de la méthode proposée, sans entrer dans les détails de chaque étape.
Le développement de chaque étape de la méthode sera fait dans les chapitres cinq à douze.
Les deux points qui suivent présentent les deux phases de la méthode adaptée à IMS.

4.3.1. L'extraction des structures de données

La phase d'extraction des structures de données présentée ici diffère quelque peu de la phase d'extraction « générale » vue auparavant. En effet, vu que nous allons analyser deux codes DDL (Data Description Language) différents (DL/1 et COBOL), il y aura une phase d'intégration supplémentaire, qui permettra d'intégrer les deux schémas issus de l'analyse des deux codes. Cette intégration supplémentaire est absente de la méthode générale. Dans certaines applications de bases de données IMS, le code DDL COBOL peut ne pas exister. Dans ce cas, nous nous retrouvons dans le cas de la méthode générale.

4.3.1.1. L'analyse des structures explicites

A. L'analyse du code DDL (DL/1)
L'analyse du code DL/1 consiste à analyser le langage de description des données du SGBD IMS dans le but de découvrir tous les éléments susceptibles de venir enrichir le premier schéma.

B. L'analyse du code DDL (COBOL)
Il est possible que le code DL/1 soit incomplet quant à la description des champs d'un type de segment défini dans un code DL/1. Dans ce cas, la description complète des champs de ce type de segment est faite dans un fichier (que nous appellerons COPYBOOK), qui est un fichier ne contenant que des déclarations de variables COBOL[2]. L'analyse des COPYBOOKS est donc fort semblable à celle du DL/1. A la fin de cette analyse, nous obtenons un deuxième schéma.

C. L'intégration des deux schémas issus des deux premières étapes
Les étapes d'analyses du code DL/1 et des COPYBOOKS correspondants aux types de segments définis dans le DL/1 analysé fournissent deux schémas distincts, mais qui concernent néanmoins les mêmes types de segments. Il faut donc intégrer ces deux schémas en un seul, sans rien perdre de l'information présente dans les deux schémas mais en évitant les doublons.

[2] En fait, comme nous le verrons dans le chapitre cinq, le DL/1 peut très bien ne contenir, pour un type de segment donné, que la description des champs identifiants ou sur lesquels il y a un index, autrement dit des champs qui ont des caractéristiques que le SGBD doit connaître pour pouvoir les implémenter. Si, dans le DL/1, on observe des absences de champs au vu de la longueur du type de segment, la description COMPLETE des champs de ce type de segment se retrouvera alors dans un COPYBOOK, un fichier qui contient les déclarations COBOL des variables correspondant aux champs du type de segment. Si l'on somme les longueurs des variables du COPYBOOK, on obtiendra la longueur du type de segment correspondant défini dans le DL/1. Ce COPYBOOK sera utilisé par les programmes d'application via l'instruction COPY, qui fera du fichier une variable COBOL.

4.3.1.2. L'analyse des structures implicites

Les structures implicites peuvent être découvertes de huit façons différentes, comme le propose **[HAINAUT, 96b]**:

A. L'analyse des programmes
Comme nous l'avons déjà dit, le SGBD IMS ne permet de déclarer qu'un minimum de structures et de contraintes au niveau du DL/1. C'est pourquoi une analyse des programmes en langage hôte est primordiale pour trouver des structures non-déclaratives, qu'il est impossible de déclarer dans le SGBD.

B. L'analyse des données
L'analyse des données est, avec l'analyse des programmes, un des moyens les plus pertinents pour trouver des indices de structures implicites.
Ainsi, on peut confirmer l'existence de certaines contraintes en réalisant de petits programmes consultant les données. Certaines confirmations peuvent également être faites sur la base d'une compréhension de la signification du contenu de la base de données. Enfin, il est aussi possible de générer automatiquement des programmes qui analysent les données.

C. La connaissance du domaine
Une autre façon de découvrir des structures implicites est de posséder une certaine connaissance du domaine d'application. Pour acquérir cette connaissance, les utilisateurs actuels et éventuellement les anciens peuvent apporter à l'analyste des informations capitales sur les fonctions du système.

D. La documentation existante
Dans la plupart des projets de rétro-ingénierie, une documentation est disponible. Cependant, cette documentation est souvent incomplète ou même erronée. Malgré cela, elle peut être une source précieuse d'information.

E. Les écrans et les rapports
Les écrans et les rapports peuvent être considérés comme des vues dérivées des données. L'affichage des données en output, des titres et des commentaires peuvent fournir de l'information essentielle au sujet des données.

F. L'exécution des programmes
L'exécution des programmes combinée avec l'analyse des données fournit un moyen d'examen puissant pour détecter les structures et les propriétés des données.

G. L'analyse des noms
Parfois, le nom d'une variable ou d'un champ suggère la signification de celle(celui)-ci.

H. Les constructions d'accès
Certaines constructions techniques (clé d'accès, champ anormalement long,...) peuvent fournir des indices concernant certaines structures et contraintes logiques.

4.3.1.3. La détermination des éléments à insérer au schéma et l'intégration finale

Le but de cette phase d'intégration est de déterminer, parmi toutes les structures et les contraintes découvertes lors de l'analyse des structures implicites, celles qui doivent être

insérées au schéma. Cette phase est une concertation entre le rétro-ingénieur et un expert qui possède une connaissance approfondie de l'application. Lorsque les décisions sont connues, il convient d'ajouter les structures et les contraintes que l'expert a validées au schéma intégré. Cela nous donne le *schéma logique enrichi conforme au SGBD.*

4.3.2. La conceptualisation des structures de données

Cette phase de conceptualisation est identique à celle de la méthode générale. Nous exposerons les différentes étapes de chaque sous-phase en se référant aux caractéristiques du SGBD IMS et à ce que cela implique.

4.3.2.1. La conceptualisation de base

A. La préparation du schéma

Cette phase consiste à nettoyer le schéma de ses caractéristiques physiques et à donner des noms significatifs aux éléments constituant le schéma.

B. La détraduction du schéma

Nous exposerons dans cette partie comment l'on peut repérer les structures propres à IMS et les traduire en structures conceptuelles.

C. La désoptimisation du schéma

Le concepteur de la base de données peut avoir introduit des structures particulières et ce à des fins d'optimisation du temps de réponse,... Dans cette phase nous allons repérer ces structures et les transformer ou les supprimer.

4.3.2.2. La normalisation conceptuelle

Cette phase consiste à faire subir au schéma certaines modifications pour qu'il devienne un schéma vraiment conceptuel.

4.4. Conclusion

Le point 4.3 nous a permis d'énumérer les grandes étapes d'un processus de rétro-ingénierie de bases de données IMS. Il convient maintenant d'expliciter chaque étape de ce processus. Les chapitres suivants aborderont donc chaque étape:

Chapitre cinq: L'analyse du code DDL(DL/1)
Chapitre six: L'analyse du code DDL(COBOL)
Chapitre sept: Le processus d'intégration des résultats issus des deux premières étapes
Chapitre huit: L'analyse des programmes
Chapitre neuf: L'analyse des données

Nous aborderons très brièvement les points suivants dans le chapitre dix, qui sont aussi des sources à ne pas négliger dans un processus de rétro-ingénierie, mais que nous avons moins étudié:

La connaissance du domaine
La documentation existante
Les écrans et les rapports
L'exécution des programmes
L'analyse des noms
Les constructions d'accès

Le chapitre onze abordera l'intégration des éléments implicites dans le schéma. Cette intégration ne sera pas immédiate, car des conflits peuvent apparaître entre plusieurs sources, concernant le même élément. Il faudra donc prendre des décisions à propos des structures ou contraintes à insérer dans le schéma.

Enfin, le chapitre douze abordera successivement la conceptualisation de base et la normalisation conceptuelle.

CHAPITRE 5: L'analyse du code DL/1 (DDL)

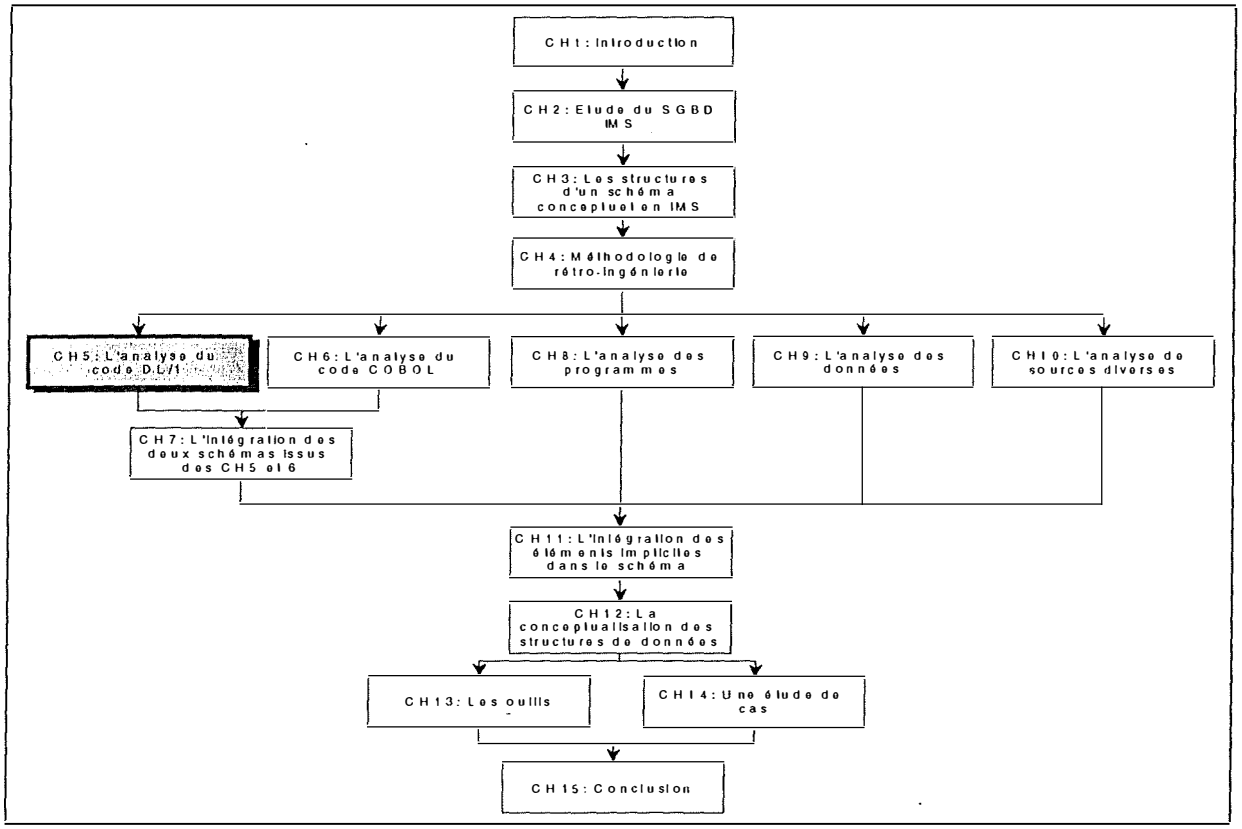


Figure 5-1: Le chapitre 5

5.1. Introduction

Le DL/1 est le DDL (Data Description Language) d'IMS. C'est un langage essentiel, il est le coeur d'IMS. En effet, c'est là que les bases de données sont décrites, ainsi que les éléments qui les composent. Dans un processus de rétro-ingénierie d'une base de données IMS, le DL/1 est certainement la première source à analyser pour le rétro-ingénieur, car cela lui permettra sans grande difficulté d'obtenir un premier schéma de la base de données (BD).

Par mesure de clarté, nous allons différencier les apports possibles du DL/1 en quatre catégories:

1. Les apports des déclarations de la BD physique
2. Les apports des déclarations de la BD logique
3. Les apports des déclarations d'index secondaires
4. Les apports des déclarations des vues

Notre but ici n'est pas de décrire la syntaxe du langage DL/1. Nous décrirons chaque instruction brièvement, sans en expliquer tous les détails. Nous nous attacherons plus particulièrement aux instructions (ou parties d'instructions) apportant quelque chose au schéma. Pour une description plus complète, consultez **[RICHARD, 95]**, ce chapitre étant une synthèse du mémoire de PH. RICHARD.

Dans les brèves descriptions syntaxiques de ce chapitre, les conventions sont les suivantes:

⇒ Les mots en lettres capitales sont des mots réservés au DDL.
⇒ Les mots en lettres minuscules sont des noms donnés par l'utilisateur.
⇒ [] indique que ce qui est entre crochets est facultatif.
⇒ { } indique qu'un choix de paramètres doit être fait.
⇒ Lorsqu'on devra faire un choix de paramètres, le paramètre par défaut sera souligné.
⇒ La présence de points d'extension (...) signifie que l'on ne détaille pas l'instruction ou la partie d'instruction.

Pour chaque instruction du DL/1, nous commencerons par en donner une syntaxe; ensuite, nous donnerons quelques explications concernant cette instruction; et enfin, nous préciserons quel(s) est (sont) l'(les) apport(s) de cette instruction au schéma entité-association que l'on veut générer. Vu que DB-MAIN constitue notre environnement de référence, nous supposerons que le schéma est construit dans l'atelier.

5.2. Les apports des déclarations de la BD physique

La Figure 5-2 montre la séquence d'enchaînement des instructions DL/1 dans la déclaration d'une BD physique. Cette déclaration doit commencer par l'instruction DBD et s'achever par les deux instructions DBDGEN et END; ces trois instructions n'apparaissent qu'une seule fois dans une même déclaration. L'instruction DBD est suivie d'une ou plusieurs instructions DATASET ou AREA. Chaque instruction DATASET peut englober plusieurs instructions SEGM. Les instructions LCHILD et FIELD sont rattachées à l'instruction SEGM; il peut y en avoir plusieurs par instruction SEGM. Enfin, il est à noter que l'ordre est important en ce qui concerne l'instruction SEGM: la suite des instructions SEGM indique l'ordre hiérarchique de la base de données.

Il ne peut y avoir que quinze niveaux dans la structure hiérarchique; il ne peut y avoir plus de 255 types de segments par DBD et plus de 255 champs par type de segment, avec un maximum de 1000 champs pour une DBD.

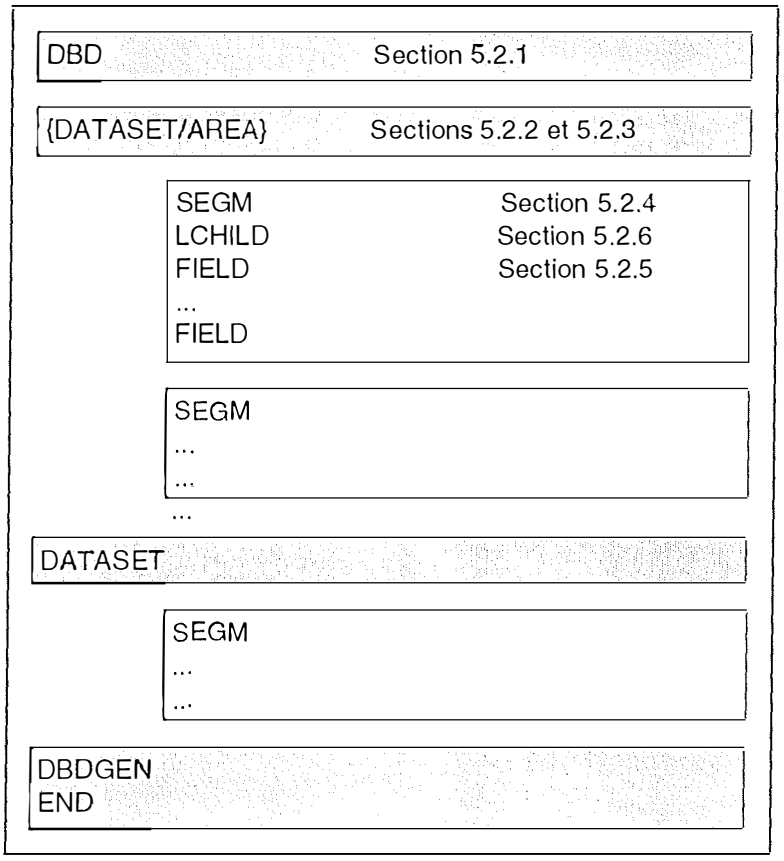


Figure 5-2: Enchaînement des instructions DL/1 pour une BD physique

Dans la suite, nous allons décrire les instructions présentées dans la Figure 5-2. Pour chaque instruction, nous donnerons une brève syntaxe, ainsi que les apports au schéma. Nous donnerons finalement un exemple illustrant l'apport de ces six instructions (section 5.2.7).

5.2.1. L'instruction DBD

L'instruction DBD identifie la base de données physique et en décrit l'organisation.

A. Brève syntaxe

```
DBD        NAME=(dbname1)
           ,ACCESS={....}
           [,EXIT=(....)]
           [,VERSION='n']
           [,RMNAME=(....)]
           [,PASSWD={....}]
```

Figure 5-3: Description de l'instruction DBD

B. Explication

NAME: *dbname1* est le nom de la BD.
ACCESS: spécifie la méthode d'accès DL/1.
EXIT: spécifie qu'une ou plusieurs routine(s) de capture de données est(sont) utilisée(s) sur l'ensemble des types de segments de la BD.
VERSION: spécifie la version de la BD.
RMNAME, PASSWD: paramètres techniques.

C. Apport

L'apport premier de cette instruction DBD 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. Le mot-clé ACCESS nous permet d'avoir une information technique importante relative au design physique de la BD. Ces informations techniques sont stockées dans la description technique du type d'entité racine de la BD analysée[1].
Le mot clé VERSION donne la version de la BD. On garde cette information dans la description sémantique du type d'entité racine.
Le mot clé RMNAME nous renseigne sur la méthode de hachage utilisée, tandis que PASSWD donne une information de sécurité technique. Ces deux dernières informations sont donc ajoutées à la description technique du type d'entité racine.

5.2.2. L'instruction DATASET

L'instruction DATASET identifie le support physique sur lequel les types de segments et leurs champs sont stockés. Elle indique également plusieurs détails relatifs au stockage physique. Toutes les instructions SEGM qui suivent un DATASET en font partie.

[1] La description technique dont on parle ici est en fait un lien avec l'atelier DB-MAIN, qui contient pour chaque objet d'un schéma une description technique et sémantique.

A. Brève syntaxe

```
DATASET        DD1=ddname
               [,DD2=ddname2]
               [,DEVICE=...]
               [,MODEL=...]
               [,OVFLW=...]
               [,BLOCK=...]
               [,SIZE=...]
               [,RECORD=...]
               [,SCAN=...]
               [,FRSPC=...]
               [,SEARCHA=...]
               [,REL=...]
```

Figure 5-4: Description de l'instruction DATASET

B. Explication

DD1: *ddname* est le nom identifiant du DATASET.
DD2, DEVICE, MODEL, OVFLW, BLOCK, SIZE, RECORD, SCAN, FRSPC, SEARCHA, REL: paramètres techniques.

C. Apport

Cette déclaration définit donc le type de support sur lequel seront stockés les types de segments s'y rattachant. La contribution au schéma de l'atelier se fait sous la forme d'une *collection* portant le nom indiqué par le mot-clé DD1 et dans laquelle se retrouvent les types d'entités provenant des types de segments du DATASET. L'ensemble des paramètres techniques de l'instruction DATASET se retrouve dans la description technique de la collection créée. Cependant, dans le cas des bases de données MSDB, SHSAM et SHISAM, celles-ci ne possédant qu'un seul type de segment, il n'est pas pertinent de créer une collection. Les renseignements fournis par le DATASET sont alors ajoutés à la description technique du type d'entité correspondant à l'unique type de segment.

5.2.3. L'instruction AREA

L'instruction AREA est équivalente à l'instruction DATASET pour les BD DEDB.

A. Brève syntaxe

```
AREA           DD1=ddname1
               ,SIZE=size
               ,UOW=(...)
               ,ROOT=(...)
```

Figure 5-5: Description de l'instruction AREA

B. Explication

DD1: *ddname1* est le nom de l'area.
SIZE, UOW et ROOT: caractéristiques techniques.

C. Apport

L'apport de l'instruction AREA au schéma est identique à celui de l'instruction DATASET.

5.2.4. L'instruction SEGM

Cette instruction décrit un type de segment.

A. Brève syntaxe

```
SEGM
        NAME=segname1
        [,PARENT={0
                  (segname2[,{SNGL DBLE}])}]
        ,BYTES=...
        [,TYPE=...]
        [,FREQ=...]
        [,POINTER=...]
        [,RULES=...]
        [,SSPTR=...]
        [,EXIT=...]
        [,COMPTRN=...]
```

Figure 5-6: Description de l'instruction SEGM

B. Explication

NAME: *segname1* est le nom identifiant du type de segment.
PARENT: spécifie le nom du type de segment parent physique (*segname2*) de ce type de segment. Les mots-clés SNGL et DBLE indiquent le type de pointeur utilisé par les segments parents pour accéder aux enfants (SNGL: le pointeur placé dans le préfixe du segment parent pointe vers le premier enfant, DBLE: deux pointeurs sont placés dans le préfixe du segment parent et pointent l'un vers le premier enfant, l'autre vers le dernier).
BYTES: donne la longueur en octets du type de segment.
TYPE: décrit le type des types de segments dépendants.
FREQ: indique la fréquence moyenne des occurrences du type de segment.
POINTER: spécifie le type de la chaîne de pointeurs utilisé pour chaîner les types de segments.

- 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: une seule occurrence permise pour ce type de segment, donc pas de pointeur.

RULES: spécifie où les segments sont insérés.
SSPTR: caractéristique technique propre aux BD DEDB.
EXIT: spécifie qu'une ou plusieurs routine(s) de capture de données est(sont) utilisée(s) sur l'ensemble des types de segments de la BD.
COMPTRN: compression de données.

C. Apport

L'apport de cette instruction est évidemment énorme. En effet, le type de segment défini par cette instruction devient au niveau du schéma de l'atelier un *type d'entité*. Son nom est celui spécifié dans le mot-clé NAME.
Grâce au mot-clé PARENT, on peut aussi déterminer le lien existant entre ce type d'entité (enfant) et son parent (*segname2*). Cela permet de créer des *types d'associations one-to-many* entre enfants et parent, avec la *cardinalité* du rôle joué par le type d'entité enfant égale à [1-1],

et celle du rôle joué par le type d'entité parent égale à [0-N]. Deux éléments conceptuels fondamentaux ont donc été ajoutés au schéma. Concernant les paramètres SNGL ou DBLE utilisés dans le mot-clé PARENT, ils apportent une description technique du type d'entité. Le mot-clé BYTES donne la longueur totale des attributs du type d'entité. Cette valeur, comparée avec la somme des longueurs des champs définis dans le DL/1, permet de détecter l'absence de champs. Nous ajoutons donc cette information à la description technique du type d'entité.
Le mot-clé FREQ donne 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*. Il peut cependant amener une suspicion sur cette cardinalité, qu'il faut vérifier dans les programmes et les données.
Le mot-clé POINTER apporte une information technique, et on ajoute donc cette information à la description technique du type d'entité. Cependant, lorsque la valeur du mot-clé est NOTWIN, le type d'association *one-to-many* est transformé en un type d'association *one-to-one*. La *cardinalité du rôle* joué par le type d'entité enfant reste [1-1], mais celle du rôle joué par le type d'entité parent devient [0-1][2].
Les quatre derniers mots-clés (RULES, SSPTR, EXIT et COMPRTN) apportent des informations qui sont ajoutées à la description technique du type d'entité.

5.2.5. L'instruction FIELD

L'instruction FIELD décrit un champ d'un type de segment.

A. Brève syntaxe

```
FIELD
        NAME=(fldname[,SEQ[,{M U}]])
        ,BYTES=bytes
        ,START=startpos
        [,TYPE=...]
```

Figure 5-7: Description de l'instruction FIELD

B. Explication

NAME: spécifie le nom du champ dans le segment. Le paramètre SEQ indique que ce champ est un champ de séquence pour ce segment. Le paramètre U (valeur par défaut) signifie que les occurrences du champ de séquence sont uniques dans la BD pour les segments racines et uniques sous un segment parent donné pour les autres segments. Le paramètre M permet les valeurs multiples.
BYTES: spécifie la longueur du champ en octets.
START: spécifie la position de départ du champ dans le type de segment.
TYPE: spécifie le type de données contenu dans le champ.

C. Apport

Chaque instruction FIELD rencontrée donnera lieu à la création d'un *attribut* dans le schéma, dont le nom sera celui spécifié dans le mot-clé NAME. Le paramètre SEQ, utilisé sans le paramètre M, spécifie l'attribut comme *identifiant* et *clé d'accès* du type d'entité si le type de segment est un type de segment racine. Si le type de segment n'est pas un type de segment racine, alors l'identifiant du type d'entité sera le champ de séquence ainsi que le rôle joué par le

[2]La valeur NOTWIN est évidemment trouvée dans l'instruction SEGM définissant le segment enfant.

type d'entité parent dans le type d'association le liant avec le type d'entité analysé. Lorsqu'on trouve dans un même segment plusieurs champs possédant les paramètres SEQ et U, le premier champ trouvé est identifiant, et les autres sont des identifiants secondaires (tenir compte du rôle si c'est un type de segment dépendant).
Les valeurs des mots-clés BYTES et TYPE sont ajoutées aux propriétés de l'attribut, dans le champ d'édition longueur et la combo-box type.
Le mot-clé START donne une information d'ordre technique et sa valeur est ajoutée à la description technique de l'attribut.
La combinaison des valeurs des mots-clés BYTES et START permet de retrouver des phénomènes de recouvrement d'attributs par un ou plusieurs autres. Nous parlerons plus en détail de ce phénomène du recouvrement[3] dans le chapitre sept. On se bornera ici à montrer deux petits exemples de recouvrement.
Le *premier exemple* est le cas de redéfinition non conflictuelle le plus connu: le cas des attributs décomposables. La combinaison des mots-clés BYTE et START permet évidemment de repérer une telle situation, en analysant la superposition éventuelle des valeurs. La Figure 5-8 montre un exemple d'un champ *A* décomposable en trois champs *B*, *C* et *D*. Si une telle situation est découverte, elle est modélisée dans le schéma (attribut décomposable créé).

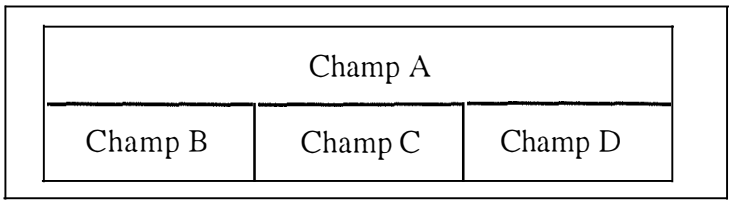


Figure 5-8: Attribut décomposable

Le *deuxième exemple* est le cas le plus simple de redéfinition conflictuelle. Il est illustré à la Figure 5-9; c'est en fait le cas d'un champ C qui recouvre partiellement deux champs A et B.

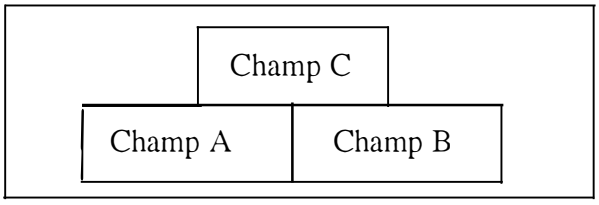


Figure 5-9: Recouvrement d'attributs

Chaque fois qu'un tel cas est découvert, une *contrainte de coexistence* est créée pour les champs *A* et *B*. Les cardinalités minimum des ces deux champs sont mises à 0, car les composants d'une contrainte de coexistence doivent être facultatifs. Ensuite, on crée une *contrainte d'exactement-un* entre le groupe *{A,B}* et le champ *C*, c'est-à-dire que dans une même occurrence du type d'entité contenant les champs *A*, *B* et *C*, on peut seulement trouver, soit le champ *C*, soit les champs *A* et *B*. Dans une même occurrence, on ne peut donc retrouver que *(A,B)* $\text{ou}_{\text{exclusif}}$ *C*. La Figure 5-10 montre le résultat de l'analyse de la Figure 5-9.

[3] Le phénomène de recouvrement est appelé REDEFINITION de façon plus générale.

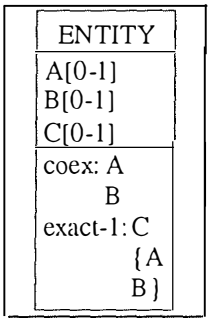


Figure 5-10: Résultat de l'analyse de la situation décrite à la Figure 5-9

5.2.6. L'instruction LCHILD

Dans le cadre des déclarations des bases de données physiques, l'instruction LCHILD est utilisée pour référencer la base de données index primaire d'une BD en accès HIDAM[4]. Pour ses autres utilisations, voyez le paragraphe 5.3.

A. Brève syntaxe

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

Figure 5-11: Description de l'instruction LCHILD pour les BD HIDAM

B. Explication

Sans entrer dans les détails, cette instruction permet de faire le lien entre la BD HIDAM et sa BD index. L'instruction sera présente dans les deux BD. Pour plus de détails concernant son utilisation, voir **[RICHARD, 95]**.

C. Apport

L'apport de cette instruction n'est que d'ordre technique. On ajoute donc à la description technique du type d'entité correspondant au type de segment racine de la BD HIDAM le nom de la BD index, ainsi que le nom de son unique type de segment. Les informations concernant le DATASET de la BD index y sont également ajoutées.

5.2.7. Un exemple illustrant les apports des instructions DBD, DATASET, SEGM et FIELD

```
DBD       NAME=FIRMDB,
          ACCESS=HDAM,RMNAME=(DFSHDC40,3,100)
DATASET   DD1=FIRMDBV,
          DEVICE=3380,SIZE=2048
SEGM NAME=FIROOT,
          BYTES=140,
          PTR=T,
          PARENT=0
FIELD     NAME=(KEY,SEQ,U),
          BYTES=38,
          START=1
FIELD     NAME=PARTNO,
          BYTES=21,
          START=1
FIELD     NAME=MU,
          BYTES=2,
          START=22
FIELD     NAME=WHNO,
          BYTES=2,
          START=24
```

[4] Les BD en accès HIDAM sont en effet composées de deux BD, la BD proprement dite et la BD index.

```
FIELD      NAME=ORDNO,
           BYTES=7,
           START=26
FIELD      NAME=SUPPNO,
           BYTES=6,
           START=33
SEGM       NAME=FIWHSE,
           BYTES=40,
           PTR=T,
           PARENT=((FIROOT,SNGL))
FIELD      NAME=(KEY,SEQ,U),
           BYTES=2,
           START=1
FIELD      NAME=CORDITM,
           BYTES=7,
           START=3
FIELD      NAME=CORDDEL,
           BYTES=2,
           START=10
FIELD      NAME=RECOUVRE,
           BYTES=7,
           START=5
DATASET    DD1=ens2,
           DEVICE=3280,SIZE=2048
SEGM  NAME=FIREQS,
      BYTES=140,
      PTR=T,
      PARENT=((FIWHSE,SNGL))
FIELD NAME=(KEY,SEQ,U),
      BYTES=10,
      START=1
SEGM  NAME=FIGREC,
      BYTES=32,
      PTR=T,
      PARENT=((FIWHSE,SNGL))
FIELD NAME=(KEY,SEQ,U),
      BYTES=9,
      START=1
FIELD NAME=(KEY2,SEQ,M),
      BYTES=6,
      START=10
DBDGEN
END
```

Figure 5-12: Source IMS illustrant l'apport des instructions DBD, DATASET, SEGM et FIELD

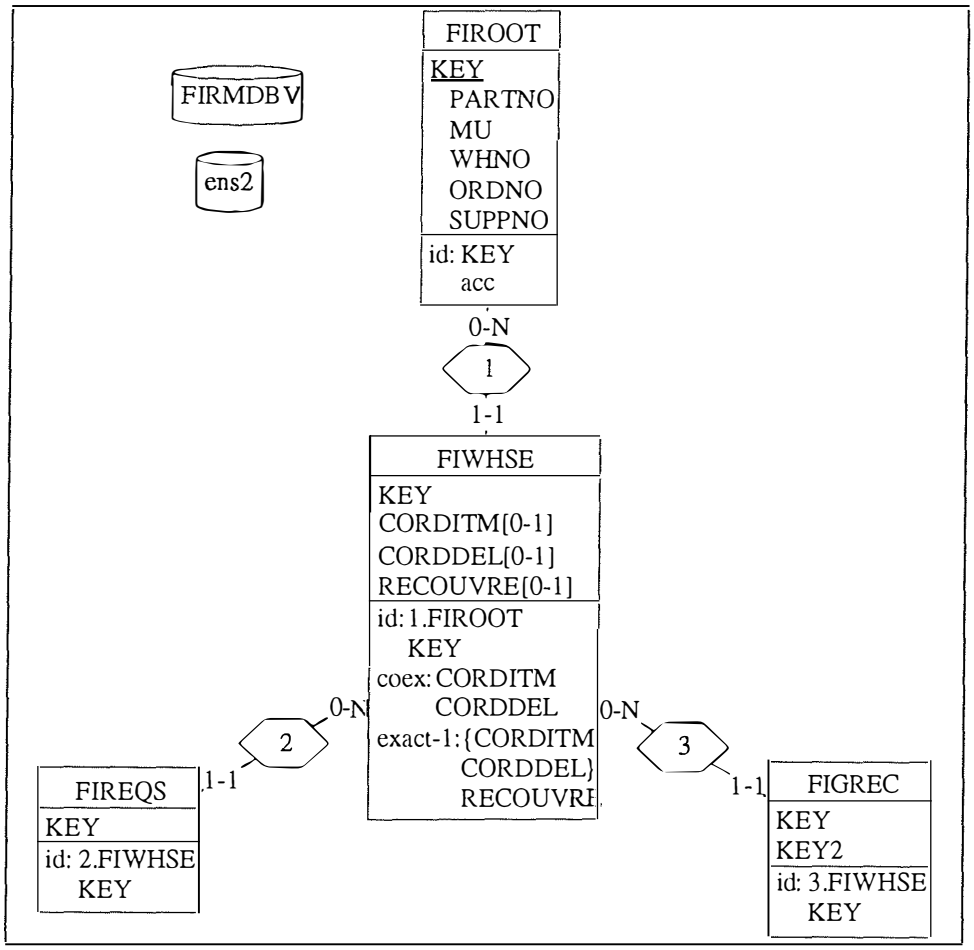


Figure 5-13: Résultat de l'analyse de la Figure 5-12

Si on examine le code source, on se rend compte qu'il contient deux instructions DATASET, ce qui a permis la création de deux collections contenant chacune leurs types d'entités associés. A chaque instruction SEGM du code source correspond un type d'entité dans le schéma de la Figure 5-13. On peut aussi remarquer que la hiérarchie représentée à la Figure 5-13 correspond bien à l'ordre de déclaration des instructions SEGM dans le code source. On voit aussi que les relations *one-to-many* entre un enfant et son parent respecte bien ce qui est décrit dans le code source (mot-clé PARENT de l'instruction SEGM).
Au niveau des champs, on remarque que les identifiants des types d'entités dépendants sont composés de leur champ identifiant et du rôle joué par leur parent dans la relation. On a également trouvé un attribut décomposable dans le type d'entité FIROOT, via l'étude des mots-clés BYTE et START des instructions FIELD. Enfin, on a aussi créé des contraintes de coexistence et d'exactement-un dans le type d'entité FIWHSE, suite à la découverte dans le code source de chevauchements entre attributs.

5.3. Les apports des déclarations de la BD logique

Deux phases sont nécessaires pour l'implémentation d'une BD logique:

1. La déclaration d'un ou plusieurs type(s) de relation(s) logique(s) dans une BD physique. Pour ce faire, les instructions SEGM et LCHILD vues au point 5.2 sont étendues.
2. La déclaration de la BD logique, qui se base sur la déclaration des types de relations logiques.

5.3.1. La déclaration d'une relation logique

Par rapport à la déclaration d'une base de données physique, les instructions suivantes restent inchangées: DBD, FIELD, DBDGEN et END. Les instructions SEGM et LCHILD sont étendues pour certains paramètres.

Comme nous l'avons déjà vu au chapitre deux, une relation logique implique trois ou quatre types de segments. Nous allons d'abord décrire les changements de l'instruction SEGM pour ces quatre types de segments. Ensuite, nous analyserons l'instruction LCHILD. Nous terminerons en analysant l'apport des trois instructions au schéma.

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

A.1. Brève syntaxe

Voici une brève syntaxe de l'instruction SEGM pour l'enfant logique réel. Pour plus de détails concernant les paramètres, consultez **[RICHARD, 95]**.

```
SEGM
        NAME=...
        ,PARENT=(segname2[,{SNGL DBLE},...]
        [(segname3[,{VIRTUAL PHYSICAL}][,dbname2])])
        ,BYTES=...
        ,POINTER=...
        ,RULES=...
```

Figure 5-14: Instruction SEGM pour l'enfant logique réel et ses parents

A.2. Explication

NAME: nom du type de segment enfant logique réel.
PARENT: *segname2* est le nom du type de segment parent physique de ce type de segment enfant logique. *Segname3* est le nom du type de segment parent logique de ce type de segment. *Dbname2* est le nom de la BD dans laquelle se trouve le type de segment parent logique. Il peut être omis si le parent se trouve dans la même BD.
BYTES: a la même signification que précédemment.
POINTER: spécifie les différents types de pointeurs utilisés. Les paramètres sont les suivants:

- LTWIN (vers l'avant) et LTWINBWD (vers l'avant et l'arrière): nous renseignent sur le type de chaîne de pointeurs utilisé dans le cadre d'un type de relation logique bidirectionnel virtuel. Ces pointeurs chaînent des enfants logiques réels.
- LPARNT: indique qu'un pointeur vers le type de segment parent logique est inséré dans le préfixe du segment.

- CTR: réserve un compteur dans le préfixe des types de segments parents logiques. Ce compteur indique le nombre d'enfants logiques attachés à un même parent.
- PAIRED: est indiqué pour les deux types de segments enfants logiques réels d'un type de relation bidirectionnel physique.

RULES: spécifie le type de chemin qui doit être utilisé pour insérer, effacer et remplacer un segment.

B. L'instruction SEGM pour l'enfant logique virtuel

B.1. Brève syntaxe

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

Figure 5-15: Instruction SEGM pour l'enfant logique virtuel

B.2. Explication

NAME: *virtchild* est le nom du type de segment enfant logique virtuel.
PARENT: *segname2* est le nom du type de segment parent logique, c'est-à-dire le type de segment parent physique du type de segment enfant logique virtuel.
SOURCE: *segname3* est le nom du type de segment enfant logique réel et *dbname1* est le nom de la BD dans laquelle se trouve ce type de segment enfant logique réel.
POINTER=PAIRED: définit ce type de segment comme type de segment enfant logique virtuel.

C. L'instruction LCHILD

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

C.1. Brève syntaxe

Le format de l'instruction est donné à la Figure 5-16.

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

Figure 5-16: Instruction LCHILD pour le type de segment enfant logique

C.2. Explication

NAME: *segname1* est le nom du type de segment enfant logique qui se trouve dans la BD de nom *dbname*.
POINTER: a la même signification que pour les pointeurs physiques.
PAIR: *segname2* est le nom du type de segment enfant logique réel avec lequel le type de segment est pairé.
RULES: paramètre technique.

D. L'apport des trois instructions au schéma

L'apport de ces trois instructions au schéma est évidemment assez riche. La première étape consiste à déterminer le type de relation logique (unidirectionnel ou bidirectionnel) et ensuite à identifier les types de segments qui en font partie.

A. Détermination du type de relation logique

A.1. Type de relation logique bidirectionnel

La présence d'un type de relation logique bidirectionnel nous est donnée par le paramètre PAIRED du mot-clé POINTER de l'instruction SEGM décrivant le type segment enfant logique réel (ou virtuel). Une fois que l'on a identifié la bidirectionnalité, deux cas sont possibles:

A.1.1. Type de relation logique bidirectionnel physique

Le mot-clé PAIR des deux instructions LCHILD permet de déterminer les deux types de segments pairés.

A.1.2. Type de relation logique bidirectionnel virtuel

Ce type de relation 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 de segments (pairés).

A.2. Type de relation logique unidirectionnel

La présence d'un type de relation logique unidirectionnel nous est donnée par l'absence du paramètre PAIRED du mot-clé POINTER de l'instruction SEGM pour l'enfant logique.

B. Détermination des types de segments faisant partie du type de relation logique

La présence d'un type de 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.

B.1. Type de relation logique unidirectionnel

On crée un *type d'entité* à partir du type de segment enfant logique réel. Les deux *types d'associations* sont alors créés entre le type d'entité enfant et les deux parents, indiqués dans le mot-clé PARENT de l'instruction SEGM de l'enfant logique et par l'instruction LCHILD dans la déclaration du type de segment parent logique. Le type d'association issu du type de relation logique reçoit comme nom la lettre 'L' (pour *logique*), suffixé par un numéro incrémenté à chaque nouvelle relation logique.

B.2. Type de relation logique bidirectionnel physique

On crée un *type d'entité* à partir des deux types de segments enfants logiques réels. Le nom de ce type d'entité est la concaténation des noms des deux segments. Les deux *types d'associations* sont alors créés entre le type d'entité enfant et les deux parents, indiqués dans les mots-clés PARENT des instructions SEGM des enfants logiques et par les instructions LCHILD dans la déclaration des types de segments parents logiques. Les types d'associations issus des types de relations logiques reçoivent comme noms la lettre'L' (pour *logique*), suffixé par un numéro incrémenté à chaque

nouvelle relation logique. Pour faire le lien entre les deux types d'associations, on suffixe ce nom par le caractère '-', suivi respectivement des chiffres 1 et 2.

B.3. Type de relation logique bidirectionnel virtuel

On crée un *type d'entité* à partir du type de segment enfant logique réel et à partir du type de segment enfant logique virtuel. Le nom de ce type d'entité est la concaténation des noms des deux segments. Les attributs de ce type d'entité proviennent de la description du type de segment enfant logique réel. Les deux *types d'associations* sont alors créés entre le type d'entité enfant créé et les deux parents, indiqués dans le mot-clé PARENT de l'instruction SEGM de l'enfant logique et par l'instruction LCHILD dans la déclaration du type de segment parent logique. Les noms des types d'associations sont donnés suivant la règle énoncée dans le cas précédent.

On ajoute également à la description technique de chaque type d'entité créé les informations issues de l'instruction SEGM (mots-clés POINTER et RULES, ainsi que les paramètres VIRTUAL/PHYSICAL) et de l'instruction LCHILD (mots-clés POINTER et RULES).

5.3.2. La déclaration de la BD logique

Comme nous l'avons vu au chapitre deux, une base de données logique peut être considérée comme une vue sur des bases de données physiques existantes. Pour une BD logique, les instructions DBD, DATASET et SEGM sont modifiées, tandis que les instructions LCHILD et FIELD sont interdites.

A. Brève syntaxe

```
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 5-17: Description de la déclaration d'une BD logique

B. Explication

DBD:

NAME: *dbname* est le nom de la BD logique. Il est unique pour tout l'environnement.

ACCESS=LOGICAL: définit la BD comme une BD logique.

DATASET LOGICAL: cette instruction doit être codée telle quelle.

SEGM:

NAME: *segname1* est le nom du type de segment.

PARENT: *segname2* spécifie le nom du type de segment parent de ce type de segment. Le parent doit avoir été défini avant.

SOURCE: spécifie la ou les source(s) du type de segment. Il existe deux cas:

- les types de segments concaténés.
- les types de segments non concaténés.

Pour les types de segments non concaténés, *segname3* spécifie le type de segment source et *dbname1* la BD physique dans laquelle il se trouve.
Pour les types de segments concaténés, *segname3* spécifie le type de segment enfant logique comme défini dans la BD physique, et *dbname1* est la BD physique dans laquelle *segname3* se trouve. *Segname4* est le type de segment parent destination, et *dbname2* est le nom de la BD physique dans laquelle *segname4* est défini.

C. Apport

Cette déclaration n'apporte rien au niveau du schéma conceptuel. Elle sert uniquement à indiquer dans quel sens sera parcouru le type de relation logique. Etant donné qu'un PCB est toujours défini sur la BD logique et que ce PCB sert à indiquer le chemin d'accès, l'apport au schéma se fera lors de l'analyse de ce PCB. Toutefois, les noms utilisés par le PCB sont ceux définis dans la BD logique, pas physique. Il faut donc garder une trace de la correspondance entre types de segments de la BD logique et types de segments de la BD physique. Cela se fait par le mot-clé SOURCE de l'instruction SEGM pour la déclaration d'une BD logique. **[RICHARD, 95]**.

5.3.3. Trois exemples illustrant les apports des instructions SEGM et LCHILD dans le cas des relations logiques

A. EXEMPLE 1: le type de relation logique unidirectionnel

La Figure 5-18 est un code source illustrant un type de relation logique unidirectionnel. En effet, le mot-clé POINTER de l'instruction SEGM pour SegB n'a pas la valeur PAIRED. La deuxième étape consiste à déterminer qui est parent physique et parent logique de SegB (mot-clé PARENT de l'instruction SEGM pour SegB et instruction LCHILD de BD2). Dans la Figure 5-19, on remarque que l'on a un type d'association *L1*, qui représente le type de relation logique. Les cardinalités sont également conformes.

```
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,BD2)),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,BD1)
DBDGEN
FINISH
END
```

Figure 5-18: Source IMS illustrant un type de relation logique unidirectionnel

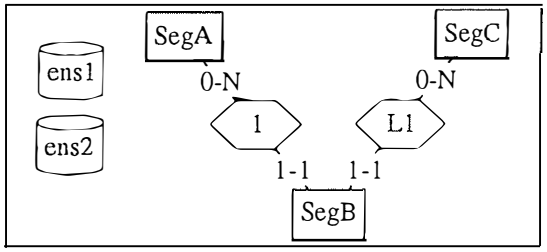


Figure 5-19: Résultat de l'analyse de la Figure 5-18

B. EXEMPLE 2: le type de relation logique bidirectionnel physique

La Figure 5-20 est un code source illustrant un type de relation logique bidirectionnel physique. Tout d'abord, on identifie ce type de relation logique par les paramètres PAIRED des mots-clés POINTER des instructions SEGM pour SegB et SegD. Ensuite, les deux types de segments pairés sont repérés par la présence des mots-clés PAIR des instructions LCHILD. Dans la Figure 5-21, on a regroupé au sein d'un même type d'entité les deux types de segments pairés SegB et SegD.

```
DBD     NAME=BD1,ACCESS=(HDAM,VSAM)
DATASET DD1=ens1
SEGM    NAME=SegA,PARENT=0,BYTES=53,RULES=PLV,FIRST
LCHILD  NAME=(SegD,BD2),PAIR=SegB
SEGM    NAME=SegB,PARENT=((SegA),(SegC,PHYSICAL,BD2)),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,BD1),PAIR=SegD
SEGM    NAME=SegD,PARENT=((SegC),(SegA,PHYSICAL,BD1)),BYTES=48,POINTER=LPARNT,PAIRED
      ,RULES=PLV,FIRST
DBDGEN
FINISH
END
```

Figure 5-20: Source IMS illustrant un type de relation logique bidirectionnel physique

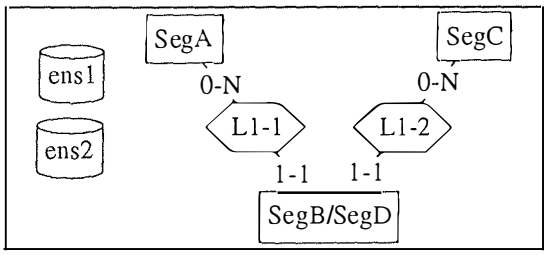


Figure 5-21: Résultat de l'analyse de la Figure 5-20

C. EXEMPLE 3: le type de relation logique bidirectionnel virtuel

La Figure 5-22 est un code source illustrant un type de relation logique bidirectionnel virtuel. Le caractère virtuel de ce type de relation logique est indiqué par le mot-clé SOURCE de l'instruction SEGM pour l'enfant logique virtuel. Ce même mot-clé permet de retrouver l'enfant logique réel. Comme on le voit à la Figure 5-23, la représentation d'un type de relation bidirectionnel virtuel se fera de la même manière que dans le cas physique.

```
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,BD2)),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,BD1),POINTER=SNGL,PAIR=SegD
SEGM    NAME=SegD,PARENT=SegC,SOURCE=((SegB,DATA,BD1)),POINTER=PAIRED
DBDGEN
FINISH
END
```

Figure 5-22: Source IMS illustrant un type de relation logique bidirectionnel virtuel

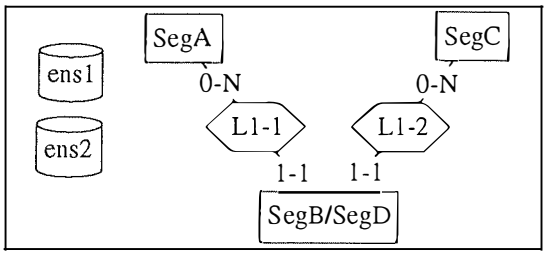


Figure 5-23: Résultat de l'extraction de la Figure 5-22

5.4. Les apports des déclarations d'index secondaires

L'implémentation d'un index secondaire se décompose en trois phases:
1. La déclaration du type de segment cible de l'index;
2. La déclaration du type de segment source de l'index;
3. La déclaration de la BD index secondaire.

Les deux premières déclarations se retrouvent dans la même BD physique. La troisième fait l'objet d'une BD particulière.

5.4.1. La déclaration du type de segment cible

Cette déclaration se fait via deux instructions « inséparables »: LCHILD et XDFLD.

A. L'instruction LCHILD

L'instruction LCHILD fait le lien avec la BD index.

A.1. Brève syntaxe

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

Figure 5-24: Description de l'instruction LCHILD pour le segment cible

A.2. Explication

NAME: *segname1* est le nom du type de segment pointeur défini dans la BD index dont le nom est *dbname*.
POINTER: indique le type de pointeur utilisé.

B. L'instruction XDFLD

L'instruction XDFLD spécifie les champs servant à l'indexation.

B.1. Brève syntaxe

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

Figure 5-25: Description de l'instruction XDFLD pour le segment cible

B.2. Explication

NAME: indique le nom du champ d'index secondaire. Il doit être unique parmi tous les noms des champs du type de segment indexé.
SEGMENT: *segname* est le nom du type de segment source. S'il n'est pas mentionné, le type de segment cible est considéré comme source aussi.
CONST: spécifie un caractère qui identifie chaque segment pointeur de la BD index.

SRCH: spécifie le ou les champ(s) du type de segment index source qui doi(ven)t être utilisé(s) comme champ(s) de recherche de l'index secondaire.
SUBSEQ: spécifie le ou les champ(s) du type de segment index source qui doi(ven)t être utilisé(s) comme champ(s) de sous-séquence de l'index secondaire.
DDATA: spécifie le ou les champ(s) du type de segment index source qui doi(ven)t être utilisé(s) comme champ(s) de données dupliqué(s) de l'index secondaire.
NULLVAL, EXTRTN: caractéristiques techniques.

C. L'apport des deux instructions au schéma

L'apport au schéma de ces deux instructions se fait sous la forme d'une clé d'accès pour le type d'entité provenant du type de segment cible. C'est le(s) champ(s) spécifié(s) par le mot-clé SRCH qui sera(ont) cette clé d'accès. Il y a cependant une exception: en effet, l'atelier ne permet pas de modéliser les clés d'accès dont l'origine est différente de la cible. Dans le cas où le mot-clé SEGMENT spécifie un type de segment autre que le type de segment cible, la clé d'accès est simplement spécifiée dans la description technique du type d'entité provenant du type de segment cible.
Les autres renseignements provenant des deux instructions sont ajoutés à la description technique de la clé d'accès correspondante.

5.4.2. La déclaration du type de segment source

Un type de segment est reconnu comme type de segment source via l'instruction XDFLD (mot-clé SEGMENT) vue précédemment dans la déclaration du type de segment cible. Il n'y a donc pas d'instruction particulière. Cependant, il existe la possibilité de déclarer des champs dits champs_reliés_système:

◊ un champ reprenant tout ou partie de l'identifiant du type de segment source. Ce champ commence obligatoirement par les trois caractères «/CK». Ce champ est déclaré pour permettre l'utilisation de l'identifiant du type de segment source dans les mots-clés SUBSEQ et DDATA de l'instruction XDFLD.
◊ un champ qui assure l'unicité des valeurs de champ de séquence dans un index secondaire. Ce champ doit obligatoirement commencer par «/SX».

L'apport au schéma se limite à l'ajout des champs_reliés_système à la description technique du type d'entité provenant du type de segment source.

5.4.3. La déclaration de la BD index

La déclaration d'une BD index ressemble à celle d'une BD physique. Toutes les instructions vues précédemment sont utilisées, à l'exception de XDFLD. Tout comme dans une BD physique, on peut avoir plusieurs instructions SEGM et LCHILD pour un DATASET, ainsi que plusieurs instructions FIELD pour un même segment. La figure 5-26 montre l'enchaînement des instructions pour une BD index.

A. Brève syntaxe

```
DBD
        NAME=(dbname1,...)
        ,ACCESS=INDEX
        [,PASSWD= ]
DATASET
        DD1=ddname1
        [,BLOCK= ]
        [,SIZE= ]
        [,RECORD= ]
SEGM
        NAME=segname1
        [,PARENT=0]
        ,BYTES= bytes
        [,FREQ=freq]
LCHILD
        NAME=(segname1,dbname)
        ,POINTER=
        ,INDEX=fldname
FIELD
        NAME=(fldname1,SEQ,U)
        ,BYTES=bytes
        ,START=1

DBDGEN
END
```

Figure 5-26: Enchaînement et syntaxe des instructions pour une BD index

B. Explication

DBD: Les mots-clés ont la même signification que pour les BD physiques.
DATASET: Les mots-clés ont la même signification que pour les BD physiques.
SEGM:

NAME: *segname1* est le nom du type de segment pointeur.
PARENT, BYTES et FREQ: Ces mots-clés ont la même signification que pour les BD physiques.

LCHILD:

NAME: *segname1* est le nom du type de segment cible se trouvant dans la BD de nom *dbname*.
POINTER: indique le type de pointeur utilisé.
INDEX: *fldname* est le nom du champ indexé

FIELD: ce mot-clé a la même signification que pour les BD physiques.

C. Apport

L'apport au schéma est minime. Les renseignements qu'on peut glaner (le nom du type de segment pointeur, le type de pointeur utilisé et les informations fournies par les instructions DBD et DATASET) sont ajoutées à la description technique des différentes clés d'accès pour lesquelles cette BD a été créée.

5.4.4. Un exemple illustrant l'apport des BD index

La Figure 5-27 ci-dessous est un code source constitué de la déclaration d'une base de donnée physique (première instruction DBD) et de celle d'une base de donnée index (deuxième

instruction DBD). Comme nous venons de le dire, la déclaration de la BD index n'apportera que des informations techniques. Nous nous limiterons donc à détailler la première DBD, où nous avons mis en gras les instructions directement concernées par les déclarations d'index.
Sur la Figure 5-28, nous remarquons la présence d'une clé d'accès sur le type d'entité CUROOT. Cette clé d'accès provient des instructions LCHILD et XDFLD du type de segment CUROOT. Remarquons aussi que le champ /SXROOT n'est pas repris dans le schéma de la Figure 5-28 car c'est un champ technique.

```
DBD       NAME=CUSTDB,
          ACCESS=HDAM,RMNAME=(DFSHDC40,5,5000)
DATASET DD1=CUSTDBV,
        DEVICE=3380,SIZE=4096
SEGM  NAME=CUROOT,
        BYTES=412,
        PTR=T,
        COMPRTN=INFCMPPF,
        PARENT=((0))
FIELD NAME=(KEY,SEQ,U),
      BYTES=8,
      START=1
FIELD NAME=/SXROOT
FIELD NAME=(AINDEX),
        BYTES=12,
        START=297
LCHILD NAME=(CUSEC1,CUSTS1),
        POINTER=INDX
 XDFLD NAME=AINDEX2,
        NULLVAL=X'40',
        SUBSEQ=/SXROOT,
        SRCH=AINDEX
DBDGEN
END

DBD  NAME=CUSTS1,ACCESS=INDEX
DATASET DD1=CUSTS1V,
                DEVICE=3380
SEGM  NAME=CUSEC1,
          BYTES=16
FIELD     NAME=(AINDEX2,SEQ,U),
          BYTES=16,
          START=1
LCHILD NAME=(CUROOT,CUSTDB),
          INDEX=AINDEX2,
          PTR=SNGL
DBDGEN
END
```

Figure 5-27: Source IMS illustrant l'apport des instruction LCHILD et XDFLD

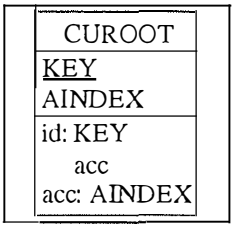


Figure 5-28: Résultat de l'analyse de la Figure 5-27

5.5. Les apports des déclarations des vues

Un programme d'application doit toujours passer par une vue pour accéder à une BD physique ou logique. Pour ce faire, il définit un PSB[5] regroupant les différentes vues nécessaires à la bonne exécution des programmes. Un PCB contient les types de segments et les champs d'une BD qui sont sensibles au programme. L'enchaînement des instructions d'un PCB est montré à la Figure 5-29.

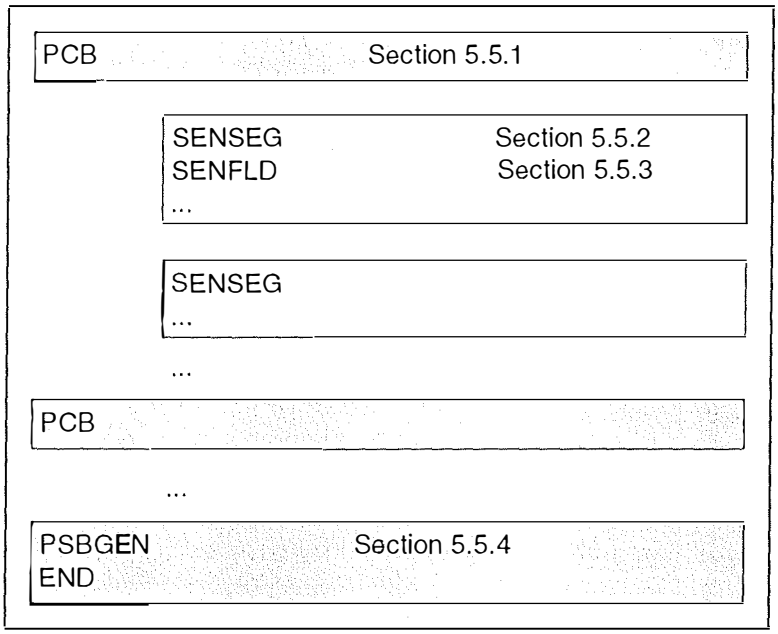


Figure 5-29: Enchaînement des instructions pour la déclaration d'un PCB

5.5.1. L'instruction PCB

L'instruction PCB a pour but de définir une vue d'une base de données que le programme a l'intention d'utiliser. La Figure 5-30 montre son format.

A. Brève syntaxe

```
PCB
        PCBTYPE=DB
        ,{DBNAME NAME}=name
        [,PCBNAME=pcbname]
        [,PROCOPT=...]
        [,SB=...]
        ,KEYLEN=value
        [,POS=...]
        [,PROCSEQ=indexdbname]
        [,LIST=...]
```

Figure 5-30: Instruction PCB

B. Explication

PCBTYPE=DB: identifie ce PCB comme une vue sur une BD.
DBNAME(NAME): indique le nom de la BD sur laquelle porte la vue.

[5]Voir infra CH2.

PCBNAME: spécifie le nom du PCB.
PROCOPT: spécifie les opérations permises sur les types de segments sensibles déclarés dans le PCB.
SB, KEYLEN, POS, PROCSEQ, LIST: paramètres techniques.

C. Apport

L'apport des vues au schéma existant se fait par la création de sous-schémas externes, qui montrent la vue qu'ont les programmes d'application de la BD. A un PCB correspondra donc un sous-schéma externe. L'instruction PCB, pas plus que les suivantes, n'apporte de changements au schéma existant. L'instruction PCB nous apporte notamment le nom du PCB. Ce nom sera utile; en effet, les types d'entités créés seront dotés d'un suffixe, ce suffixe étant constitué d'un chiffre et du nom du PCB. On fait cela pour maintenir l'unicité des noms au sein d'un même schéma.
Les vues sont surtout utiles pour montrer les chemins d'accès. On représente ceux-ci par les sous-schémas externes.
Les vues peuvent aussi nous apporter des renseignements 'cachés'. Ainsi, si pour chaque PCB défini sur un type de segment donné, ce sont toujours les trois mêmes champs qui sont sensibles, et que ceux-ci s'avèrent être facultatifs, cela peut être un indice de l'existence d'une contrainte de coexistence entre ces trois champs au niveau de la base de donnée physique.

5.5.2. L'instruction SENSEG

Cette instruction spécifie quels sont les types de segments auxquels le programme est sensible.

A. Brève syntaxe

```
SENSEG
        NAME=name
        ,PARENT=
        [,PROCOPT= ]
        [,SSPTR= ]
        [,INDICES=list1]
```

Figure 5-31: Instruction SENSEG

B. Explication

NAME: nom d'un type de segment défini auparavant dans une BD.
PARENT: parent de ce type de segment. Ce champ vaut zéro si le segment est un segment racine.
PROCOPT, SSPTR, INDICES: paramètres techniques.

C. Apport

Quand on rencontre l'instruction SENSEG, on peut créer un type d'entité qui recevra un nom selon la technique décrite au point précédent. Signalons enfin que les informations fournies par le mot-clé PROCOPT sont ajoutées à la description technique de chaque type d'entité recopié. On recrée les types d'associations entre les types d'entités créés grâce au mot-clé PARENT de l'instruction SENSEG.

5.5.3. L'instruction SENFLD

Cette instruction spécifie quels sont les champs d'un type de segment auxquels le programme est sensible.

A. Brève syntaxe

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

Figure 5-32: Instruction SENFLD

B. Explication

NAME: *name* est le nom d'un champ défini dans une BD.
START: *startpos* est 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 être altéré dans un CALL de remplacement.

C. Apport

Quand on rencontre l'instruction SENFLD, on peut créer un champ dans le type d'entité qui en est propriétaire.

5.5.4. L'instruction PSBGEN

Cette instruction spécifie les caractéristiques du programme d'application.

A. Brève syntaxe

```
PSBGEN
        PSBNAME=name
        [,LANG= ]
        [,MAXQ= ]
        [,CMPAT= ]
        [,IOASIZE= ]
        [,SSASIZE= ]
        [,IOERPN= ]
        [,OLIC= ]
        [,LOCKMAX= ]
```

Figure 5-33 : Instruction PSBGEN

B. Explication

PSBNAME: *name* est le nom du PSB.
LANG: indique le langage dans lequel le programme d'application est écrit.
MAXQ, CMPAT, IOASIZE, SSASIZE, IOERPN, OLIC, LOCKMAX: paramètres techniques.

C. Apport

Le nom et les renseignements concernant le PSB sont ajoutés à la description technique du type d'entité racine du PCB.

5.5.5. Un exemple illustrant l'apport des vues au schéma

```
DBD      NAME=ADVNDB,
         ACCESS=HDAM,
         RMNAME=(DFSHDC40,1,100)
DATASET  DD1=ADVNDBV,
         DEVICE=3380,SIZE=2048
SEGM     NAME=ADROOT,
         BYTES=180,
         PTR=T,
         PARENT=((0))
FIELD    NAME=(KEY,SEQ,U)
         BYTES=5
         START=1
FIELD    NAME=STATUS
         BYTES=2
         START=6
FIELD    NAME=ADDATE
         BYTES=4
         START=7
FIELD    NAME=PCKDATE
         BYTES=10
         START=10
FIELD    NAME=WEIGHT
         BYTES=4
         START=19
SEGM     NAME=ADLINE,
         BYTES=240,
         PTR=T,
         PARENT=((ADROOT,DBLE))
FIELD    NAME=(KEY,SEQ,U)
         BYTES=3
         START=1
FIELD    NAME=MU-ORDNO
         BYTES=10
         START=3
FIELD    NAME=SEQNO
         BYTES=4
         START=12
DBDGEN
END

PCB      PCBTYPE=DB,DBNAME=ADVNDB,PCBNAME=vue1,PROCOPT=A,KEYLEN=100
         SENSEG NAME=ADROOT,PARENT=0
         SENFLD NAME=KEY, START=1
         SENFLD NAME=ADDDATE, START=7
         SENFLD NAME=PCKDATE, START=10
         SENSEG NAME=ADLINE,PARENT=ADROOT
         SENFLD NAME=KEY, START=1
         SENFLD NAME=MU-ORDNO, START=3
PCB      PCBTYPE=DB,DBNAME=ADVNDB, PCBNAME=vue2,PROCOPT=G,KEYLEN=100
         SENSEG NAME=ADROOT,PARENT=0
         SENFLD NAME=KEY, START=1
         SENFLD NAME=WEIGHT, START=19
PSBGEN LANG=COBOL,PSBNAME=ADVNPSB
END
```

Figure 5-34: Source IMS illustrant l'apport pour l'instruction PCB

La Figure 5-34 est un code source dont le schéma résultant se trouve à la Figure 5-35. La partie gauche du schéma représente une hiérarchie issue de la déclaration de la DBD physique définie dans la première partie du code source de la Figure 5-34. Des deux instructions PCB, on a déduit les deux sous-schémas de la partie droite de la Figure 5-35. On remarque dans ces sous-schémas la suffixation des types d'entités et des types d'associations par le nom des PCB et un numéro.

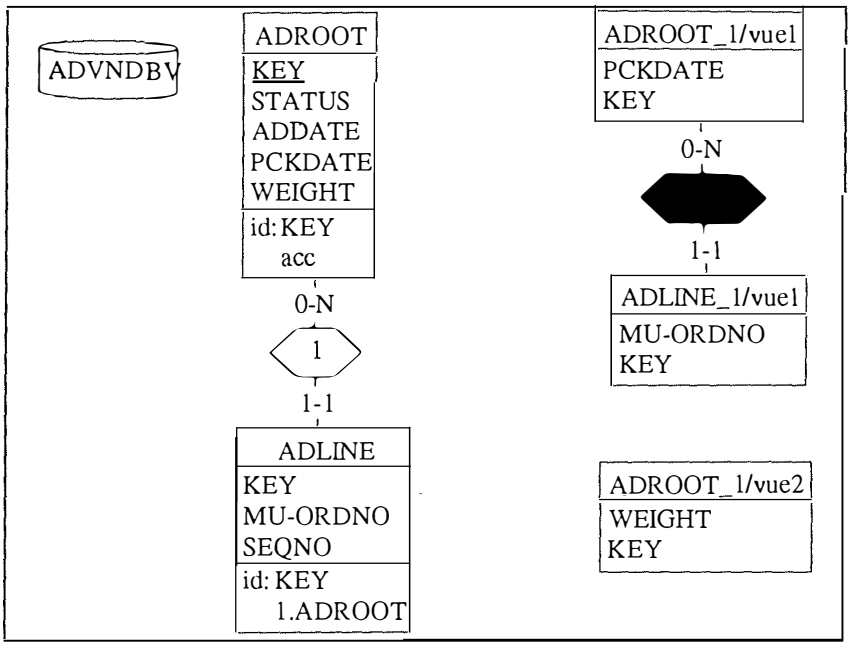


Figure 5-35: Résultat de l'extraction de la Figure 5-34

5.6. Conclusion

Ce chapitre cinq nous a permit de faire connaissance avec le code DL/1 et d'en retiter toute une série d'informations « sûres », à partir desquelles on peut construire un premier schéma. Cette analyse servira de spécification à l'extracteur IMS de DB-MAIN.

CHAPITRE 6: L'analyse du code COBOL (DDL)

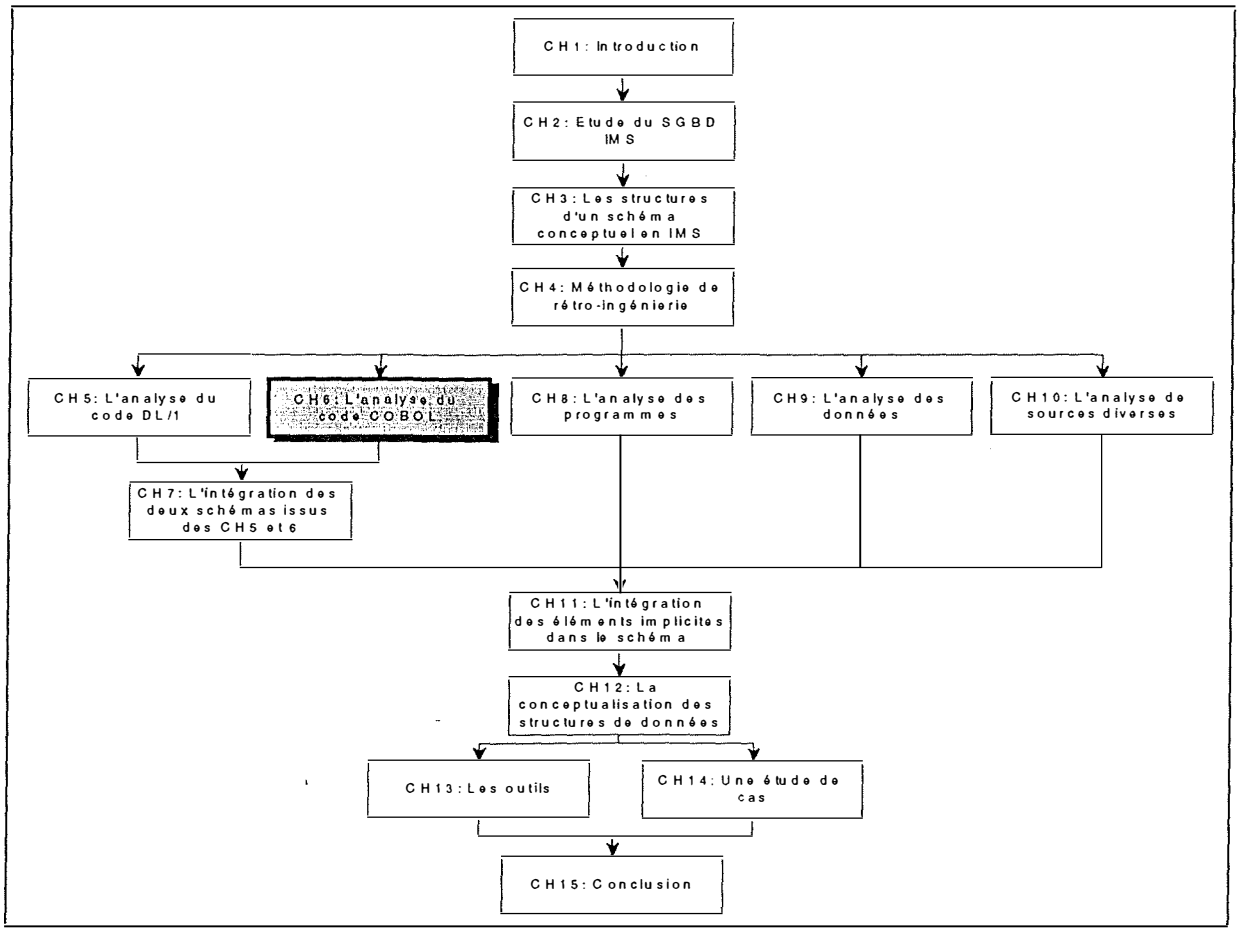


Figure 6-1: Le chapitre 6

6.1. Introduction: la notion de COPYBOOK

Comme nous l'avons déjà expliqué, il est possible que le code DL/1, qui décrit une base de données IMS, soit incomplet quant à la description des champs d'un type de segment. Dans ce cas, la description complète des champs d'un type de segment est faite dans un fichier (que nous appellerons dans la suite un "COPYBOOK"), qui est une déclaration COBOL habituelle. Ainsi, l'expérience vécue chez d'Ieteren nous a montré que la plupart des champs d'un type de segment n'étaient pas déclarés au niveau du Data Description Language, le DL/1. En fait, dans l'application étudiée chez d'Ieteren, on ne décrit dans le DL/1 que les informations nécessaires pour organiser les données, comme la taille d'un type de segment, l'identifiant d'un type de segment, les index. Donc, la description complète des champs d'un type de segment n'est pas vitale, obligatoire au niveau du DL/1[1]. C'est pourquoi on peut créer un fichier par type de segment qui décrit la structure complète de celui-ci. Ce fichier est appelé un COPYBOOK. L'exemple de la Figure 6-2 nous montre comment se présente un COPYBOOK.

```
05 ADLINE-KEY.
  10 ADLINE-CSEQNO     PIC 9(5).
05 ADLINE-CUSTNO       PIC X(2).
05 ADLINE-DELADNO      PIC 9(2).
```

Figure 6-2: Un exemple de COPYBOOK

Ces COPYBOOKS seront utilisés par les programmes d'application via l'instruction COPY[2], qui fera d'un COPYBOOK une variable conventionnelle au niveau du programme. Cette pratique offre certains avantages, comme de pouvoir modifier la structure d'un type de segment sans toucher au code DL/1, cela uniquement si on ne dépasse pas les bornes fixées pour ce type de segment dans le DL/1.
Evidemment, le recours à des COPYBOOKS n'a rien d'obligatoire; tous les champs d'un type de segment peuvent être décrits au niveau du DL/1.
Il se peut aussi qu'un COPYBOOK pour un type de segment ait une structure contradictoire par rapport à la description faite dans le DL/1 correspondant. L'exemple de la Figure 6-3 décrit une telle situation. Le DL/1 (colonne de gauche) décrit une commande, le COPYBOOK correspondant (colonne de droite) les détails d'une commande. C'est le champ RECTYPE qui permettra de savoir si on a affaire à une commande ou un détail de commande. Nous reparlerons en détail de ces situations contradictoires dans le chapitre sept.

[1] Il suffit en effet de donner la taille du type de segment.
[2] On peut comparer cette instruction COPY à l'instruction INCLUDE en C.

Déclaration DL/1 du type de segment ORDER	COPYBOOK ORDER_LINE, correspondant au type de segment ORDER
SEGM NAME=ORDER, BYTES=30, PARENT=CUSTOMER FIELD NAME=(NUM,SEQ,U), BYTES=10, START=1 FIELD NAME=CUS_CODE, BYTES=10, START=11 FIELD NAME=RECTYPE, BYTES=1, START=21	05 ORD_NUM PIC 9(10). 05 STK_CODE PIC 9(5). 05 ORD_QTY PIC 9(5). 05 RECTYPE PIC X(1).

Figure 6-3: Exemple de contradiction entre un type de segment et son COPYBOOK

Dans ce chapitre six, nous allons dans un premier point expliquer comment se présente les déclarations de données en COBOL, puisqu'elles sont le coeur d'un COPYBOOK. Dans un deuxième point, nous préciserons quels peuvent être les apports des COPYBOOKS au schéma.

Remarque: Dans le texte qui suit, nous allons parfois utiliser des descriptions syntaxiques d'instructions COBOL. Les conventions sont les mêmes que dans le chapitre précédent. Pour rappel:

⇒ Les mots en lettres capitales sont des mots réservés à COBOL.
⇒ Les mots en lettres minuscules sont des noms donnés par l'utilisateur.
⇒ [] indique que ce qui est entre crochets est facultatif.
⇒ { } indique qu'un choix de paramètres doit être fait.
⇒ Lorsqu'on devra faire un choix de paramètres, le paramètre par défaut sera souligné.

6.2. Le format général des COPYBOOKS[3]

Un COPYBOOK est un fichier contenant une série de déclarations de variables COBOL. Dans ce paragraphe, nous donnons donc le format général d'une déclaration d'une donnée en COBOL, à la Figure 6-4.

```
(a) déclaration d'une donnée:
                    numero-de-niveau  [nom-de-donnee]
                              [REDEFINES]
                              [[IS] GLOBAL]
                              [[IS] EXTERNAL]
                              [OCCURS]
                              [PIC[TURE]]
                              [BLANK WHEN ZERO]
                              [JUSTIFIED RIGHT]
                              [USAGE]
                              [SIGN]
                              [SYNCHRONIZED]
                              [VALUE]
(b) déclaration d'une chaîne de caractères:
                    66 [nom-de-donnee]
                              RENAMES.
(c) déclaration d'un nom de condition:
                    88 [nom-de-condition]
                              VALUE[S].
```

Figure 6-4: Formats généraux des déclarations de données en COBOL

Une première remarque concernant la Figure 6-4, est que le numéro de niveau, le nom de la donnée et la clause REDEFINES doivent être, dans l'ordre, les premiers éléments de la rubrique. Les autres clauses peuvent figurer dans un ordre quelconque.

La partie (a) de la Figure 6-4 déclare une donnée.

⇒ Le *numero-de-niveau* situe cette donnée dans la structure logique.

⇒ Le *nom-de-donnee* est le nom déclaré pour la donnée.

⇒ La clause REDEFINES signale que la description en cours redéfinit, par une nouvelle structure logique, une structure déjà définie.

⇒ La clause GLOBAL signale que les noms déclarés sont globaux.

⇒ La clause EXTERNAL signale que la structure de données est commune à plusieurs programmes.

⇒ La clause OCCURS signale que la donnée forme une table.

⇒ La clause PICTURE détermine le type et la longueur de la donnée.

⇒ La clause BLANK WHEN ZERO détermine un format particulier pour les données numériques: mise à blanc de la donnée quand sa valeur numérique est zéro.

⇒ La clause JUSTIFIED RIGHT inverse le sens du cadrage pour une donnée non numérique.

⇒ La clause technique USAGE détermine le système de codification de la donnée.

⇒ La clause technique SIGN détermine le mode de représentation du signe algébrique dans une donnée numérique.

⇒ La clause technique SYNCHRONIZED détermine l'alignement de la donnée.

[3] Tous les formats COBOL de ce chapitre sont issus de **[CLARINVAL,91]**.

⇒ La clause VALUE assigne une valeur initiale à la donnée.

La partie (b) de la Figure 6-4 redéfinit sous forme d'une chaîne de caractères une donnée déjà définie.
La partie (c) de la Figure 6-4 associe à une donnée un *nom-de-condition* identifiant un sous-ensemble des valeurs possibles de cette donnée.

A partir de ce format de déclaration COBOL, nous pouvons extraire des informations utiles pour construire notre schéma issu des COPYBOOKS.

6.3. L'apport des COPYBOOKS au schéma

Comme nous l'avons déjà dit, un COPYBOOK peut contenir la description complète des champs du type de segment auquel il correspond. Il peut donc compléter utilement les informations apportées par le DL/1. Néanmoins, il peut aussi affiner les informations découvertes dans le DL/1 (répétitivité de champs, décomposition de champs).

L'apport des COPYBOOKS au schéma est le suivant:

1. Le nom et le niveau des champs
2. Le type et la longueur (physique et logique) des champs
3. La répétitivité des champs
4. La redéfinition structurée d'un espace déjà défini: la clause REDEFINES
5. La redéfinition d'une chaîne de caractères: la rubrique 66, clause RENAMES
6. La rubrique 88

Pour chaque apport, nous expliquerons comment l'information trouvée dans un COPYBOOK peut être traduite dans le schéma.

Remarque: Notre méthodologie implique que le rétro-ingénieur crée un schéma qui englobera les apports de l'ensemble des COPYBOOKS existant pour une BD donnée. Le rétro-ingénieur devra donc s'assurer qu'il a tous les COPYBOOKS[4] pour une BD donnée à sa disposition. De ce fait, nous aurons d'une part, le schéma issu de l'analyse du DL/1 et d'autre part celui issu de l'analyse des COPYBOOKS. Il nous suffira alors de les intégrer en un troisième pour avoir un premier schéma logique enrichi complet.

Remarque: Comme un COPYBOOK contient la description d'un type de segment donné, il convient de créer dans le schéma un type d'entité pour chaque COPYBOOK. Il faudra donc créer autant de types d'entités dans le schéma qu'il y a de COPYBOOKS à analyser.

6.3.1. Le nom et le niveau des champs

En COBOL, toute variable doit être située dans un ensemble structuré. En outre, elle est identifiée par un nom. Pour plus de détails syntaxiques, voyez **[CLARINVAL, 91]**.

A. Brève syntaxe

numero-de-niveau [nom-de-donnee]

Figure 6-5: Format du niveau et du nom pour les données

La Figure 6-6 donne un exemple de la structuration des données en COBOL.

[4] Nous ne voulons pas dire par « tous les COPYBOOKS » qu'à chaque type de segment d'une BD correspond nécessairement un COPYBOOK. On veut juste dire que le rétro-ingénieur doit reprendre tous les COPYBOOKS existant pour une BD donnée, car ils sont susceptibles de lui apporter de l'information.

```
05 PERSONNE
  10 NOM
  10 PRENOM
  10 DATE-NAISSANCE
      15 ANNEE
      15 MOIS
      15 JOUR
  10 SEXE
05 VOITURE
```

Figure 6-6: Exemple de la structuration des données en COBOL

B. Apport

Le niveau et le nom d'une donnée sont évidemment deux informations importantes apportées par les COPYBOOKS. Le numéro de niveau permet de structurer les données dans le schéma, c'est-à-dire de créer des attributs décomposables quand il le faut. Le nom de la donnée devient dans le schéma créé le nom d'un attribut. Remarquons dès à présent qu'il se peut qu'un champ déclaré dans le DL/1 soit redéclaré dans le COPYBOOK avec un nom différent de celui utilisé dans le DL/1. Ce problème ne nous intéresse pas pour le moment mais devra être réglé dans le chapitre sept, lorsque nous aborderons l'intégration des deux schémas, le premier issu de l'analyse du DL/1, le deuxième issu de l'analyse des COPYBOOKS correspondants.

Remarque: La déclaration des champs dans un COPYBOOK n'est pas nécessairement la plus fine, c'est-à-dire que certains champs pourraient être déclarés comme atomiques dans le COPYBOOK, mais que l'on découvre par la suite par une analyse des programmes que ce sont des champs décomposables. La structure n'est donc pas nécessairement définitive dès ce stade. Il faudra encore affiner le schéma obtenu par la recherche d'éléments implicites, sujet que nous aborderons dans les chapitres huit, neuf et dix.

Le schéma de la Figure 6-7 montre le résultat de l'analyse de la Figure 6-6.

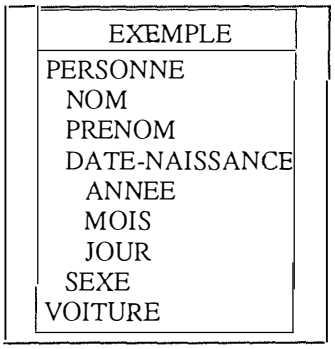


Figure 6-7: Résultat de l'analyse de la Figure 6-6

6.3.2. Le type et la longueur (physique et logique) des champs

Le type et la longueur logique des données élémentaires sont des informations fournies par la clause PICTURE.

A. Brève syntaxe

```
{PICTURE  PIC} [IS] image
```

Figure 6-8: Format de la clause PICTURE

B. Apport

Cette clause PICTURE permet d'ajouter à un attribut du schéma son type et sa longueur logique. Il nous faut cependant préciser notre propos, car l'apport sera plus ou moins important en fonction du type de la donnée.

En COBOL, trois types de données sont possibles:

- Les données numériques
- Les données alphabétiques
- Les données alphanumériques

B.1. Les données numériques

En COBOL, on peut compacter les données de type numérique, ce qui implique donc que la longueur logique d'un champ sera différente de sa longueur physique.
L'image d'une donnée numérique, c'est-à-dire l'expression de sa longueur logique, peut être d'une des formes suivantes (Figure 6-9).

```
[S]9(i)V9(j)
[S]9(i)[V]
[S]V9(j)
[S]9(i)P(j)[V]
[S][V]P(i)9(j)
```

Figure 6-9: Formats possibles pour une donnée numérique

Le symbole 'S' spécifie la donnée comme signée.
Le symbole '9' représente toute position occupée par un chiffre décimal.
Le symbole 'V' sépare les parties entière et fractionnaire.
Le symbole 'P' représente une position décimale implicite; les positions symbolisées par 'P' n'existent pas dans la représentation physique de la donnée mais elles sont occupées par des zéros dans la valeur de cette donnée pour les opérations de comparaison et d'affectation. Les symboles 'P' ne doivent donc pas être comptés dans la longueur physique d'une donnée.

La longueur logique d'une donnée numérique est égale au nombre de symboles '9' et 'P' indiqués par son image.
La longueur physique d'une donnée numérique dépend de la présence de la clause USAGE et du symbole 'P' dans la définition de la donnée. Cette longueur physique aura une valeur différente suivant les options de la clause USAGE utilisées. Voici le format de cette clause.

```
[ USAGE IS ]
        { COMPUTATIONAL
          COMP
          COMPUTATIONAL-1
          COMP-1
          COMPUTATIONAL-2
          COMP-2
          COMPUTATIONAL-3
          COMP-3
          DISPLAY
          INDEX
          POINTER
        }
```

Figure 6-10: Format de la clause USAGE

L'option COMP-1 est une abréviation pour COMPUTATIONAL-1. Il en va de même pour COMP-2 et COMP-3, respectivement pour les options COMPUTATIONAL-2 et COMPUTATIONAL-3.
Si les options COMP-1 et COMP-2 sont utilisées dans la description d'une donnée, la clause PICTURE ne peut être présente.
Les différentes options possibles pour la clause USAGE permettent de savoir quelle est la longueur physique de la donnée. Les tableaux 1 et 2 de l'annexe A montrent la valeur qu'aura la longueur physique d'une donnée suivant la clause USAGE utilisée.

Ainsi, dans l'exemple de la Figure 6-11, la longueur logique de IDENT est de 9 bytes. Si on se réfère à l'annexe A, tableau 1, sa longueur physique est de 5 bytes ((9+1)/2).

```
05 IDENT PIC 9(9) COMP-3
```

Figure 6-11: Exemple de calcul de la longueur logique et physique d'un champ

En conclusion, les déclarations COBOL permettent de différencier les longueurs physique et logique des données numériques. Dans le cas des données numériques, l'apport des clauses PICTURE et USAGE est donc triple: outre le type et la longueur logique, il y a aussi la longueur physique du champ. Deux cas sont possibles:

1. Les longueurs logiques et physiques sont identiques (pas de clause USAGE, ni de symboles P)
2. Les longueurs logiques et physiques sont différentes (clause USAGE, symbole 'P')

B.2. Les données alphabétiques

Le symbole 'A' représente toute position d'un caractère de l'alphabet. Les longueurs logique et physique d'une donnée alphabétique sont égales au nombre de symboles 'A' indiqués par son image.

B.3. Les données alphanumériques

Le symbole 'X' représente la position d'un caractère quelconque. Les longueurs logique et physique d'une donnée alphanumérique sont égales au nombre de symboles 'X' indiqués par son image.

La Figure 6-12 montre un exemple concernant le type et les longueurs des champs. La Figure 6-13 reprend pour chaque champ défini dans l'exemple de la Figure 6-12 son type et ses longueurs logique et physique.

```
05 IDENT PIC S9(9)V99 COMP-3
05 PERSONNE
  10 NOM PIC A(15)
  10 PRENOM PIC X(20)
  10 SEXE PIC X(1)
  10 AGE PIC 9(2)
```

Figure 6-12: Un exemple concernant le type et les longueurs physique et logique des champs

	Type	Longueur logique	Longueur physique
IDENT	numérique	11	6
NOM	alphabétique	15	15
PRENOM	alphanumérique	20	20
SEXE	alphanumérique	1	1
AGE	numérique	2	2

Figure 6-13: Tableau reprenant le type et les longueurs physique et logique des champs de la Figure 6-12

6.3.3. La répétitivité des champs

L'instruction OCCURS permet de déclarer une table et indique donc le caractère multivalué d'un champ.

A. Brève syntaxe

```
OCCURS entier [TIMES]
        [{ASCENDING  DESCENDING} [KEY IS] nom-de-donnee]
        [INDEXED [BY] nom-d-index]
```

Figure 6-14: Format de la clause OCCURS

Les mots-clés ASCENDING et DESCENDING spécifient comment la table est triée (ordre croissant ou décroissant). Ils indiquent aussi sur quelle donnée s'effectue ce tri. L'option INDEXED BY déclare les noms des index associés à la table.

La Figure 6-15 donne un exemple de l'utilisation de la clause OCCURS.

```
01 BAREME
 02 Poste-Baremique  OCCURS 20 times
        ASCENDING key is No-Bareme
  03 No-Bareme PIC 99
  03 Montant-Bareme OCCURS 5 times  PIC S9(6)
```

Figure 6-15: Exemple de clause OCCURS

B. Apport

Chaque fois qu'on trouve l'instruction OCCURS dans la déclaration d'un champ, on peut créer dans le schéma un attribut multivalué. Comme la clause OCCURS a pour effet de réserver en mémoire la longueur de la variable multipliée par la valeur de la clause, les bornes minimales et maximales de l'attribut multivalué seront égales à la valeur de la clause OCCURS. Si l'option de tri est rencontrée, on l'ajoute à la description technique de l'attribut. De même, si l'option INDEXED BY est présente, on ajoute les indices à la description technique de l'attribut.
La Figure 6-16 donne le résultat de l'analyse des déclarations de la Figure 6-15.

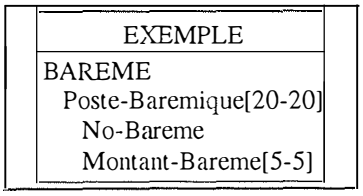


Figure 6-16: Résultat de l'analyse de la Figure 6-15

6.3.4. La redéfinition structurée d'un espace déjà défini: la clause REDEFINES

C'est la clause REDEFINES qui permet de redéfinir un champ.

A. Brève syntaxe

```
numero-de-niveau [sujet] REDEFINES objet
```

Figure 6-17: Format de la clause REDEFINES

Une rubrique contenant la clause REDEFINES déclare une donnée appelée *sujet* redéfinissant; la clause REDEFINES fait référence à une donnée appelée *objet* redéfini.
La clause REDEFINES signifie que la donnée *sujet* occupe le même emplacement que la donnée *objet*. On peut donner plusieurs redéfinitions du même objet. Objet et sujets doivent être déclarés sous le même numéro de niveau. On peut aussi fournir des redéfinitions partielles.
Pour plus de détails concernant les conditions d'utilisation de cette clause, consultez **[CLARINVAL, 91]**.

Le problème de la redéfinition abordé ici est identique à celui abordé dans le chapitre précédent, lorsque la combinaison des mots-clés BYTES et START nous permettait de découvrir des recouvrements de champs. Nous n'avions alors montré que deux possibilités de recouvrement, un conflictuel, l'autre non conflictuel, tout en ajoutant que d'autres possibilités seraient abordées au chapitre sept.
Nous procéderons de la même manière ici. Notons dès maintenant que le problème est un peu plus compliqué que dans le cas du DL/1: en COBOL, avec la clause REDEFINES, il est obligatoire que l'objet et les sujets soient déclarés sous le même numéro de niveau. Nous n'avions pas ce problème dans le DL/1.

En COBOL, on peut faire deux types de redéfinitions:
- La redéfinition totale: un champ peut être déclaré comme redéfinition d'un autre, pas forcément de même type, mais de même longueur. La Figure 6-18 montre un tel cas.

```
05 ADLINE-1-DATA PIC X(30).
05 ADLINE-2-DATA PIC 9(30) REDEFINES ADLINE-1-DATA.
```

Figure 6-18: Un exemple de redéfinition totale

Pour rappel, la clause REDEFINES de la Figure 6-18 signifie donc que *ADLINE-2-DATA* occupe le même emplacement que *ADLINE-1-DATA*.

- La redéfinition partielle: il se peut aussi que la redéfinition soit partielle. Ainsi, l'exemple de la Figure 6-19 montre que *ECRAN-2* redéfinit seulement partiellement *ECRAN-1*. En effet, la longueur totale du champ *ECRAN-1* est de 88, celle d'*ECRAN-2* étant seulement de 24.

```
05 ECRAN-1.
 10 IDENTITE.
   15 Nom        PIC X(24).
   15 Prenom     PIC X(20).
 10 Adresse.
  15 Rue         PIC X(24).
  15 No-Rue      PIC X(6).
 10 No-postal    PIC X(4).
 10 Localite     PIC X(20).

05 ECRAN-2 REDEFINES ECRAN-1.
 10 Domicile-banque.
   15 No-Banque PIC 999.
   15 No-compte PIC 9(7).
   15 cle-compte PIC 99.
 10 date-naissance.
```

```
 15 Jour          PIC 99.
 15 Mois          PIC 99.
 15 Annee         PIC 99.
10 Etat-civil     PIC X.
10 Pers-a-charge PIC 99.
10 No-bareme      PIC XXX.
```

Figure 6-19: Un exemple de redéfinition partielle

Il se peut aussi que l'on rencontre des situations plus compliquées. Imaginons que l'on soit en présence d'une structure comme celle de la Figure 6-20. Imaginons maintenant que l'on désire redéfinir les champs B2 et C1 en un troisième. Vu que l'objet et le sujet doivent être de même niveau, le seul moyen pour redéfinir les champs B2 et C1 est celui décrit à la Figure 6-21.

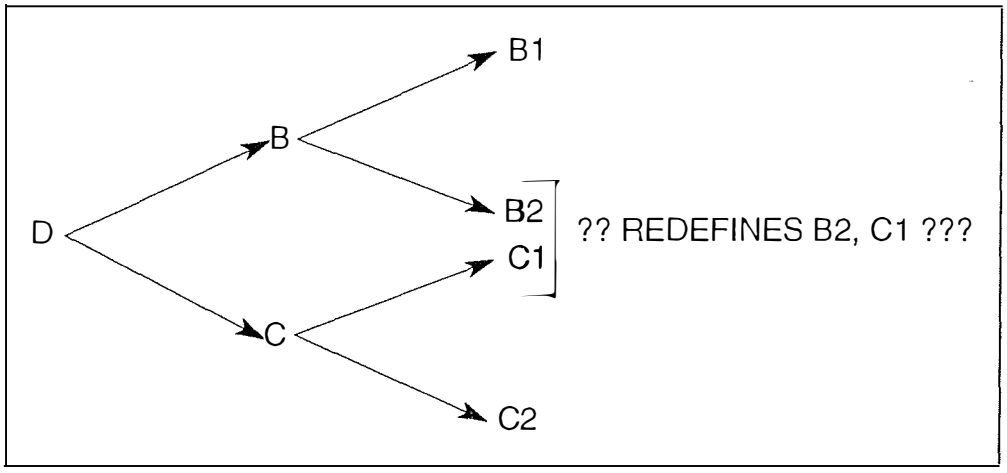


Figure 6-20: Cas du recouvrement de deux sous-champs par un troisième

```
05 D
 10 B
  15 B1 PIC X(5)
  15 B2 PIC X(5)
 10 C
  15 C1 PIC X(5)
  15 C2 PIC X(5)

05 E REDEFINES D
 10 FILLER PIC X(5)
 10 A
  15 A1 PICX(5)
  15 A2 PIC X(5)
 10 FILLER PIC X(5)
```

Figure 6-21: Redéfinition des champs B2 et C1

La Figure 6-22 est l'équivalent graphique de la Figure 6-21.

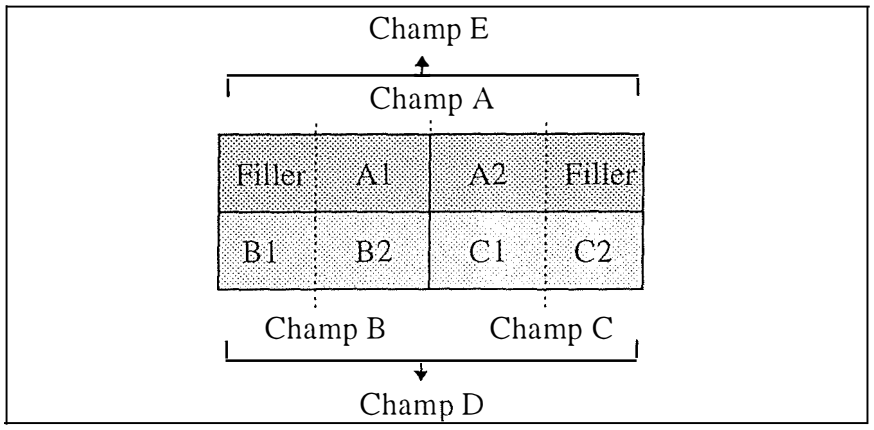


Figure 6-22: Equivalent graphique de la solution de la Figure 6-21

Nous avons donc créé un champ E de même longueur et de même niveau que le champ D. Nous avons ensuite créé un champ décomposable A dont le premier composant recouvre B2 et le second composant recouvre le champ C1.
Pour rappel, nous aborderons plus complètement ces situations de recouvrement dans le chapitre sept.

B. Apport

La clause REDEFINES signifie qu'un seul des deux champs intervenant dans la clause peut-être utilisé à la fois, car physiquement les champs consomment le même espace.
Chaque fois qu'on rencontre une clause REDEFINES dans une déclaration, on crée un groupe avec les champs objet et sujet comme composants, et une contrainte d'exactement-un concernant ce groupe. Cette contrainte modélise le fait que dans une même occurrence, on ne pourra avoir D et E, puisqu'ils consomment le même espace. Les composants du groupe deviennent facultatifs. Un champ redéfinissant devient donc un attribut comme un autre dans le schéma.
La Figure 6-23 représente le schéma issu de l'analyse de la Figure 6-21.

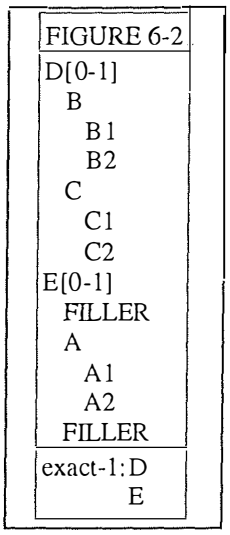


Figure 6-23: Résultat de l'analyse de la Figure 6-21

6.3.5. La redéfinition d'une chaîne de caractères: la rubrique 66, clause RENAMES

La clause RENAMES, rédigée dans une rubrique spéciale, redéfinit sous la forme d'une chaîne de caractères une partie de l'espace occupé par un article.

A. Brève syntaxe

```
66 nom-de-donnee-1
        RENAMES nom-de-donnee-2 [THRU nom-de-donnee-3].
```

Figure 6-24: Format de la rubrique 66

Une variable *nom-de-donnee-1* redéfinit une donnée *nom-de-donnee-2*. Elle peut aussi redéfinir un ensemble de données contiguës, qui sont définies entre *nom-de-donnee-2* et *nom-de-donnee-3*.
Contrairement à la clause REDEFINES, les données décrites dans la rubrique 66 ne peuvent être décomposées, structurées. Par cette rubrique 66, on peut renommer toute donnée, quel que soit son niveau, ce qui n'est pas le cas avec la clause REDEFINES. Il faut également que la première position de *nom-de-donnee-2* soit inférieure à la première position de *nom-de-donnee-3*, et que la dernière position de *nom-de-donnee-3* soit supérieure à la dernière position de *nom-de-donnee-2*.

Cette rubrique 66 aborde aussi le problème du recouvrement, qui sera abordé plus complètement au chapitre sept. Nous nous contentons ici de donner quelques exemples éclairants.

Le Figure 6-25 montre un exemple de trois rubriques 66.

```
02 No-postal.
  03 Post1 PIC X.
  03 Post2 PIC X.
  03 Post3 PIC XX.
66 Tri-principal RENAMES Post1.
66 Tri-secondaire RENAMES Post1 THRU Post2.
66 Distribution RENAMES Post1 THRU Post3.
```

Figure 6-25: Un exemple de l'utilisation de la clause RENAMES

Examinons la dernière clause 66 de la Figure 6-25. La donnée *Distribution* redéfinit l'ensemble des données définies à partir de la position de départ de *Post1*, jusqu'à la position de fin de *Post3*. Graphiquement, la situation sera celle de la Figure 6-26.

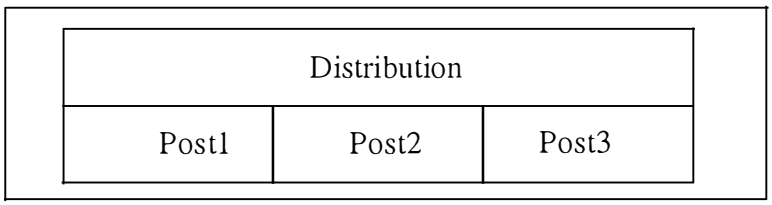


Figure 6-26: Situation graphique pour la déclaration <66 Distribution RENAMES Post1 THRU Post3>

B. Apport

L'apport de la rubrique 66 est comparable à l'apport de la clause REDEFINES. La seule différence est qu'une clause RENAMES peut redéfinir plusieurs données de la structure logique, contrairement à la clause REDEFINES.

Lorsqu'on rencontre une rubrique 66, clause RENAMES, notre comportement varie en fonction de l'utilisation ou non du mot-clé THRU.

- Si le mot-clé THRU n'est pas utilisé, la clause RENAMES ne concerne donc qu'un champ de la structure logique. Nous nous retrouvons donc dans une situation comparable à ce qui se passe avec la clause REDEFINES. De ce fait, une contrainte d'exactement-un est créée avec comme composants les deux attributs correspondant aux deux champs, à savoir le champ déclaré au niveau de la rubrique 66 et celui qu'il renomme. Les composants du groupe deviennent facultatifs.
- Si le mot-clé THRU est utilisé, la clause RENAMES concerne plus de deux champs de la structure logique. La première étape consiste donc à repérer quels sont les champs de la structure logique concernés et à regrouper les attributs correspondants dans une contrainte de coexistence, ce qui implique qu'ils deviennent facultatifs. La deuxième étape consiste à construire une contrainte d'exactement-un avec comme composants le groupe de coexistence qui vient d'être créé et l'attribut correspondant au champ déclaré au niveau 66.

La Figure 6-27 représente le schéma issu de l'analyse de la Figure 6-25.

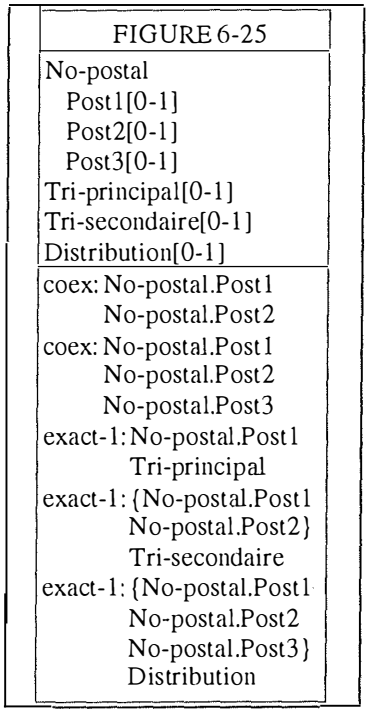


Figure 6-27: Résultat de l'analyse de la Figure 6-25

6.3.6. La rubrique 88

Une rubrique du niveau spécial 88 associe à une donnée un nom de condition grâce auquel son contenu pourra être testé.

A. Brève syntaxe

```
88 nom-de-condition
        {VALUE [IS]  VALUES [ARE]}  constante-1 [THRU constante-2]
```

Figure 6-28: Format de la rubrique 88

La clause VALUE(S) identifie un sous-ensemble de valeurs possibles pour la donnée. Ce sous-ensemble est formé de la réunion de toutes les valeurs citées. Chaque indication *constante-1 THRU constante-2* identifie un intervalle fermé dans l'ensemble ordonné des valeurs possibles. Ainsi, dans l'exemple ci-dessous, *ADLINE-NUMBER* peut prendre des valeurs entre 40 et 79. L'utilisation par un programme de *nom-de-condition* équivaut à un test d'appartenance de la valeur actuelle de la donnée à l'ensemble de valeurs identifié.

La Figure 6-29 montre un exemple de l'utilisation de la clause 88.

```
05 ADLINE-PRIORITY  PIC X.
 88 ADLINE-PRIORITY-ORDER  VALUE 'Y'
 88 ADLINE-FIXED-PRIORITY    VALUE 'F'
 88 ADLINE -NUMBER VALUES ARE 40 THRU 79
```

Figure 6-29: Un exemple de clauses 88

B. Apport

Lorsque nous rencontrons un niveau 88, nous ajoutons à la description technique de l'attribut correspondant les valeurs qu'il peut prendre.

6.4. Conclusion

Ce deuxième point nous a permis d'introduire la notion de COPYBOOK, qui est une notion très courante dans les systèmes IMS. L'analyse de ces COPYBOOKS nous a permis de créer un nouveau schéma; ce schéma complète le schéma issu de l'analyse du code DL/1. Vu que ces deux schémas modélisent la même situation, il peut paraître évident de rassembler toutes ces informations dans un seul schéma. Le chapitre suivant s'y attache.

CHAPITRE 7: L'intégration des deux schémas issus des analyses des codes DL/1 et COBOL

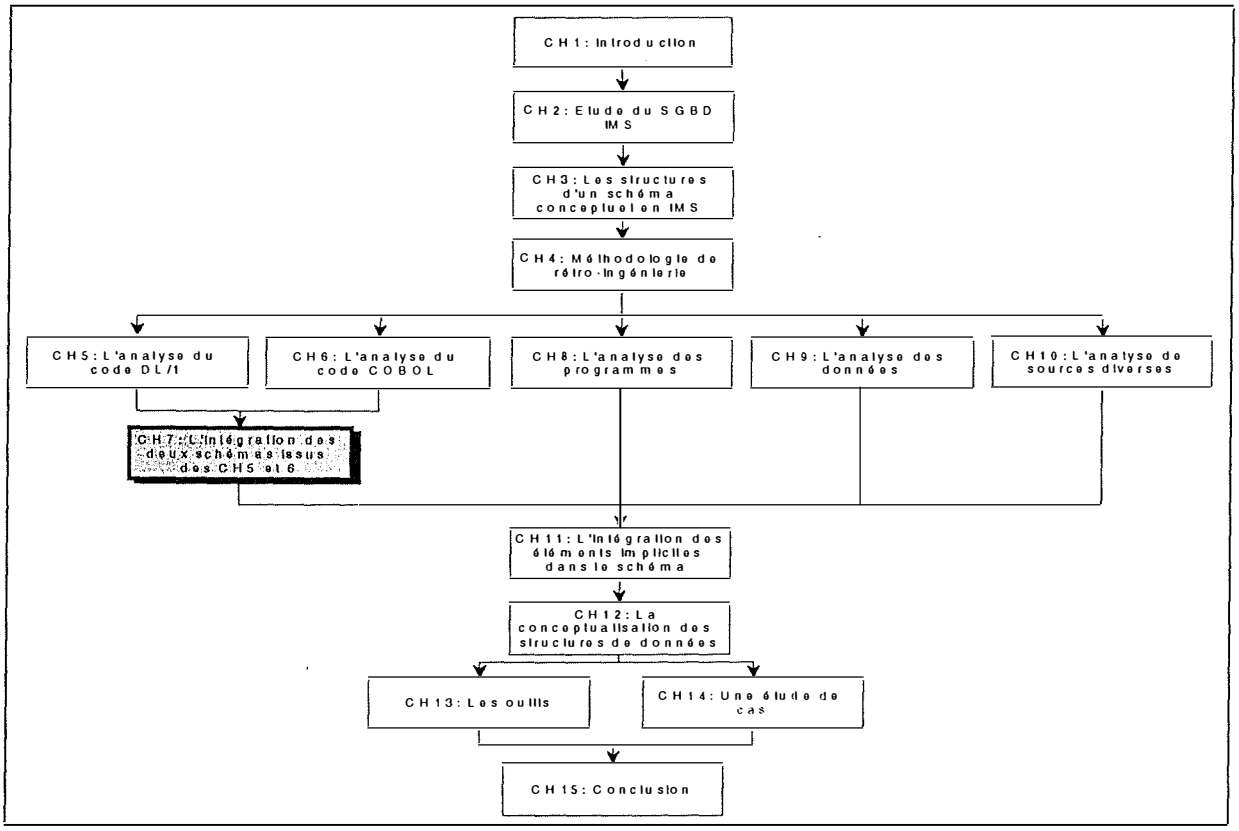


Figure 7-1: Le chapitre 7

7.1. Introduction

Les étapes d'analyse du code DL/1 et des COPYBOOKS correspondant aux types de segments définis dans le DL/1 fournissent deux schémas distincts. Néanmoins, ces deux schémas concernent la même base de donnée, puisque les COPYBOOKS sont la description complète de certains types de segments de la base de données. Il nous faut donc regrouper l'information présente dans ces deux schémas dans un troisième.

Ce chapitre abordera d'abord quelques généralités concernant l'intégration des deux schémas. Ensuite, comme nous l'avions mentionné dans les chapitres cinq et six, nous aborderons de façon approfondie les problèmes de redéfinition de champs ou de types de segments. Nous distinguerons les situations conflictuelles et les situations non conflictuelles. Nous consacrerons aussi un paragraphe au cas particulier de la redéfinition d'un type de segment par plusieurs COPYBOOKS. Le paragraphe 7.5 donnera un exemple complet d'intégration. Enfin, nous conclurons brièvement ce chapitre.

7.2. Généralités

Etant en présence de deux schémas qui concernent la même base de données, l'intégration consiste à générer à partir de ces deux schémas un troisième, tel que ce nouveau schéma englobe toute l'information de ses deux schémas sources. Autrement dit, nous ne pouvons pas perdre de l'information en intégrant les deux schémas, ni générer des doublons. Il y aura deux 'types' d'informations à intégrer:

- Les informations présentes dans les deux schémas; dans ce cas, il faudra veiller à ne pas avoir de doublons, de redondance dans le schéma intégré.
- Les informations présentes uniquement dans un des deux schémas; dans ce cas, il faut prendre en compte cette information.

Formalisons un peu plus cette intégration:

<u>Etape 1:</u> Notre schéma de base est celui issu de l'analyse des COPYBOOKS. En effet, nous considérons d'une part que les noms de champs donnés au niveau des COPYBOOKS sont plus pertinents que ceux issus de l'analyse du DL/1, d'autre part qu'il y a de fortes chances que le COPYBOOK soit plus complet quant à la structure du type de segment qu'il décrit. Nous recopions donc le schéma issu de l'analyse des COPYBOOKS. Nous avons donc le squelette du schéma intégré.

<u>Etape 2:</u> Une fois le schéma issu de l'analyse des COPYBOOKS recopié, nous pouvons y intégrer toutes les informations issues de l'analyse du DL/1 qui n'y sont pas encore présentes.

Le schéma issu de l'analyse du DL/1 pourrait être choisi aussi comme schéma de base.

Par l'intégration décrite ci-dessus, nous obtenons un schéma non redondant et complet. Il n'y a aucune perte d'information.

La description ci-dessus ne tient pas compte d'une situation particulière: il se peut qu'entre la définition d'un type de segment dans le DL/1 et son COPYBOOK, il y ait des différences au niveau de la structure des champs, donc au niveau des longueurs physiques. Ces différences peuvent être conflictuelles ou non. Le point suivant aborde ce problème important.

Remarque: Le schéma intégré n'est plus un schéma à 100% conforme à IMS puisqu'il englobe les constructions trouvées dans les COPYBOOKS et qui ne sont pas conformes à IMS. On pense notamment aux attributs décomposables, facultatifs et multivalués.

7.3. Le problème des redéfinitions[1]

Comme nous l'avons déjà dit, il se peut que la définition, la structure interne d'un type de segment dans le DL/1 soit différente de celle de son COPYBOOK. Un exemple d'une telle situation a été donné à la Figure 6-3 du chapitre six.
Ce paragraphe a pour but d'approfondir ce problème et de proposer des solutions au rétro-ingénieur.
Il existe deux types de situations: les situations non conflictuelles et les situations conflictuelles.

7.3.1. Les situations non conflictuelles

Dans ce premier paragraphe, nous allons examiner un ensemble de situations non conflictuelles qui peuvent survenir entre la structure d'un type de segment défini dans le DL/1 et son COPYBOOK. Le but de ce paragraphe est de montrer ce que l'on peut faire lorsque l'on se trouve en présence d'une telle situation. Il convient de remarquer que dans la suite, nous raisonnerons sur les attributs des schémas générés, mais que le terme 'attribut' peut être remplacé par celui de 'type d'entité'. Autrement dit, toutes les solutions présentées ci-dessous sont également valables dans le cadre des définitions des types d'entités. De même, ces problèmes de redéfinitions peuvent survenir à l'intérieur même d'une déclaration DL/1 ou COBOL.

A. Situation 1

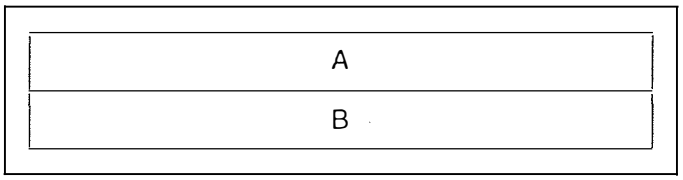


Figure 7-2: Situation 1

Ce cas est le plus simple auquel le rétro-ingénieur peut être confronté. En effet, les deux définitions qui doivent être intégrées possèdent la même taille et débutent à la même position. Elles sont donc en tout point identiques. La représentation de cette situation est simple; on crée dans le schéma intégré un type d'entité et un attribut. Le nom de cet attribut peut être l'un des deux noms des attributs initiaux ou peut être choisi de manière à rappeler que l'attribut est le fruit d'une intégration. Néanmoins, comme nous l'avons déjà dit, nous faisons le choix de garder le nom utilisé dans le schéma issu de l'analyse des COPYBOOKS, supposons *A*.

Figure 7-3: Résultat de l'intégration de la Figure 7-2

On suppose dans la Figure 7-3 que les deux attributs *A* et *B* représentent la même information. Dans le cas contraire, il faudrait créer une contrainte d'exactement-un entre ces deux attributs, qui deviendraient facultatifs.

[1] Ce point aborde les problèmes vus brièvement dans le chapitre cinq, paragraphe 5.2.5 et dans le chapitre six, paragraphes 6.3.4 et 6.3.5.

B. Situation 2

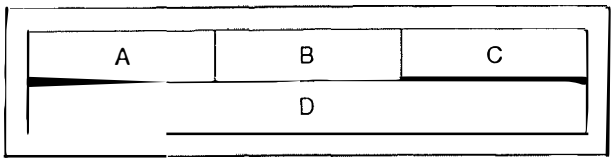


Figure 7-4: Situation 2

Dans cette situation, trois attributs (*A*,*B*,*C*) sont redéfinis par un seul (*D*). La longueur de l'attribut *D* est égale à la somme des longueurs des attributs *A*, *B*, *C*.
Deux possibilités s'offrent au rétro-ingénieur. Il peut d'abord considérer que cette situation représente un attribut décomposable *D* dont les composants sont *A*, *B* et *C*. Cette solution est décrite à la Figure 7-5. Le rétro-ingénieur considère donc que dans tous les cas, les deux types d'entités représentent la même information. Il doit s'en être assuré.

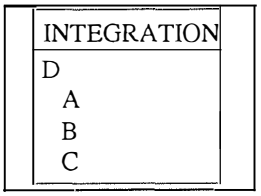


Figure 7-5: Résultat de l'intégration de la Figure 7-4

Il peut également englober les deux définitions au sein d'un seul et même schéma. Cette possibilité est exposée à la Figure 7-6. Dans ce cas, il considère que les deux découpes représentent deux informations différentes usant le même espace physique.

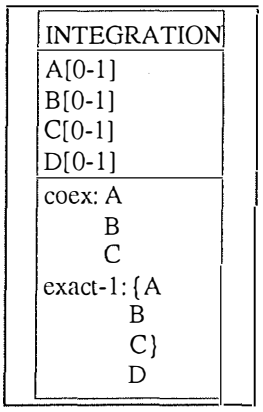


Figure 7-6: Résultat de l'intégration de la Figure 7-4

Nous pouvons remarquer, dans la Figure 7-6, une contrainte de coexistence entre les trois attributs de la première définition. Cela signifie que si un de ces attributs existe, alors ils existent tous. La seconde contrainte est une contrainte d'exactement-un entre le groupe formé par les attributs *A*, *B* et *C* et l'attribut *D*. Cela signifie qu'à tout moment, on peut être soit dans le cas de la définition en trois attributs, soit dans le cas de la définition comprenant un seul attribut. Il y a toujours une et une seule définition qui est utilisée.

C. Situation 3

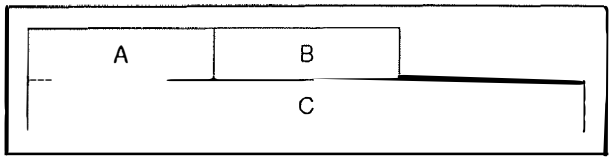


Figure 7-7: Situation 3

La situation 3 est identique à la situation 2 à ceci près que la seconde définition est plus longue que la première. Il est possible d'intégrer cette situation de deux manières.
La première solution consiste à considérer que la première définition est complétée par un champ *filler*. Cet attribut n'existe pas en réalité et peut être ajouté afin de conserver la cohérence des longueurs. Ainsi, les deux définitions possèdent la même longueur. La représentation de cette situation se trouve à la Figure 7-8. La situation est identique à la précédente (l'attribut *C* est remplacé par l'attribut *filler*).

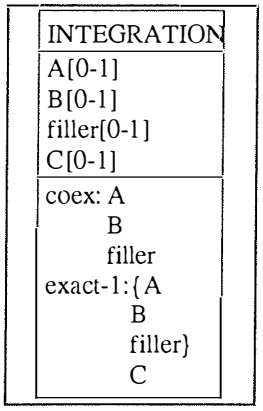


Figure 7-8: Résultat de l'intégration de la Figure 7-7

La seconde possibilité consiste à ne pas ajouter d'attribut supplémentaire. Il y a donc deux contraintes identiques aux précédentes. Le problème de cette solution est la longueur. En effet, si on examine la Figure 7-7, on pourrait croire que suivant la situation, les longueurs sont différentes. Or, un type d'entité avec deux longueurs différentes n'est pas envisageable. Il suffit de donner la plus grande longueur au type d'entité *INTEGRATION*, ici la longueur de l'attribut *C*.

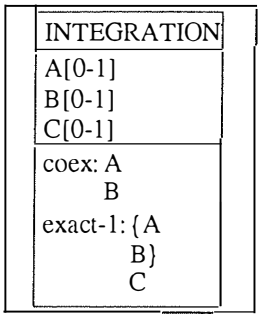


Figure 7-9: Résultat de l'intégration de la Figure 7-7

D. Situation 4

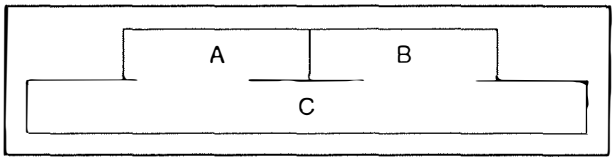


Figure 7-10: Situation 4

Cette situation est encore plus complexe que la précédente. En effet, les deux définitions ne sont pas de mêmes longueurs et de plus elles ne débutent pas à la même position.
La Figure 7-11 représente la première solution pour le rétro-ingénieur. Elle consiste à ajouter deux attributs fictifs *filler* qui ne sont présents que pour assurer l'adéquation des longueurs des deux définitions.

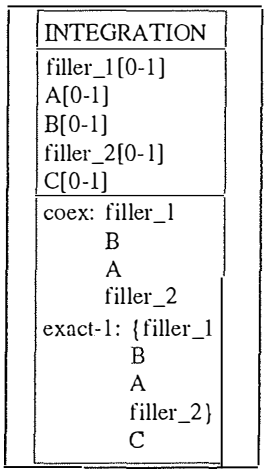


Figure 7-11: Résultat de l'intégration de la Figure 7-10

La Figure 7-12 représente une deuxième solution. Elle pose de nouveau le problème des positions physiques (absence d'attributs *filler*). On donne au type d'entité la longueur de *C* dans tous les cas.

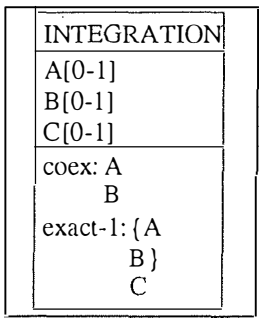


Figure 7-12: Résultat de l'intégration de la Figure 7-10

E. Situation 5

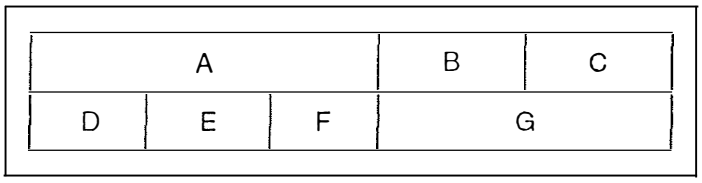


Figure 7-13: Situation 5

Cette situation est en fait identique à la situation 2. En effet, la situation des attributs *B*, *C* et *G* est identique à la situation 2. De la même façon la situation des attributs *A*, *D*, *E* et *F* est aussi identique à la situation 2. Elle sera donc traitée de la même manière. Nous obtenons alors les solutions des Figures 7-14 et 7-15.

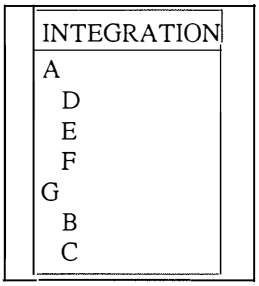


Figure 7-14: Résultat de l'intégration de la Figure 7-13

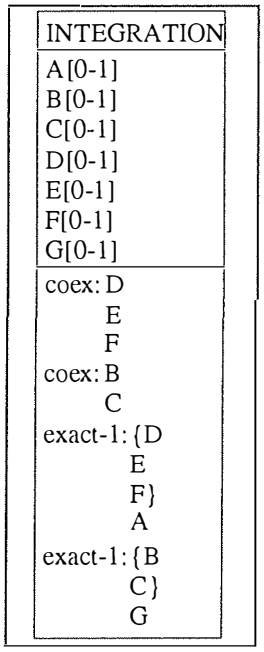


Figure 7-15: Résultat de l'intégration de la Figure 7-13

7.3.2. Les situations conflictuelles

Dans ce paragraphe, nous étudions de plus près trois situations conflictuelles.

A. Situation 6

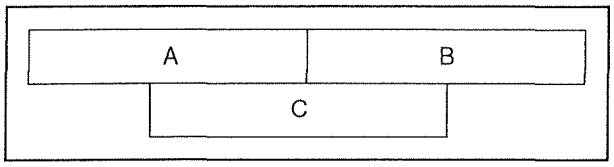


Figure 7-16: Situation 6

Dans la situation 6, nous sommes en présence d'un chevauchement d'attributs. L'attribut *C* reprend une partie de l'attribut *A* et une partie de l'attribut *B*. Deux intégrations sont possibles pour représenter une telle situation.

La première solution est représentée à la Figure 7-17. Nous ajoutons de nouveau deux attributs facultatifs *filler_1* et *filler_2*. Nous sommes alors en présence de deux groupes

d'attributs coexistants: *{A,B} et {filler_1, C, filler_2}*. Il y a également une contrainte d'exactement-un entre ces deux groupes.

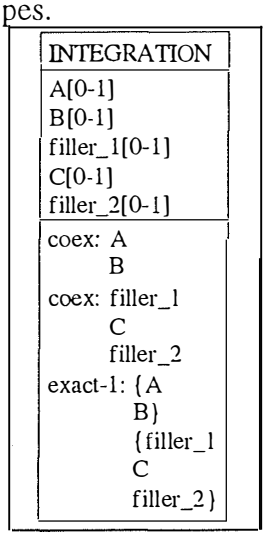


Figure 7-17: Résultat de l'intégration de la Figure 7-16

La seconde solution est à la Figure 7-18 ci-dessous. Elle pose le problème des longueurs physiques. Elle suppose que la longueur des attributs *A* et *B* est la seule possible.

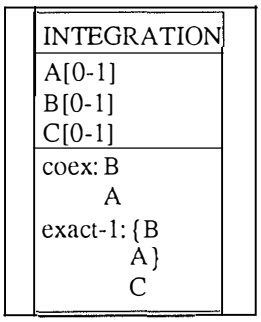


Figure 7-18: Résultat de l'intégration de la Figure 7-16

B. Situation 7

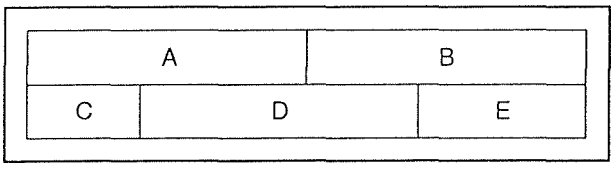


Figure 7-19: Situation 7

Les deux définitions possèdent la même longueur physique. Cependant, les deux découpes ne sont pas compatibles.

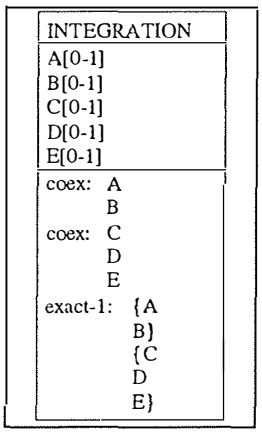


Figure 7-20: Résultat de l'intégration de la Figure 7-19

La solution de la Figure 7-20 consiste comme pour les autres cas à créer deux groupes d'attributs et à les lier par une contrainte d'exactement-un.

C. Situation 8

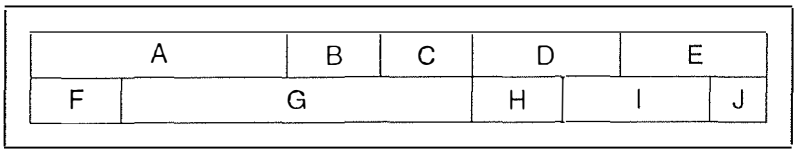


Figure 7-21: Situation 8

La situation 8 de la Figure 7-21 est évidemment conflictuelle, vu qu'on y trouve plusieurs chevauchements. La solution pour intégrer une telle situation (Figure 7-22) est inspirée des précédentes.

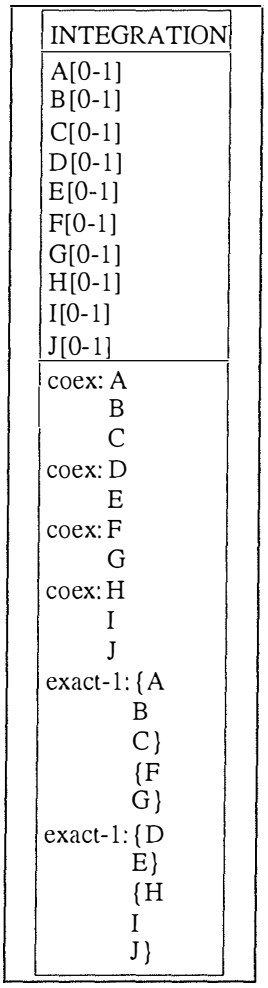


Figure 7-22: Résultat de l'intégration de la Figure 7-21

7.3.3 Un exemple concret

Nous donnons ci-dessous (Figures 7-23 et 7-24) un exemple de contradictions entre deux schémas qui représentent la même situation. Ici, nous avons trouvé un type de segment *COMMANDE* dans le code DL/1, avec quatre attributs. Nous avons également trouvé un COPYBOOK correspondant à ce type de segment, avec aussi quatre attributs, mais dont les longueurs physiques ne correspondent pas aux longueurs physiques des attributs issus du code DL/1. En fait, un même type de segment représente à la fois une entête de commande et une ligne de commande, d'où des différences au niveau des attributs.

Schéma issu de l'analyse du code DL/1

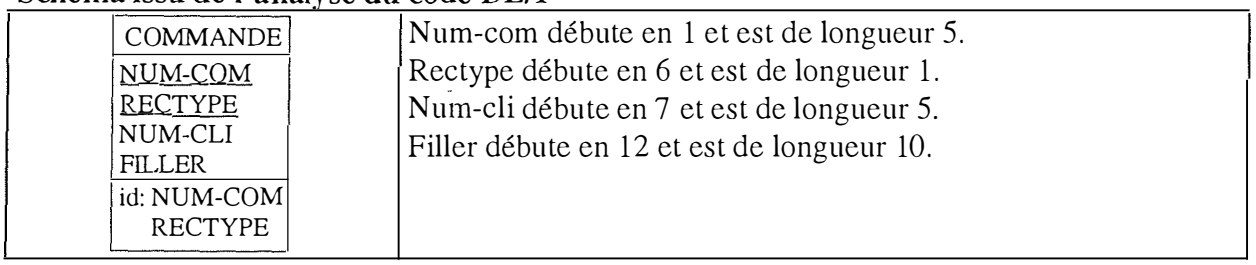


Figure 7-23: Schéma représentant une entête de commande

Schéma issu de l'analyse du code COBOL (COPYBOOK)

COMMANDE NUM-COM RECTYPE NUM-PROD FILLER		Num-com débute en 1 et est de longueur 5. Rectype débute en 6 et est de longueur 1. Num-prod débute en 7 et est de longueur 7. Filler débute en 14 et est de longueur 8.

Figure 7-24: Schéma représentant une ligne de commande

La Figure 7-25 montre l'agencement des attributs des deux types d'entités décrits aux Figures 7-23 et 7-24.

Ligne de Commande	Num-com	Re.	Num-prod	Filler
Commande	Num-com	Re.	Num-cli	Filler

Figure 7-25: Agencement des attributs

La Figure 7-26 représente une solution possible d'intégration. Nous avons, pour plus de clarté, décidé de regrouper en deux attributs décomposables les attributs se chevauchant.

COMMANDE
NUM-COM RECTYPE EN-TETE[0-1] NUM-PROD[0-1] FILLER[0-1] LIGNECOM[0-1] NUM-CLI[0-1] FILLER[0-1]
id: NUM-COM RECTYPE coex: EN-TETE.NUM-PROD EN-TETE.FILLER coex: LIGNECOM.NUM-CL LIGNECOM.FILLER exact-1: EN-TETE LIGNECOM

Figure 7-26: Exemple d'intégration des Figures 7-23 et 7-24

7.4. La redéfinition d'un type de segment par plusieurs COPYBOOKS

Ce paragraphe aborde un problème particulier rencontré un grand nombre de fois dans l'application étudiée chez d'Ieteren. Ce problème est qu'il se peut qu'un type de segment défini dans le DL/1 soit redéfini un certain nombre de fois dans les COPYBOOKS. Autrement dit, à un même type de segment correspond plusieurs COPYBOOKS.

Par exemple, soit *MEMBRE* un type de segment extrait du code DL/1, de longueur 70.
soient *JOUEUR* et *ARBITRE* deux COPYBOOKS de longueur 70 qui représentent tous les deux le type de segment *MEMBRE*. Ces deux COPYBOOKS sont des redéfinitions du type de segment *MEMBRE*.

Autrement dit, nous avons dans le schéma issu du code DL/1 un type d'entité *MEMBRE*, et dans le schéma issu des deux COPYBOOKS deux types d'entités *JOUEUR* et *ARBITRE*. Quand nous rencontrons une telle situation, nous procédons en deux étapes:

- la première étape consiste à créer un type d'entité *générique* au niveau du schéma issu de l'analyse des COPYBOOKS. Cette étape est représentée à la Figure 7-27. Nous créons pour chaque redéfinition (les types d'entités *JOUEUR* et *ARBITRE*) un attribut facultatif qui sera la représentation de la redéfinition. Une contrainte d'exactement-un est créée entre les attributs facultatifs créés. En effet, nous ne pourrons avoir en même temps qu'un seul des attributs facultatifs puisqu'ils consomment le même espace physique. Les attributs qui interviennent dans les deux redéfinitions sont extraits des attributs facultatifs créés et ajoutés au niveau du type d'entité générique (une seule fois). C'est le cas des attributs *KEY*, *NOM* et *PRENOM* dans l'exemple de la Figure 7-27.
- A la fin de la première étape, nous avons le schéma issu du DL/1 inchangé, tandis que celui issu des COPYBOOKS a été modifié, puisqu'il ne contient plus qu'un seul type d'entité regroupant les types d'entités *JOUEUR* et *ARBITRE*. Nous sommes donc dans la situation de base (à un type d'entité correspond SON COPYBOOK). Nous pouvons donc procéder à l'intégration standard décrite au paragraphe 7.2.

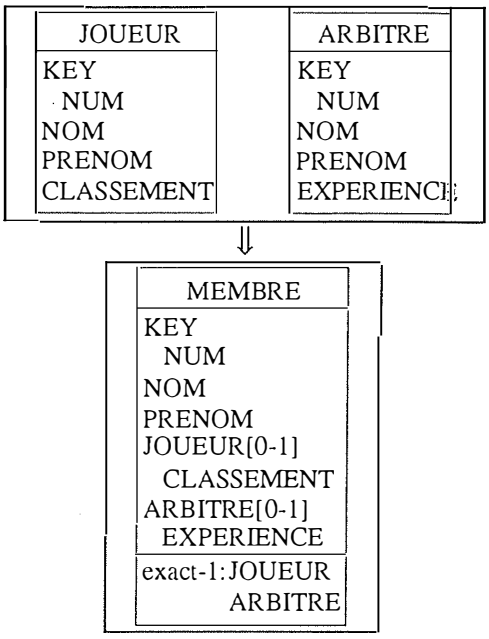


Figure 7-27: Création d'un type d'entité générique MEMBRE

Il nous semble que le véritable problème ici n'est pas la modélisation d'une telle situation, mais sa découverte. Elle ne peut se faire que par une analyse des longueurs physiques de chaque type de segment, au niveau du DL/1 et des COPYBOOKS. Si les longueurs sont identiques, on peut soupçonner quelque chose, sans même en être sûr totalement. L'aide d'une personne connaissant l'application ne serait pas inutile.

7.5. Un exemple concret d'intégration

Soit le schéma de la Figure 7-28 celui issu de l'analyse d'un code DL/1.

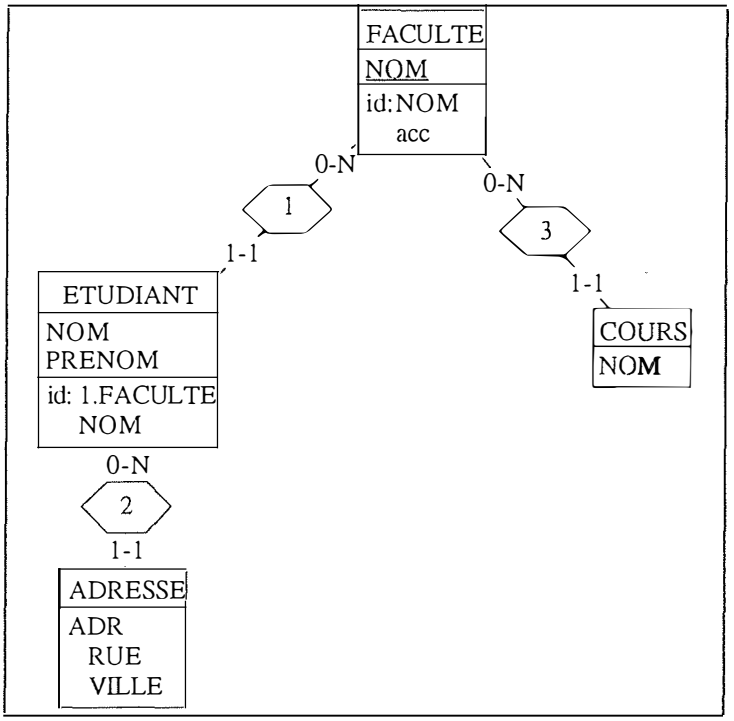


Figure 7-28: Un schéma issu de l'analyse d'un code DL/1

Soit le schéma de la Figure 7-29 celui issu de l'analyse des COPYBOOKS correspondant aux types de segments de la Figure 7-28.

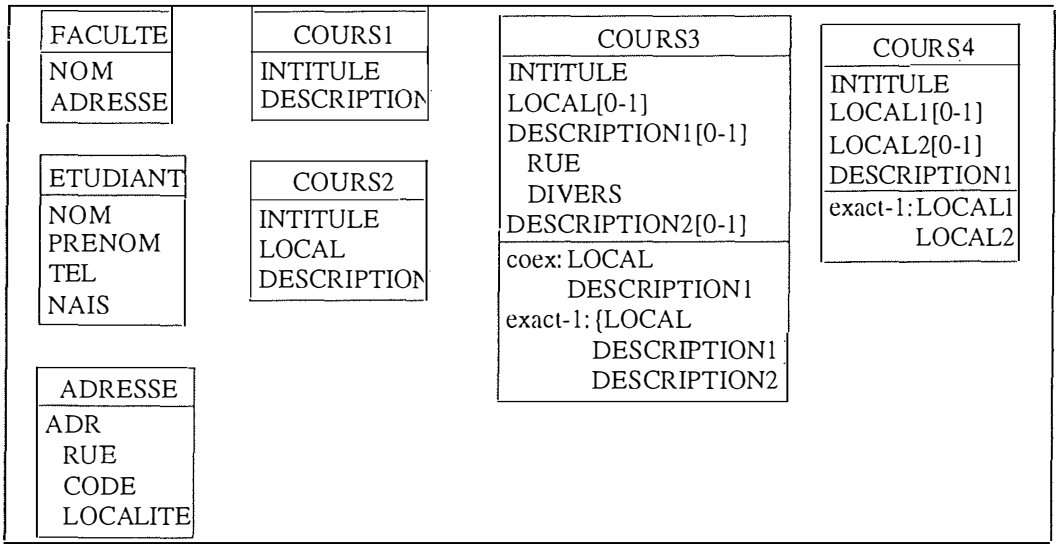


Figure 7-29: Un schéma issu de l'analyse des COPYBOOKS

Comme nous pouvons le constater à la Figure 7-29, au type d'entité *COURS* de la Figure 7-28 correspond quatre COPYBOOK nommés *COURS1*, *COURS2*, *COURS3* et *COURS4*. Il nous faut, avant de pouvoir intégrer les deux schémas, rassembler ces quatre COPYBOOKS en un

seul (nous supposons donc que ces quatre COPYBOOKS redéfinissent tous *COURS*). La Figure 7-30 montre le schéma des COPYBOOKS modifié.

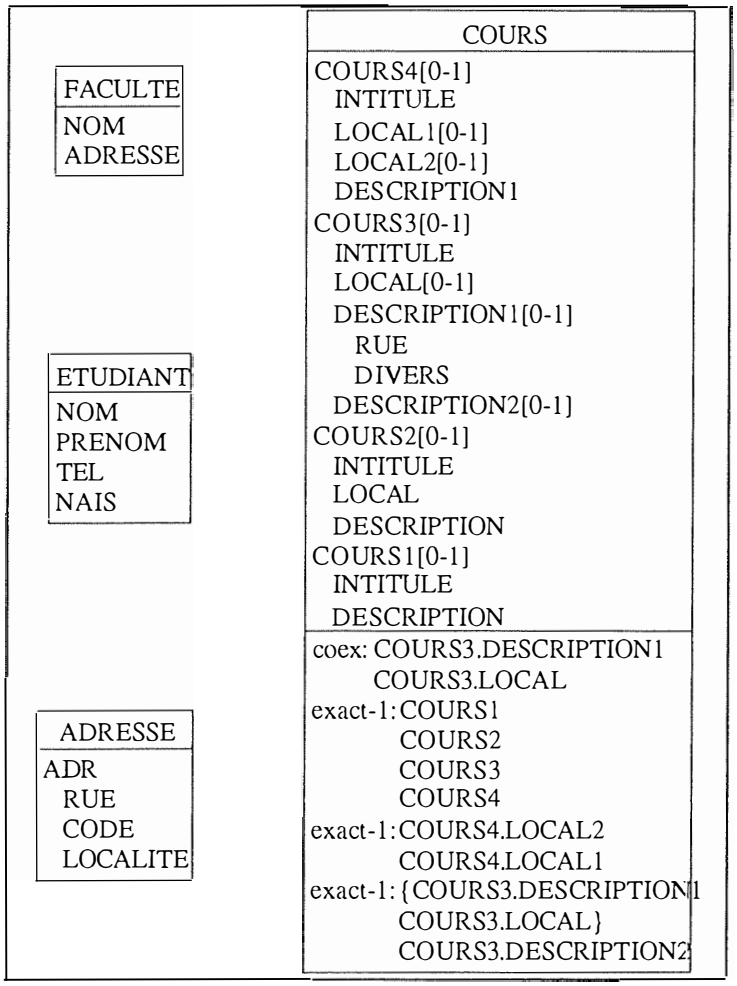


Figure 7-30: Le schéma des COPYBOOKS modifié

Nous avons un type d'entité *COURS* qui englobe quatre attributs facultatifs représentant les quatre types d'entités originaux. Entre ces quatre attributs, nous avons créé une contrainte d'exactement-un reflétant le fait qu'au même moment on ne peut avoir qu'un seul des quatre attributs.

Nous pouvons maintenant procéder à l'intégration proprement dite, entre les schémas des Figures 7-28 et 7-30. La Figure 7-31 montre le résultat de cette intégration.

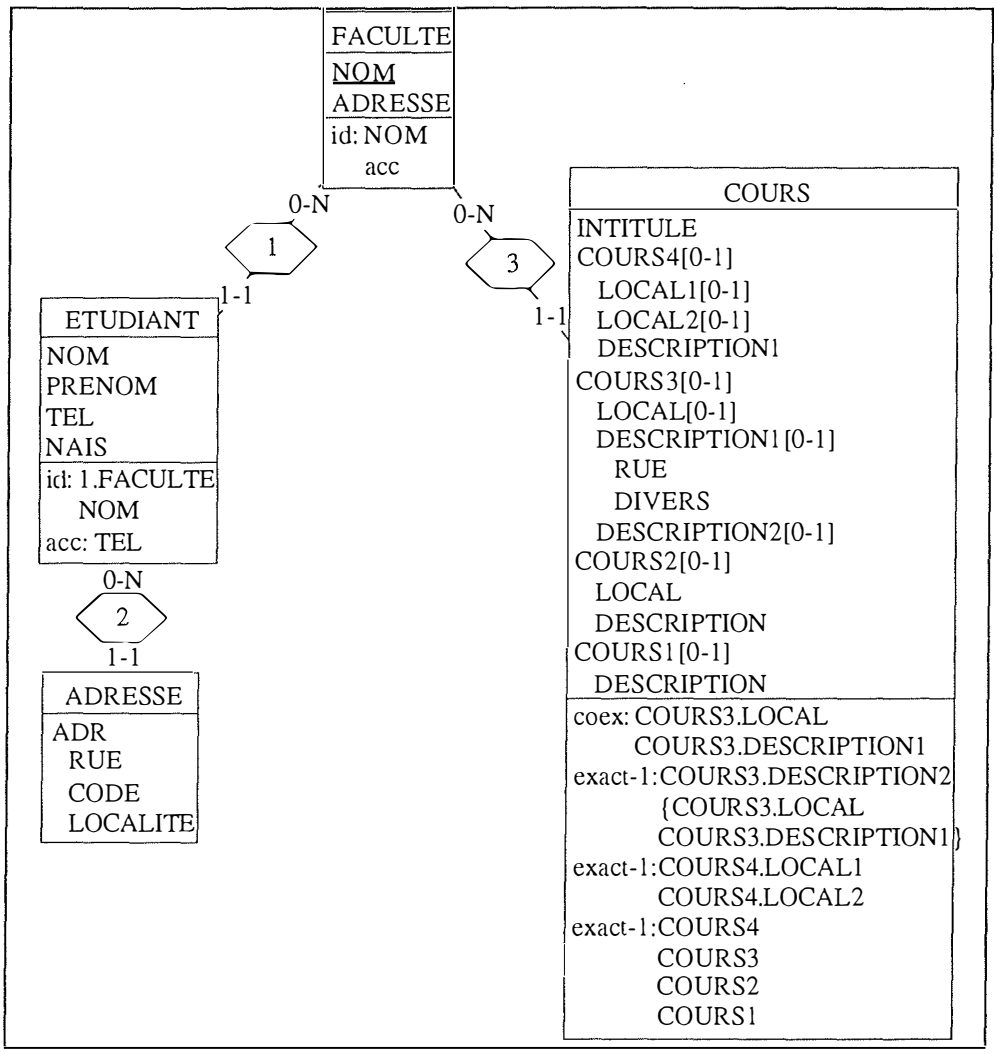


Figure 7-31: Le schéma intégré des Figures 7-28 et 7-30

7.6. Conclusion

Cette étape d'intégration nous permet d'avoir un premier schéma qui contient toute l'information EXPLICITE déclarée à la fois dans le DL/1 et les COPYBOOKS. Ce schéma sera la base de toutes les recherches de structures implicites, et sera donc enrichi au fur et à mesure des découvertes. Le lecteur peut consulter le chapitre quatorze relatif aux outils pour découvrir les deux applications d'intégration que nous avons développées.

CHAPITRE 8: L'analyse des programmes

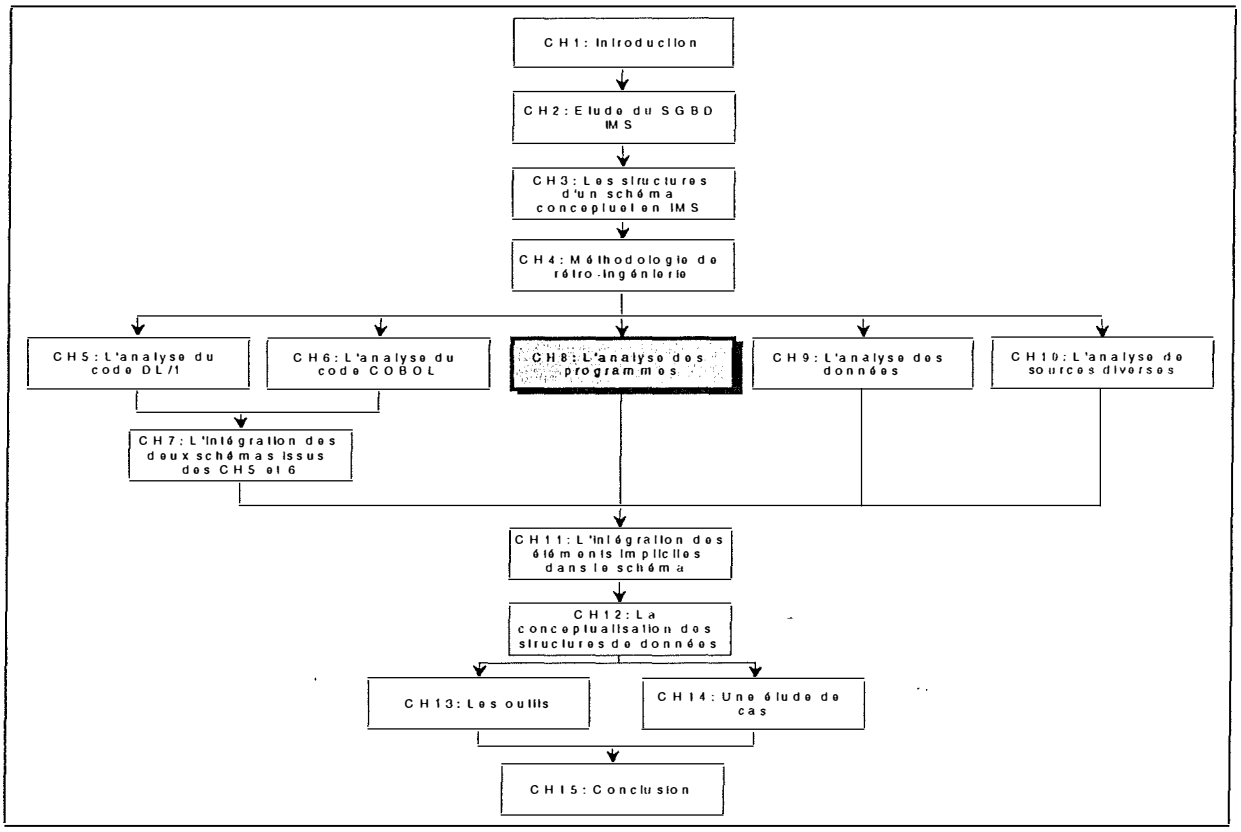


Figure 8-1: Le chapitre 8

8.1. Introduction

Le chapitre huit aborde l'analyse des programmes. IMS n'offre pas de langage de programmation. Les programmes manipulant les données sont donc écrits en langage hôte, et notamment en COBOL, certainement le langage le plus couramment utilisé avec IMS.

L'analyse des programmes consiste à trouver des indices qui permettraient de confirmer ou de supposer l'existence de contraintes ou de structures non implémentées explicitement dans le SGBD IMS. L'analyse des programmes est en fait une des sources les plus riches pour découvrir des structures implicites. L'analyse consiste donc à parcourir le code COBOL directement ou indirectement en rapport avec l'instruction CALL du DML (Data manipulation Language).

Ce chapitre huit commencera par décrire les méthodes d'analyse utilisées pour faire ces recherches. Ensuite, nous présenterons certaines structures implicites que l'on peut découvrir grâce aux méthodes d'analyse présentées dans le point précédent. Nous terminerons par une brève conclusion.

8.2. Les méthodes d'analyse des programmes

Dans cette phase d'analyse des programmes, trois types de méthodes sont utiles dans le but de trouver des structures implicites.

Ces méthodes d'analyse sont:
1. *L'analyse du flux de données* d'un programme (graphe de dépendance)
2. *La recherche de patterns d'analyse*
3. *La fragmentation des programmes* (program slicing)

Les trois paragraphes suivants décrivent brièvement ces trois méthodes.

8.2.1. L'analyse du flux de données

Pour **[ANDERSSON, 96]**, l'analyse du flux de données consiste à suivre à la trace une variable à travers le code du programme. Pour établir le flux de données dans un programme, nous devons donc définir, pour une variable qui a une certaine valeur, dans quelles autres variables elle est propagée. Il faut remarquer que, suivant l'input du programme, des branchements conditionnels peuvent modifier le flux. Cela signifie que toutes les possibilités doivent être prises en considération.
Nous devons déterminer, étant donné l'assignation d'une valeur dans une variable x, à quelles autres variables y_i est assignée la valeur de x avant que x ne reçoive une autre valeur. Le résultat d'une telle recherche dans un programme s'appelle le **graphe de dépendance**, qui montre l'ensemble des variables qui sont pertinentes, c'est-à-dire qui ont été en contact avec la variable d'origine. Cela nous permet donc d'avoir une vue sélective, pertinente sur le code et de pouvoir tirer des conclusions beaucoup plus facilement et plus rapidement.
La Figure 8-2 montre un exemple de graphe de dépendance.

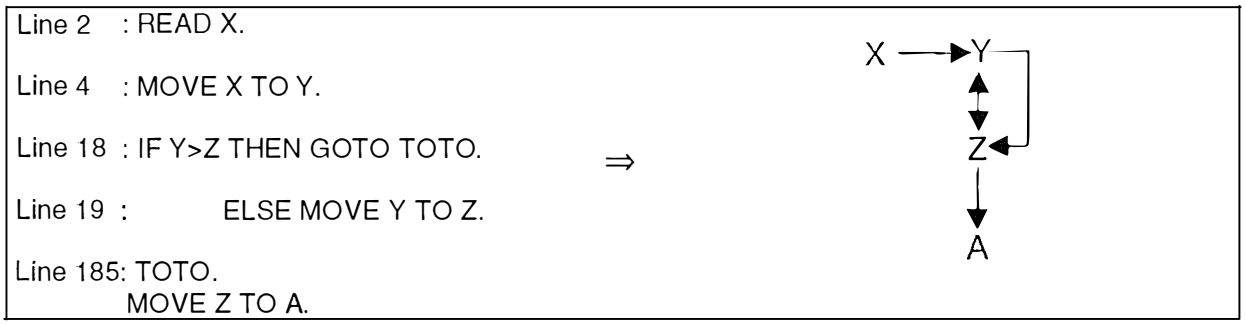


Figure 8-2: Exemple de graphe de dépendance

La Figure 8-2 nous montre qu'à partir des lignes de code de la partie gauche, nous avons créé le graphe de dépendance à droite. La ligne 18 est une instruction IF, qui compare *Y* et *Z* (d'où la flèche orientée dans les deux sens dans le graphe de dépendance à droite). Si *Y* est plus grand que *Z*, on doit aller en *TOTO*, et donc aller vérifier le code à cet endroit, où l'on découvre qu'on assigne *Z* à *A*. Par contre, si *Y* est plus petit que *Z*, on assigne la valeur de *Y* à *Z*. Le graphe de dépendance nous permet entre autres de faire le lien entre la variable *X* et la variable *Z*, via *Y*.

8.2.2. La recherche de patterns d'analyse

La recherche de patterns d'analyse est selon nous la plus intéressante des méthodes d'analyse des programmes. Elle consiste à extraire des parties de code « standard » à partir des

programmes, ces parties de code ayant été spécifiées auparavant par l'utilisateur. Un pattern est défini dans un langage de définition (dans DB-MAIN, il s'agit du PDL, proche de la notation BNF; on peut trouver la syntaxe du PDL à l'annexe B; ce langage de définition dispose de variables qui sont instanciables avant la recherche d'un pattern[1]). Il faut noter que cette méthode est sûrement celle qui est la plus difficile à automatiser, et qui demande encore beaucoup de recherches, notamment au niveau de DB-MAIN et son langage PDL.
Chaque programmeur étant susceptible de coder différemment une même contrainte (clé étrangère,...), la définition d'un seul pattern ne suffira sans doute pas à trouver toutes les parties de code relatives à la contrainte recherchée. De nombreuses variantes d'un pattern sont donc susceptibles d'exister.

8.2.3. La fragmentation des programmes (program slicing)

La technique de fragmentation des programmes (program slicing) permet d'extraire d'un programme le fragment nécessaire et suffisant pour comprendre le fonctionnement de ce programme à un point déterminé. Ce point est appelé *critère de fragmentation.* Un *fragment de programme* est constitué des parties du programme qui affectent les variables référencées par le critère de fragmentation. Idéalement, un fragment est constitué de toutes les valeurs calculées par rapport au critère de fragmentation et uniquement celles-là. Le concept de fragment de programme a été introduit par Mark Weiser **[WEISER, 84]**.

[1] Dans la suite, nous exprimerons nos patterns en COBOL et dans un pseudo-langage, le PDL n'étant pas encore assez puissant pour les exprimer pleinement.

8.3. Les diverses structures cachées

Le but de ce point est de présenter un panel de structures que l'on peut découvrir dans les programmes. Les méthodes théoriques présentées au paragraphe 8.2 sont très utiles pour rechercher des **indices** concernant différentes structures implicites possibles. De nouveau, nous ferons la remarque que tous les éléments découverts appuyant l'hypothèse que l'on désire vérifier ne sont que des **indices** et n'induisent aucune certitude sur l'exactitude de cette hypothèse.
Il faut aussi remarquer que dans certains cas, on ne sait pas exactement ce que l'on cherche, c'est-à-dire que l'on n'a pas de suspicions préalables; dans ces cas, les structures implicites sont d'autant plus difficiles à trouver, vu qu'on ne sait pas ce qu'on recherche.

Cette section a pour but de présenter une série de découvertes que l'on peut faire dans les codes applicatifs, au moyen des méthodes d'analyse présentées dans le point précédent.
Nous nous sommes plus particulièrement penchés sur:

1. Les identifiants,
2. Les clés étrangères,
3. Les champs:
   - décomposables,
   - agrégés,
   - multivalués,
   - facultatifs,
   - manquants.
4. Les contraintes:
   - de coexistence,
   - d'au-moins-un,
   - d'exclusion,
   - d'exactement-un,
   - de dépendances fonctionnelles,
   - de redondance,
   - de cardinalités.

Remarque: Dans les paragraphes suivants, nous allons parfois décrire des patterns d'analyse. Ceux-ci seront décrits en COBOL (cadre de gauche des figures), ainsi qu'en pseudo-code (cadre de droite des figures), pour faciliter la compréhension. La plupart des patterns décrits en COBOL, comprendront l'instruction d'accès à la BD "CALL". Pour la syntaxe et la sémantique de cette instruction capitale, veuillez consulter le chapitre deux consacré à IMS.
Nous attirons aussi l'attention du lecteur sur le fait que dans les patterns en COBOL, nous ne ferons pas directement référence à des types de segments d'une base de données mais à des variables COBOL correspondant à ces types de segments et donc à leurs champs (En fait, ce sont les variables des COPYBOOKS). Nous rappellerons systématiquement ce fait dans les explications relatives à un pattern.

8.3.1. Les identifiants

Dans le cas du SGBD IMS, les contraintes d'identifiants peuvent être exprimées explicitement dans le DL/1 (voyez le chapitre cinq). Cependant, il se peut que le concepteur exprime cette

contrainte dans les programmes. Dans ce cas, il nous faut analyser le code à la recherche d'indices prouvant qu'un ou plusieurs champ(s) est (sont) identifiant(s).
La technique la plus efficace pour la recherche d'indices concernant un identifiant est la recherche de patterns.
Les deux patterns décrits ci-après ont été décrits par **[HAINAUT, 95a]**. Nous les avons traduit en COBOL.

A. Pattern 1: Recherche dans la BD selon une valeur

Un accès spécial à la base de données peut confirmer la présence d'un identifiant.

Si l'on recherche une valeur d'un champ dans une base de données,
si dès qu'elle est découverte, on arrête la recherche et
que l'on ne s'occupe pas des valeurs suivantes
alors, cela peut être un indice de la présence d'un identifiant.

MOVE VAR1 TO ASSA-A1 CALL BD GU Apcb AVAR ASSA IF STATUS-CODE = SEGMENT-FOUND THEN EXIT PROGRAM	GET-FIRST A(A1=VAR1); IF RECORD-FOUND THEN process this record Ignore the others

Figure 8-3: Pattern 1 concernant les identifiants

Avec A un type de segment,
A1 un champ de A,
ASSA, une variable représentant un SSA de A,
ASSA-A1, une variable appartenant à ASSA, et correpondant au champ A1 de A,
VAR1 une variable,
AVAR une variable recevant le résultat de la requête (CALL),
Apcb, un PCB.

Explication:
On recherche la première occurrence du segment qui contient dans son champ A1 la valeur contenue dans VAR1. La fonction GU (Get Unique) signifie que l'on ne prend en compte que la première valeur correspondant aux critères de recherches présents dans le SSA (Search Segment Arguments) ASSA. Cela peut donc signifier que la valeur du champ A1 est unique, et donc identifiante.

B. Pattern 2: Vérification avant une insertion

Si, avant d'insérer une nouvelle occurrence d'un type de segment, on s'assure que la valeur donnée au champ suspecté d'être un identifiant de ce type de segment n'est pas déjà présente dans la base de données, cela peut être un indice de la présence d'un identifiant. La Figure 8-4 décrit ce pattern.

MOVE VAR1 TO ASSA-A1 CALL BD GU Apcb AVAR ASSA IF STATUS-CODE = SEGMENT-NOT-FOUND THEN CALL BD INSERT Apcb AVAR ASSA	READ-FIRSTA(A1=VAR1); IF record-not-found THEN INSERT A;

Figure 8-4: Pattern 2 concernant les identifiants

Avec A un type de segment,
A1 un champ de A,
ASSA, une variable représentant un SSA de A,
ASSA-A1, une variable appartenant à ASSA, et correpondant au champ A1 de A,

VAR1 une variable,
AVAR une variable recevant le résultat de la requête (CALL),
Apcb, un PCB.

Explication:
Les deux premières instructions COBOL ont pour but de trouver dans la base de données la première occurrence du segment qui répond aux critères de recherches contenus dans le SSA ASSA. Si aucun segment répondant à ces critères n'est trouvé, on insère un nouveau segment avec les données contenues dans le SSA ASSA. Le pseudo-code de la partie droite de la Figure 8-4 est plus lisible.

Remarque: La variable STATUS-CODE renferme des informations sur la terminaison de l'appel à la BD.

8.3.2. Les clés étrangères

Une clé étrangère est un champ (ou combinaison de champs) qui est utilisé pour référencer un type de segment dans une autre (ou une même) base de données. Dans le SGDB IMS, il est impossible de déclarer explicitement une clé étrangère dans le DL/1, comme c'est possible dans le modèle relationnel; il est donc nécessaire de trouver ces clés étrangères par une recherche dans les programmes.
Il est possible de découvrir des indices de clés étrangères au moyen des trois techniques décrites au paragraphe 8.2.

A. L'analyse du flux de données

Si une clé étrangère existe entre deux types de segments, alors nous devrions trouver dans certains programmes, des flux de données entre variables représentant la clé étrangère et le champ référencé.
La Figure 8-5 constitue un premier exemple de graphe de dépendance.

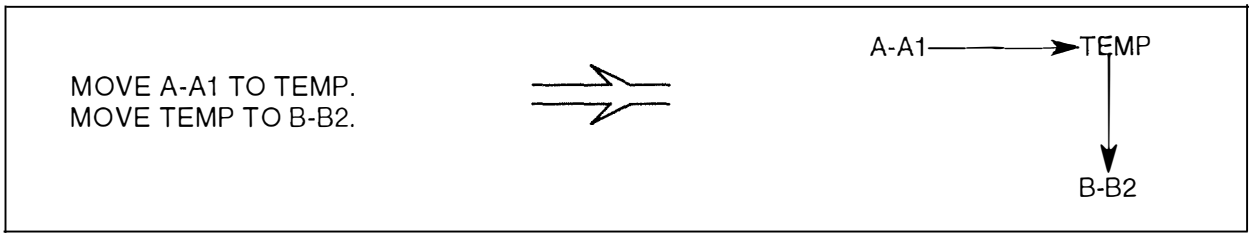


Figure 8-5: exemple de graphe de dépendance

Si A et B sont deux types de segments,
si A1 est un champ de A,
si B2 est un champ de B,
si A1 est identifiant du type de segment A ou
si B2 est identifiant du type de segment B,
si A-A1 et B-B2 sont deux variables correspondant aux deux champs A1 et B2 des deux types de segments A et B,
si TEMP est une variable intermédiaire,
alors, on peut découvrir un indice de clé étrangère du champ non identifiant vers le champ identifiant.

Le graphe de dépendance va permettre de repérer la variable TEMP qui reçoit une valeur qui nous intéresse. Il va également repérer que TEMP est en relation avec B-B2, donc que A-A1 est en relation avec B-B2 à un moment donné. On a découvert que A-A1 et B-B2 partageaient la même valeur à un moment donné. Une telle découverte n'apporte aucune preuve de l'existence d'une clé étrangère; elle est simplement un indice qui doit être confirmé par la suite. Il convient de s'assurer que les valeurs assignées aux variables A-A1 et B-B2 sont bien des valeurs provenant des types de segments correspondants. Autrement dit, il convient de repérer la partie de code où les variables ont reçu leurs valeurs actuelles.
Nous donnons un autre exemple à la Figure 8-6.

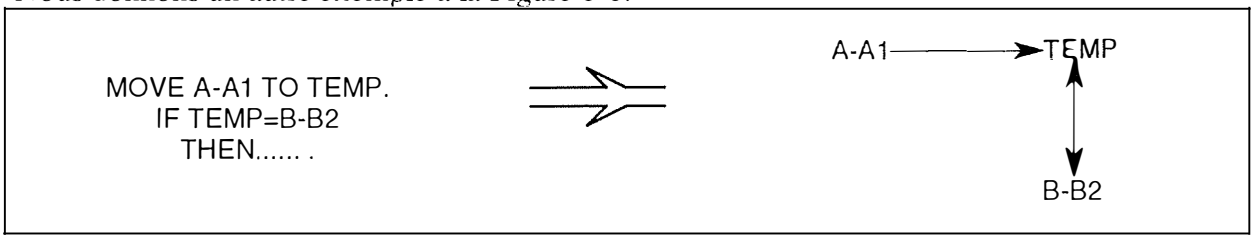


Figure 8-6: exemple de graphe de dépendance

Si A et B sont deux types de segments,
si A1 est un champ de A,
si B2 est un champ de B,
si A1 est identifiant du type de segment A ou
si B2 est identifiant du type de segment B,
si A-A1 et B-B2 sont deux variables correspondant aux deux champs A1 et B2 des deux types de segments A et B,
si TEMP est une variable intermédiaire,
alors, on peut découvrir un indice de clé étrangère du champ non identifiant vers le champ identifiant.

B. La recherche de patterns d'analyse

La recherche d'indices de clés étrangères peut aussi se faire par la recherche de patterns d'analyse. Nous allons décrire dans ce paragraphe toute une série de patterns qui nous semblent pertinents, que l'on a adapté de **[HAINAUT, 95a]**.
Les patterns ci-dessous se basent sur l'exemple de la Figure 8-7.

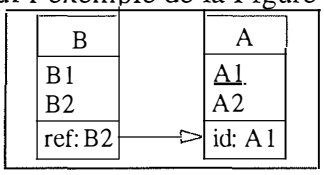


Figure 8-7: un exemple de base aux patterns concernant les clés étrangères

Un moyen très simple de trouver des indices confirmant l'existence d'une clé étrangère est de s'intéresser plus particulièrement aux parties de code où se font les accès à la base de données, et de voir ce qui se passe aux alentours directs de ces appels.

B.1. Pattern 1: La jointure procédurale

MOVE B-B2 TO ASSA-A1. CALL BD GU Apcb Avar ASSA. IF STATUS-CODE = SEGMENT-NOT-FOUND THEN ERROR.	READ A(A1=B.B2); IF NOT FOUND THEN ERROR;

Figure 8-8: Pattern 1 concernant les clés étrangères

Explication:
Si A et B sont deux segments,
si A1 est un champ de A,
si B2 est un champ de B,
si ASSA est une variable constituant un SSA du segment A,
si ASSA-A1 est une variable appartenant à ASSA et correspondant au champ A1 de A,
si B-B2 est une variable correspondant au champ B2 de B,
si Apcb est un PCB,
si Avar est une variable destinée à recevoir le résultat de la requête (CALL),
alors ce pattern signifie que l'on affecte à la variable ASSA-A1 la valeur du champ B2 du segment B courant, présente dans la variable B-B2. Ensuite, on effectue la recherche dans la BD via le CALL. Si elle ne donne rien, un message d'erreur est affiché; cela signifie donc que toute valeur du champ B2 doit être une valeur présente dans le champ A1 d'un segment A.

Ce premier pattern est une situation commune que **[HAINAUT, 96b]** appelle *procedural join*, par analogie avec la jointure SQL: lire un segment (A), identifié par une valeur de champ (B2) venant d'un autre segment (B). La probabilité que B2 soit une clé étrangère de A est très grande.

B.2. Pattern 2: Recherche de tous les segments dont la valeur d'un champ est identique à la valeur d'un autre champ

MOVE VAR1 TO ASSA-AS1. CALL BD GU Apcb Avar ASSA. IF STATUS-CODE=SEGMENT-FOUND THEN MOVE ASSA-AS1 TO BSSA-BS2 CALL BD GU Bpcb Bvar BSSA PERFORM CALL BD GN Bpcb Bvar BSSA UNTIL STATUS-CODE =SEGMENT-NOT-FOUND END-PERFORM END-IF.	READ-FIRST B(B2=A.A1); WHILE FOUND DO READ-NEXT B(B2=A.A1) END-WHILE;

Figure 8-9: Pattern 2 concernant les clés étrangères

Explication:
Si A et B sont deux segments,
si A1 est le champ identifiant du segment A,
si B2 est un champ du segment B,
si ASSA est une variable constituant un SSA de A,
si ASSA-AS1 est une variable appartenant à ASSA, correspondant au champ A1 de A,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS2 est une variable appartenant à BSSA, correspondant au champ B2 de B,
si VAR1 est une variable,
si Apcb et Bpcb sont deux PCB,
si Avar et Bvar sont deux variables destinées à recevoir les résultats des requêtes (CALL),
alors ce pattern signifie qu'on lit tous les segments B dont le champ B2 (clé étrangère) est identique à l'identifiant A1 de A.

B.3. Pattern 3: Création d'un segment

MOVE VALB2 TO BSSA-BS2. MOVE VALB2 TO ASSA-AS1. CALL BD GU Apcb Avar ASSA. IF STATUS-CODE = SEGMENT-FOUND THEN CALL BD INSERT Bpcb Bvar BSSA.	READ A (A1=B.B2); IF FOUND THEN CREATE B;

Figure 8-10: Pattern 3 concernant les clés étrangères

Explication:
Si A et B sont deux segments,
si A1 est un champ du segment A,
si B2 est un champ du segment B,
si ASSA est une variable constituant un SSA de A,
si ASSA-AS1 est une variable appartenant à ASSA, correspondant au champ A1 de A,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS2 est une variable appartenant à BSSA, correspondant au champ B2 de B,
si VALB2 est une variable,
si Apcb et Bpcb sont deux PCB,
si Avar et Bvar sont deux variables destinées à recevoir les résultats des requêtes (CALL),
alors, ce pattern 3 signifie qu'avant toute insertion d'un nouveau segment B, on vérifie que la valeur que l'on veut donner au champ B2 de B est une valeur existante pour le champ identifiant A1 de A. Il y a donc une forte chance pour que B2 soit une clé étrangère pour A.

B.4. Pattern 4: Suppression d'un segment

MOVE VALA1 TO BSSA-BS2. CALL BD GHU Bpcb Bvar BSSA. PERFORM BOUCLE UNTIL STATUS-CODE = SEGMENT-NOT-FOUND. MOVE VALA1 TO ASSA-AS1. CALL BD GHU Apcb Avar ASSA. CALL BD DELETE Apcb Avar ASSA. BOUCLE. CALL BD DELETE Bpcb Bvar BSSA. CALL BD GHN Bpcb Bvar BSSA.	READ-FIRST B(B2=A.A1); WHILE FOUND DO DELETE B READ-NEXT B(B2=A.A1) END-WHILE; DELETE A;

Figure 8-11: Pattern 4 concernant les clés étrangères

Explication:
Si A et B sont deux segments,
si A1 est un champ du segment A,
si B2 est un champ du segment B,
si ASSA est une variable constituant un SSA de A,
si ASSA-AS1 est une variable appartenant à ASSA, correspondant au champ A1 de A,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS2 est une variable appartenant à BSSA, correspondant au champ B2 de B,
si VALA1 est une variable contenant une valeur existant pour le champ A1 d'un segment A,
si Apcb et Bpcb sont deux PCB,
si Avar et Bvar sont deux variables destinées à recevoir les résultats des requêtes (CALL),
alors, la suppression du segment référencé A entraîne la suppression de tous les segments B référençant ce champ. Dans ces instructions, on détruit en premier lieu toutes les occurrences

des segments B dont le champ B2 a la même valeur que A1 (clés étrangères) et ensuite, on détruit l'occurrence du segment A (identifiant).

Remarque importante: les clés étrangères « hiérarchiques »

Les quatre patterns ci-dessus ne sont valables que dans le cas d'un identifiant composé d'un ou plusieurs attributs. Or, le modèle hiérarchique est composé de deux types de types de segments:

- Les types de segments racines, qui n'ont aucun parent.
- Les types de segments dépendants, qui eux, ont un et un seul parent.

Cet état de fait a pour conséquence qu'il y a deux types d'identifiants possibles pour les types de segments dans un modèle hiérarchique:

- Pour les types de segments racines, les identifiants composés uniquement d'un ou plusieurs champs.
- Pour les types de segments dépendants, les identifiants composés d'un ou plusieurs champs <u>et</u> de tous les identifiants concaténés des types de segments ancêtres. La présence des identifiants des ancêtres dans l'identifiant de l'enfant se justifie par le fait que les occurrences de ce type de segment enfant sont uniques sous les mêmes segments ancêtres mais pas pour toute la BD.

La présence de deux types d'identifiants implique qu'il y a aussi deux types de clés étrangères:

- Dans le cas d'une clé étrangère vers un type de segment racine, on a affaire à une clé étrangère "habituelle", qui sera composée du ou des champs composant l'identifiant.
- Dans le cas des types de segments dépendants, on aura affaire à un type particulier de clé étrangère: les clés étrangères hiérarchiques. Ces clé étrangères sont composées:
  - du ou des champs composant l'identifiant du type de segment référencé,
  - et de tous les identifiants **concaténés** des types de segments supérieurs à ce type de segment référencé (parent et ancêtres, jusqu'à la racine y compris).

Donc, dans le cas de la recherche de clés étrangères vers des identifiants de types de segments dépendants, les quatre patterns ci-dessus devront être adaptés. La Figure 8-12 donne un exemple de clé étrangère hiérarchique.

```
MOVE COROOT-CUSTNO TO CUROOTQ-CUSTNO.
MOVE COROOT-CUSTMU TO CUROOTQ-CUSTMU.
MOVE COROOT-DELADNO TO CUDELVQ-KEY.
CALL 'LS3DLIB' USING DB-USER-AREA FUNC-GU LS-PCB6 CUDELV CUROOTQ CUDELVQ.
IF STATUS-CODE = SEGMENT-FOUND
THEN CALL 'LS3DLI' USING DFHEIBLK DB-USER-AREA FUNC-ISRT LS-PCB1 COROOT
COROOTQ.
```

Figure 8-12: Exemple de clé étrangère hiérarchique

L'exemple ci-dessus est l'adaptation « hiérarchique » du pattern 3. La situation graphique correspondante est décrite à la Figure 8-13.

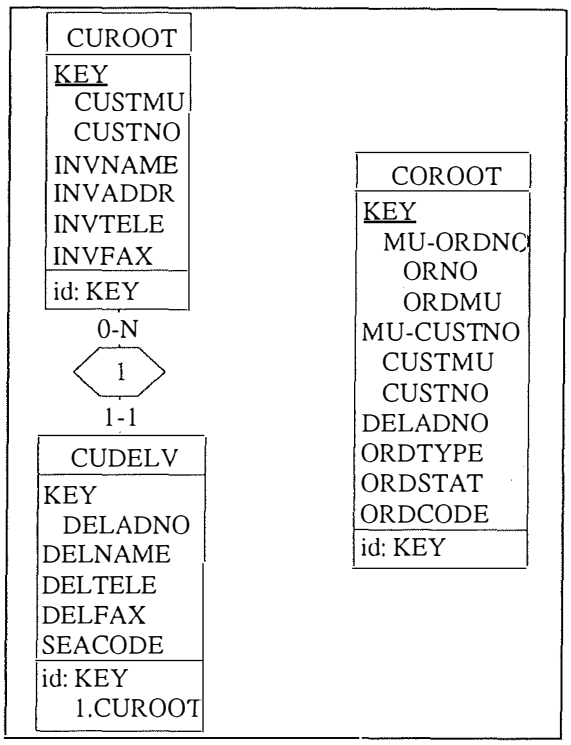


Figure 8-13: Situation graphique

En examinant le code de la Figure 8-12, on peut supposer l'existence d'une clé étrangère constituée des champs MU-CUSTNO et DELADNO de COROOT vers le type de segment référencé CUDELV, identifié par:

- le champ DELADNO,
- mais aussi l'identifiant concaténé du type segment parent de CUDELV, qui est CUROOT (champ KEY).

La Figure 8-14 montre comment représenter cette situation avec DB-MAIN.

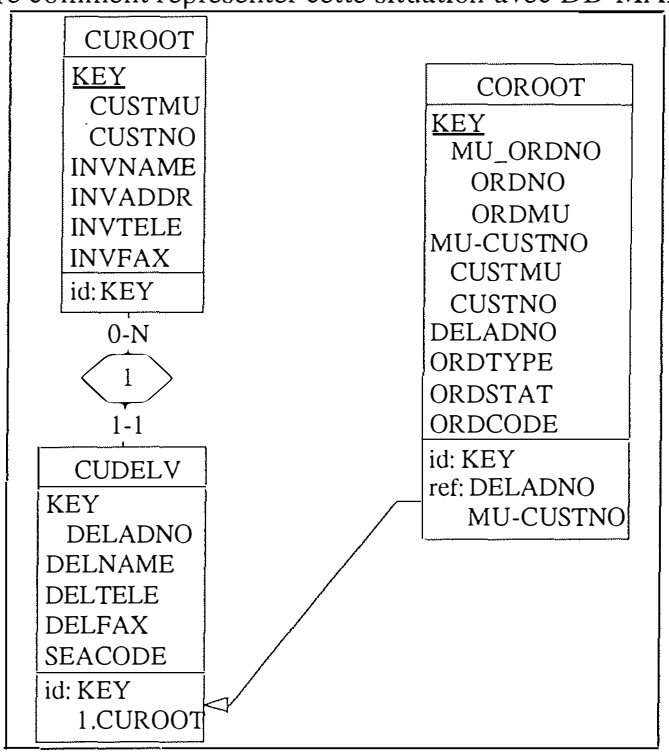


Figure 8-14: la situation en DB-MAIN

C. La fragmentation des programmes (program slicing)

La méthode de fragmentation de programmes peut aussi être très utile dans le cadre de la recherche d'indices de clés étrangères. Une technique que nous avons utilisée lors de notre stage consistait à sélectionner les instructions d'appel à la base de données, ce qui nous permettaient de voir si des liens existaient entre champs de types de segments différents et dans quel cadre ces liens existaient.
Par exemple:

a.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CLIENT-COMMANDE.

ENVIRONMENT DIVISION.
...

DATA DIVISION.
...

WORKING-STORAGE SECTION.
01 NOM              PIC X(9).
01 COMM-CLI          PIC X(9).
01 CLIENT-SSA        PIC X(9).
01 CLIENT-VAR        PIC X(9).
01 CLIENT-PCB        PIC X(9).
01 IND               PIC 99.
...

PROCEDURE DIVISION.
MAIN.
            PERFORM TRAITEMENT.
            STOP RUN.
CHERCOM.
            ...
TRAITEMENT.
            ACCEPT NOM.
            ...
            MOVE NOM TO COMM-CLI
            ...
            MOVE COMM-CLI TO CLIENT-SSA.
            ...
            CALL BD GU  CLIENT-PCB CLIENT-VAR CLIENT-SSA.
            IF STATUS-CODE = SEGMENT-NOT-FOUND THEN ERROR.
            DISPLAY CLIENT-VAR.
            ...
            MOVE 1 TO IND.
            PERFORM UNTIL IND=10
                  CHERCOM.
ERROR.
            ...
```

b. Si nous utilisons le program slicing à partir du CALL en gras, nous obtenons le fragment de programme suivant:

```
            ACCEPT NOM.
            MOVE NOM TO COMM-CLI
            MOVE COMM-CLI TO CLIENT-SSA.
            CALL "BD" GU  CLIENT-PCB CLIENT-VAR CLIENT-SSA.
```

Figure 8-15: a un exemple de programme; b. le fragment du programme par rapport à la ligne en gras

Ces quelques lignes fournies par le *program slicing* indique que l'on recherche un CLIENT dont le nom est égale à la valeur de COMM-CLI. Les lignes après le CALL nous montrent que si on ne trouve pas un tel CLIENT, il y a erreur. Si COMM-CLI contient des valeurs qui proviennent du champs CLI du type de segment COMM et si CLIENT-SSA est un SSA du type de segment CLIENT alors on peut conclure qu'il y a probablement un indice de la présence d'une clé étrangère entre COMM-CLI et CLIENT.

8.3.3. Les champs

Dans cette section, nous allons nous intéresser à la problématique des champs. Il s'agira de résoudre des problèmes de structures non-déclaratives (champs décomposables, agrégation de champs, champs multivalués, facultatifs et manquants).

A. Les champs décomposables (le raffinement de champs)

Le DDL d'IMS (DL/1) ne permet pas de déclarer explicitement un champ comme décomposable. On peut néanmoins découvrir le caractère décomposable d'un champ dans le DL/1 par la combinaison des mots-clés BYTE et START de l'instruction SEGM (ou encore par une analyse des COPYBOOKS). Cependant, il se peut qu'un champ considéré comme atomique après ces analyses, soit en réalité un champ décomposable. Pour le découvrir, il nous faut analyser les programmes. L'analyse du flux de données et la recherche de patterns seront les deux méthodes utilisées.

A.1. L'analyse du flux de données (couplée à une analyse des déclarations de variables)

Lorsqu'on assigne le contenu d'une variable dans une autre variable, et que leurs décompositions sont différentes, on peut conclure, en prenant l'exemple ci-dessous, que la variable *ACTEUR*, correspondant à un champ d'un type de segment, peut se décomposer de la même manière que *PERSON-ID* (Figure 8-16).

```
01 PERSON-ID.
        05 NOM PIC X(10).
        05 PRENOM PIC X(10).
        05 DATE-NAIS PIC X(6).

01 ACTEUR PIC X(26).
MOVE PERSON-ID TO ACTEUR.
```

Figure 8-16: Exemple

Si l'instruction MOVE s'effectue dans le sens inverse (MOVE ACTEUR TO PERSON-ID), on peut tirer les mêmes conclusions.

Pour que cette instruction soit utile, il est nécessaire que le contenu de la variable *ACTEUR* se décompose d'une manière telle que les champs de *PERSON-ID* soient remplis d'une manière cohérente. Dans l'exemple, nous connaissons la signification de chaque champ. Il est alors aisé de conclure que la variable *ACTEUR* doit contenir:

dans ses 10 premiers bytes un nom,
dans les 10 bytes suivants un prénom
et dans les 6 derniers bytes une date de naissance.

Lors de l'analyse du contenu de la variable *ACTEUR*, il sera possible de se rendre compte du format de cette variable. Cela n'est pas nécessairement le cas avec une variable dont le contenu n'aurait pas de sens évident. Par exemple, l'identifiant d'une pièce qui se diviserait en plusieurs

parties. L'analyse du flux de données peut donc nous aider à suivre à la trace une variable, un champ que l'on suspecte d'avoir une structure interne « cachée ».

A.2. La recherche de patterns d'analyse

Certains programmes ou certaines requêtes extraient ou utilisent des parties de champs. Cela apporte l'information que cette partie de champ possède une existence autonome. On peut en tirer comme conséquence que ce champ peut être divisé en plusieurs parties; ce champ est peut-être décomposable. Un de ces composants est la partie extraite. Un pattern, adapté de **[HAINAUT, 95a]**, peut donc être défini à la Figure 8-17.

MOVE A(1:15) TO A1. MOVE A(16:10) TO A2.	Select substr(A,1,15):A1 Select substr(A,16,26):A2

Figure 8-17: Pattern 1 concernant les champs décomposables

Explication:
Si A est une variable correspondant à un champ atomique d'un segment de la base de donnée,
si A1 et A2 sont deux variables recevant les deux sous-chaînes extraites de A,
si l'instruction *MOVE A(16:10) TO A2* signifie que l'on assigne à A2 les valeurs de A à partir de la position 16 sur une longueur de 10,
alors le champ A peut être suspecté d'avoir une structure qui peut être décomposée en A1 et A2.
On peut aussi découvrir le caractère décomposable d'un champ via l'instruction COBOL UNSTRING, comme le montre la Figure 8-18.

MOVE 1 TO pointeur.
PERFORM UNTIL pointeur>25
UNSTRING A
DELIMITED BY "/"
INTO resultat
COUNT IN long
WITH POINTER pointeur
END-UNSTRING
IF long=15
THEN MOVE resultat TO A1
END-IF
IF long=9
THEN MOVE resultat TO A2
END-IF
END-PERFORM.
	i=1;
	while (i<=25) and (A[i]<>'/') do
	A1[i]=A[i];
	i=i+1;
	endwhile;
	i=i+1;
	while (i<=25) do
	A2[i]=A[i];
	i=i+1;
	endwhile;

Figure 8-18: Pattern 2 concernant les champs décomposables

Explication:
Si l'instruction UNSTRING est une instruction COBOL qui extrait une sous-chaîne d'une chaîne de caractère,
si A est une variable correspondant à un champ atomique d'un segment de la base de donnée,
si pointeur est un pointeur pointant sur A,
si long est une variable contenant la longueur de la sous-chaîne extraite,
si resultat est une variable contenant la sous-chaîne extraite,
si A1 et A2 sont deux variables recevant les deux sous-chaînes extraites de A,
alors le champ A peut être suspecté d'avoir une structure qui peut être décomposée en A1 et A2.
Nous remarquons que l'instruction UNSTRING est imbriquée dans une boucle qui stopppe quand la variable *pointeur* dépasse 25, c'est-à-dire la longueur de A. Nous remarquons aussi

la présence d'un délimiteur (/) entre les deux sous-chaînes, ce qui permet à l'instruction UNSTRING d'extraire les deux sous-chaînes.

B. L'agrégation de champs

Des champs apparemment indépendants sont en fait les composants d'un champ décomposable « caché ». Le problème à résoudre est de reconstruire ce champ décomposable source. Les techniques utilisées seront l'analyse du flux de données et la recherche de patterns d'analyse.

B.1. L'analyse du flux de données

Il suffit de suivre à la trace le groupe de champs suspectés et on pourra peut-être découvrir qu'à un moment donné, ils sont affectés dans une seule variable décomposable. On aura ainsi découvert la structure de base, qui avait été dissimulée.

B.2. La recherche de patterns d'analyse

On peut observer que ces champs sont utilisés en même temps par les programmes.

C. Les champs multivaluéss

Le DDL d'IMS ne permet pas de déclarer un champ comme multivalué. Cette contrainte doit donc être exprimée dans les programmes. Nous devrons donc trouver un champ F, déclaré comme monovalué, qui en fait implémente le champ G tel que F=G[1]+G[2]+ ... +G[n]. Les techniques de recherche de patterns et d'analyse de flux de données sont les mieux à même de nous aider dans la recherche d'indices.

C.1. L'analyse du flux de données

Si un champ suspecté d'être multivalué est assigné à une variable déclarée comme multivaluée (mot-clé OCCURS), ce champ est multivalué. Suivre à la trace le champ permettra d'arriver plus aisément à l'instruction d'assignation à cette variable multivaluée.

C.2. La recherche de patterns d'analyse

Si dans un programme, l'analyste s'aperçoit qu'à partir d'un champ, on extrait plusieurs variables, cela peut-être le signe de la présence d'un champ multivalué.

MOVE 0 TO i.
PERFORM UNTIL i>N
MOVE A(i*10+1:10) TO resultat(i)
MOVE i+1 TO i
END-PERFORM.
	for i:=0 to N do
	begin
	  substr(A,i*10+1,10)+ " "=resultat(i)
	end

Figure 8-19: Pattern concernant les champs multivalués

Explication:
si A est une variable correspondant à un champ atomique d'un segment de la base de donnée,
si resultat est une variable multivaluée contenant les sous-chaînes extraites,
alors ce pattern signifie qu'à partir de la variable monovaluée *A,* on extrait N sous-champs stockés dans la variable multivaluée *resultat.*

D. Les champs facultatifs

Il est impossible en DL/1 de déclarer des champs facultatifs. Encore une fois, une telle contrainte doit donc être exprimée dans les programmes.
Si l'analyste repère dans le code des programmes l'utilisation de valeurs que nous avons appelé vides (NULL, VOID, " ", ...), et que ces valeurs sont affectées à des variables qui elles-mêmes sont enregistrées dans un champ, alors on peut découvrir l'existence d'un champ facultatif.

L'analyse du flux de données peut donc être utile dans ce cadre, de même que la recherche dans le programme du mot-clé représentant la valeur vide utilisée.

E. Les champs manquants

Un champ manquant est un champ qui n'a pas été déclaré explicitement mais qui existe. La recherche de champs manquants peut être facilitée par l'analyse du flux de données.
Supposons que dans un programme, on découvre une instruction du type MOVE B TO A. avec les déclarations suivantes pour *A* et *B* (Figure 8-20).

```
05 A.
 10 A1 PIC X(10).
 10 FILLER PIC X(10).
05 B.
 10 B1 PIC X(10).
 10 B2 PIC X(10).
```

Figure 8-20: Un exemple de déclaration COBOL

Si A est une variable COBOL représentant un champ d'un type de segment de la base de donnée,
si B est une variable COBOL utilisée dans un programme,
alors on se trouve dans une situation où une valeur est affectée à un champ "de remplissage" (*FILLER*), un champ qui n'a pas de sens pour l'application.
On se trouve en présence d'un indice qui confirme la présence d'un champ manquant, dont il faudra essayer de comprendre la signification.

8.3.4. Les contraintes

Nous nous sommes intéressés aux contraintes:

- *de coexistence,*
- *d'au-moins-un,*
- *d'exclusion,*
- *d'exactement-un,*
- *de dépendance fonctionnelle,*
- *de redondance,*
- *de cardinalité.*

Ce genre de contraintes est très difficile à repérer car, souvent, on n'a pas de suspicion préalable. Néanmoins, on peut raisonnablement penser que les contraintes seront codées aux alentours des appels à la base de données (CALL). La recherche de patterns d'analyse et dans une moindre mesure l'analyse du flux de données peuvent être utiles pour la recherche de ce genre de contraintes, très difficiles à repérer.

A. Les contraintes de coexistence

Une contrainte de coexistence s'applique entre des éléments facultatifs d'un type d'entité. Elle signifie que les éléments visés par la contrainte ont une valeur en même temps dans la BD ou n'en ont pas.

La recherche de patterns d'analyse peut apporter des résultats intéressants. En effet, il suffit d'analyser les parties de code aux alentours des appels à la base de données qui insèrent le type de segment contenant les champs "suspectés".

MOVE VAL1 TO BSSA-BS1. MOVE VAL2 TO BSSA-BS2. CALL BD INSERT Bpcb Bvar BSSA.	B.B1=VAL1; B.B2=VAL2; INSERT B;

Figure 8-21: Pattern concernant les contraintes de coexistence

Explication:
Si B est un type de segment,
si B1 et B2 sont deux champs facultatifs de B suspectés d'être liés par une contrainte de coexistence,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS1 et BSSA-BS2 sont deux variables appartenant à BSSA, correspondant aux champs B1 et B2 de B,
si Bvar est une variable devant recevoir le résultat de la requête,
si Bpcb est un PCB,
si VAL1 et VAL2 sont deux variables,
alors on voit qu'avant d'insérer le segment B, on assigne des valeurs aux champs B1 et B2.

Si, lors de chaque insertion de B, il en va ainsi, la probabilité qu'il existe une contrainte de coexistence entre B1 et B2 est grande.

B. Les contraintes d'au-moins-un

Une contrainte d'au-moins-un s'applique entre des éléments facultatifs d'un type d'entité. Cette contrainte indique qu'au moins un des éléments facultatifs doit être présent.
La recherche de patterns d'analyse est la méthode la plus efficace pour la découvrir.
En effet, il suffit d'analyser les parties de code aux alentours des CALL à la base de donnée qui insèrent le type de segment contenant les champs "suspectés". En effet, si à chaque fois qu'on insère une occurrence de ce type de segment, au moins un des champs suspecté reçoit une valeur, alors c'est un indice supplémentaire de l'existence d'une contrainte d'au-moins-un.
Nous avons défini trois patterns possibles pour repérer des contraintes d'au-moins-un.

MOVE VAL1 TO BSSA-BS1. CALL BD INSERT Bpcb Bvar BSSA.	B.B1=VAL1; INSERT B;
MOVE VAL2 TO BSSA-BS2. CALL BD INSERT Bpcb Bvar BSSA.	B.B2=VAL2; INSERT B;
MOVE VAL1 TO BSSA-BS1. MOVE VAL2 TO BSSA-BS2. CALL BD INSERT Bpcb Bvar BSSA.	B.B1=VAL1; B.B2=VAL2; INSERT B;

Figure 8-22: Trois patterns concernant les contraintes d'au-moins-un

Explication:
Si B est un type de segment,
si B1 et B2 sont deux champs facultatifs de B suspectés d'être liés par une contrainte d'au-moins-un,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS1 et BSSA-BS2 sont deux variables appartenant à BSSA, correspondant aux champs B1 et B2 de B,
si Bvar est une variable devant recevoir le résultat de la requête,
si Bpcb est un PCB,
si VAL1et VAL2 sont deux variables,
alors ces trois patterns montrent qu'il pourrait exister une contrainte d'au-moin-un entre B1 et B2.

En effet, si lors de chaque insertion de B, un de ces trois patterns est utilisé, la probabilité d'avoir une telle contrainte augmente. Le pattern 1 donne une valeur uniquement à B1; le pattern 2 donne une valeur uniquement à B2 et le pattern 3 donne une valeur à B1 et B2.

C. Les contraintes d'exclusion

Une contrainte d'exclusion s'applique entre des éléments facultatifs d'un type d'entité. La contrainte d'exclusion indique que si un élément visé par la contrainte est présent, alors les autres sont exclus.

C.1. L'analyse du flux de données

En analysant le flux de données des variables suspectées, le rétro-ingénieur peut parvenir à déterminer si à un moment donné, il n'y a qu'un seul des champs qui contient une valeur.

C.2. La recherche de patterns d'analyse

En général, lorsque l'on recherche la présence de patterns d'analyse, c'est dans le but de découvrir l'existence d'une structure. Il est également possible de rechercher la présence de patterns d'analyse pour prouver que la structure n'existe pas. C'est le cas ici. En effet, nous proposons un pattern qui prouve que la contrainte d'exclusion n'est pas envisageable entre deux champs.

MOVE VAL1 TO BSSA-BS1. MOVE VAL2 TO BSSA-BS2. CALL BD INSERT Bpcb Bvar BSSA.	B.B1=VAL1; B.B2=VAL2; INSERT B;

Figure 8-23: Pattern concernant les contraintes d'exclusion

Explication:
Si B est un type de segment,
si B1 et B2 sont deux champs facultatifs de B suspectés d'être liés par une contrainte d'exclusion,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS1 et BSSA-BS2 sont deux variables appartenant à BSSA, correspondant aux champs B1 et B2 de B,
si Bvar est une variable devant recevoir le résultat de la requête,
si Bpcb est un PCB,
si VAL1et VAL2 sont deux variables,
alors ce pattern signifie que les champs B1 et B2 possèdent une valeur au même moment.

La conclusion est évidente: une contrainte d'exclusion ne peut être envisagée entre ces deux champs facultatifs.

D. Les contraintes d'exactement-un

Cette contrainte est l'union des contraintes d'au-moins-un et d'exclusion, et concerne donc des éléments facultatifs d'un type d'entité.
La différence entre une contrainte d'exactement-un et une contrainte d'exclusion est que dans la première, au moins un des éléments visé par la contrainte doit recevoir une valeur, contrairement à la contrainte d'exclusion.

La recherche de patterns d'analyse est la méthode la plus efficace pour découvrir des indices de contraintes d'exactement-un.

D.1. Pattern 1: Modification d'un segment

Le rétro-ingénieur peut renforcer son hypothèse de présence d'une contrainte d'exactement-un s'il découvre le pattern de la Figure 8-24.

MOVE VAL1 TO BSSA-BS1. CALL BD GHU Bpcb Bvar BSSA. IF STATUS-CODE=SEGMENT-FOUND THEN CALL BD DELETE Bpcb Bvar BSSA MOVE VAL2 TO BSSA-BS2 CALL BD INSERT Bpcb Bvar BSSA END-IF.	Get-First B(B1=VAL1); IF record-found THEN DELETE B; INSERT B (B2=VAL2);

Figure 8-24: Pattern 1 concernant les contraintes d'exactement-un

Explication:
Si B est un type de segment,
si B1 et B2 sont deux champs facultatifs de B suspectés d'être liés par une contrainte d'exactement-un,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS1 et BSSA-BS2 sont deux variables appartenant à BSSA, correspondant aux champs B1 et B2 de B,
si Bvar est une variable devant recevoir le résultat de la requête,
si Bpcb est un PCB,
si VAL1et VAL2 sont deux variables,
alors, cela signifie qu'avant d'insérer une valeur d'un champ (B2) , on enlève la valeur contenue dans le champ B1.

D.2. Pattern 2: Insertion sélective d'un segment

IF test1 THEN MOVE VAL1 TO BSSA-B1 CALL BD INSERT Bpcb Bvar BSSA ELSE MOVE VAL2 TO BSSA-B2 CALL BD INSERT Bpcb Bvar BSSA.	IF test1 THEN INSERT B(B1=VAL1) ELSE INSERT B(B2=VAL2);

Figure 8-25: Pattern 2 concernant les contraintes d'exactement-un

Explication:
Si B est un type de segment,
si B1 et B2 sont deux champs facultatifs de B suspectés d'être liés par une contrainte d'exactement-un,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS1 et BSSA-BS2 sont deux variables appartenant à BSSA, correspondant aux champs B1 et B2 de B,
si Bvar est une variable devant recevoir le résultat de la requête,
si Bpcb est un PCB,
si VAL1et VAL2 sont deux variables,
alors, cela signifie qu'avant d'insérer un segment B, on teste s'il s'agit d'assigner une valeur au champ B1 ou au champ B2.

Si, lors de chaque insertion de B, ce pattern est présent, la probabilité de se trouver face à une contrainte d'exactement-un est grande.

E. Les contraintes de dépendance fonctionnelle

Une ou plusieurs dépendance(s) fonctionnelle(s) peu(ven)t exister entre des champs d'un type de segment. Le problème est qu'il faut d'abord déterminer les champs formant une dépendance fonctionnelle. Il faut ensuite prouver qu'on a bien affaire à une dépendance fonctionnelle. Ce genre de contrainte est très difficile à prouver. Il nous semble que l'analyse du flux de données est la plus susceptible de donner des résultats.

En utilisant le graphe de dépendance, le rétro-ingénieur peut repérer plus aisément l'exactitude de son hypothèse concernant l'existence de la dépendance fonctionnelle suspectée. En effet, si les variables sont liées dans les graphes de dépendance, c'est un indice qui renforce l'idée de la dépendance fonctionnelle.
D'autres part, si l'on découvre dans les programmes des instructions ou des séquences d'instructions qui lient les champs suspectés tout en établissant la dépendance fonctionnelle qui les relient, alors nous sommes en présence d'une possible confirmation de dépendance fonctionnelle.
Par exemple, on suspecte une dépendance fonctionnelle:
TOTAL = SOUS-TOTAL * (1+TVA)
Si l'on découvre dans un programme l'instruction

```
COMPUTE TOTAL = SOUS-TOTAL * (1+TVA)
```

ou encore

```
COMPUTE TEMP = 1+TVA
COMPUTE TOTAL = SOUS-TOTAL*TEMP,
```

et que cette suite d'instruction se déroule pour chaque valeur des champs concernés,
alors, on obtient un indice susceptible de confirmer la dépendance fonctionnelle entre TOTAL, SOUS-TOTAL et TVA.

F. Les redondances

Un type de segment inclut des champs dont les valeurs sont dérivées d'autres valeurs de champs, issus du même type de segment ou d'un autre.
Les redondances sont un cas particulier des dépendances fonctionnelles (simple égalité de champ à champ). Une contrainte de redondance implique que les ensembles concernés sont identiques. L'analyse du flux de données et la recherche de patterns d'analyse peuvent apporter des résultats.

F.1. L'analyse du flux de données

Tout comme pour les dépendances fonctionnelles, le graphe de dépendance permet au rétro-ingénieur de vérifier plus aisément l'hypothèse de la redondance suspectée. En effet, si les deux variables sont liées dans le graphe de dépendance, c'est un indice qui renforce l'idée de redondance.

F.2. La recherche de patterns d'analyse

Les redondances sont très difficiles à repérer à travers les programmes. On peut cependant remarquer qu'une instruction du type MOVE suivie d'un accès à la BD peut être un indice de la présence de redondance.
Plusieurs cas de figure sont possibles concernant les accès à la BD.

- Pattern 1: L'accès pour suppression

Soit la suite d'instructions de la Figure 8-26.

MOVE VAR1 TO BSSA-BS2. CALL BD GHU Bpcb Bvar BSSA. IF STATUS-CODE=SEGMENT-FOUND THEN CALL BD DELETE Bpcb Bvar BSSA MOVE VAR1 TO ASSA-AS1 CALL BD GHU Apcb Avar ASSA CALL BD DELETE Apcb Avar ASSA END-IF.	READ-FIRST B(B2=A.A1); IF FOUND THEN DELETE B; DELETE A;

Figure 8-26: Pattern 1 concernant les redondances

Explication:
Si A est le nom d'un segment,
si A1 est un champ de A,
si B est le nom d'un segment,
si B2 est un champ de B,
si ASSA est une variable constituant un SSA de A,
si ASSA-AS1 est une variable appartenant à ASSA, correspondant au champ A1 de A,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS2 est une variable appartenant à BSSA, correspondant au champ B2 de B,
si Apcb, Bpcb sont deux PCB,
si Avar, Bvar sont des variables destinées à recevoir les résultats des requêtes,
si VAR1 est une variable contenant la valeur d'un champ A1 d'un segment A.
alors, cette suite d'instructions peut être un indice de la présence de cette redondance.

En effet, on affecte d'abord la valeur de la variable VAR1, donc celle du champ A1, au champ BS2 du SSA de B. Ils ont donc même valeur. On consulte ensuite la BD à la recherche de l'occurrence du type de segment B contenant la valeur présente dans le SSA de B. On efface le type de segment B pointé, et on fait de même avec le type de segment A. On aura donc toujours les mêmes valeurs dans les deux champs. Cependant, cet indice est très faible car rien ne dit que dans une autre partie de code on ne découvrira pas une affectation d'une valeur dans l'un des champ et pas dans l'autre.

- Pattern 2: L'accès pour insertion

Ce pattern est le pendant du précédent. Soit la suite d'instructions de la Figure 8-27.

MOVE VALB2 TO BSSA-BS2. CALL BD INSERT Bpcb Bvar BSSA. MOVE VALB2 TO ASSA-AS1. CALL BD INSERT Apcb Avar ASSA.	INSERT B(B2=VALB2); INSERT A(A1=VALB2);

Figure 8-27: Pattern 2 concernant les redondances

Explication:
Si A et B sont deux segments,
si A1 est un champ de A,
si B2 est un champ de B,
si ASSA est une variable constituant un SSA de A,
si ASSA-AS1 est une variable appartenant à ASSA, correspondant au champ A1 de A,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS2 est une variable appartenant à BSSA, correspondant au champ B2 de B,
si Apcb, Bpcb sont deux PCB,
si Avar, Bvar sont des variables destinées à recevoir les résultats des requêtes,
si VALB2 est une variable,

alors, cette suite d'instructions peut être un indice de la présence de cette redondance.

En effet, si l'on insère un segment B avec la variable VALB2 comme valeur du champ B2 de B, et qu'ensuite on insère un type de segment A avec le champ A1 de valeur égale à VALB2 (deuxième instruction MOVE), on est peut-être face à une redondance. En effet, lorsque l'on insère une valeur dans le champ B2, on l'insère également dans le champ A1.

- Pattern 3: L'accès pour remplacement

Soit la suite d'instructions de la Figure 8-28.

MOVE VALB2 TO BSSA-BS2. CALL BD GHU Bpcb Bvar BSSA. MOVE VALB3 TO BSSA-BS2. CALL BD REPLACE Bpcb Bvar BSSA. MOVE VALB2 TO ASSA-AS1. CALL BD GHU Apcb Avar ASSA. MOVE VALB3 TO ASSA-AS1. CALL BD REPLACE Apcb Avar ASSA.	READ-FIRST B(B2=VALB2); REPLACE B(B2=VALB3); READ-FIRST A(A1=VALB2); REPLACE A(A1=VALB3);

Figure 8-28: Pattern 3 concernant les redondances

Explication:
Si A et B sont deux segments,
si A1 est un champ de A,
si B2 est un champ de B,
si ASSA est une variable constituant un SSA de A,
si ASSA-AS1 est une variable appartenant à ASSA, correspondant au champ A1 de A,
si BSSA est une variable constituant un SSA de B,
si BSSA-BS2 est une variable appartenant à BSSA, correspondant au champ B2 de B,
si Apcb, Bpcb sont deux PCB,
si Avar, Bvar sont des variables destinées à recevoir les résultats des requêtes,
si VALB2 et VALB3 sont deux variables,
alors, cette suite d'instructions peut être un indice de la présence de cette redondance.

En effet, si l'on remplace la valeur du champ B2 du segment B par la valeur de la variable VALB3, et qu'ensuite on remplace de la même façon la valeur du champ A1 du type de segment A, on est peut-être face à une redondance.

- Conclusion

Les trois patterns que l'on vient de décrire peuvent aider à détecter une situation de redondance. Cependant, il faut remarquer que tous les remplacements, insertions et suppressions concernant les champs suspectés devront suivre le même canevas. Ces trois patterns doivent donc être considérés comme un ensemble.

G. Les contraintes de cardinalité supérieure

Le DDL du SGBD IMS ne permet pas de définir directement une limite supérieure d'occurrences pour un type de relation physique.
Le mot-clé FREQ de l'instruction SEGM nous renseigne sur le nombre moyen d'occurrences du type de segment enfant attachées à une même occurrence du type de segment parent. Il ne peut être considéré comme une limite de cardinalité supérieure du rôle joué par le type d'entité parent dans le type d'association *one-to-many*.

Cependant, si par une étude statistique, il s'avère que ce nombre moyen d'occurrences est une limite supérieure constante pour toutes les occurrences du type de relation physique, alors la cardinalité supérieure du rôle joué par le type d'entité parent dans le type d'association issu de ce type de relation physique, sera modifiée par la valeur du mot-clé FREQ.

Cas particulier:
Lorsque le mot-clé POINTER de l'instruction SEGM du DL/1 a la valeur NOTWIN, cela signifie qu'il n'y a qu'une seule occurrence permise pour ce type de segment (défini par l'instruction SEGM).
La cardinalité supérieure du rôle joué par le type d'entité parent devient dans ce cas égale à 1.
Toute autre information éventuelle sur une limite supérieure de cardinalité peut être découverte dans les programmes.
Les programmes peuvent contenir des procédures dont le but est de maintenir une contrainte de cardinalité supérieure à ne pas dépasser. Mais les procédures d'insertion ou de consultation des segments enfants qui jouent un rôle dans le type de relation physique « suspecté » doivent être analysées avec attention.
Si l'insertion (ou même la consultation) se fait au sein d'une boucle ayant comme limite un nombre, il peut s'agir d'une limite de cardinalité supérieure. La recherche de patterns d'analyse semble donc un bon outil.

G.1. Pattern 1: L'insertion de segments enfants au sein d'une boucle

PERFORM INSERT-SEGM-PARENT. PERFORM INSERT-SEGM-ENFANT VARYING compteur FROM 1 BY 1 UNTIL compteur > 10. INSERT-SEGM-PARENT. CALL BD INSERT parent-pcb parent-var parent-SSA. INSERT-SEGM-ENFANT. CALL BD INSERT enfant-pcb enfant-var enfant-SSA.	INSERT SEGM_PARENT; WHILE 1<=10 DO INSERT SEGM_ENFANT;

Figure 8-29: Pattern 1 concernant les cardinalités supérieures

Explication:
Si SEGM-PARENT et SEGM-ENFANT sont deux segments,
si parent-SSA est le nom d'un SSA de SEGM-PARENT,
si enfant-SSA est le nom d'un SSA de SEGM-ENFANT,
si parent-pcb et enfant-pcb sont deux PCB,
si parent-var et enfant-var sont des variables destinées à recevoir les résultats des requêtes,
alors, cette suite d'instructions peut être un indice de la présence d'une limite supérieure de cardinalité(boucle).
Dans ce pattern, l'analyste voit clairement qu'après avoir inséré une occurrence du type de segment parent, le programme insère 10 occurrences d'un segment enfant de ce parent. Si, à chaque insertion d'un type de segment parent, on insère toujours dix occurrences du type de segment enfant, l'analyste peut en déduire que la cardinalité maximale et la cardinalité minimale sont de 10. Les conclusions peuvent donc aussi concerner une limite inférieure. La découverte de ce pattern doit être accompagnée d'autres, au niveau des données notamment.

G.2. Pattern 2: La vérification avant insertion

MOVE 0 TO CP. MOVE 0 TO STOP. PERFORM VERIFICATION UNTIL STOP=1. IF CP<NB-MAX THEN CALL BD INSERT enfant-pcb enfant-var enfant-SSA. VERIFICATION. CALL BD GN enfant-pcb enfant-var enfant-SSA. IF STATUS-CODE = SEGMENT-NOT-FOUND THEN STOP=1 ELSE COMPUTE CP=CP+1.	STOP=0. CP=0. WHILE STOP<>1 DO READ(SEGM_ENFANT); IF status<>OK THEN STOP=1 ELSE CP=CP+1; IF CP < NB-MAX THEN INSERT(SEGM-ENFANT);

Figure 8-30: Pattern 2 concernant les cardinalités supérieures

Explication:
Si SEGM-PARENT et SEGM-ENFANT sont deux segments,
si parent-SSA est le nom d'un SSA de SEGM-PARENT,
si enfant-SSA est le nom d'un SSA de SEGM-ENFANT,
si parent-pcb et enfant-pcb sont deux PCB,
si parent-var et enfant-var sont des variables destinées à recevoir les résultats des requêtes,
alors, cette suite d'instructions peut être un indice de la présence d'une limite supérieure de cardinalité (boucle).
Le programme insère une nouvelle occurrence du type de segment enfant si le nombre maximum de types de segments enfants n'est pas déjà atteint. La valeur de la cardinalité supérieure est donc la valeur de la variable NB-MAX. Lorsque ce pattern a été repéré, l'analyste doit déterminer la valeur de NB-MAX. Il peut effectuer cette tâche au moyen de l'analyse du flux des données ou du program slicing.

H. Les contraintes de cardinalité inférieure

Il n'est pas possible de spécifier le nombre minimum d'enfants qu'un segment parent doit avoir dans le DL/1. C'est donc dans les programmes qu'il nous faut trouver des preuves pour connaître la limite de cardinalité inférieure. Une fois encore, les procédures d'insertion sont importantes. Par exemple, si on insère une occurrence du segment parent, et que dans tous les cas on insère ensuite au moins une occurrence de son enfant, la cardinalité inférieure du rôle joué par le type d'entité parent dans le type d'association est de 1.

8.4. Remarque à propos des commentaires

La recherche d'informations et d'indices peut se faire dans les divers *commentaires* faits dans les programmes. L'information fournie ne confirme peut-être pas toujours la présence de l'élément suspecté. Cependant, elle permet à l'analyste d'avoir une idée générale de la structure du programme. A partir de ces informations, la recherche d'indices peut s'effectuer d'une manière plus efficace.

Les commentaires présents dans les programmes sont une source importante d'information. Il faut cependant vérifier leur fiabilité et leur correction. Certains programmes sont parfois totalement dépourvus de commentaires.

Nous avons pris l'exemple de commentaires trouvés dans des programmes analysés lors du stage, qui permettent à l'analyste de renforcer l'hypothèse de l'existence d'une clé étrangère.

Certains de ces commentaires mentionnent explicitement la présence de clés étrangères.

Par exemple:

```
* CHECK RSROOT-PARTNO IN PAROOT-KEY *
```

D'autres en suggèrent la présence.

Par exemple:

```
* THIS SECTION TRIES TO INSERT A GRROOT. *
```

Si on suspecte une clé étrangère d'un champ de GRROOT vers un autre champ, il convient de vérifier que dans le morceau de code relatif au commentaire, on effectue bien un contrôle sur la présence de la valeur à insérer dans le champ référé.

8.5. Conclusion

L'analyse des programmes est une des étapes la plus délicate à aborder pour le rétro-ingénieur, car il ne dispose pas d'outils aussi directs que des extracteurs. De plus, les programmes à analyser peuvent être très nombreux et sont souvent mal documentés. Malgré toutes ces difficultés, un schéma conceptuel ne sera complet que si ces programmes ont été bien analysés, surtout dans le cas de SGBD comme IMS, dans lesquels on ne peut exprimer qu'un minimum de contraintes de façon explicite dans le DDL.

CHAPITRE 9: L'analyse des données

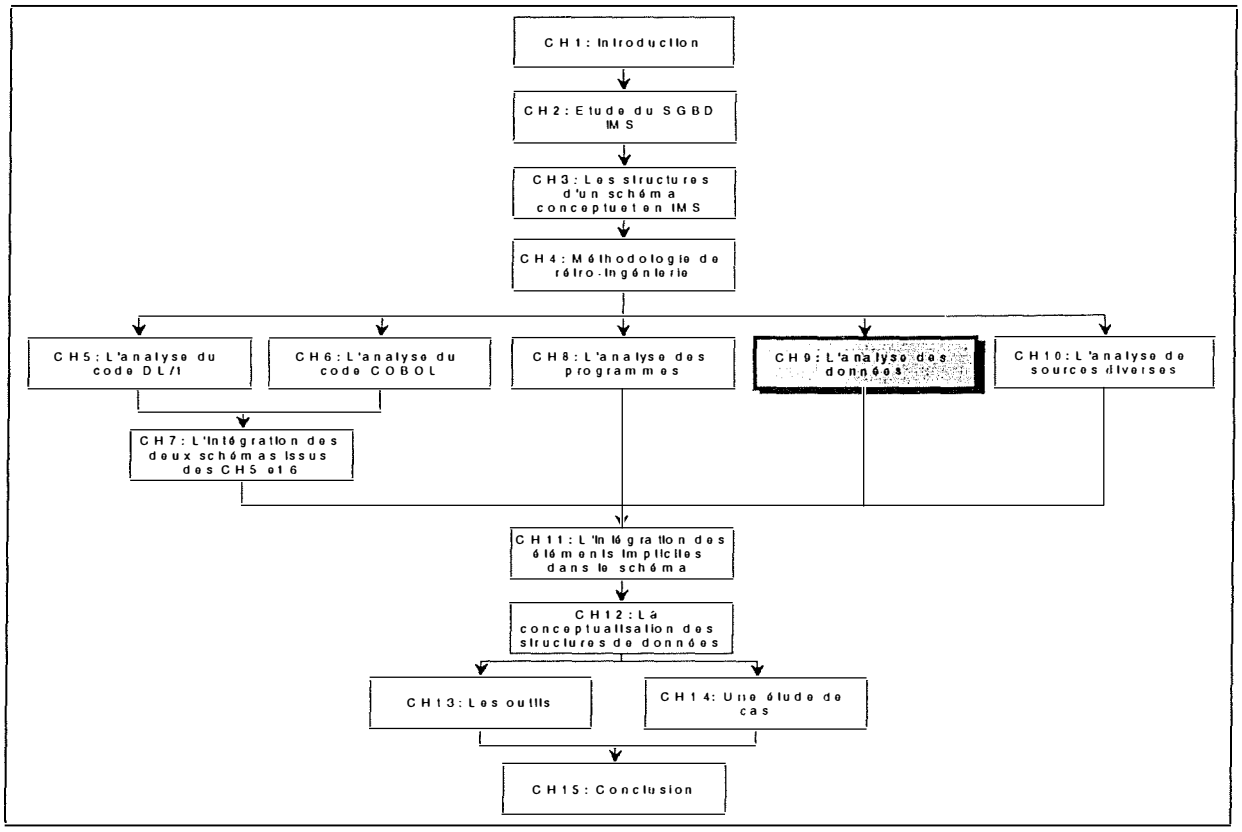


Figure 9-1: Le chapitre 9

9.1. Introduction

L'analyse des données est une étape importante dans le processus de rétro-ingénierie, et en particulier pour la découverte et la confirmation d'éléments cachés (structures implicites). Cette étape permet de confirmer certaines informations découvertes auparavant et pour lesquelles une confirmation est nécessaire. Ainsi, on peut compléter un indice découvert dans un programme par une analyse des données. Cette analyse permet également de découvrir de nouveaux éléments inconnus avant ce stade.

Pour effectuer cette analyse, il peut être intéressant de réaliser des programmes qui parcourent automatiquement le contenu de la base de données. Ces programmes ont pour objet d'analyser les BD dans le but de confirmer certaines hypothèses, ou d'en découvrir de nouvelles. Ils ne doivent en aucun cas altérer la BD. Dans la mesure du possible, ils ne doivent pas alourdir les performances du système.

Afin que l'analyse de la BD soit efficace et donne des résultats intéressants et exploitables, il est nécessaire d'exécuter ces programmes d'analyse sur des BD de tailles importantes. En effet, l'analyse d'une BD de taille modeste pourrait confirmer des découvertes erronées.
Par exemple,

1. La présence d'un champ identifiant pourrait être confirmée en analysant une BD peu volumineuse et pourrait être infirmée sur une BD de taille plus importante.
2. Pour les recherches sur les cardinalités, la taille de la BD doit impérativement être importante pour pouvoir poser des hypothèses sûres. Plus la taille de la BD sera grande, plus les éléments découverts et les hypothèses posées seront certains.

D'une manière générale, on peut dire que tous les éléments découverts peuvent être confirmés de nouveau par l'analyse des données. Cependant, certaines informations sont ici plus aisément découvertes que d'autres. Cela s'explique par la possibilité variable d'automatiser les recherches concernant certains éléments.
Par exemple,

1. Il est très aisé de construire un programme qui confirmera la présence d'un identifiant.
2. La construction d'un programme vérifiant qu'un champ est en réalité décomposable est plus difficile et nécessite, dans la plupart des cas, la connaissance de la signification des données.

Il y a principalement trois méthodes d'analyse des données:

1. La construction de programmes d'analyse des données.
2. L'analyse des données sur base de la signification du contenu.
3. La construction de programmes d'analyse automatique (générateurs).

Ce chapitre neuf détaille ces trois méthodes.

9.2. Les constructions supposées

Les constructions supposées sont des constructions que l'analyste aimerait confirmer. Elles peuvent avoir été découvertes lors de l'analyse des programmes ou lors de l'analyse d'autres sources. Le rétro-ingénieur décide d'analyser les données, soit pour obtenir une première confirmation de ses suppositions (dans le cas où il n'a pas réussi à en découvrir lors des phases précédentes), soit pour faire apparaître une confirmation supplémentaire (dans le cas où il a déjà découvert des confirmations de la présence de telles structures lors des phases précédentes).
Nous allons examiner successivement trois méthodes d'analyse des données, qui sont la construction de programmes d'analyse de données, l'analyse des données sur base de la signification du contenu et la construction de programmes d'analyse automatique.

9.2.1. La construction de programmes d'analyse des données

Construire un petit programme chargé de consulter une base de données pour confirmer une hypothèse est la première méthode que nous développons. **[RICHARD, 95]** donne également un aperçu de cette technique.
Les éléments suivants sont susceptibles d'être confirmés suite à une analyse automatisée des données:

1. Les identifiants,
2. Les clés étrangères,
3. Les dépendances fonctionnelles,
4. Les cardinalités,
5. Les contraintes de coexistence,
6. Les contraintes d'au-moins-un,
7. Les contraintes d'exclusion,
8. Les attributs facultatifs.

1. Les identifiants

La confirmation de la présence d'identifiants est ce qu'il y a de plus aisé à automatiser. Il suffit en effet de construire un programme qui consulte toute la BD en vérifiant qu'une même valeur ne se retrouve pas plus d'une fois dans le champ suspecté.
Si les résultats du programme montrent qu'il existe plusieurs occurrences d'une même valeur pour le champ étudié, on peut conclure que l'hypothèse de la présence d'un identifiant n'est pas correcte.
D'autre part, si les résultats n'indiquent pas qu'une même valeur est présente plusieurs fois dans le champ étudié, alors on ne peut en aucun cas affirmer que la présence d'un identifiant est prouvée. En effet, il est possible qu'au moment de l'analyse, l'état de la BD est tel que l'on est dans une situation où l'hypothèse est vérifiée et que dans le futur, cela ne soit plus le cas.

2. Les clés étrangères

La confirmation de la présence de clés étrangères est elle aussi très aisément automatisable. Il suffit de construire un programme qui pour chaque valeur présente dans le champ référençant vérifie la présence de cette même valeur dans le champ référencé.
Comme pour le cas des identifiants, si l'on découvre une occurrence qui ne vérifie pas la contrainte de clé étrangère, on peut penser que l'hypothèse n'est pas correcte.

Dans le cas contraire, cela renforce l'hypothèse de l'existence de la structure cachée sans pour autant la confirmer.

3. Les dépendances fonctionnelles

Une dépendance fonctionnelle est définie sur plusieurs champs. Il suffit de créer autant de listes que de champs présents dans la dépendance et de vérifier la relation suspectée pour chaque ligne.
Par exemple, si l'on suspecte que TOTAL = TOT1 + TOT2, on construit trois listes (Figure 9-2).

TOT1	TOT2	TOTAL
15	16	31
25	4	29
...	...	...

Figure 9-2: Exemple de table de dépendance fonctionnelle

Il faut évidemment que cette relation soit vérifiée pour chaque ligne. Si elle est vérifiée dans tous les cas, alors l'hypothèse de la présence de la dépendance fonctionnelle est renforcée.

4. Les cardinalités

Il est aisé d'estimer les cardinalités minimales et maximales par l'analyse des données. En effet, une suite de *CALL* (au sein d'une boucle) de type *GET NEXT WITHIN PARENT* montre le nombre de segments enfants que possède chaque segment parent.
Si l'on retient les valeurs minimale et maximale pour l'ensemble de la BD, nous obtenons ainsi des cardinalités. Ces cardinalités seules n'ont pas de sens. En effet, elles sont susceptibles d'être modifiées à chaque nouvelle insertion d'un élément dans la BD. Elles n'ont de sens que si elles confirment une suspicion découverte lors de l'analyse des programmes.

5. Les contraintes de coexistence

On peut également vérifier l'existence de cette contrainte. Il suffit de s'assurer que pour chaque occurrence d'un champ, le(s) autre(s) champ(s) suspecté(s) d'être visé(s) par la contrainte a(ont) aussi une valeur (ou aucun des champs n'a de valeur).

6. Les contraintes d'au-moins-un

Ce type de contrainte se vérifie de la même manière que la précédente à ceci près qu'un seul des champs ou tous les champs suspecté(s) d'être visé(s) par la contrainte a(ont) aussi une valeur.

7. Les contraintes d'exclusion

Les contraintes d'exclusion se vérifient de la même manière que les précédentes. Il convient de rechercher si un seul des champs suspectés d'être visés par la contrainte a une valeur.

8. Les attributs facultatifs

Nous pouvons ici faire apparaître des éléments qui n'étaient pas visibles dans le DL/1. Il convient de vérifier les valeurs des champs suspectés et si les valeurs caractéristiques NULL, " ", HIGH-VALUE, VOID ou d'autres plus spécifiques apparaissent, on peut conclure que le champ est bien facultatif.

9.2.2. L'analyse des données sur base de la signification du contenu

La confirmation de la présence de certaines structures cachées au travers de l'analyse des données exige parfois la connaissance de la signification des éléments de la BD ou tout au moins de la partie étudiée. Pour cette raison, ces confirmations ne sont pas toujours réalisables. Dans certains cas, détecter la décomposition d'un champ en analysant uniquement les données implique la connaissance précise de la signification de celles-ci. Il existe cependant des cas relativement évidents:
soient deux champs qui représentent une adresse:

Rue Grandgagnage	2	5000	Namur
Chaussée de Charleroi	150	1000	Bruxelles

L'analyste peut découvrir que le contenu de ce champ peut être décomposé et peut également en découvrir la signification. Dans la majorité des autres cas, la découverte de la décomposition n'est pas si aisée.
Il en va de même pour les attributs facultatifs ou multivalués.

9.2.3. La construction de programmes d'analyse automatique

Dans ce paragraphe, nous exposons une méthode de construction de programmes d'analyse de données. Deux possibilités s'offrent à l'analyste:
1. La réalisation d'un programme unique qui vérifie toutes les structures supposées.
2. La réalisation d'un programme de vérification pour chaque structure suspectée.

La première possibilité semble à première vue plus difficile à réaliser, voir irréalisable. La seconde peut être automatisée en partie. En effet, prenons l'exemple des clés étrangères. Une méthode efficace pour détecter la présence de clés étrangères ou d'autres éléments dont on cherche une confirmation est de (pour l'exemple des clés étrangères):

1. Créer un programme (P1) qui confirme une clé étrangère.
2. Créer un programme (P2) qui générera le code du programme P1 pour la confirmation d'une autre clé étrangère.

Cette méthode est également applicable à toutes les autres recherches sur les données.
Cela permet d'écrire un seul programme de vérification des données pour les clés étrangères et d'en créer automatiquement d'autres qui lui sont pratiquement identiques.
Lors de notre stage, nous avons réalisé un tel programme pour la découverte de la présence de clés étrangères. Ce programme est expliqué plus longuement dans le chapitre treize concernant les outils.

9.3. La découverte de nouvelles constructions

La découverte de nouvelles structures lors de l'analyse des données peut s'avérer bien délicate. En effet, lors de la recherche de confirmations d'éléments supposés par l'analyste, celui-ci connaît ce qu'il recherche. Le champ des recherches est alors réduit. Dans le cas qui nous préoccupe ici, l'analyste n'a pas la moindre idée de ce qu'il recherche. Il ne recherche rien de bien précis. S'il découvre l'existence de structures auxquelles il n'avait pas pensé, c'est dans beaucoup de cas le fruit du hasard.

Nous avons inséré ce point pour faire remarquer que si le rétro-ingénieur découvre de nouvelles structures cachées par hasard, celles-ci doivent être prises en compte au même titre que les autres. L'analyste peut alors envisager une analyse des programmes pour appuyer sa découverte.

9.4. Conclusion

L'analyse des données comprend deux parties bien distinctes.
D'une part, la confirmation de certaines suppositions. Cette confirmation peut être dans certains cas automatisée, ce qui facilite grandement la tâche de l'analyste. D'autres confirmations se basent sur le contenu, la structure de certains enregistrements. Ces confirmations sont évidemment beaucoup plus difficiles à obtenir étant donné qu'il n'est pratiquement pas possible de les automatiser et que l'examen "à la main" de ces confirmations peut être très fastidieux. Il existe également une autre optique qui consiste à utiliser un programme qui génère un autre programme de vérification d'une structure implicite bien précise.
D'autre part, l'analyste peut découvrir des structures auxquelles il n'avait pas pensé. Cette partie est nettement moins aisée car la découverte de nouveaux éléments peut s'avérer très aléatoire.

CHAPITRE 10: L'analyse de sources diverses

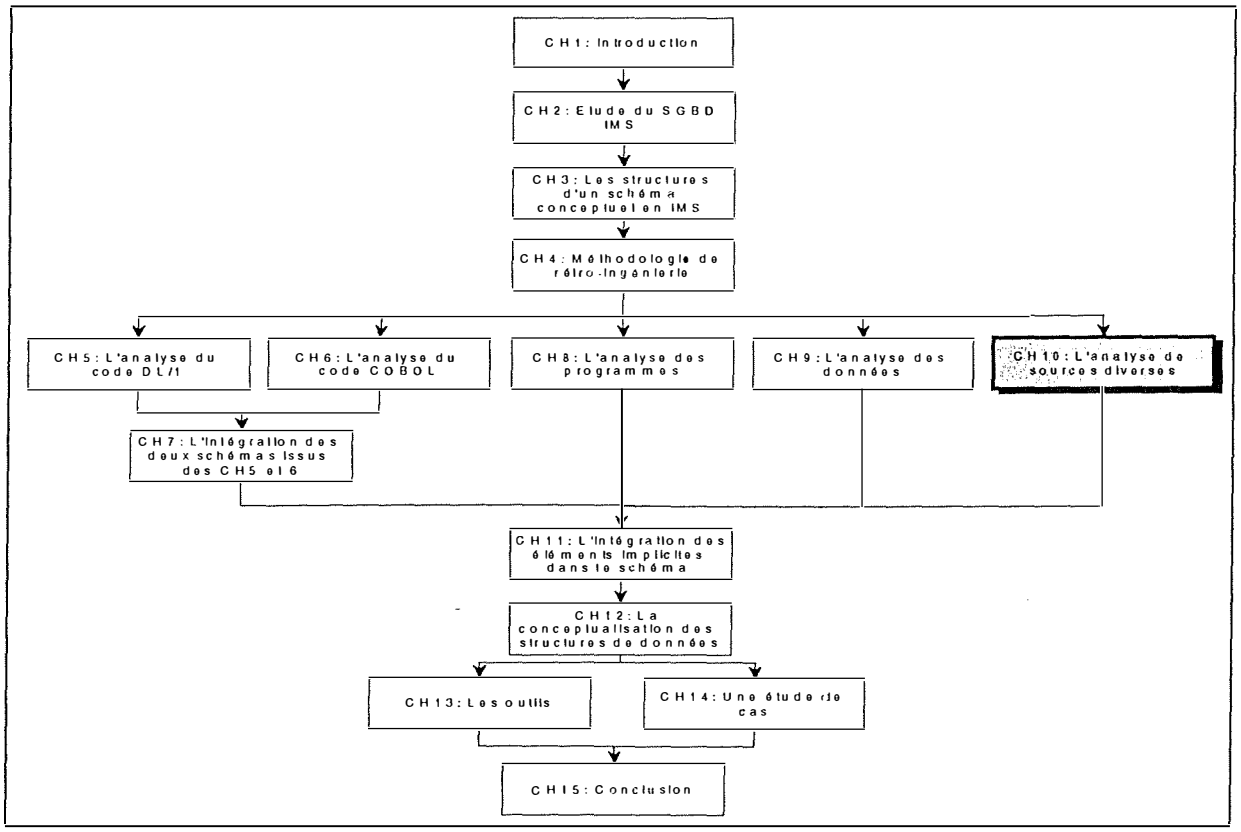


Figure 10-1: Le chapitre 10

10.1. Introduction

Le chapitre dix aborde très brièvement diverses sources qui peuvent aider le rétro-ingénieur dans sa recherche de structures ou de contraintes implicites.
Ces sources sont:

1. La connaissance du domaine
2. La documentation existante
3. Les écrans et les rapports
4. L'exécution des programmes
5. L'analyse des noms
6. Les constructions d'accès

Pour rappel, ces sources ont été proposées par **[HAINAUT, 95a]**.

Les paragraphes suivants s'attachent à développer brièvement ces diverses sources. Nous clorerons par une conclusion.

10.2. Les diverses sources

10.2.1. La connaissance du domaine

Il est inconcevable de débuter un projet de rétro-ingénierie sans posséder une certaine connaissance du domaine d'application. En effet, il est nécessaire de débuter avec une première représentation mentale (modèle mental) des objectifs et des principaux concepts de l'application. L'analyste peut considérer le système existant comme une implémentation de ce modèle. Pour acquérir cette connaissance, les utilisateurs actuels et éventuellement les anciens peuvent apporter à l'analyste des informations capitales sur les fonctions du système.
Par exemple:

1. Si le projet de rétro-ingénierie concerne une base de données des commandes d'une entreprise, il est nécessaire de savoir qu'une commande est composée d'un ensemble de lignes de commande.
2. Par leur expérience, les utilisateurs savent qu'un client doit avoir un nom et une adresse; donc, dans la structure de la Figure 10-2, le champ CLI-INFO devrait inclure ces informations.

```
01 CUTOMER-RECORD.
   02 CLI-IDENT PIC 9(6).
   02 CLI-INFO PIC X(80).
```

Figure 10-2: Un exemple

La connaissance du domaine apporte de l'information tout au long du processus de rétro-ingénierie.

10.2.2. L'analyse de la documentation existante

Dans la plupart des projets de rétro-ingénierie, une documentation est disponible. Cependant, cette documentation est souvent incomplète ou même erronée. Malgré cela, elle peut être une source précieuse d'information. Les éléments les plus importants (entités, associations, identifiants, ...) y sont décrits. Les commentaires ajoutés par les programmeurs sont eux aussi très intéressants. Il faut cependant évaluer leur correction ainsi que leur fiabilité.

10.2.3. L'analyse des écrans et des rapports

Les écrans et les rapports peuvent être considérés comme des vues dérivées des données. L'affichage des données en output, des titres et des commentaires peuvent fournir de l'information essentielle au sujet des données.
Par exemple:
La valeur du champ CLI-DATA est affichée dans le champ ADRESSE de la boîte de dialogue CUSTOMER. Donc, on pourrait renommer CLI-DATA en ADRESSE.

10.2.4. L'exécution des programmes

Le comportement dynamique d'un programme sur les données fournit de l'information au sujet des caractéristiques que les données doivent posséder pour être enregistrées dans les fichiers et des liens entre les données enregistrées. L'exécution des programmes combinée avec l'analyse des données fournit un moyen d'examen puissant pour détecter les structures et les propriétés des données.

Par exemple, si une application demande que l'utilisateur introduise les noms de fournisseurs pour un produit et que lorsqu'il a introduit le onzième nom, un message d'erreur s'affiche, alors on peut conclure que la cardinalité maximale du nombre de fournisseurs par produit est de 10.

10.2.5. L'analyse des noms

Parfois, le nom d'une variable ou d'un champ suggère la signification de celle(celui)-ci.
Par exemple, le champ CLI-ID pourrait être l'identifiant du type de segment CUSTOMER.

10.2.6. Les constructions d'accès

Certaines constructions techniques (clé d'accès, champ anormalement long, ...), peuvent fournir des indices concernant certaines structures et contraintes logiques.
Par exemple, la plupart des clés étrangères sont supportées par un index; donc, la présence d'un index pourrait être un indice de la présence de clés étrangères.

10.3. Conclusion

Tout au long du processus de rétro-ingénierie, une analyse des sources vues au paragraphe 10.2 peut apporter des renseignements capitaux. Ces sources ne doivent pas être négligées.

CHAPITRE 11: L'intégration des éléments implicites découverts dans le schéma

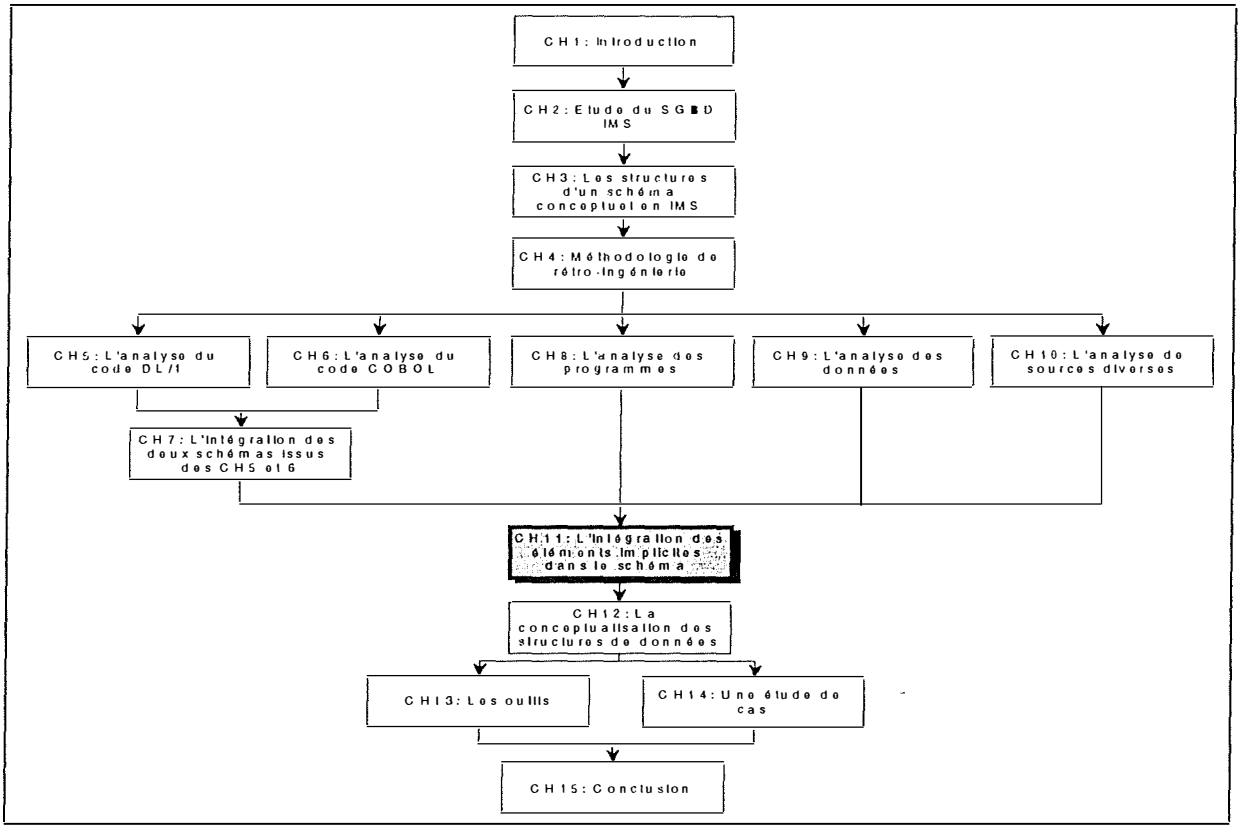


Figure 11-1: Le chapitre 11

11.1. Introduction

Le chapitre onze décrit la phase d'intégration durant laquelle le rétro-ingénieur, parfois accompagné d'un expert de l'application analysée, examine(nt) les résultats des recherches d'éléments implicites dans les différentes sources, afin de déterminer les éléments qu'il convient d'insérer au schéma du chapitre sept.

Les décisions prises lors de cette étape sont loin d'être évidentes. Bien sûr, dans certains cas, on aura trouvé tellement d'indices concernant un même élément dans plusieurs sources différentes que l'on décidera de l'insérer au schéma, **en ayant soin de toujours conserver la trace des indices ayant permis de trouver cet élément**. Mais dans d'autres cas, le choix d'insérer un élément dont on soupçonne l'existence en se basant uniquement sur un ou deux indices peut être plus risqué, ou du moins plus subjectif.

Avant de décider l'insertion ou non de structures ou de contraintes dans le schéma, il nous semble que le rétro-ingénieur doit élaborer un rapport reprenant toutes les contraintes et structures cachées qu'il a découvertes lors des phases précédentes, avec la liste des indices les confirmant ou les infirmant. Ce sera la base des discussions avec l'expert de l'application.

11.2. L'expert et les degrés de certitude

Il est possible que le rétro-ingénieur soit une personne qui ne travaille que depuis peu sur l'application étudiée. Il semble donc évident qu'il n'est pas à même de décider seul si telle ou telle structure ou contrainte s'avère réaliste. L'avis d'un expert de l'application étudiée doit alors être demandé. Sur base de ses connaissances et des indices infirmant ou confirmant chaque hypothèse, l'expert décidera en connaissance de cause si oui ou non il convient d'intégrer une nouvelle structure implicite au schéma.

Comme nous venons de le dire dans l'introduction, le rétro-ingénieur peut produire un rapport qui contient pour chaque élément suspecté:

1. les éléments qui prouvent son existence et
2. les éléments qui infirment la suspicion

Comme il arrive souvent qu'on ne soit pas sûr à cent pourcent de l'existence réelle d'un élément donné, il peut être utile, dans ce rapport, de prévoir une case vide pour chaque structure ou contrainte implicite présente dans celui-ci. Dans cette case, un expert de l'application pourra 'quantifier' la certitude de l'existence de chaque structure/contrainte par un *degré de certitude*.

Ce degré de certitude est un nombre qui représente, qui quantifie la certitude que possède l'expert quant à l'exactitude de l'hypothèse. Ce degré de certitude peut prendre différentes formes. Nous en proposons deux. Cependant, on peut utiliser sa propre méthode d'évaluation. On peut imaginer que l'on définisse trois niveaux de certitude: 1, 2 et 3. Si, pour une structure donnée, la suspicion possède le degré de certitude 1, cela signifie que l'expert suspecte l'existence de celle-ci mais qu'il n'a pas en mains assez d'éléments la confirmant. Si le degré de certitude vaut 2, cela signifie que des indices ont été découverts et que l'expert possède des éléments confirmant son existence; cependant, il ne peut être sûr à cent pourcent de son existence 'réelle'. Le niveau de certitude 3 est réservé lorsque l'expert est pratiquement certain de l'exactitude de l'hypothèse, parce que les indices sont nombreux et parce qu'il connaît l'existence de cette structure, grâce à sa connaissance de l'application.
On peut aussi concevoir le degré de certitude sous la forme d'un rapport. Par exemple, une cote sur 10 qui indiquerait le niveau de certitude de l'expert.

Il est important de remarquer que le degré de certitude ne veut rien dire en lui-même. En effet pour pouvoir l'évaluer, l'apprécier, il convient qu'il soit accompagné des éléments positifs et négatifs qui amènent l'expert à le définir. Finalement, ce degré de certitude n'est rien d'autre qu'un moyen formel de 'coter ' la fiabilité des indices trouvés concernant une structure ou contrainte implicite.

La Figure 11-2 représente le processus de décision entre le rétro-ingénieur et l'expert, s'il y en a un. Nous y remarquons la présence de deux boucles de rétroaction. Cela signifie que l'expert et le rétro-ingénieur peuvent estimer que les indices obtenus pour certaines structures ne sont pas suffisants pour trancher.
Trois possibilités sont alors envisageables:

1. La structure qui pose problème n'est pas ajoutée au schéma.

2. La structure qui pose problème est ajoutée au schéma avec un degré de certitude peu élevé.
3. Le rétro-ingénieur approfondit son travail de recherche d'indices:
   - en réexaminant les informations qu'il a déjà découvertes,
   - en recommençant ou en complétant certaines analyses.

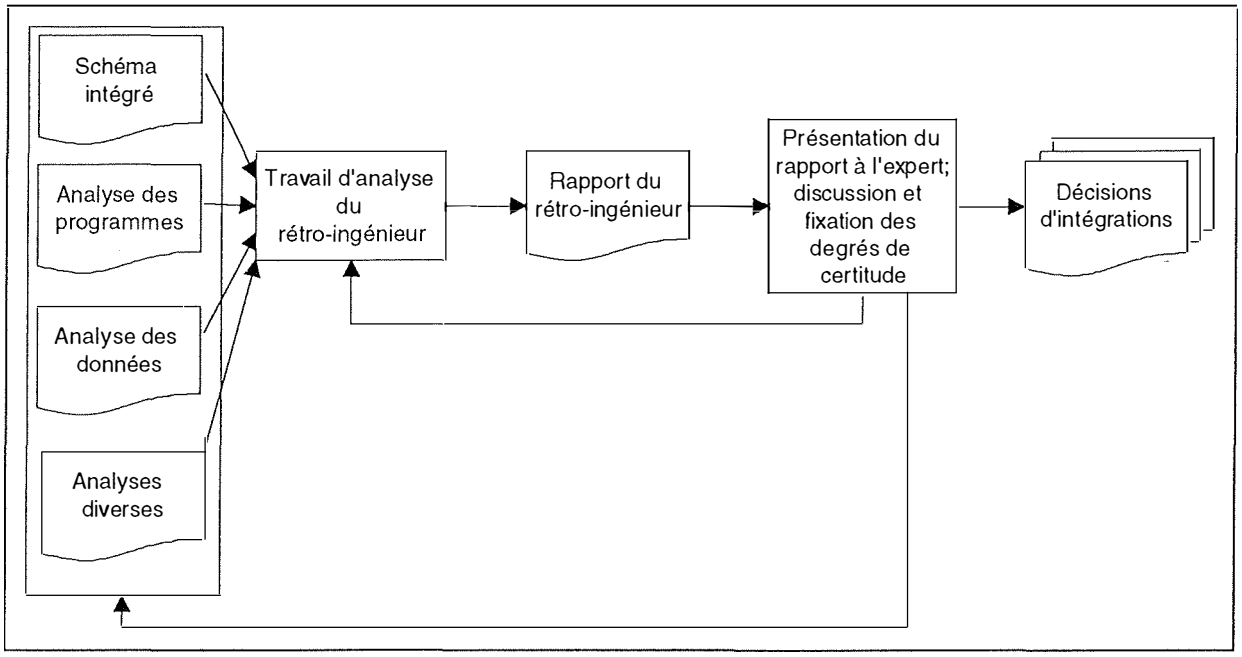


Figure 11-2: Le processus de décision

11.3. L'intégration des éléments retenus

Quand les éléments à intégrer ont été déterminés, on peut les intégrer au schéma issu du chapitre sept. Nous proposons deux stratégies d'intégration:

- la première stratégie consiste à créer un schéma pour chaque étape de l'analyse. Autrement dit, à partir du schéma intégré du chapitre sept, nous aurons trois schémas qui le suivent:
  - Le premier (appelé SCHEMA A), qui intégrera les résultats de l'analyse des programmes au schéma du chapitre sept.
  - Le deuxième (appelé SCHEMA B), qui intégrera les résultats de l'analyse des données au SCHEMA A.
  - Le troisième (appelé SCHEMA C), qui intégrera les résultats de l'analyse des sources diverses au SCHEMA B.

  Cette façon de faire permet de bien différencier les étapes de la recherche de structures implicites, mais présente l'inconvénient de ne pas être très souple. En effet, il est rare de consulter seulement une seule source pour confirmer l'existence d'une structure implicite. Lier une découverte à une seule source peut donc paraître un peu surfait. Il vaut mieux utiliser cette stratégie prudemment.

- La deuxième stratégie consiste à construire un seul schéma, qui englobera toutes les découvertes faites par le rétro-ingénieur. L'inconvénient de la première stratégie disparaît. De plus, lorsque le rétro-ingénieur doit faire approuver ses découvertes par un utilisateur chevronné de l'application, il lui sera plus simple de travailler sur un seul schéma.
  La Figure 11-3 représente cette deuxième stratégie. Chaque ellipse représente un schéma, avec le chapitre correspondant.

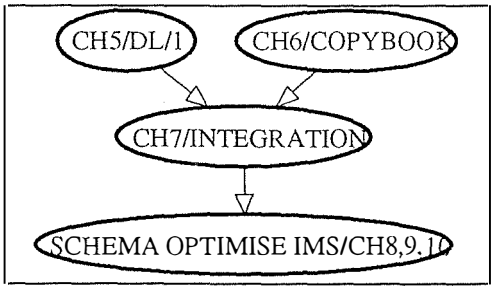


Figure 11-3: Deuxième stratégie d'intégration

Il est évident que ces deux stratégies ne sont rien d'autres que des propositions que le rétro-ingénieur est libre de suivre ou non. En ce qui nous concerne, la deuxième stratégie nous semble la meilleure, car la plus flexible, et la plus propre pour travailler.

11.4. Conclusion

A la fin de cette étape d'intégration, nous sommes en présence du schéma logique enrichi orienté IMS complet (ou du moins le plus complet possible). L'expert et le rétro-ingénieur se sont mis d'accord sur un schéma et c'est à partir de celui-ci que l'on peut commencer la phase de conceptualisation, qui nous emmènera vers le schéma conceptuel final.

CHAPITRE 12: La conceptualisation des structures de données

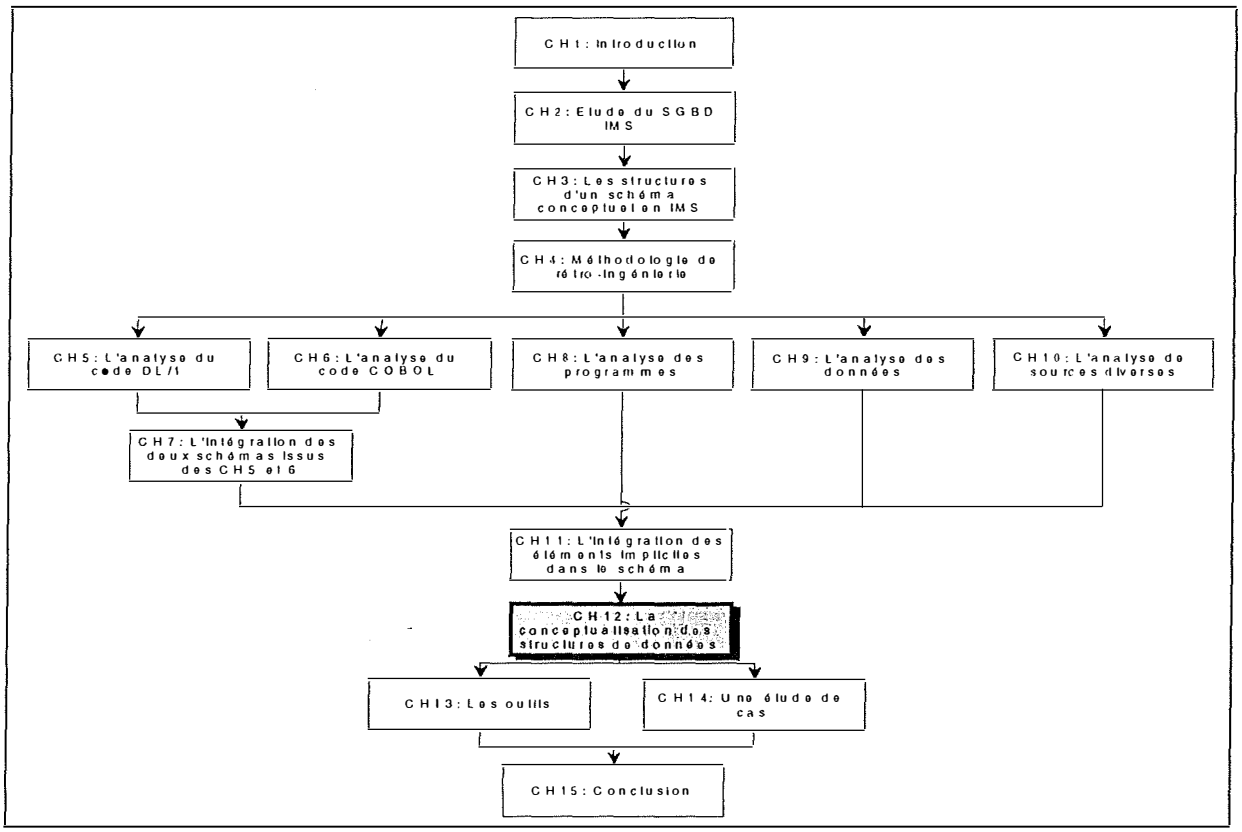


Figure 12-1 : Le chapitre 12

12.1. Introduction

Une fois que nous avons procédé à l'extraction du schéma logique IMS enrichi, il convient de le libérer de ses optimisations et des structures propres à IMS. Le but de ce chapitre est de montrer les différentes actions à accomplir pour obtenir un schéma conceptuel normalisé.
Pour rappel, la conceptualisation consiste en:

- **La préparation du schéma**, qui consiste à nettoyer, à modifier les différents noms appartenant au schéma pour les rendre plus compréhensibles.
- **La détraduction du schéma**, qui consiste à retrouver les structures conceptuelles desquelles le concepteur a dérivé les structures logiques.
- **La désoptimisation du schéma**, qui a pour but de retrouver les structures provenant d'une optimisation et à les éliminer si cela est nécessaire.
- **La normalisation conceptuelle**, qui consiste à donner au schéma des qualités telles que la simplicité, la lisibilité et la minimalité.

Ces quatre phases feront l'objet des quatre paragraphes suivants.

12.2. La préparation du schéma

Avant de commencer les phases de détraduction et de désoptimisation, il convient de passer par la phase préliminaire de préparation du schéma. Cette phase consiste en la modification de certains noms de manière à les rendre plus significatifs. Il faut éclaircir le schéma et le débarrasser de ses éléments techniques. Les points suivants reprennent une liste de tâches à accomplir lors de cette phase préliminaire.

12.2.1. Les noms des types d'entités et des attributs

Il convient de nettoyer, de modifier les noms des types d'entités et des attributs. Ces noms sont choisis par les concepteurs dans un double souci d'efficacité. Il n'est pas rare qu'un concepteur donne un nom à un type d'entité ou à un attribut qui lui rappelle à quel projet il appartient.
Par exemple, au cours de notre stage, nous avons remarqué (dans l'application étudiée) que les deux premières lettres de tous les types de segments rappellent le nom de la base de données à laquelle ils appartiennent: la base de données des clients porte le nom de CUSTDB et tous les types de segments de cette BD commencent par CU: CUROOT, CUDELV, CUWHSE,... Dans un tel cas, on pourrait modifier les noms de la manière suivante: CUROOT devient ROOT, CUDELV devient DELIVERY et CUWHSE devient WAREHOUSE[1].
Le concepteur peut être limité dans le choix des noms. Par exemple, dans certains SGBD, les noms doivent avoir un maximum de 8 caractères: STAT pour STATISTIQUES. Nous ferons remarquer qu'une autre possibilité pourrait être de traduire les noms de l'anglais vers le français pour les rendre encore plus clairs pour un lecteur francophone. Dans ce cas, CUROOT devient CLIENT, CUDELV devient LIVRAISON, et CUWHSE devient ENTREPOT.
Le but de cette première partie est donc de modifier tous les noms qui ne possèdent pas de signification claire par des noms dont la signification n'est pas ambiguë.

12.2.2. Les noms des rôles et des types d'associations du schéma

Il convient de donner des noms significatifs aux rôles et aux types d'associations du schéma. La structure hiérarchique ne permet d'avoir qu'un type de relation physique et un type de relation logique entre un type de segment parent et un type de segment enfant. Les noms des rôles et des types d'associations n'ont pas de raison d'exister dans ce modèle[2]. L'analyste doit essayer de retrouver des noms à ces types d'associations et à ces rôles qui seront adéquats. Pour cela, l'analyste ne peut compter que sur son bon sens et la connaissance de la BD qu'il a pu acquérir. Les noms des types d'associations et des rôles contribuent également fortement à éclaircir le schéma.

12.2.3. Les collections et les clés d'accès

Les collections et les clés d'accès peuvent être supprimées. En effet, elles n'apportent que des informations techniques liées à la conception physique. Les détails sur les supports de stockage et les moyens par lesquels on accède aux données sont sans importance.

---

[1] Il faut évidemment savoir ce que représente chaque type de segment.
[2] Pour rappel, les types d'associations du schéma logique sont numérotés suivants l'ordre de la hiérarchie et les rôles n'ont pas de noms.

12.2.4. Les descriptions techniques

Les descriptions techniques de chaque élément du schéma peuvent elles aussi être supprimées. Elles ne fournissent que des informations techniques sur les différents éléments. Les descriptions sémantiques sont par contre de la plus haute importance et sont précieusement conservées. Ces descriptions sont des éléments très importants au niveau conceptuel.

12.3. La détraduction du schéma

Le schéma de départ de cette phase de détraduction est conforme au SGBD utilisé, ici IMS. Cette phase consiste à détecter toutes les structures de données spécifiques à IMS et à les transformer en structures de données conceptuelles équivalentes. Durant la phase de détraduction, l'analyste tente de retrouver les structures conceptuelles desquelles le concepteur a dérivé les structures logiques du schéma source. Nous allons indiquer les moyens de détecter et d'appliquer les transformations que nous avons exposées au chapitre trois.

Remarque: Dans ce paragraphe, nous illustrerons notre propos d'exemples qui se présenteront toujours de la même façon. Les parties gauches des figures correspondront aux schémas conformes à IMS, tandis que les parties droites représenteront le schéma conceptuel équivalent.

12.3.1. L'attribut décomposable

Un attribut décomposable n'est pas directement représentable en IMS, mais on peut l'avoir détecté dans l'analyse des COPYBOOKS[3], des programmes et/ou des données. Dans un schéma optimisé orienté IMS, il peut être représenté sous la forme d'un type d'entité ou sous la forme d'une liste d'attributs.

A. Représentation d'un attribut décomposable par un type d'entité

Un attribut décomposable peut être représenté sous la forme d'un type d'entité. Il existe deux possibilités, une représentation par les instances ou une représentation par les valeurs.

A.1. Représentation par les instances

La Figure 12-2 montre comment transformer un type d'entité en un attribut décomposable. La cardinalité minimale de *possède.PERSONNE* est une contrainte qui doit être découverte par le rétro-ingénieur.

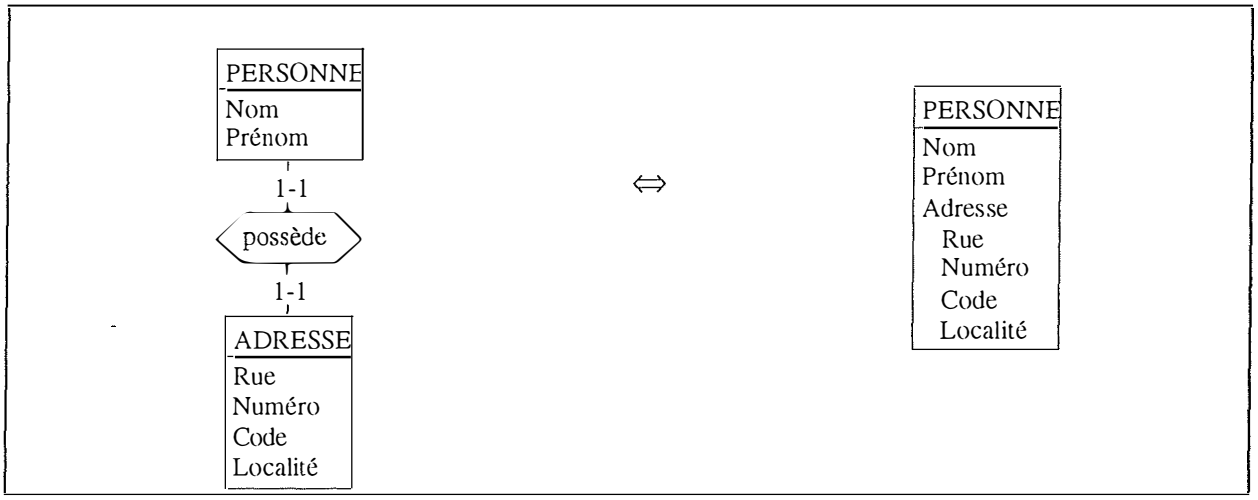


Figure 12-2: Transformation d'un type d'entité (instances) en un attribut décomposable

[3] Si on a découvert un attribut décomposable dans les COPYBOOKS, le schéma IMS source contient déjà cet attribut, même s'il n'est pas conforme à IMS.

A.2. Représentation par les valeurs

La Figure 12-3 représente la transformation à effectuer. La cardinalité minimale 1 de *possède.ADRESSE* a été découverte dans les programmes et/ou dans les données.

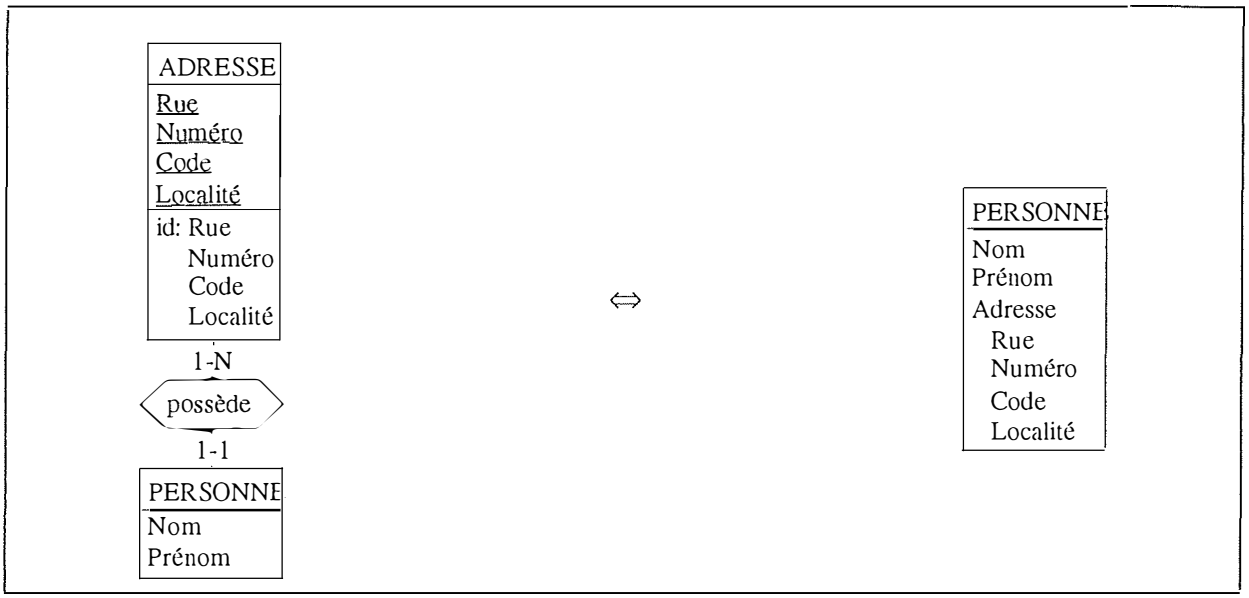


Figure 12-3: Transformation d'un type d'entité (valeurs) en un attribut décomposable

B. Représentation d'un attribut décomposable par une liste d'attributs

Un ensemble d'attributs qui possèdent un même préfixe peut également être le signe de la présence d'un attribut décomposable. Parfois, le bon sens et la connaissance du domaine suffisent pour détecter une telle situation; un préfixe commun n'est donc pas indispensable (voyez la figure du bas de la Figure 12-4).

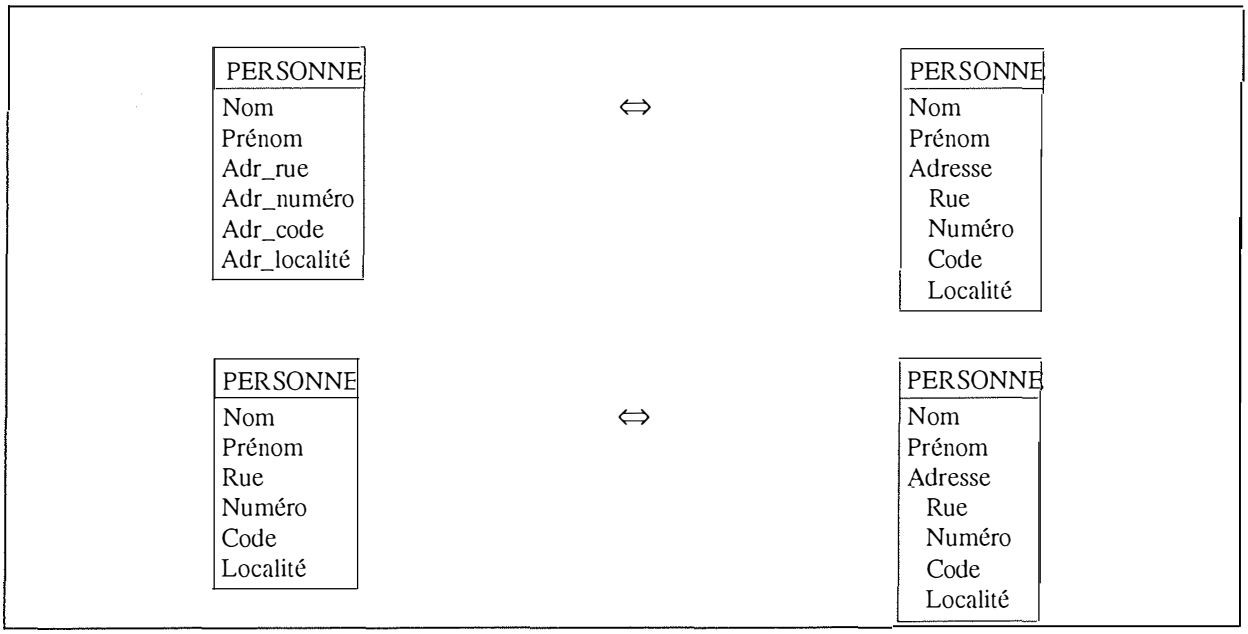


Figure 12-4: Transformation d'une liste d'attributs en un attribut décomposable

12.3.2. L'attribut multivalué

Il n'est pas non plus possible de représenter directement un attribut multivalué en IMS. On peut cependant le découvrir par l'analyse des programmes et des données.
Dans un schéma orienté IMS, il y a trois manières principales pour représenter un attribut multivalué. Il est possible de le représenter par un type d'entité, par une liste d'attributs similaires et par un attribut long.

A. Représentation d'un attribut multivalué par un type d'entité

La Figure 12-5 représente la transformation d'un type d'entité en un attribut multivalué. Dans l'exemple, la cardinalité supérieure de valeur 5 du rôle *possède.PERSONNE* doit être découverte par l'analyse des programmes et des données pour obtenir un schéma conceptuel correct.
Un type d'entité ne possédant qu'un seul attribut peut être également l'indice de la présence d'un attribut multivalué. Cependant, dans le cas d'un attribut multivalué et décomposable, le type d'entité possède plusieurs attributs.

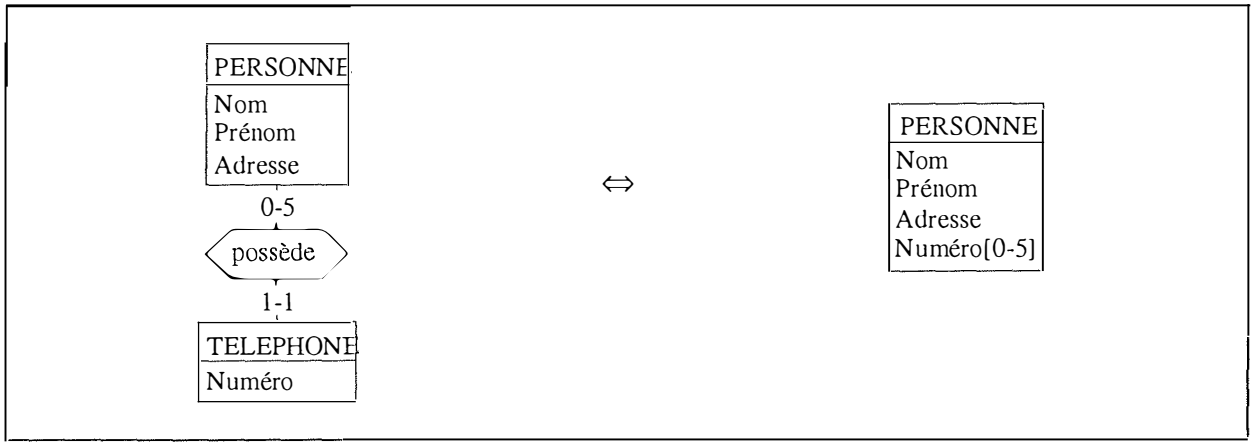


Figure 12-5: Transformation d'un type d'entité en un attribut multivalué

B. Représentation d'un attribut multivalué par une liste d'attributs similaires

Un ensemble de plusieurs attributs possédant le même préfixe et qui diffèrent par un suffixe sous forme de numéro peut être l'indice de la présence d'un attribut multivalué (partie gauche de la Figure 12-6). La cardinalité inférieure doit être découverte par l'analyse des données.

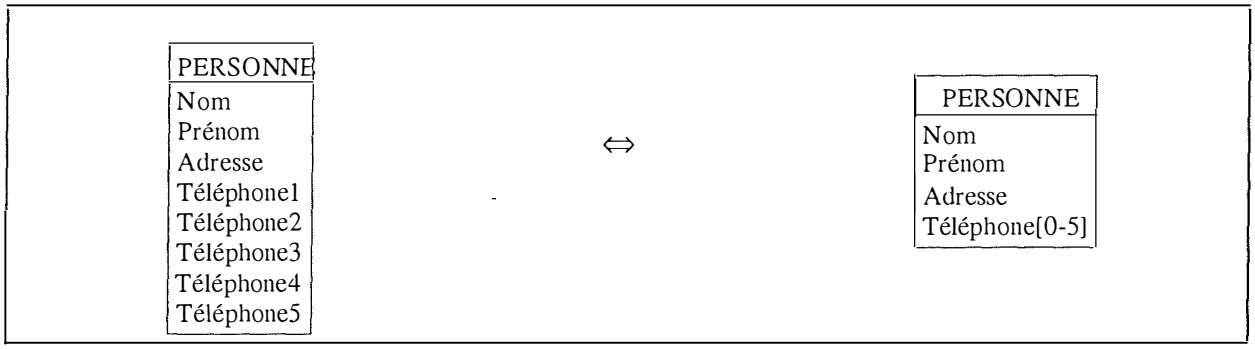


Figure 12-6: Transformation d'une liste d'attributs similaires en attribut multivalué

C. Représentation d'un attribut multivalué par un attribut long

Une autre possibilité de modélisation d'un attribut multivalué est de le représenter par un attribut long. Un attribut long se caractérise principalement par une longueur physique ou logique importante. Par exemple, un champ *Téléphone* de longueur 120 peut être une indication de la présence d'une telle construction.

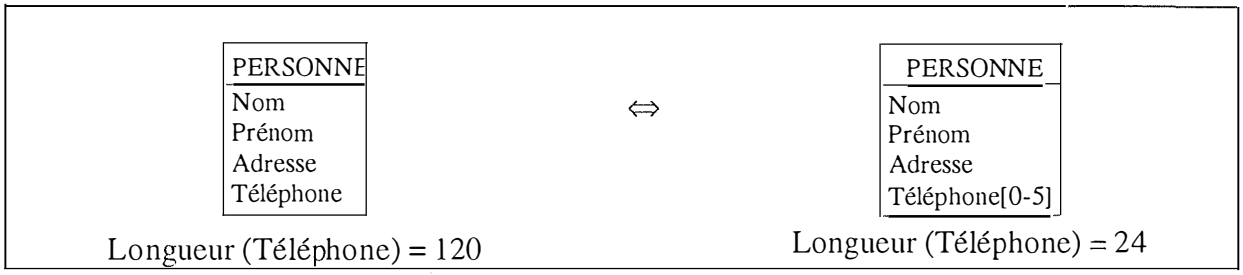


Figure 12-7: Transformation d'un attribut long en un attribut multivalué

12.3.3. L'attribut facultatif

L'attribut facultatif n'est pas non plus directement représentable en IMS. Il peut cependant être repéré dans l'analyse des programmes et l'analyse des données. Nous avons remarqué qu'il était possible de le modéliser sous la forme d'un type d'entité indépendant.
Un type d'entité transformable en attribut facultatif est repérable par le fait que la cardinalité du rôle joué par l'autre type d'entité est de [0-1]. Il faut noter également qu'il ne possède généralement qu'un seul attribut sauf dans le cas d'un attribut facultatif décomposable.

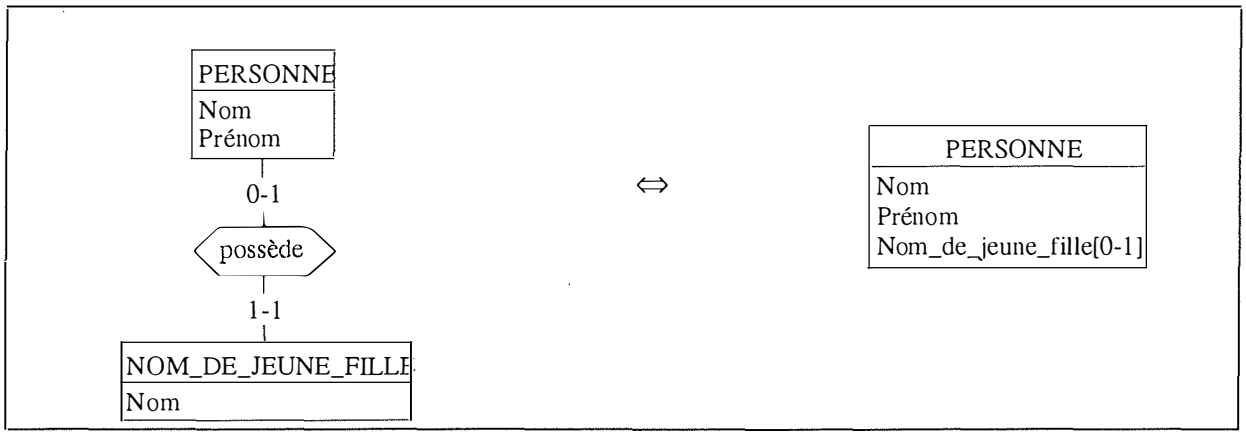


Figure 12-8: Transformation d'un type d'entité en attribut facultatif

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

Nous avons vu qu'il existait plusieurs possibilités pour modéliser un type d'association *many-to-many* dans un schéma conforme à IMS. Nous avons dégagé trois cas différents:

- une modélisation par un type d'entité
- une modélisation par une clé étrangère
- une modélisation par des clés étrangères

A. Représentation d'un type d'association *many-to-many* par un type d'entité

Cette modélisation d'un type d'association *many-to-many* par un type d'entité est possible grâce aux facilités offertes par les relations logiques. Ce type de construction peut donc être repéré par la présence d'une relation logique. La Figure 12-9 montre une telle transformation.

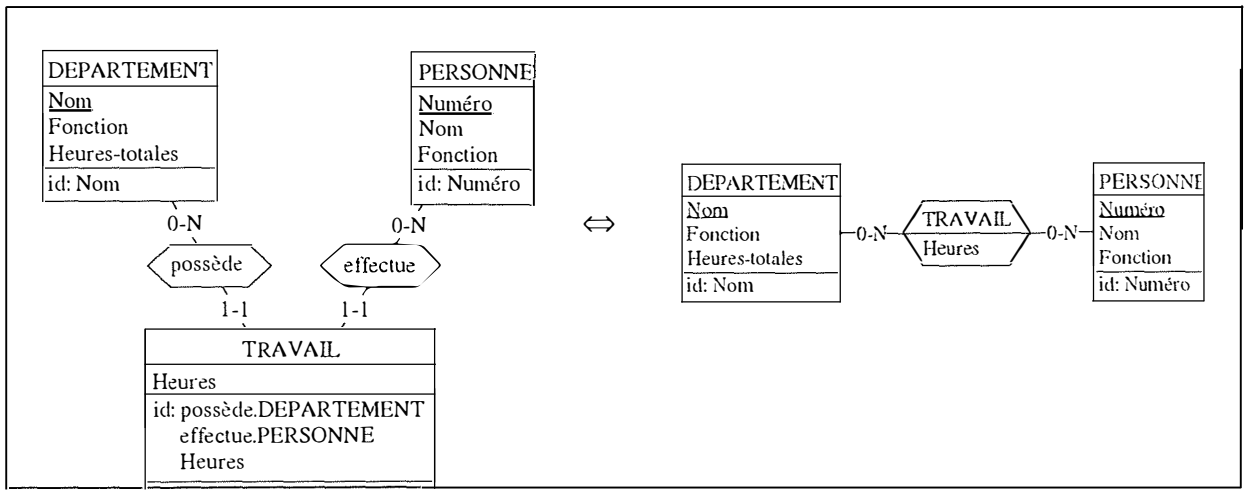


Figure 12-9: Transformation d'un type d'entité en relation *many-to-many*

B. Représentation d'un type d'association *many-to-many* par une clé étrangère

La présence d'une clé étrangère d'un attribut ayant une cardinalité [0-N] vers un autre attribut identifiant peut être l'indice de la présence d'une relation *many-to-many*. La clé étrangère du schéma de gauche de la Figure 12-10 est évidemment une contrainte implicite qui a été découverte lors d'une analyse.

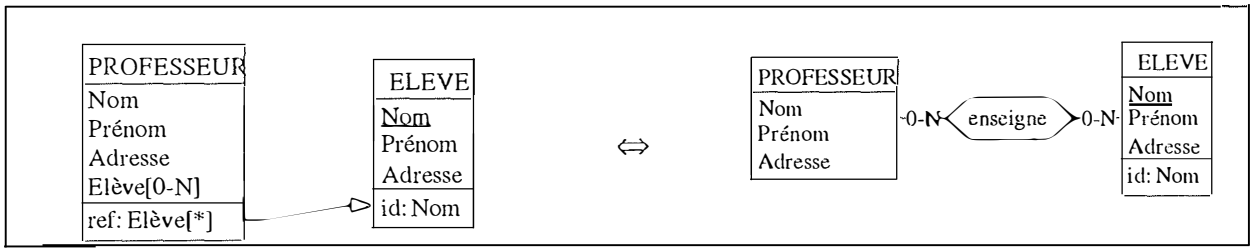


Figure 12-10: Transformation d'une clé étrangère en une relation *many-to-many*

C. Représentation d'un type d'association *many-to-many* par des clés étrangères

La présence de deux clés étrangères au sein d'un même type d'entité qui font référence à des attributs n'appartenant pas aux mêmes types d'entités peut être l'indice de la présence d'une relation *many-to-many*.

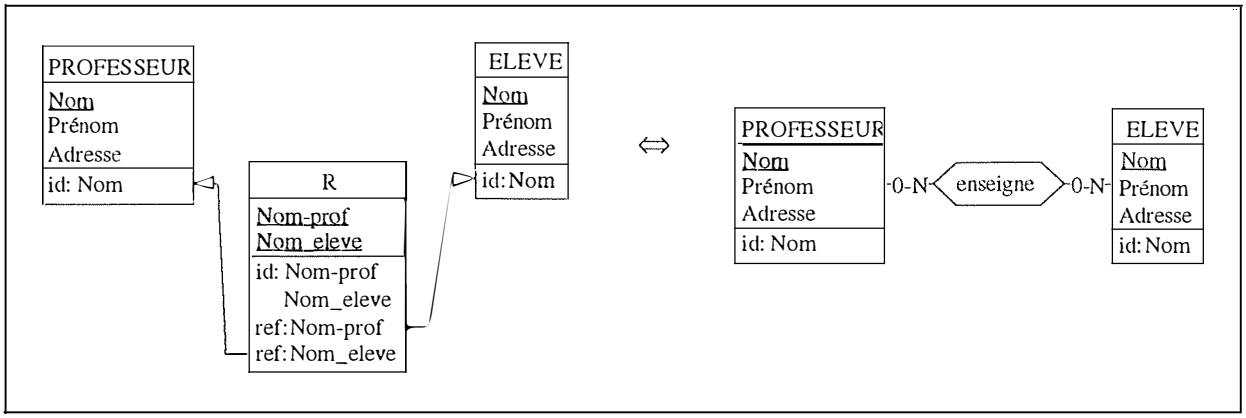


Figure 12-11 : Transformation des clés étrangères en une relation *many-to-many*

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

Nous avons vu au chapitre trois que le type d'association *one-to-one* pouvait être représenté par une clé étrangère.

Représentation d'un type d'association *one-to-one* par une clé étrangère

La Figure 12-12 montre la transformation de la clé étrangère en un type d'association *one-to-one*. La présence d'une clé étrangère peut donc être l'indice de la présence d'une relation *one-to-one*. Cette clé étrangère a évidemment été découverte dans les programmes et/ou les données.

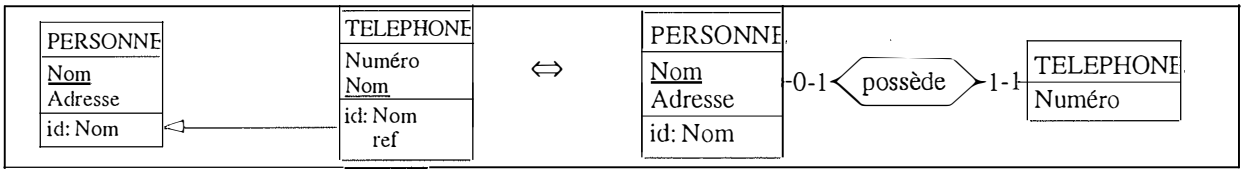


Figure 12-12: Transformation d'une clé étrangère en un type d'association *one-to-one*

12.3.6. Le type d'association récursif

De nombreuses modélisations permettent au concepteur de représenter un type d'association récursif en IMS. Ainsi, on peut représenter un type d'association par un type d'entité, ou par une clé étrangère.

A. Représentation d'un type d'association récursif par un type d'entité

On peut soupçonner l'existence d'un type d'association récursif si le schéma examiné est comparable au schéma de gauche de la Figure 12-13. Ce schéma implique la présence d'une relation logique.

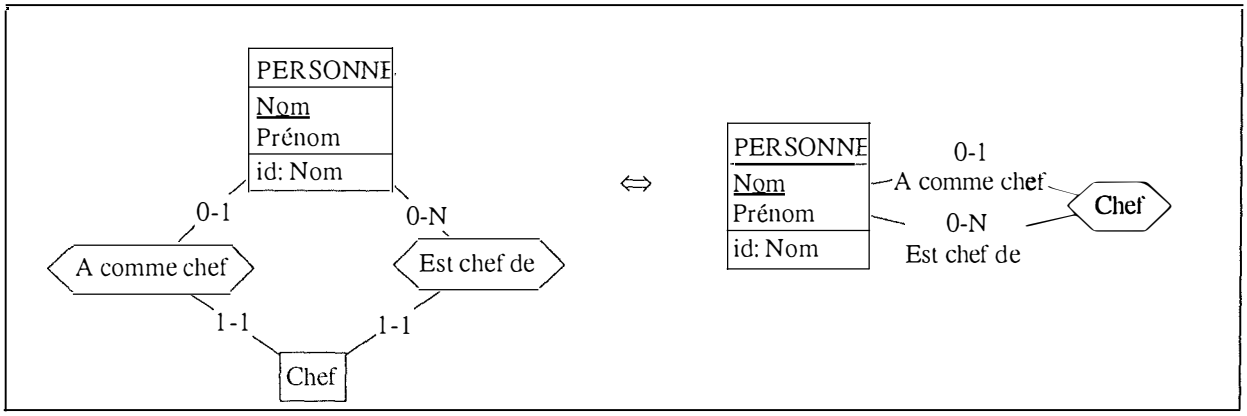


Figure 12-13: Transformation d'un type d'entité en un type d'association récursif

B. Représentation d'un type d'association récursif par une clé étrangère

Si nous sommes en présence d'un schéma comme celui représenté à gauche de la Figure 12-14, où il y a une clé étrangère au sein d'un même type d'entité, alors nous pouvons conceptualiser cette construction par un type d'association récursif. Il faut remarquer que le schéma de gauche contient un attribut facultatif, qui n'est pas une construction conforme à IMS. Cet attribut a donc été découvert auparavant, soit lors de la phase d'extraction ou encore dans la phase de détraduction.

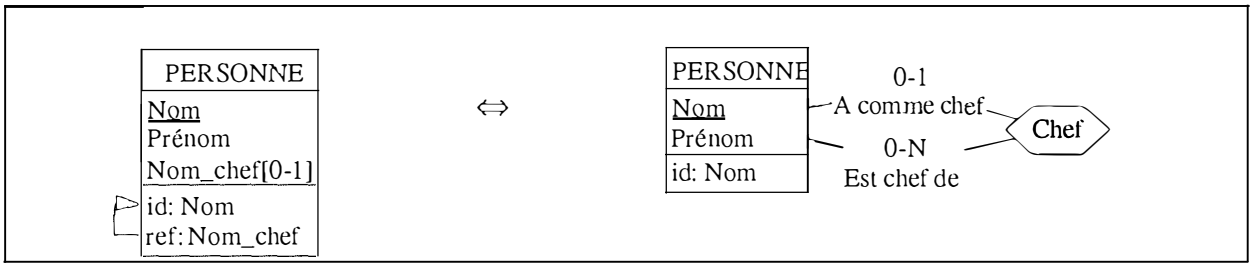


Figure 12-14: Transformation d'une clé étrangère en un type d'association récursif

12.3.7. Le type d'association n-aire

Les types d'association n-aires n'étant pas conformes à IMS, nous avons montré dans le chapitre trois comment il était possible de modéliser ces types d'associations. Il nous suffit maintenant d'inverser ces transformations, qui étaient symétriquement réversibles.

A. Représentation d'un type d'association n-aire par décomposition

A.1. Situation 1

La première modélisation possible d'un type d'association n-aire en IMS est présentée par la figure 12-15.

Il faut s'assurer que les cardinalités du coté des types de segments enfants sont bien des [0-1] et que la contrainte de coexistence entre les deux rôles est bien présente. Notons que le schéma de gauche n'est déjà plus conforme à IMS. En effet, les types de relation [0-N]-[0-1] sont impossibles en IMS. Ce schéma a donc déjà été détraduit partiellement.

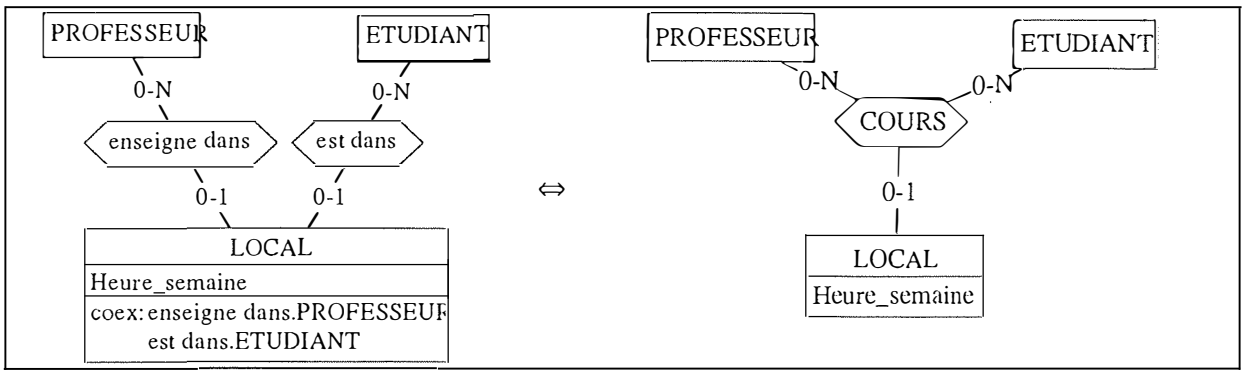


Figure 12-15: Transformation de deux types d'association *one-to-many* en un type d'association n-aire

A.2. Situation 2

Dans cette situation , nous sommes en présence de trois types d'associations *one-to-many*, comme le montre le schéma de gauche de la Figure 12-16. Ce schéma n'est bien sûr pas conforme à IMS, car on ne peut avoir trois types de relations entre deux types d'entités. Ce schéma a donc déjà subi des transformations.

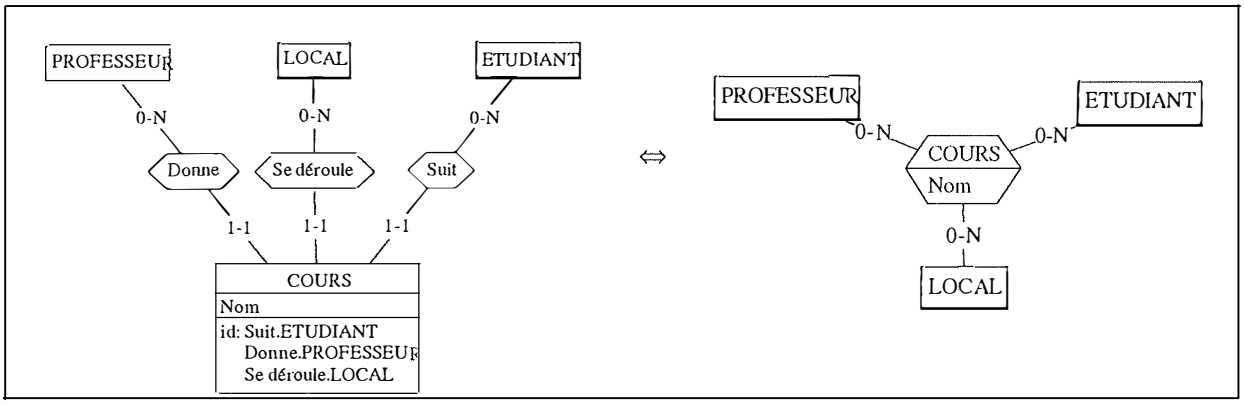


Figure 12-16: Transformation de trois types d'associations *one-to-many* en un type d'association n-aire

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

En IMS, un type de segment ne peut avoir plus de deux parents. C'est pourquoi il faut représenter de façon détournée ces situations en IMS. On peut modéliser cette situation par un type d'association ou par des clés étrangères.

A. Représentation d'un type d'entité avec plus de deux parents par un type d'association

Le concepteur de la BD peut avoir choisi de représenter un type de segment avec plus de deux parents par le schéma de gauche de la Figure 12-17. Cela lui permet d'implémenter une situation qui n'est pas directement représentable en IMS. Il faut remarquer que ce schéma n'est pas conforme à IMS, étant donné la présence d'un type d'association *one-to-one*. On a donc découvert dans les programmes ou dans les données la cardinalité [1-1] de R.cours_1. Une telle situation peut être repérée facilement par la présence du type d'association *one-to-one*, qui permet de réunir les deux types d'entités qu'il relie.

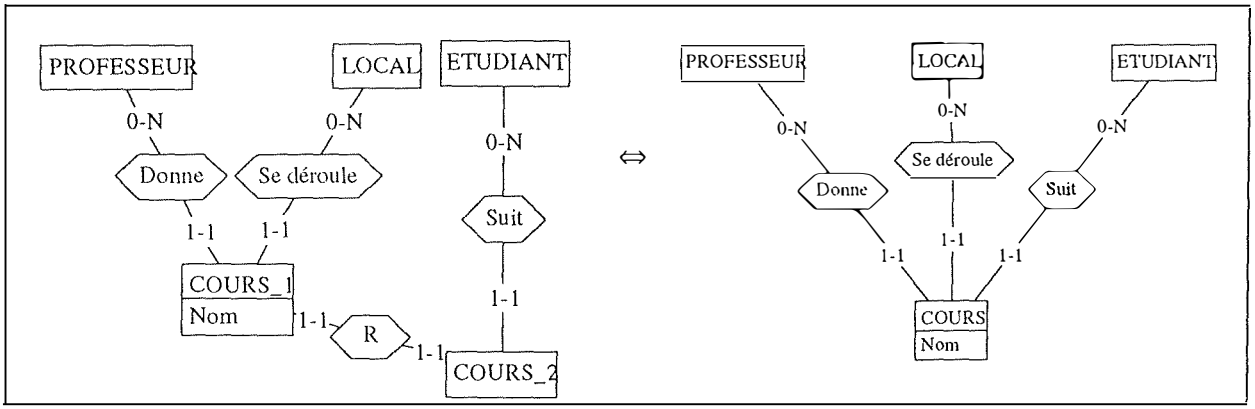


Figure 12-17: Réunion de deux types d'entités en un seul

B. Représentation d'un type d'entité avec plus de deux parents par des clés étrangères

On peut également représenter un type d'entité ayant plus de deux parents par des clés étrangères. La Figure 12-18 montre un tel cas.

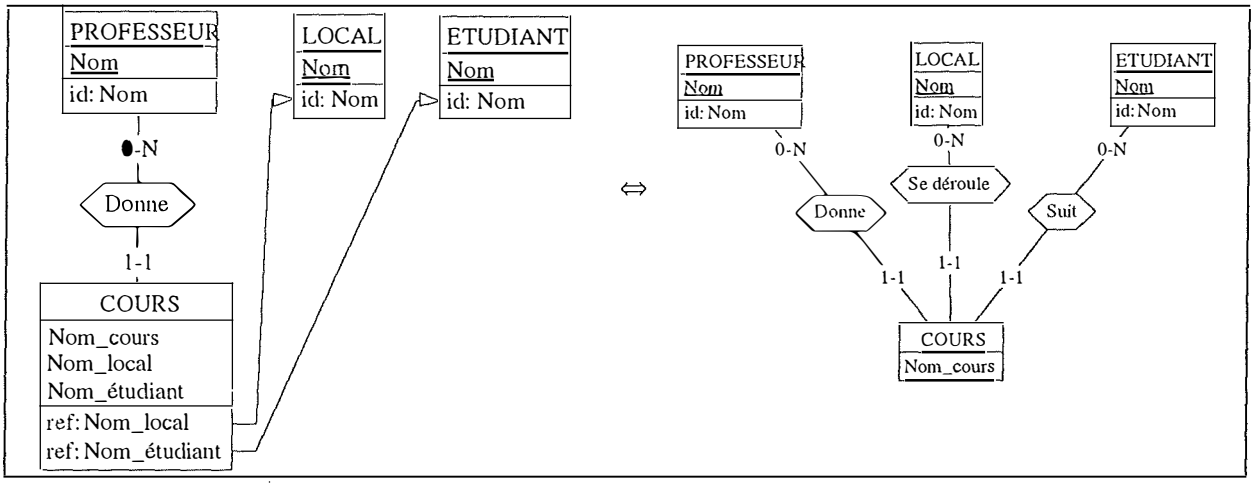


Figure 12-18: Transformation des clés étrangères en des types d'associations *one-to-many*

12.4. La désoptimisation du schéma

Lors de la conception de la base de données, le schéma conceptuel a été progressivement restructuré et enrichi pour des raisons d'optimisation. Il faut détecter et éliminer ces structures. Le SGBD IMS amène souvent le concepteur de la base de données à faire des choix d'optimisation. Nous allons essayer de repérer les structures qui découlent de ces choix. Pour ce faire, nous analyserons les différentes familles d'optimisation à savoir la dénormalisation, la redondance structurelle et la restructuration.

12.4.1. La dénormalisation

Selon **[RICHARD, 95]**, 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. Il est possible de repérer cette structure. La présence d'un déterminant d'une dépendance fonctionnelle qui n'est pas identifiant du type d'entité peut être l'indice d'une telle structure.
Si l'existence d'une telle construction est établie alors, il convient d'éclater le type d'entité en deux types d'entité reliés par un type d'association *one-to-one*, comme le montre la Figure 12-19.

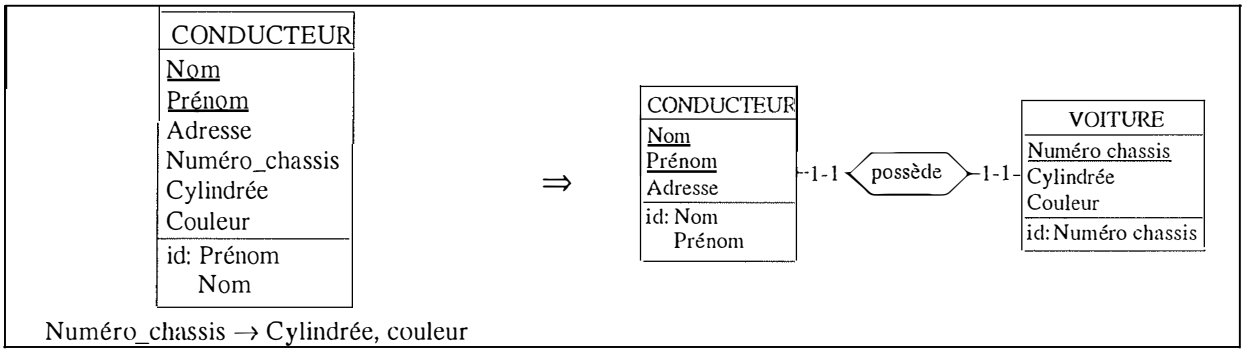


Figure 12-19: Eclatement d'un type d'entité

Une autre technique de dénormalisation consiste à remplacer un type d'association *one-to-many* entre deux types d'entités par une clé étrangère. Si une telle structure est repérée, il convient de remplacer cette clé étrangère par un type d'association *one-to-many*, comme le montre la Figure 12-20.

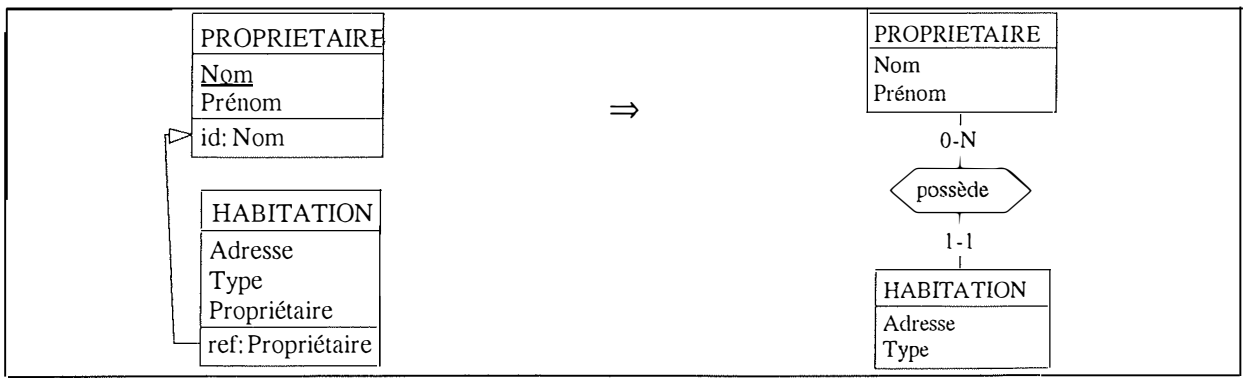


Figure 12-20: Transformation d'une clé étrangère en un type d'association *one-to-many*

12.4.2. La redondance structurelle

Selon [**RICHARD, 95**], la redondance structurelle consiste à ajouter des éléments au schéma qui sont calculables à partir d'autres éléments.
La redondance structurelle la plus fréquente est la duplication d'un même champ dans plusieurs types de segments. Cette redondance d'attributs est repérable par l'analyse des programmes et des données. Cette redondance est pratique et fréquente en IMS car elle permet d'éviter le parcours de toute une hiérarchie pour retrouver une information. Cette construction implique cependant un lourd travail de mise à jour de tous les champs redéfinis. En effet, la modification, l'ajout ou le retrait d'une valeur pour ce champ entraîne autant d'opérations sur toutes les redéfinitions de ce champ. Une telle construction doit être modifiée. Le champ dupliqué est enlevé. Il faut cependant garder trace de ces duplications. [**RICHARD, 95**] propose de mettre cette information dans la description sémantique du type d'entité où l'attribut correspondant au champ a été supprimé.
Une seconde redondance structurelle possible est la redondance des types de segments. Pour des raisons identiques à celles des champs, il est parfois nécessaire de dupliquer un type de segment dans plusieurs hiérarchies. Une telle structure nécessite également un lourd travail de mise à jour au sein des programmes. L'existence d'une telle structure peut donc être repérée dans l'analyse des programmes. Cette structure est transformée en enlevant les types de segments dupliqués. Cette suppression doit faire l'objet d'une grande attention. Si, au niveau du schéma, les types d'entités supprimés jouaient des rôles avec d'autres types d'entités, alors ces derniers doivent être reliés au type d'entité restant.

12.4.3. La restructuration

[**RICHARD, 95**] repère quatre types d'*optimisations de restructuration*: le partionnement horizontal, le partionnement vertical, le rassemblement horizontal et le rassemblement vertical.

A. Le partitionnement horizontal

Le partitionnement horizontal consiste à diviser un type d'entité en plusieurs types d'entités identiques de telle manière que la population du type d'entité initial soit répartie dans les types d'entités ainsi créés.

Cette situation peut être repérée par une redéfinition dans l'analyse des données mais pas dans l'analyse des programmes, car alors elle passerait du côté des redondances structurelles. Il n'y a pas de mise à jour directe à effectuer dans les programmes. Une occurrence d'un type d'entité n'est présente que dans ce type d'entité, ce qui est l'opposé de ce qui se passe dans le cas de la redondance structurelle.

Si une telle structure est découverte, il convient de rassembler ces différents types d'entités au sein d'un même type d'entité (comme le montre la figure 12-21). Les rôles que jouaient ces types d'entités sont rattachés au nouveau type d'entité.

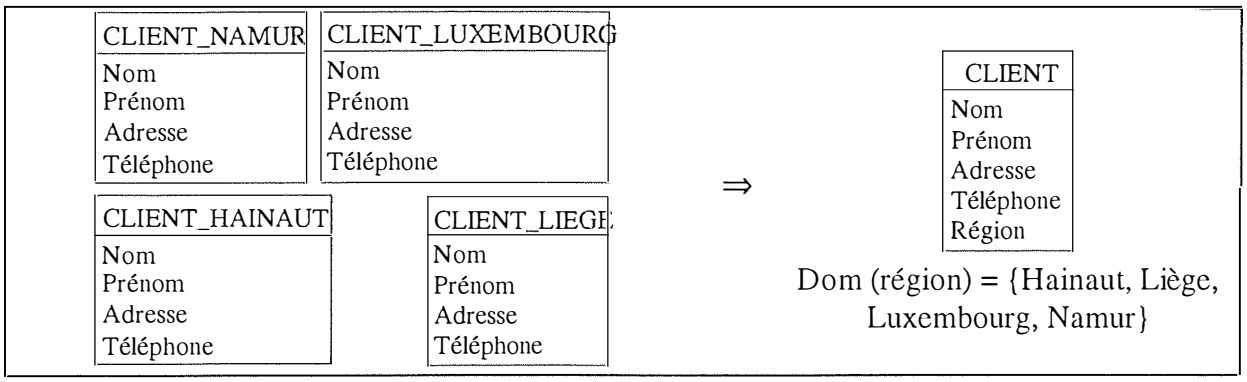


Figure 12-21: Le partitionnement horizontal

B. Le partitionnement vertical

Le partitionnement vertical est courant en IMS à cause de la structure hiérarchique. En effet, le stockage des types de segments est tel que le concepteur doit apprécier correctement la taille des différents types de segments. Les types de segments de taille trop importante augmentent les temps d'accès. En effet, si nous sommes en présence d'un type de segment racine de grande taille, il faudra lire de nombreux blocs avant de pouvoir accéder aux types de segments enfants. C'est principalement pour cette raison que le partitionnement vertical est utilisé en IMS. Les types de segments enfants créés sont 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 de segments dépendants qui sont en fait des attributs du type d'entité parent.

Si cette situation est découverte, les types de segments enfants sont transformés en attributs du type de segment parent comme à la figure 12-22.

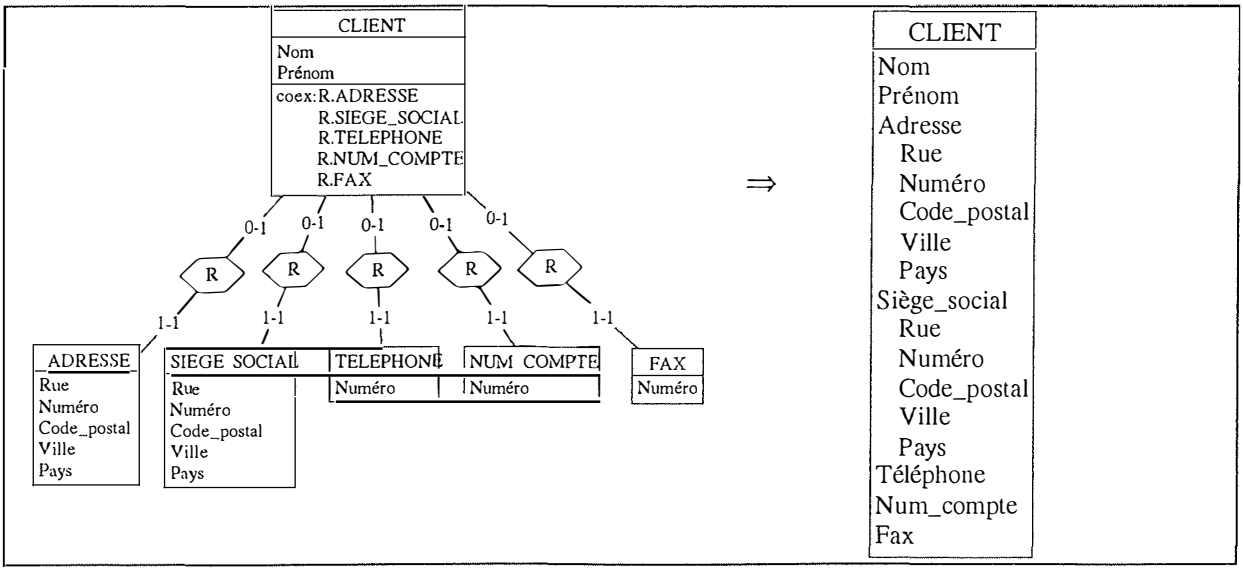


Figure 12-22: Le partitionnement vertical

C. Le rassemblement horizontal

Le rassemblement horizontal est l'inverse du partitionnement horizontal c'est-à-dire que deux types d'entités différents sont rassemblés en un seul.

Si cette situation est repérée, le type d'entité qui est le résultat du rassemblement horizontal est éclaté en plusieurs types d'entités distincts.

D. Le rassemblement vertical

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.

Remarque: Il convient de faire remarquer que le concepteur de la base de données est totalement libre d'employer les transformations qu'il désire et cela dans n'importe quel ordre. De plus, elles mettent toutes en présence les mêmes concepts, à savoir les clés étrangères, les types d'associations provenant de types de relations logiques et les hiérarchies. Le travail du rétro-ingénieur est dès lors très ardu car il doit essayer de repérer les différentes structures qui ont été implémentées et surtout il ne doit pas les confondre. L'aide d'une personne ayant participée à la conception de la BD est dès lors un atout précieux.

12.5. La normalisation conceptuelle

Cette étape est un processus de normalisation conceptuelle normale. Selon **[HAINAUT, 95a]** les objectifs principaux de cette phase sont:

- donner au schéma des qualités telles que la simplicité, la minimalité, la force d'expression, la lisibilité,
- retrouver des constructions sémantiques avancées telles que les types d'associations n-aires et IS-A,
- rendre le schéma obtenu conforme à certains standards désirés.

12.6. Conclusion

Comme nous avons pu le voir, la conceptualisation des structures de données n'est pas chose aisée. Il est en effet très difficile de repérer dans le schéma la structure que le concepteur a voulu représenter initialement. Le SGBD IMS ne permet pas de représenter des constructions telles que les clés étrangères, les attributs multivalués,... Toutes ces structures sont remplacées par des transformations conformes à IMS. Cela implique que l'on retrouve de nombreuses constructions qui peuvent représenter plusieurs structures différentes. Le rétro-ingénieur doit donc faire preuve de beaucoup de prudence lors de ce processus.

PARTIE 4: COMPLEMENTS DE MISE EN OEUVRE

CHAPITRE 13: Les outils

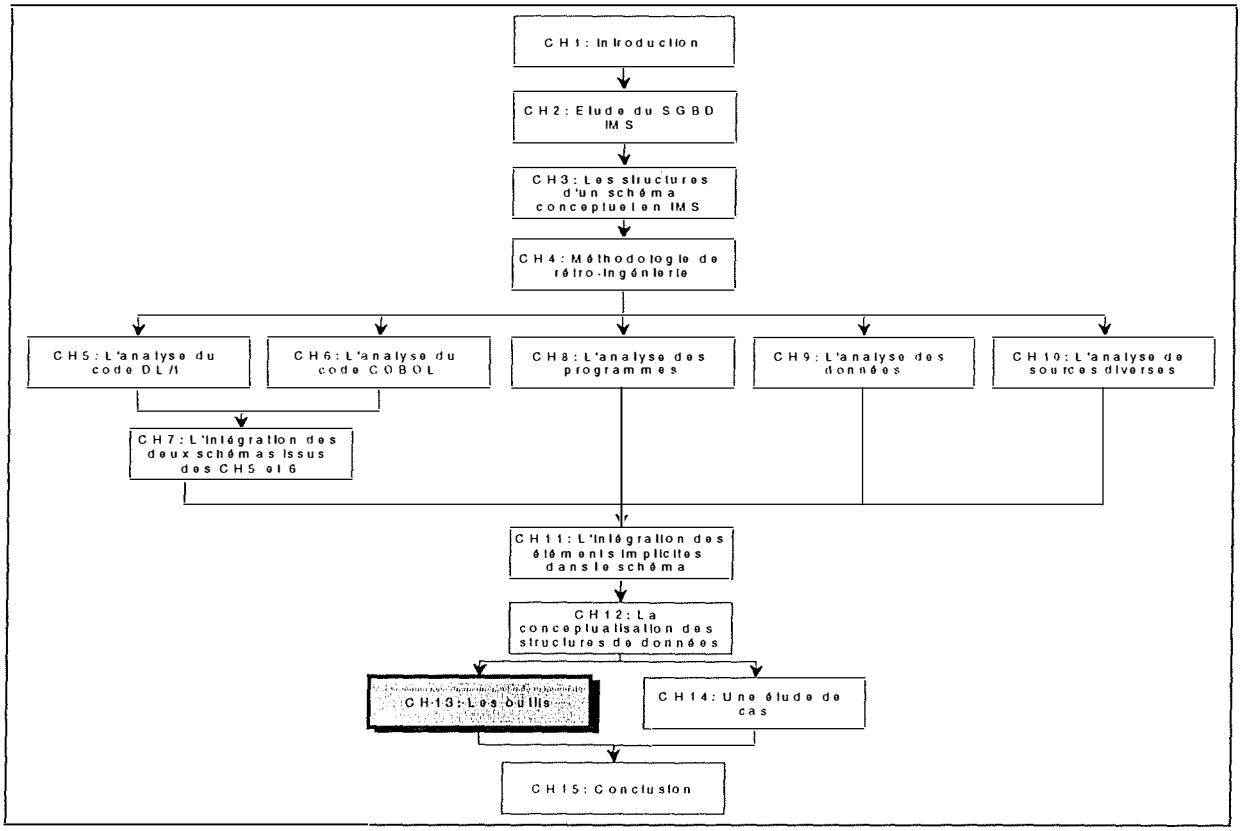


Figure 13-1: Le chapitre 13

13.1. Introduction: les besoins en outils

Dans ce chapitre treize, nous allons spécifier quels outils seraient intéressants pour faciliter le travail du rétro-ingénieur qui suit notre méthodologie. Avant de décrire les outils développés dans le paragraphe 13.2, nous allons, dans cette introduction, préciser pour chaque phase décrite aux chapitres cinq à douze quels seraient les développements intéressants. Il est important de préciser que les outils proposés dans cette introduction ne sont pas tous développés pour l'instant; on ne se limite pas à décrire ce qui existe déjà, mais on spécifie quels genres d'outils seraient utiles pour chaque étape de la méthodologie, indépendamment de leurs implémentations.

A. L'analyse du code DL/1

Pour rappel, cette étape consiste à créer un schéma entité-association à partir d'un code DL/1, qui est la description de la base de données IMS. Il s'agit donc d'un travail assez répétitif, qui pourrait être automatisé sans trop de problèmes. Un extracteur automatique pourrait faire gagner un temps non négligeable au rétro-ingénieur. Cet extracteur prendrait en entrée un code DL/1 sans erreurs et donnerait en sortie un schéma physique entité-association.
Cet extracteur a été écrit en C++ par Ph.Richard il y a deux ans, et est intégré dans l'atelier DB-MAIN. Il est capable d'analyser un code DDL correct.
Nous avons cependant complété celui-ci d'un programme en VOYAGER 2. En effet, nous avons remarqué que l'extracteur était incapable de détecter la présence d'attributs décomposables ou recouvrants (grâce aux mots-clés BYTES et START de l'instruction FIELD). Nous avons donc écrit un petit programme en VOYAGER 2 (dcompims), qui modifie le schéma extrait, et qui détecte les attributs décomposables et recouvrants.

B. L'analyse du code COBOL (COPYBOOKS)

Cette analyse consiste à créer un schéma entité-association à partir d'un certain nombre de fichiers de déclarations COBOL (qui, pour rappel, décrivent la structure d'un type de segment donné), qui sont appelés COPYBOOKS chez d'Ieteren. Comme dans l'étape précédente, il serait utile de disposer d'un extracteur automatique de l'information disponible dans ces déclarations, qui prendrait donc en entrée un code COBOL (déclarations de variables), et donnerait en sortie le schéma entité-association correspondant. On ferait gagner du temps au rétro-ingénieur, en annulant aussi le risque d'erreurs d'une extraction manuelle.
Nous avons développé un programme en VOYAGER 2 (extr_cop) qui analyse un COPYBOOK et crée le schéma DB-MAIN correspondant à ce COPYBOOK. On le décrira dans le paragraphe 13.2.3.

C. L'intégration des schémas issus des deux premières étapes

Nous avons vu que l'extraction du code DL/1 et des COPYBOOKS correspondant donnait deux schémas différents, mais qui concernent néanmoins la même situation[1]. Il nous faut donc regrouper l'information présente dans ces deux schémas dans un troisième, sans qu'il y ait la moindre perte d'information.

[1] En effet, nous demandons au rétro-ingénieur d'extraire au sein d'un même schéma tous les COPYBOOKS qui correspondent aux types de segments de la base de données analysée. Nous avons donc dans un premier schéma les types d'entités issus de l'analyse du DL/1, et dans l'autre les types d'entités issus de l'analyse des COPYBOOKS et qui correspondent aux types d'entités du schéma DL/1.

Plusieurs outils seraient intéressants pour cette étape d'intégration:

1. Il faudrait un outil capable de détecter la redéfinition d'un même type de segment dans plusieurs COPYBOOKS[2], c'est-à-dire un outil capable, au sein du schéma issu des COPYBOOKS, de détecter des types d'entités ayant la même longueur.
2. Il faudrait un analyseur capable de détecter s'il y a des contradictions entre le schéma DL/1 et celui des COPYBOOKS. Cet analyseur ne modifierait pas les schémas et produirait un rapport que le rétro-ingénieur pourrait consulter. Cet outil nous semble le plus utile, car aucune contradiction ne sera manquée grâce à lui.
3. Enfin, pour les cas simples, c'est-à-dire des situations sans contradictions, un intégrateur automatique serait intéressant (gain de temps, pas de risque d'erreur). Il prendrait en entrée les deux schémas et donnerait en sortie le schéma intégré.

En fait, ces trois outils ont été développés dans le cadre de ce mémoire. Tous les trois l'ont été en VOYAGER 2. Le premier se nomme copy; Le second analint et le troisième integims. Il faut remarquer qu'integims est un intégrateur « aveugle », c'est-à-dire qu'il est incapable de détecter des contradictions. On pourrait imaginer créer un intégrateur intelligent qui arrête son travail s'il détecte une contradiction. Ces trois outils seront décrits plus loin.

D. L'analyse des programmes

Cette étape est l'une des plus difficiles à automatiser complètement. En effet, lancer un programme sur un code COBOL pour y repérer des structures implicites comme les clés étrangères est illusoire. Tout ce qu'on peut faire, c'est lancer le rétro-ingénieur sur des pistes de recherche. Les outils utiles dans cette étape feraient avant tout gagner du temps, en facilitant le travail de recherche du rétro-ingénieur, en le lançant sur des pistes de recherche.
Comme nous l'avons vu dans le chapitre huit, il y trois grandes méthodes d'analyse des programmes:

1. L'analyse du flux de données
2. La fragmentation des programmes
3. La recherche de patterns d'analyse

Ces trois méthodes sont disponibles dans l'atelier DB-MAIN. Nous n'allons pas expliquer ici en quoi consistent ces méthodes, nous l'avons déjà fait au chapitre huit.
L'outil d'analyse du flux de données ou graphe de dépendance est un des outils d'analyse des programmes les plus au point dans DB-MAIN.
L'outil de fragmentation des programmes est aussi un outil puissant. Cependant, pour l'instant, il n'est dédié qu'à COBOL. Quelques adaptations ont dû être effectuées par J. HENRARD pour prendre en compte l'instruction CALL d'appel à la base de données.

Enfin, le moteur de patterns d'analyse est sûrement celui qui demande encore quelques développements supplémentaires. Il est encore limité par la complexité des patterns à rechercher. Si cette méthode nous semble puissante, elle est aussi très difficile à utiliser, et cela pour deux raisons:

- Cette technique est très difficile à automatiser, car il faut des outils d'analyse très puissants, capables de lier des parties de programmes parfois dispersées.
- Les patterns d'analyse ne sont pas des structures figées. Cela signifie que chaque programmeur ne codera pas de la même façon une même contrainte. Les patterns proposés ne sont donc pas figés, de nombreuses variantes sont susceptibles d'exister.

Pour l'instant, cet outil n'est donc pas assez puissant et pourrait être développé pour prendre en compte des patterns plus compliqués.

---

[2] Autrement dit, le cas où un type de segment possède plusieurs COPYBOOKS

E. L'analyse des données

Dans le chapitre neuf, nous avons exposé un processus de génération automatique de programmes de vérification des données. Le programme GENE effectue cette génération de programmes pour des programmes vérifiant l'existence de clés étrangères.

F. L'analyse des noms

Pour cette étape d'analyse des noms, il faudrait des outils qui facilitent le travail. Par exemple, un outil de tri de tous les noms d'attributs d'un schéma.
Nous citerons dans cette analyse des noms un outil très pratique de l'atelier DB-MAIN, qui utilise notamment une analyse des noms: l'outil REFERENTIAL KEY. Cet outil permet de rechercher l'existence d'une clé étrangère dans un schéma.
Un autre outil présent dans DB-MAIN est la possibilité de trier sur les noms tous les attributs d'un schéma (SORTED VIEW).

Dans cette introduction, nous avons montré que notre travail avait permis de construire des outils utiles pour le rétro-ingénieur, mais il reste du travail, notamment en ce qui concerne le développement d'outils performants pour l'analyse des programmes en langage hôte.
Nous allons maintenant décrire les outils qui existent déjà, et qui sont utiles dans notre méthodologie.

13.2. La description des outils développés

Nous exposerons dans ce point les divers outils logiciels qui permettent de faciliter le travail du rétro-ingénieur. Ces outils seront à la fois des outils déjà intégrés à DB-MAIN, ou des outils développés dans le cadre de ce mémoire.
Il faut distinguer deux grandes catégories d'outils. D'une part, les outils qui se substituent au rétro-ingénieur en produisant un résultat « fini », qui ne doit plus être analysé par celui-ci; d'autre part les outils d'aide à l'analyse, qui ne se substituent aucunement au raisonnement humain, mais permettent de l'aiguiller utilement.

Ce paragraphe a notamment pour objet d'expliquer en quoi consistent les différents outils utilisés lors d'un processus de rétro-ingénierie. Nous y décrirons la manière de les utiliser et les résultats qu'ils produisent.

Les différents outils présentés sont:
1. L'extracteur IMS
2. Le programme DCOMPIMS
3. Le programme EXTR_COP
4. Le programme COPY
5. Les programmes d'intégration de schémas
    5.1. Le programme d'analyse ANALINT
    5.2. L'intégrateur INTEGIMS
6. Le programme GENE
7. Le program slicing
8. Le graphe de dépendance
9. Le moteur de pattern-matching
10. L'outil REFERENTIAL-KEY
11. La « Transformation Toolkit »

13.2.1. L'extracteur IMS

Cet extracteur a été réalisé par Philippe Richard dans le cadre de son mémoire "La rétro-ingénierie des bases de données, application à IMS", Institut d'informatique 1995. Le but de ce programme est d'extraire d'un fichier de définition d'une base de données IMS (DL/1) une structure graphique (types de segments, hiérarchies) représentant la situation. Nous n'allons pas expliquer ici le fonctionnement précis de cette application. Le lecteur intéressé pourra consulter **[RICHARD, 95]**. En fait, la méthode d'analyse du DL/1 vue au chapitre cinq a servi de spécification à ce programme d'extraction.

La phase d'analyse du code DL/1 peut donc se faire automatiquement grâce à ce programme. Nous reprenons dans le Tableau 13-1 les éléments principaux apportés au schéma par l'extracteur, ainsi que la provenance de ceux-ci (instructions DL/1 correspondantes).

Eléments du schéma	Instruction correspondante
Schéma	DBD
Collection	DATASET et AREA
Type d'entité	SEGM
Type d'association « physique »	PARENT dans SEGM
Cardinalité supérieure égale à 1 côté parent	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
Type d'association « logique »	LCHILD et SEGM
Clé d'accès pour un type d'entité	SRCH et XDFLD
Sous-schéma externe	PCB, SENSEG et SENFLD

Tableau 13-1: Eléments du schéma produit et sources correspondantes

13.2.2. Le programme DCOMPIMS

Lors de nos premiers tests de l'extracteur IMS de Ph. Richard, nous avons constaté que ce programme avait une faiblesse: il était incapable de détecter des situations de champs décomposables ou chevauchants. Nous avons donc développé le programme **dcompims** en VOYAGER 2, qui a pour but de détecter ces situations, à partir du schéma créé par l'extracteur IMS de l'atelier.

Ce programme comprend trois phases distinctes:
1. La création des méta-propriétés PSTART (position physique de départ) et PLENGTH (longueur physique) pour les attributs et les types d'entités du schéma,
2. La détection des situations d'*overlapping* (attributs chevauchants),
3. La détection des attributs décomposables.

Nous allons maintenant préciser quelles sont les pré et post-conditions de ce programme.

Pré-condition:
Le programme doit être lancé sur le schéma physique issu d'un code DL/1 et créé par l'extracteur. Si aucun schéma n'est sélectionné, le programme est stoppé et le message **<None schema!>** apparaît à la console VOYAGER 2.

Post-condition:
Le schéma sélectionné est modifié:
- La méta-propriété[3] PLENGTH (longueur physique) est créée pour chaque type d'entité du schéma.
- Les méta-propriétés PSTART (position de départ physique) et PLENGTH (longueur physique) sont créées pour chaque attribut de chaque type d'entité du schéma (voir la fenêtre 2 de l'annexe C).
- Une fois les méta-propriétés créées, le programme peut les utiliser pour détecter des situations d'*overlapping*. L'overlapping est le cas du recouvrement d'un ou plusieurs attributs

[3] Ces deux méta-propriétés ont pour but de compenser une légère faiblesse de DB-MAIN, qui ne fait pas de distinction « directe » entre longueurs logique et physique. En effet, le champ 'length' des boîtes de dialogue 'attributes properties' et 'entity type properties' représentent des longueurs logiques (voir la fenêtre 1 de l'annexe C). Pour la suite du processus de rétro-ingénierie, nous créons donc automatiquement ces deux propriétés, qui seront très utiles pour nos applications.

par un autre. Nous avons analysé toutes les situations possibles au chapitre sept. La Figure 13-2 montre une de ces situations.

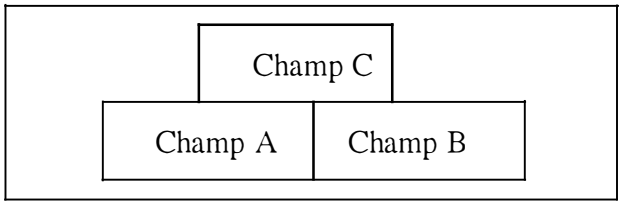


Figure 13-2: Recouvrement d'attributs

Chaque fois qu'un tel cas sera découvert, une contrainte de coexistence sera créée pour les champs A et B. Les cardinalités minimales des ces deux champs seront mises à 0, car les composants d'une contrainte de coexistence doivent être facultatifs. On créera ensuite une contrainte d'exactement-un entre le groupe {A,B} et le champ C, c'est-à-dire que dans une même occurrence du type d'entité contenant les champs A, B et C, on peut seulement trouver, soit le champ C, soit les champs A et B, ou aucun des champs. Dans une même occurrence, on ne peut donc retrouver que C $ou_{exlusif}$ (A,B). La Figure 13-3 montre le résultat donné par le programme s'il découvre la situation de la Figure 13-2.

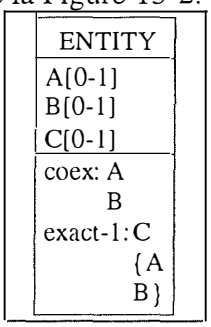


Figure 13-3: Traitement par le programme de la situation décrite à la Figure 13-2

- Après avoir traité les situations de chevauchement, le programme **dcompims** commence à analyser le schéma pour détecter des attributs décomposables. Cette analyse se fait grâce aux méta-propriétés PSTART et PLENGTH. Chaque fois qu'il détecte un attribut décomposable, il modifie le schéma en conséquence, en créant l'attribut décomposable et ses composants.

13.2.3. Le programme EXTR_COP

Dans la majorité des déclarations de bases de données IMS (DL/1), la structure interne des types de segments n'est pas complète. Cette structure sera décrite complètement dans un COPYBOOK. Un COPYBOOK est un fichier contenant les déclarations COBOL des variables qui représentent la structure interne du type de segment donné. Le programme **extr cop**, écrit en VOYAGER 2, a pour but de représenter les déclarations des variables COBOL sous forme de types d'entités.
La Figure 13-4 montre un exemple de COPYBOOK.

```
*                                          NOM ETUDIANT
  05 NOM      PIC X(30).
*                                          PRENOM ETUDIANT
  05 PRENOM   PIC X(30).
*                                          TELEPHONE ETUDIANT
  05 TEL      PIC X(20).
*                                          DATE DE NAISSANCE
  05 NAIS     PIC X(10).
```

Figure 13-4: Contenu d'un copybook à extraire

L'extraction par le programme du code de la Figure 13-4 donne le résultat suivant.

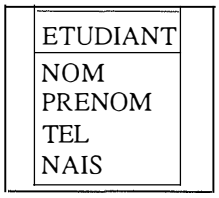


Figure 13-5: Résultat de l'extraction

La méthode d'analyse des COPYBOOKS présentée au chapitre six a servi de spécification à ce programme d'extraction des COPYBOOKS. Voyons quelles sont les pré et post-conditions de ce programme.

Pré-condition:
Comme on peut le voir dans le code source de la Figure 13-4, le nom du type de segment auquel le COPYBOOK correspond n'est pas présent. En effet, le lien entre un COPYBOOK et son type de segment correspondant est fait via l'instruction CALL à la base de données, dans les programmes d'application. Donc, le seul moyen pour nous de savoir à quel type de segment correspond le COPYBOOK analysé (pour donner au type d'entité que l'on va générer le même nom qu'à celui du schéma issu du DL/1), est de le demander à l'utilisateur. Pour cela, nous lui demandons de créer un fichier *lien* qui fera cette correspondance et qui contiendra aussi le chemin d'accès des différents COPYBOOKS à analyser.

En résumé, l'utilisateur doit créer un fichier LIEN qui:
- fait le lien entre le type de segment et le COPYBOOK dans lequel il est décrit.
- contient le chemin d'accès des différents COPYBOOKS à analyser.

Le format du fichier *lien* est le suivant:
<Nom de la base de données
Nom du type de segment/ nom du COPYBOOK
Nom du type de segment/ nom du COPYBOOK
...>
la Figure 13-6 donne un exemple de fichier *lien*.

```
student_db
ETUDIANT / c:\ims\etudiant
COURS / c:\ims\cours
ADRESSE / c:\ims\adresse
......
```

Figure 13-6: Fichier lien

Post-condition:
Le résultat de l'exécution du programme **extr_cop** est un schéma qui contiendra autant de types d'entités qu'il y avait de COPYBOOKS en entrée. Toute une série d'informations peut être tirée des COPYBOOKS, et traduite dans le schéma. Le Tableau 13-2 reprend les constructions qu'on peut rencontrer dans des COPYBOOKS et leur traitement par le programme.

Constructions trouvées	Traitement par le programme
Nom et niveau de la variable	Création d'un attribut, simple ou décomposable
Type de la variable	Remplissage du champ "type" de l'attribut
Longueur logique de la variable	Remplissage du champ "longueur logique" de l'attribut
Longueur physique de la variable (COMP)	Création des méta-propriétés PSTART et PLENGTH
Clause OCCURS	Création d'un attribut multivalué
Rubrique 66, clause RENAMES	Contraintes d'exactement-un, et de coexistence
Clause REDEFINES	Contrainte d'exactement-un

Tableau 13-2: Parallèles entre les instructions d'un COPYBOOK et le schéma généré

Intéressons nous maintenant à trois des constructions du Tableau 13-2.

1. La longueur physique de la variable

Cette information est stockée dans une méta-propriété PLENGTH identique à celle créée par le programme **dcompims**; le programme calcule la longueur physique de la variable en fonction de l'utilisation ou non de techniques de compression dans le code source du COPYBOOK. Le programme calcule également les positions de départ physiques (PSTART) des divers attributs extraits. Ces informations sont stockées dans deux propriétés dynamiques associées à chaque attribut, à savoir:

PSTART qui est la position physique de départ et
PLENGTH qui est la longueur physique de l'attribut.

Dans l'exemple de la Figure 13-4, la situation est la suivante:

	PSTART	PLENGTH
NOM	1	30
PRENOM	31	30
TEL	61	20
NAIS	81	10

Tableau 13-3: Tableau des positions physiques de l'exemple

2. L'instruction REDEFINES

L'instruction REDEFINES signifie qu'un même emplacement physique possède deux définitions possibles.
La Figure 13-7 montre un exemple de l'utilisation de l'instruction REDEFINES.

```
*                                   INTITULE DU COURS
    05 INTITULE      PIC X(10).
*                                   LOCAL DU COURS
    05 LOCAL         PIC X(5).
*                                   PREMIERE DESCRIPTION DU COURS
    05 DESCRIPTION1.
  *
        10 RUE  PIC X(10).
  *
        10 DIVERS PIC X(30).
  *                                 SECONDE DESCRIPTION DU COURS
    05 DESCRIPTION2 PIC X(40) REDEFINES DESCRIPTION1.
```

Figure 13-7: Exemple d'instruction REDEFINES

Le programme **extr-cop** traite cette instruction en créant d'abord les deux attributs et en ajoutant ensuite une contrainte d'exactement-un entre ces deux attributs. Ils deviennent facultatifs. La Figure 13-8 montre le résultat de l'extraction de la Figure 13-7 par le programme.

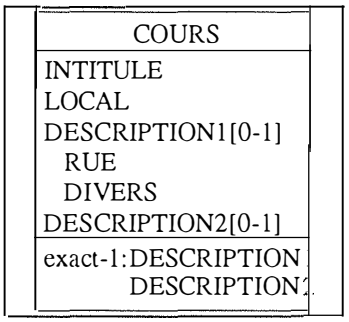


Figure 13-8: Résultat de l'extraction de la Figure 13-7

Il faut remarquer que les champs *description1* et *description2* possèdent les mêmes valeurs de début physique (PSTART) et de longueur physique (PLENGTH). Comme nous l'avons déjà dit, la contrainte d'exactement-un entre les deux attributs s'explique par le fait que, vu qu'ils consomment le même espace physique, ils ne peuvent pas exister ensemble, au même moment dans une même occurrence du type d'entité.

3. L'instruction RENAMES

L'instruction RENAMES signifie également qu'un même emplacement physique possède deux définitions possibles.
Soit l'exemple de la Figure 13-9.

```
*                                   INTITULE DU COURS
 05 INTITULE      PIC X(10).
*                                   LOCAL DU COURS
 05 LOCAL         PIC X(5).
*                                   PREMIERE DESCRIPTION DU COURS
 05 DESCRIPTION1.
*
     10 RUE  PIC X(10).
*
     10 DIVERS PIC X(30).
*                                   SECONDE DESCRIPTION DU COURS
 66 DESCRIPTION2 RENAMES DESCRIPTION1.
*
```

Figure 13-9: Premier exemple de RENAMES

On renomme *description1* par *description2*. *Description2* est de type caractère et possède la même longueur physique que *description1*.

La situation à la fin de l'extraction est la suivante (Figure 13-10).

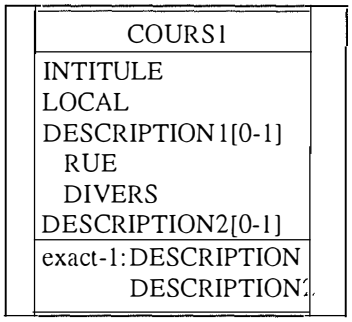


Figure 13-10: Résultat de l'extraction de la Figure 13-9

Soit l'exemple de la Figure 13-11.

```
*                                   INTITULE DU COURS
 05 INTITULE      PIC X(10).
*
*                                   DESCRIPTION DU COURS
 05 LOCAL         PIC X(5).
*
*                                   PREMIERE DESCRIPTION DU COURS
 05 DESCRIPTION1.
*
      10 RUE  PIC X(10).
*
      10 DIVERS PIC X(30).
*                                   SECONDE DESCRIPTION DU COURS
 66 DESCRIPTION2 RENAMES LOCAL THRU DESCRIPTION1.
*
```

Figure 13-11: Second exemple de RENAMES

La dernière ligne de la Figure 13-11 signifie que l'on renomme la zone allant de *local* à *description1* par la variable *description2*.
Après extraction, nous obtenons la situation de la Figure 13-12.

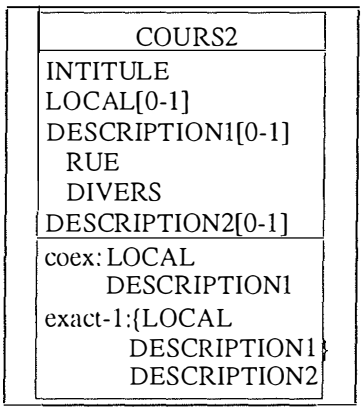


Figure 13-12: Résultat de l'extraction de la Figure 13-11

Considérons la ligne de code suivante: *66 ATTRIBUT RENAMES borne1 THRU borne2.*
On crée d'abord un groupe de coexistence qui contient les attributs qui se trouvent physiquement entre la borne1 et la borne2. Dans l'exemple de la Figure 14-11, LOCAL et DESCRIPTION1. L'un n'existe pas sans l'autre. Donc, soit les deux attributs du groupe de coexistence existent, soit le seul attribut qui est la redéfinition existe. C'est pour cela que l'on a créé une contrainte d'exactement-un entre l'attribut qui est la redéfinition et le groupe de coexistence.

13.2.4. Le programme COPY

Le programme **copy** est utile lorsque l'analyste suspecte une situation ou un même type de segment possède plusieurs définitions possibles. C'est-à-dire que plusieurs COPYBOOKS différents qui sont utilisés dans les programmes décrivent le même type de segment. Nous sommes obligés de nous baser sur les longueurs physiques des types d'entités du schéma pour détecter des redéfinitions éventuelles. Nous sommes conscients que cette situation n'est pas idéale car deux types d'entités peuvent avoir même longueur sans pour autant représenter le même type de segment. C'est pourquoi le programme ne fait rien sans l'autorisation de l'utilisateur.

Pré-condition:
Il est seulement demandé à l'utilisateur de lancer le programme **copy** sur le schéma issu de l'extraction des COPYBOOKS, pour analyser celui-ci, c'est-à-dire pour rechercher des situations de redéfinitions, et les traiter s'il en trouve.

Post-condition:
Pour les cas de redéfinitions trouvés, nous demandons à l'utilisateur s'il veut les intégrer dans un seul type d'entité. S'il accepte, chaque redéfinition est alors transformée en un attribut facultatif du nouveau type d'entité créé. De plus, nous créons une contrainte d'exactement-un entre les attributs facultatifs créés, vu que nous avons affaire à une redéfinition.
Par exemple, soit un cours qui peut avoir une des deux découpes suivantes (Figures 13-13 et 13-14).

COURS1

```
*                                          INTITULE DU COURS
 05 INTITULE      PIC X(10).
*
*                                          DESCRIPTION DU COURS
 05 DESCRIPTION    PIC X(45).
```

Figure 13-13: Description COBOL de COURS1

COURS2

```
*                                          INTITULE DU COURS
 05 INTITULE      PIC X(10).
*
*                                          DESCRIPTION DU COURS
 05 LOCAL         PIC X(5).
*
*                                          DESCRIPTION DU COURS
 05 DESCRIPTION    PIC X(40).
```

Figure 13-14: Description COBOL de COURS2

La phase d'extraction par le programme **extr_cop** de ces deux descriptions donne le résultat décrit à la Figure 13-15.

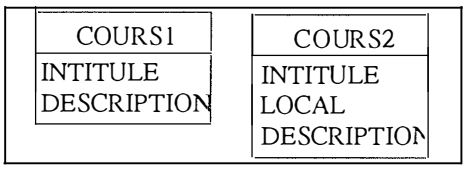


Figure 13-15: Résultat de l'extraction des Figures 13-13 et 13-14

Lorsqu'on lance le programme **copy**, on obtient la console VOYAGER 2, qui demande de choisir les différents types de segments que l'on souhaite intégrer.

Si l'utilisateur choisit les deux types de segments, le programme va alors transformer les deux types d'entités en attributs facultatifs et les rassembler au sein du même type d'entité. Le résultat est à la Figure 13-16.

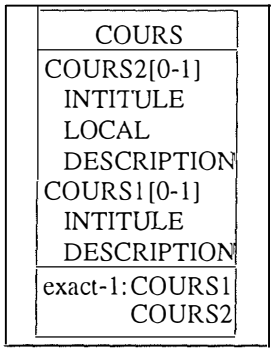


Figure 13-16: Résultat de la première fusion

Nous pouvons remarquer la contrainte d'exactement-un entre les deux attributs facultatifs.

Si nous examinons le code qui décrit COURS1 et COURS2, nous remarquons que l'attribut INTITULE est un PIC X(10) dans les deux cas. Cet attribut occupera donc le même espace physique avec la description de COURS1 et la description de COURS2. Le programme remarque cela et propose à l'utilisateur de le sortir de la définition de COURS1 et de COURS2, pour en faire un attribut obligatoire du nouveau type d'entité; il lui demande aussi le nom qu'il convient de lui donner.
Finalement, si l'utilisateur souhaite en faire un attribut obligatoire, nous obtenons la situation de la Figure 13-17.

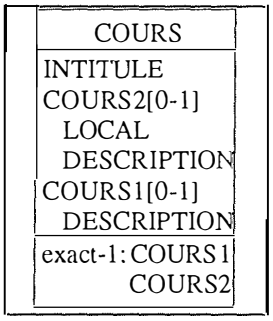


Figure 13-17: Résultat de la seconde fusion

13.2.5. Les programmes d'intégration de schémas

Nous avons développé deux outils différents pour traiter l'intégration des deux schémas correspondants, l'un étant le résultat de l'extraction d'un code DL/1, l'autre étant le résultat de l'extraction des COPYBOOKS (déclarations COBOL) correspondants.
Ces deux schémas sont donc susceptibles de contenir certaines informations dédoublées (données d'intersection) mais chaque schéma contient aussi des informations que l'autre ne contient pas. Il nous faut à tout prix conserver toutes les informations, ne rien perdre.
La création de deux outils différents s'explique par le fait qu'il peut exister certaines contradictions entre la déclaration DL/1 d'une base de données et ses COPYBOOKS correspondants. C'est pourquoi nous avons deux outils distincts:

- Le programme **Analint** est un programme d'analyse des deux schémas sélectionnés par l'utilisateur. Ce programme ne modifie ni ne crée aucun schéma; il se limite à créer un rapport d'analyse qui contiendra toutes les suspicions de contradictions. Il suffira à

l'utilisateur de consulter le fichier texte qui est généré par le programme (dont le nom est spécifié par l'utilisateur) et où il consigne toutes ses remarques.

- Le programme **Integims** est un programme qui intègre automatiquement deux schémas. Certaines précautions doivent être prises avant de lancer ce programme, notamment s'assurer de l'absence de contradictions, car ce programme est 'aveugle' sur ce point. Il paraît donc utile de lancer **Analint** dans tous les cas. A partir des schémas créés par l'extracteur IMS et par le programme **extr cop**, ce programme crée un troisième schéma qui est une intégration des deux autres.

A. Le programme d'analyse ANALINT

Pré-condition:
Il est demandé à l'utilisateur de rentrer les noms respectifs des schémas, via la console Voyager 2, comme la Fenêtre 3 de l'annexe C le montre. La partie inférieure de la console est destinée à recevoir l'input de l'utilisateur, c'est-à-dire les noms des schémas dans ce cas. Si un des noms donnés par l'utilisateur est absent du projet ouvert, le programme est stoppé avec un message d'erreur inscrit à la console.

Post-condition:
Lors de la fin de l'exécution, le fichier texte nommé par l'utilisateur contient toutes les contradictions trouvées entre les deux schémas. L'utilisateur n'a plus qu'à le consulter via un éditeur quelconque.

B. Le programme d'intégration automatique INTEGIMS

Pré-condition:
L'utilisateur doit s'assurer d'abord de l'absence de contradictions entre les deux schémas à intégrer. Lorsqu'il s'en est assuré, il peut lancer le programme. On lui demandera simplement les noms des schémas qu'il veut intégrer, via la console Voyager 2, comme cela s'est passé dans le programme **analint**.

Post-condition:
A la fin de l'exécution, le schéma intégré est créé et est relié à ses deux schémas sources.

13.2.6. Le programme GENE

Le programme GENE est un générateur de programme. Son utilisation intervient lors de la phase d'analyse des données. C'est une application concrète du générateur de programmes de confirmation renseigné dans le chapitre neuf.
Ce programme génère le code source d'un programme COBOL qui vérifie l'existence d'une clé étrangère (au sein de l'application étudiée lors du stage).

Pré-condition:
Un fichier texte contenant le code source d'un programme (COBOL) qui vérifie l'existence ou non d'une clé étrangère est disponible en input.

Post-condition:
Un fichier résultat contenant le code source d'un programme (COBOL) qui vérifie l'existence ou non d'une clé étrangère est disponible en output.

Le principe de ce programme est très simple. En input, il possède le code source d'un programme de vérification de la présence d'une clé étrangère. Le programme parcourt ce code en repérant et en modifiant certains éléments clés dans la recherche de cette vérification.
Ces éléments sont:

- le nom du champ supposé clé étrangère,
- le nom du champ supposé cible,
- la position physique du début de la clé étrangère,
- la position physique de la fin de la clé étrangère.

Les éléments du code initial sont modifiés par les éléments introduits par l'utilisateur. Dans un sens, on pourrait dire que le programme source en input est paramétré par les valeurs introduites par l'utilisateur.
Il faut remarquer que les éléments à fournir par l'utilisateur sont dépendants du programme source en input.
Le lecteur intéressé pourra trouver le code source du fichier utilisé pour l'exécution de ce programme en annexe D.

13.2.7. Le program slicing

Le « slice » d'un programme par rapport à la ligne p du programme et à la variable x, consiste en tous les prédicats et toutes les instructions du programme qui pourraient affecter la valeur de x au point p.
Le program slicing calcule le « slice » qui pourrait affecter la valeur de toute variable référencée à un point donné du programme.
Dans l'atelier DB-MAIN, pour lancer le program slicing, sélectionnez une ligne du programme courant et sélectionnez le menu Assist, puis l'item de menu Text Analysis-Prg slicing (voir la fenêtre 4 de l'annexe C). Une console apparaîtra; elle affiche certaines informations à propos de l'analyse du texte source. Quand c'est fini, il suffit de fermer la console. Les lignes du slice sont colorées.
Remarque: Pour l'instant, l'outil de program slicing de DB-MAIN est uniquement dédié à COBOL.

13.2.8. Le graphe de dépendance

Le graphe de dépendance est un graphe dans lequel les noeuds sont des variables du programme et dans lequel les arcs sont des relations entre ces variables définies par des instructions.
Les instructions qui définissent les relations entre les variables sont définies par des patterns contenant exactement deux variables.
Dans la boîte de dialogue Dependency, accessible par le menu Assist-Text Analysis-Dependency, la combo box Patterns (Fenêtre 5 de l'annexe C) est utilisée pour sélectionner des patterns qu'il faut ajouter à la liste des patterns qui définissent la relation entre variables.
La combo box Separator est le nom du pattern utilisé pour trouver le début et la fin d'une variable.
Un clic sur le bouton OK calculera le graphe de dépendance pour le programme courant. Une fois dans le texte du programme, tout clic avec le bouton droit sur le nom d'une variable entraînera la coloration de toutes les variables en relation avec cette variable.

13.2.9. Le moteur de pattern-matching

Dans le chapitre huit, dans le point consacré à l'analyse des programmes, une des méthodes d'analyse présentée était la recherche de patterns d'analyse. Comme nous l'avons déjà dit dans l'introduction de ce chapitre, cet outil est encore très limité.
L'outil DB-MAIN permet cependant à l'utilisateur de définir des patterns "simples", dans un langage appelé Pattern Definition Language (PDL), qui a une syntaxe proche du BNF. Voyez l'annexe B pour une brève syntaxe de ce langage.
Cependant, cet outil ne permet pas d'exprimer des patterns « composés » qui prennent en compte des niveaux de variables différents.
Par exemple,

```
MOVE VAR1 TO A-SSA-A1
CALL BD GU Apcb A-VAR A-SSA
```

Figure 13-18: Un exemple

Ce pattern induit que la variable A-SSA-A1 du MOVE soit une composante de la variable A-SSA de l'instruction CALL. Or, le moteur de patterns est un outil qui, pour faire ses recherches, se base sur la syntaxe, non sur la sémantique, comme le program slicing.
C'est pourquoi, pour l'instant, cette recherche de « patterns non évidents » ne peut se faire que par le program slicing; cette recherche sera évidemment moins directe et ne permettra pas de « voir » uniquement le pattern voulu, mais après une petite analyse du slice, l'utilisateur repérera sans difficulté ce genre de patterns composés.

13.2.10. L'outil REFERENTIAL KEY

Cet outil de DB-MAIN accessible par le menu Assist-Referential key permet de rechercher l'existence d'une clé étrangère dans un schéma.
La première étape est de sélectionner un groupe identifiant dans le schéma courant, puis de sélectionner le menu Assist-Referential key (voir les fenêtres 6 et 7 de l'annexe C).
L'outil permettra alors de rechercher la présence de clés étrangères par rapport au groupe sélectionné d'après plusieurs critères:

1. Le matching des noms.
2. Le matching des structures des attributs.

Si vous êtes d'accord avec ce que l'outil propose, il vous permet de créer directement la clé étrangère dans le schéma (bouton Create).
Cet outil intégré à DB-MAIN a été modifié pour prendre en compte le cas spécial des clés étrangères hiérarchiques.
Dans le modèle hiérarchique, l'identifiant d'un type de segment dépendant est composé d'un ou plusieurs de ses champs mais aussi des champs composants l'identifiant de son parent.
Dans un schéma entité-association orienté IMS, l'identifiant d'un type d'entité issu d'un type de segment dépendant est donc composé d'un ou plusieurs de ses attributs **et** du rôle joué par le type d'entité parent dans le type d'association qui les relie.
Or, il était impossible pour l'assistant de *foreign keys* de créer une clé étrangère vers un identifiant composé d'un ou plusieurs rôles. Ce problèle a été résolu.

13.2.11. La « Transformation Toolkit »

L'atelier DB-MAIN contient une boîte à outils transformationnels qui permet de faciliter grandement le processus de conceptualisation. On distingue trois niveaux de transformations possibles:

- Les transformations élémentaires, qui consistent à appliquer une transformation T à l'objet courant O. La plupart de ces transformations sont symétriquement réversibles. Il y a plus ou moins 25 transformations disponibles. Pour modifier un des objets du schéma courant, il suffit de le sélectionner, de cliquer sur le menu Transform, et de choisir l'item correspondant à la transformation voulue.
- Les transformations globales, qui consistent à appliquer la transformation T aux objets qui satisfont à la condition P. Il suffit de choisir le menu Assist, item Global transformations, pour obtenir la boîte de dialogue qui permet de définir quelles sont les transformations à effectuer globalement sur le schéma courant (voir la fenêtre 8 de l'annexe C). La partie gauche de cette fenêtre permet à l'utilisateur de choisir, pour chaque classe d'objets, une transformation qui remplace les objets appartenant à cette classe par une construction équivalente (combo-box *into*). Il suffit, une fois que l'utilisateur a fait ses choix, d'ajouter ceux-ci dans la liste de sélection intitulée "Script". En appuyant sur Le bouton OK, l'utilisateur valide le script et il est exécuté sur le schéma courant.
- Les transformations orientées-modèle, ou plans de transformations, qui consistent à appliquer les transformations nécessaires pour rendre le schéma courant satisfaisant au modèle M. En clair, cela consiste à écrire un plan de transformations pour passer par exemple d'un schéma conceptuel à un schéma SQL. Comme on peut le voir dans la fenêtre 9 de l'annexe C, un plan de transformation est fait de transformations (dont les portées peuvent être définies) et de structures de contrôle (incluant des boucles et des if-then-else). Pour en savoir plus concernant la manière d'écrire les scripts, veuillez consulter l'aide en ligne de DB-MAIN.

13.3. Conclusion

Ce chapitre nous a permis de faire le point sur les outils existants déjà et sur certains développements qui sont encore possibles. Comme nous l'avons vu, une minorité des outils existants se substituent à l'homme, la plupart sont des aides pour celui-ci. Il faut bien se dire que jamais un processus de rétro-ingénierie ne sera à cent pourcent automatisé. Le raisonnement humain sera toujours prépondérant dans ce domaine.

CHAPITRE 14: Une étude de cas

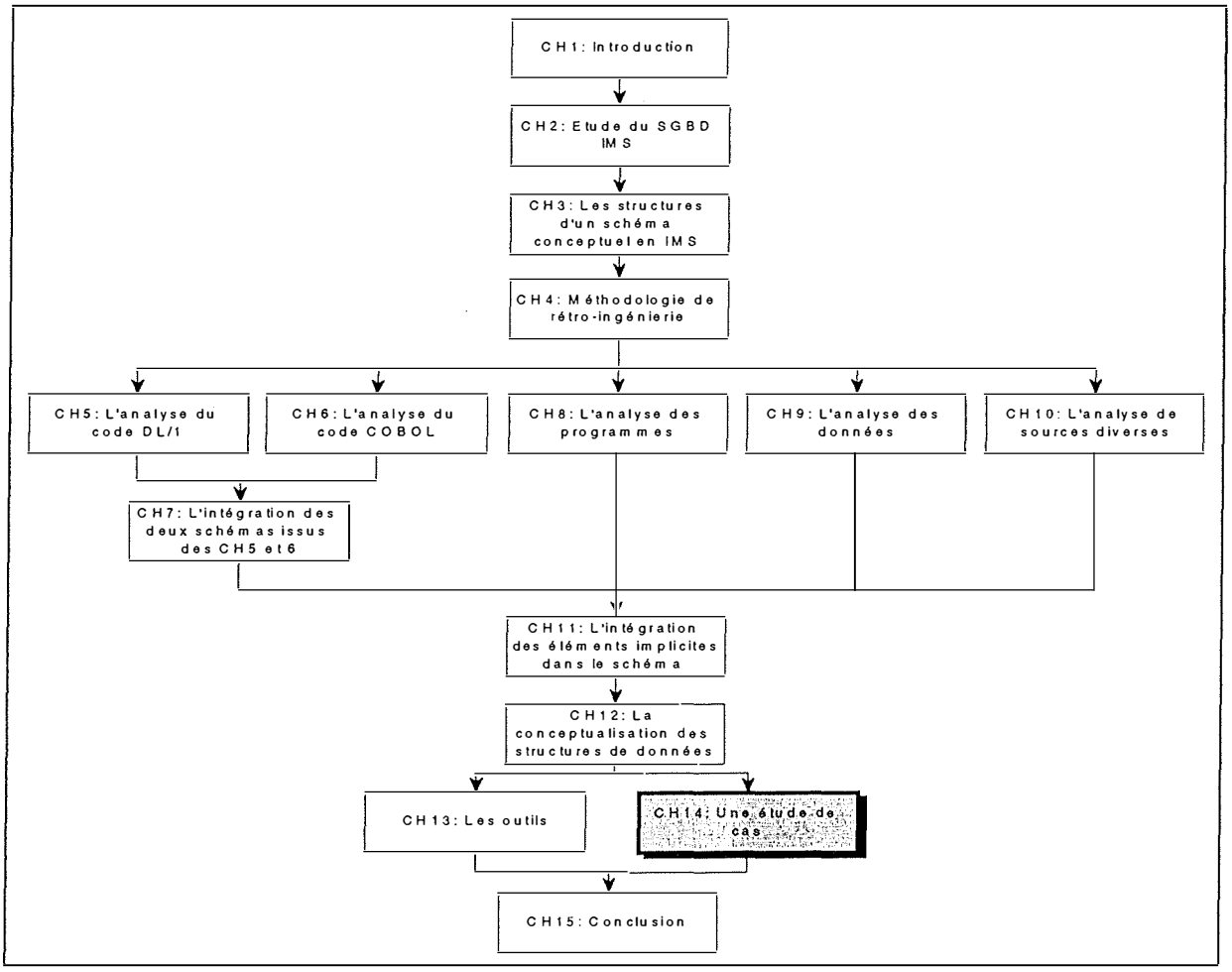


Figure 14-1: Le chapitre 14

14.1. Introduction

Dans ce chapitre quatorze, nous développerons une étude de cas. Celle-ci est tirée de l'ensemble des bases de données sur lesquelles nous avons travaillé durant notre stage. Cette étude de cas permet d'examiner comment appliquer la méthodologie exposée dans les chapitres quatre à douze. Il convient de remarquer que les bases de données choisies ne sont qu'une petite partie d'un ensemble beaucoup plus vaste.

Lors de ce processus, nous utiliserons certaines des applications que nous avons développées. Pour une compréhension approfondie des résultats de ces applications, le lecteur pourra consulter le chapitre treize consacré aux outils. Une grande partie des codes sources des exemples est disponible dans l'annexe E et ceci dans un souci de clarté.

14.2. L'extraction de l'information contenue dans le DL/1

La première phase de la méthode est l'extraction de l'information contenue dans le DDL d'IMS, autrement dit le langage de définition des bases de données. Le principe de cette première phase est exposé au chapitre cinq. Pour cette étude de cas, nous extrayons d'abord le contenu informationnel de la déclaration d'une base de données (CORDDB) qui contient des informations relatives aux commandes passées par des clients (CUSTOMER.- **ORD**ER - **DB**).

L'utilisateur peut pour ce faire utiliser l'extracteur IMS présent dans l'atelier DB-MAIN. Cette fonctionnalité construit automatiquement le schéma entité-association correspondant à la déclaration de la base de données. Le rétro-ingénieur doit spécifier à l'extracteur quel est le fichier qui contient la déclaration de la BD.

La console voyager (Figure 14-2) indique que le processus s'est bien déroulé. Nous pouvons voir que des types de relations, des types d'entités, des clés d'accès ont été créés.

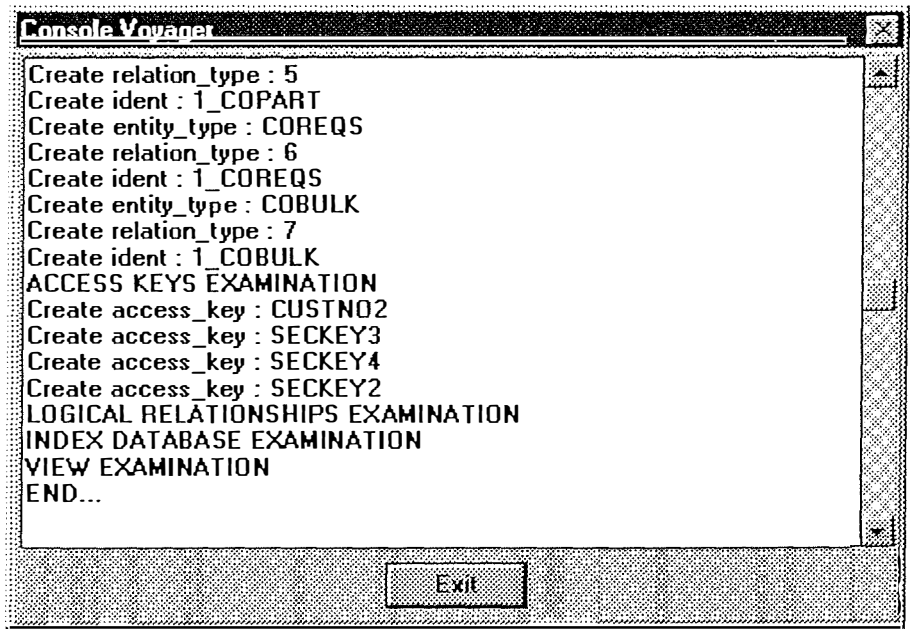


Figure 14-2: Extraction de l'information du DL/1

Après l'extraction, nous obtenons dans la fenêtre *projet* de DB-MAIN, une première figure constituée de deux ellipses reliées par une flèche. La première ellipse constituée d'un trait simple correspond au fichier qui contient la définition de la base de données. La seconde ellipse constituée d'un trait gras correspond au schéma qui résulte de l'analyse du fichier contenant le code DL/1. La flèche reliant les deux ellipses montre que le schéma résultant de l'analyse a été construit à partir du fichier de définition de la BD.

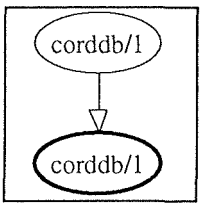


Figure 14-3: Schéma général de l'extraction de l'information contenue dans le DL/1 de CORDDB

Pour notre étude de cas, nous avons choisi d'extraire également une seconde base de données:CUSTDB. Le schéma résultant de cette nouvelle définition de base de données est ajouté au schéma obtenu initialement. Cette nouvelle base de données contient les informations relatives aux clients (**CUSTOMER DB**). Après avoir extrait les informations contenues dans la déclaration de cette autre base de données, nous obtenons la Figure 14-4. Cela signifie que le schéma résultant (ellipse en gras) a été construit à partir des deux fichiers contenant les déclarations des bases de données, à savoir *corddb* et *custdb*.

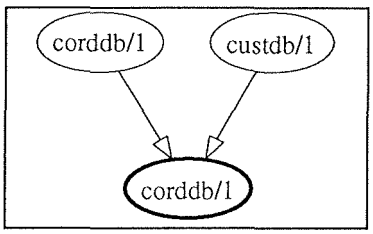


Figure 14-4: Schéma général de l'extraction de CORDDB et de CUSTDB

En cliquant sur l'ellipse en gras, nous obtenons un schéma constitué de types d'entités et de types d'associations dont la disposition ne correspond pas directement à des hiérarchies IMS. Il convient donc de replacer ces types d'entités et ces types d'associations de manière telle qu'ils forment une ou plusieurs hiérarchie(s) IMS. L'extracteur attribue des numéros croissants aux divers types d'associations, ces chiffres correspondant à l'ordre hiérarchique. Ces chiffres facilitent donc fortement le processus de mise en forme des hiérarchies[1]. Après avoir effectué ces petites transformations, nous obtenons la Figure 14-5.
Le lecteur remarquera la présence de deux hiérarchies et de deux collections distinctes qui correspondent aux deux bases de données examinées.

Si nous examinons le type de segment COROOT, nous remarquons qu'il possède plusieurs clés d'accès: (CUSTNO, SECDATE, SECORD), QPRTKEY et HOLDKEY. Ces informations ont été découvertes aux lignes 37, 41 et 46 du code DL/1. Cela est dû à l'instruction SRCH.

Le lecteur attentif pourra également remarquer que le type de relation 4 ne possède pas la cardinalité "habituelle" [0-N]-[1-1] mais une cardinalité [0-1]-[1-1]. Cela est dû à la découverte du mot-clé POINTER=NOTWIN[2] qui signifie que l'on ne peut avoir qu'une seule occurrence du type de segment, d'où la cardinalité [0-1] du rôle joué par le parent de ce type de segment.

Le nombre d'attributs présents dans les différents types de segments est assez réduit. Dans les segments de la base de données des clients (CUSTDB), il n'y a pratiquement que les attributs qui sont identifiants. D'autre part, dans les segments de CORDDB, des clés d'accès sont

[1] Pour rappel, une hiérarchie se lit de bas en haut et de gauche à droite. Il suffit donc d'arranger le schéma en mettant les numéros les plus petits à gauche et ainsi de suite.
[2] Instruction PTR=NT à la ligne 89 du DBD de corddb dans l'annexe E

également présentes. Tout cela peut indiquer qu'une découpe plus précise des types de segments sera présente au sein de COPYBOOKS.

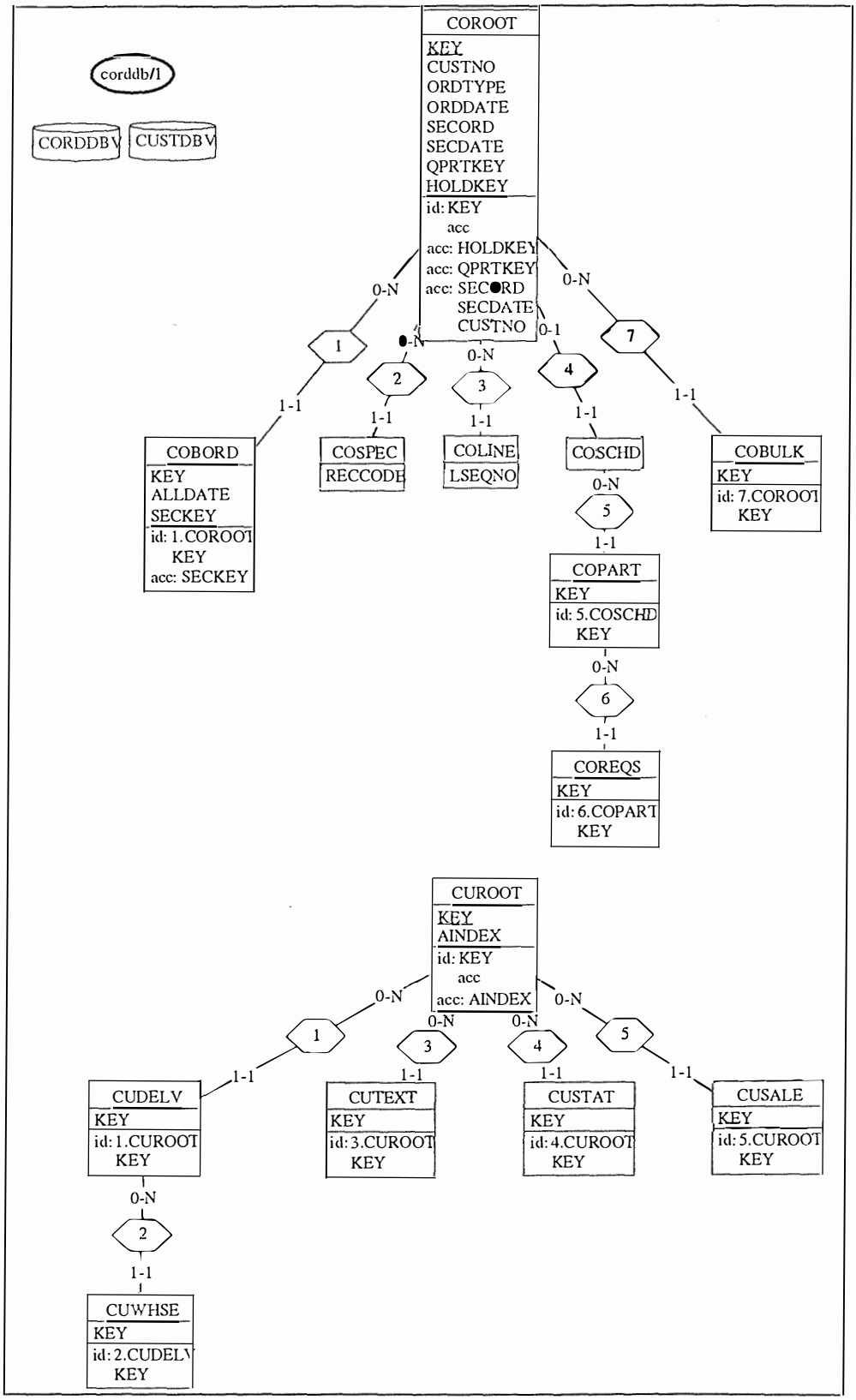


Figure 14-5: Schéma résultat de l'extraction de CORDDB et de CUSTDB

Les caractéristiques techniques des types de segments et des types d'attributs sont contenues dans la boîte de dialogue Technical Description de DB-MAIN. Ces caractéristiques ont été

découvertes lors de l'analyse des DL/1 examinés. La Figure 14-6 en donne une illustration. Ces caractéristiques peuvent être intéressantes pour le bon déroulement du processus de rétro-ingénierie.

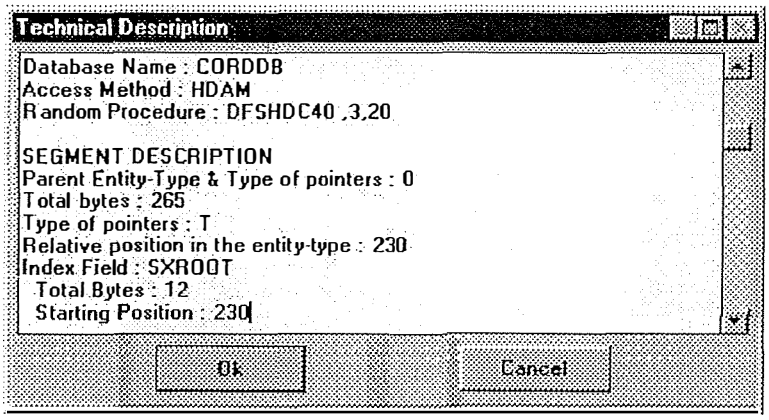


Figure 14-6: La description technique de COROOT

L'information vitale pour la suite du processus est l'information relative aux longueurs physiques. Dans cet exemple, nous pouvons retrouver la longueur du segment COROOT. Pour les différents attributs, l'extracteur repère la position physique du début de l'attribut ainsi que sa longueur. Cela permettra, dans une étape ultérieure, de vérifier l'existence d'un attribut décomposable et en l'occurrence de le créer. Cela permettra également de repérer les attributs qui représentent les mêmes éléments dans les DBD et dans les COPYBOOKS.

La taille des deux bases de données qui font l'objet de cette étude de cas est très importante. Nous avons donc décidé dans un souci de clarté de limiter notre étude à la base de données CORDDB et au segment CUROOT (segment racine de la base de données CUSTDB). Nous obtenons donc la figure 14-7.

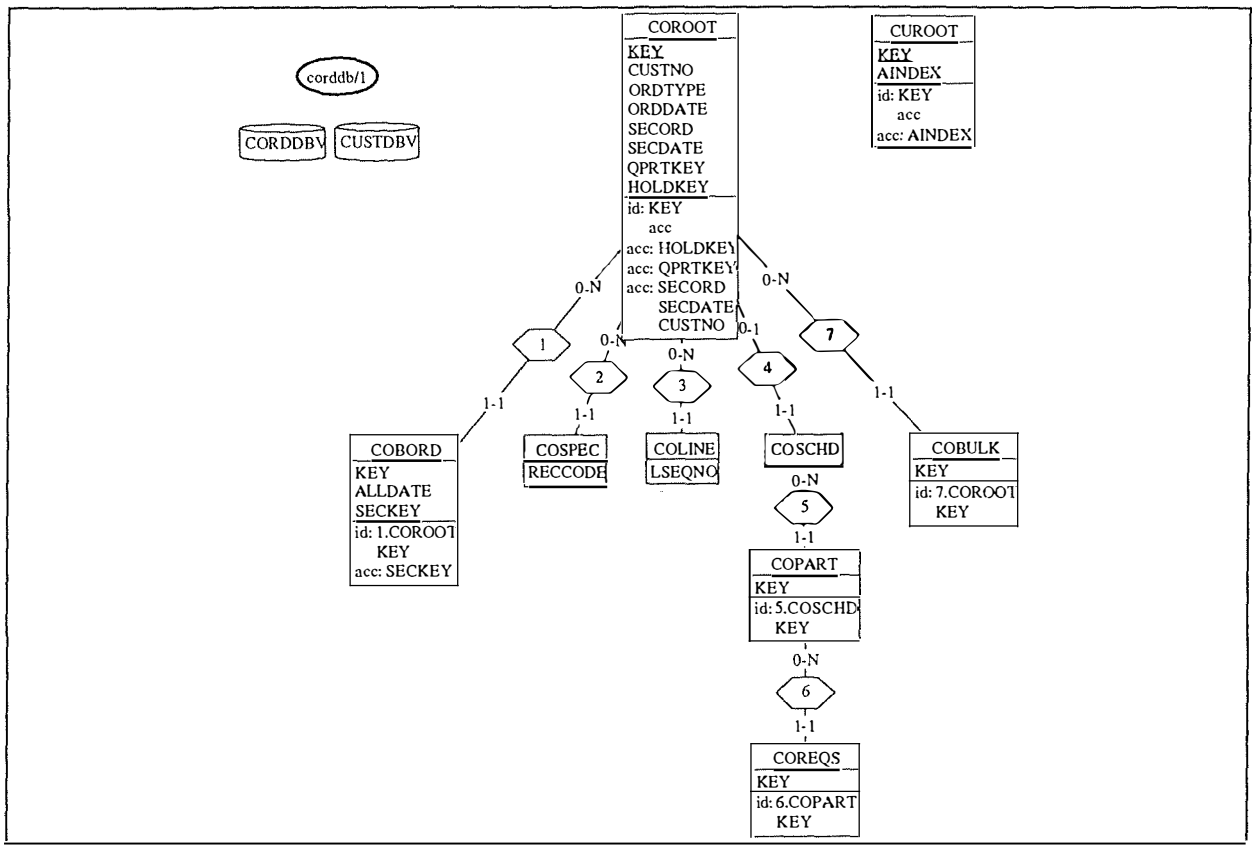


Figure 14-7: Schéma réduit résultat de l'extraction de CORDDB et de CUSTDB

La phase suivante consiste à calculer les positions physiques de tous les attributs et à repérer les attributs décomposables éventuels. Cette phase est indispensable pour réaliser l'intégration. En effet, comme nous l'avons déjà dit, cette phase se base sur les positions physiques des différents attributs. Le rétro-ingénieur peut utiliser pour cela le programme que nous avons réalisé: DCOMPIMS. Ce programme prend les positions physiques présentes dans les descriptions techniques et crée à partir de ces informations des propriétés dynamiques pour chaque attribut et pour chaque type d'entité. Il repère et crée enfin les attributs décomposables.

Dans cet exemple, il n'y a qu'un seul attribut décomposable. L'attribut décomposable KEY.ALLDATE de COBORD a donc été construit. Cette situation est exposée à la figure 14-8.

En examinant le code DDL de CORDDB, nous nous apercevons (de la ligne 53 à la ligne 58) que le champ KEY débute à la position 1 et possède une longueur de 14 tandis que ALLDATE débute à la position 6 et possède une longueur de 7. Il est donc bien clair que ALLDATE est un composant de KEY.

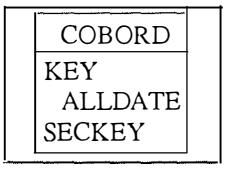


Figure 14-8: L'attribut décomposable

Le schéma devient (Figure 14-9):

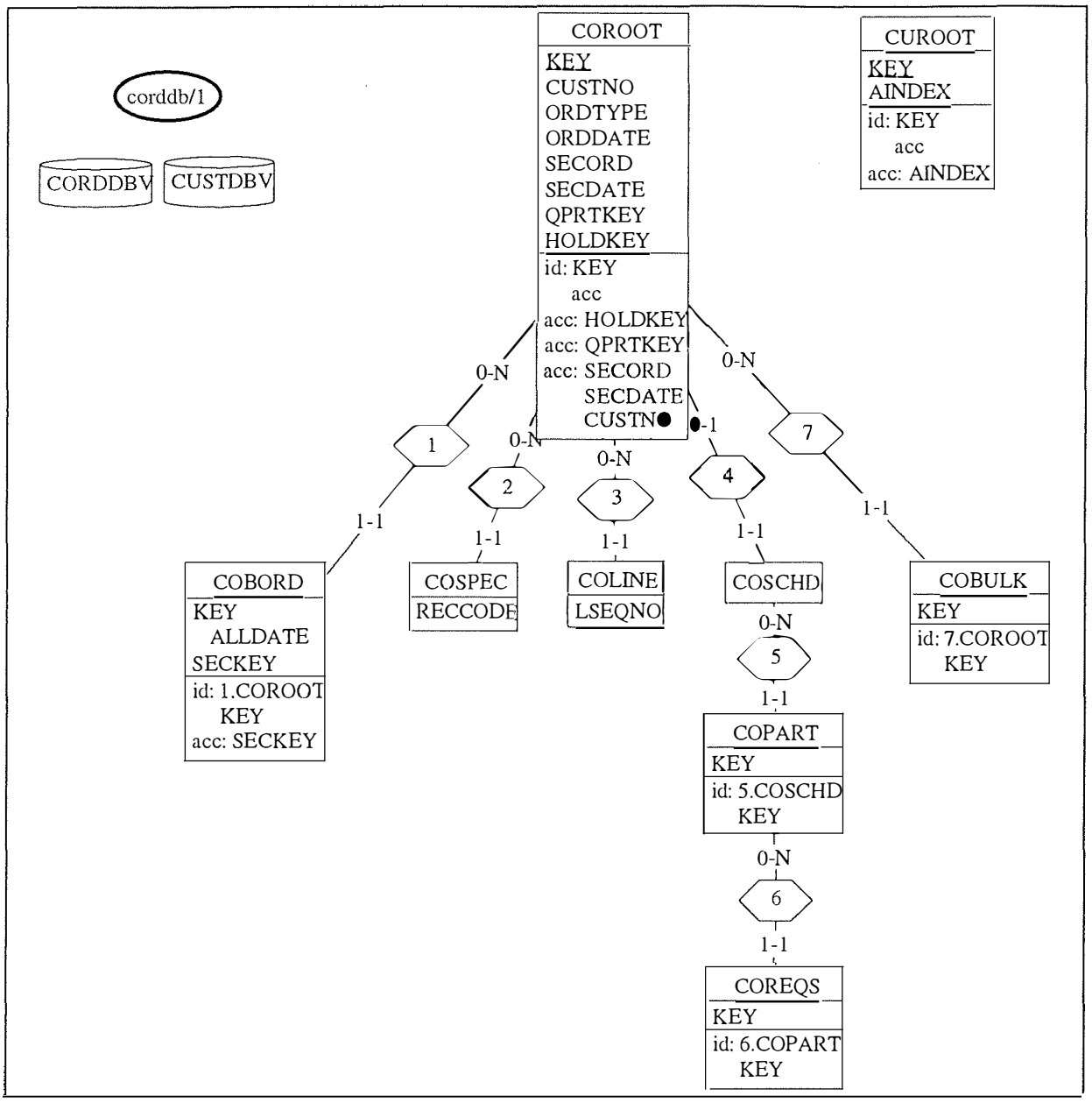


Figure 14-9: Schéma avec le champ décomposable et les positions physiques calculées

Comme nous l'avons déjà mentionné, toutes les positions physiques sont calculées. Elles sont stockées dans la liste des propriétés dynamiques des différents attributs et types d'entités. Il faut remarquer qu'il existe deux longueurs différentes: la longueur physique réelle présente dans la propriété dynamique *plength* et la longueur logique dans le champs *length*. La différence entre ces deux longueurs provient du fait que par exemple les entiers peuvent avoir une représentation physique sur un demi byte plutôt que sur un. La représentation des ces entiers est alors "compactée". Cela a été exposé dans le chapitre six. La figure 14-10 nous montre l'exemple de l'attribut CUSTNO de COROOT dont le début physique se situe à la position 10.

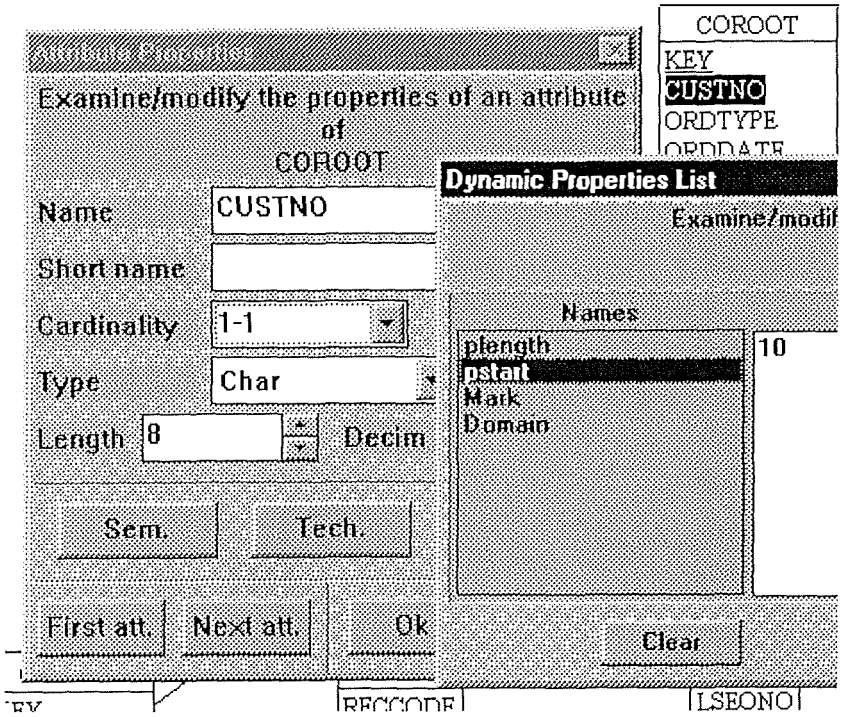


Figure 14-10: Exemple de position physique

Nous ferons remarquer que les positions physiques existaient déjà au sein du champs *Tech.* Cependant pour pouvoir les utiliser aisément, il convient de les extraire et de les stocker dans des champs propres.

14.3. L'analyse du code Cobol

L'étape suivante consiste à extraire l'information qui est contenue dans les COPYBOOKS (chapitre six). Il faut remarquer que cette étape n'existe pas dans tous les processus de rétro-ingénierie. En effet, les COPYBOOKS sont facultatifs.

Pour effectuer cette phase, le rétro-ingénieur peut utiliser le programme écrit en VOYAGER2 que nous avons réalisé. Ce programme porte le nom de EXTR_COP.

Pour utiliser ce programme, il convient de remplir le fichier LIEN de la manière suivante:

CORDDB CUROOT / c:\chemin_d'accès\CUROOT COBORD / c:\chemin_d'accès\COBORD COBULK / c:\chemin_d'accès\COBULK COLINE / c:\chemin_d'accès\COLINE COPART / c:\chemin_d'accès\COPART COREQS / c:\chemin_d'accès\COREQS COROOT / c:\chemin_d'accès\COROOT COSCHD / c:\chemin_d'accès\COSCHD COSPEC1 / c:\chemin_d'accès\COSPEC1 COSPEC2 / c:\chemin_d'accès\COSPEC2 COSPEC3 / c:\chemin_d'accès\COSPEC3 COSPEC4 / c:\chemin_d'accès\COSPEC4 COSPEC5 / c:\chemin_d'accès\COSPEC5 COSPEC6 / c:\chemin_d'accès\COSPEC6 COSPEC7 / c:\chemin_d'accès\COSPEC7	La première ligne contient le nom de la base de données à laquelle correspondent les différents COPYBOOKS. Les lignes suivantes contiennent le nom que l'utilisateur désire donner aux types d'entités et le chemin d'accès et le nom du fichier qui correspond au COPYBOOK. Pour plus d'information, nous renverrons le lecteur au chapitre treize.

Figure 14-11: le fichier LIEN

Le résultat de cette phase est la figure 14-13.

Dans ce schéma nous retrouvons plusieurs occurrences de la contrainte d'exactement-un. Si nous examinons la contrainte

exact-1 :COLINE-REC[*].COLINE-1-DATA
COLINE-REC[*].COLINE-2-DATA

nous pouvons remarquer qu'elle est due à la présence de l'instruction REDEFINES à la ligne 34 dans le COPYBOOK COLINE. La présence de ce REDEFINES peut être également découverte dans le champs Tech de COLINE-2-DATA. La Figure 14-12 montre cette description technique.

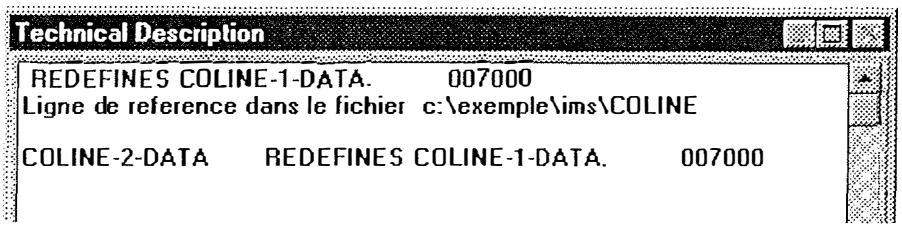


Figure 14-12: Une description technique d'un attribut avec la contrainte REDEFINES

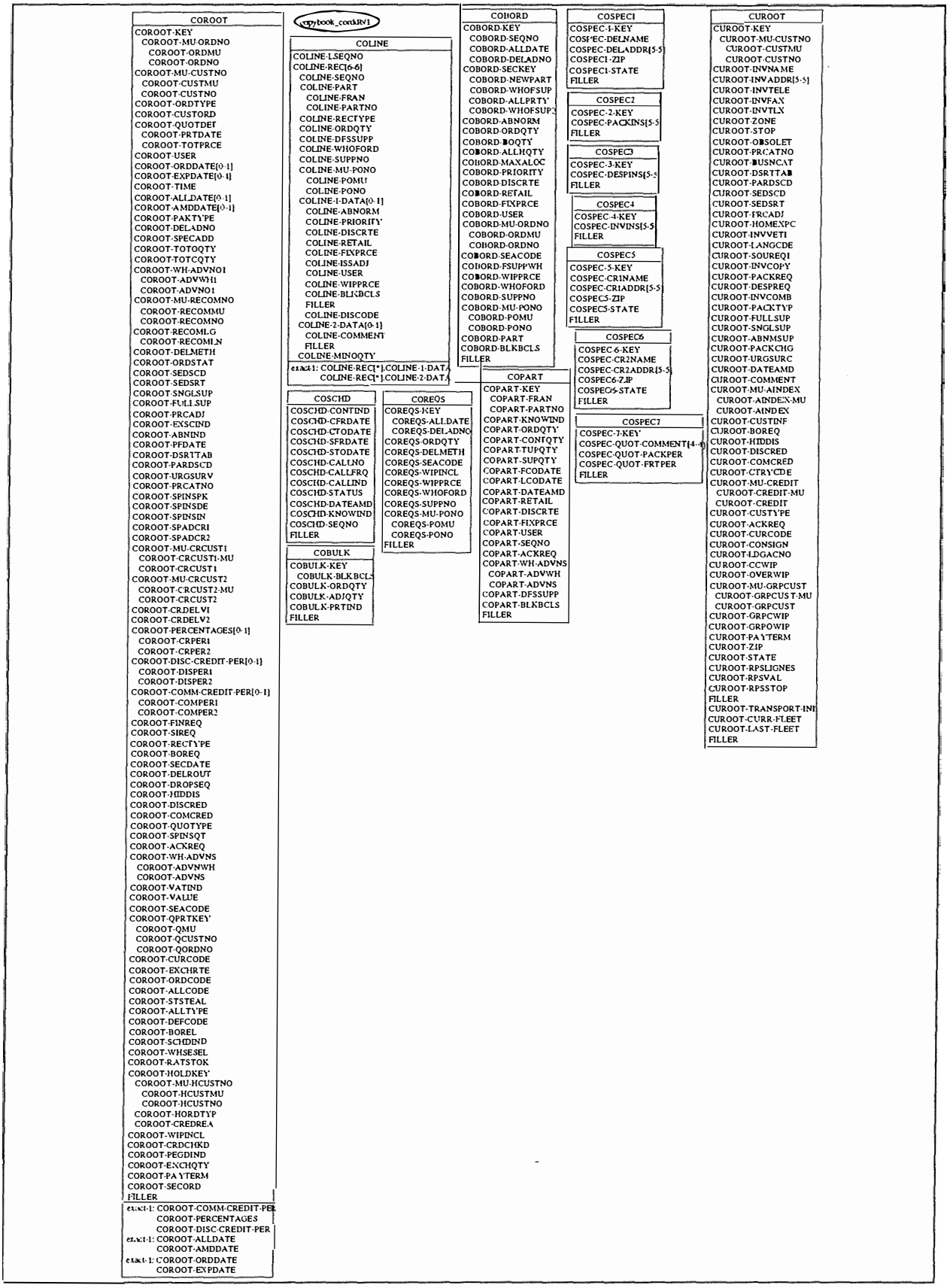


Figure 14-13: Le schéma résultant de l'extraction des COPYBOOKS

La phase suivante consiste à repérer si plusieurs types d'entités issus des COPYBOOKS analysés ne sont pas des redéfinitions d'un seul et même type de segment de la BD.

Dans notre exemple, les types d'entités COSPEC1, COSPEC2, COSPEC3, COSPEC4, COSPEC5, COSPEC6, COSPEC7 sont sans doute des redéfinitions du même type de segment. Les indices sont qu'ils possèdent tous la même longueur physique et que leurs noms sont formés à partie de la même racine: COSPEC. Avant d'effectuer une transformation concernant ces types d'entités, il convient d'être certain qu'il s'agit bien d'une redéfinition. L'avis de l'expert peut intervenir dès cette étape.

Le programme COPY permet d'analyser et de transformer directement le schéma en construisant un seul et même type d'entité pour toutes ces redéfinitions.

Le résultat est le suivant:

Figure 14-14: Schéma de l'intégration des redéfinitions

Le schéma final de l'extraction des informations contenues dans les COPYBOOKS est la Figure 14-15.

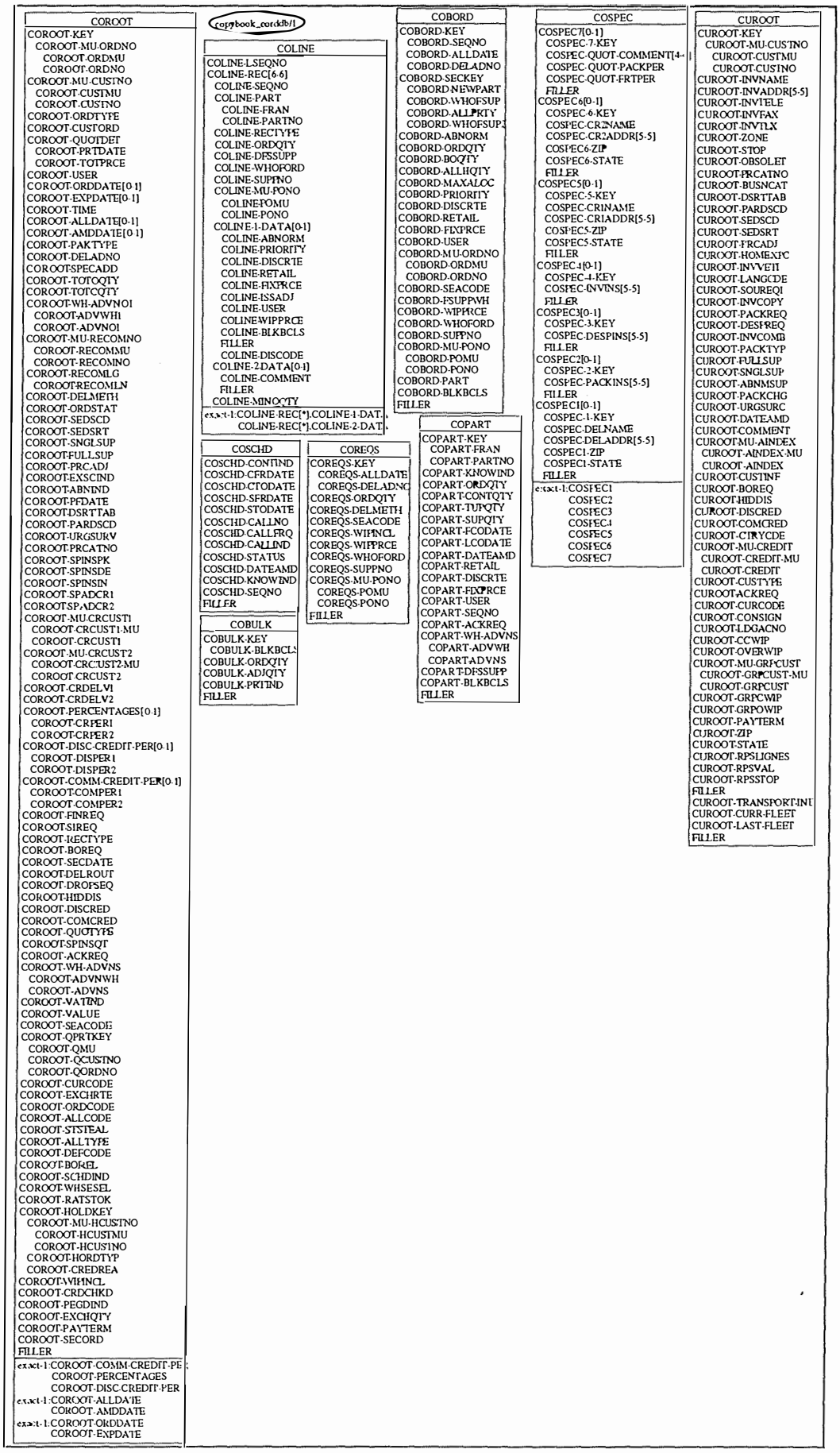


Figure 14-15: Résultat final de l'extraction des COPYBOOKS

14.4. Intégration des deux schémas

Nous possédons donc deux schémas différents. Le premier est la représentation de l'information contenue dans la définition de la base de données. Le second est la représentation de l'information contenue dans les COPYBOOKS. Nous devons donc maintenant intégrer les deux schémas. Cette phase est décrite dans le chapitre sept.

Pour se faire, nous avons réalisé deux applications.
La première (que nous n'utiliserons pas dans cette étude de cas) analyse les différents schémas et détermine les problèmes qui risquent de surgir si l'intégration est effectuée. Son nom est ANALINT.
La seconde (qui est utilisée ici) intègre automatiquement les différents schémas. Cette application s'appelle INTEGIMS.
Pour plus d'information, nous renverrons le lecteur au chapitre treize dans lequel nous expliquons les différentes applications.

L'apparition de la console (Figure 14-16) indique que l'opération s'est bien déroulée. On peut apercevoir dans cette console que différents types d'associations et différents groupes ont été créés. Tous les éléments créés lors de l'intégration n'apparaissent pas dans cette saisie de console. Cependant, ils sont bien créés.

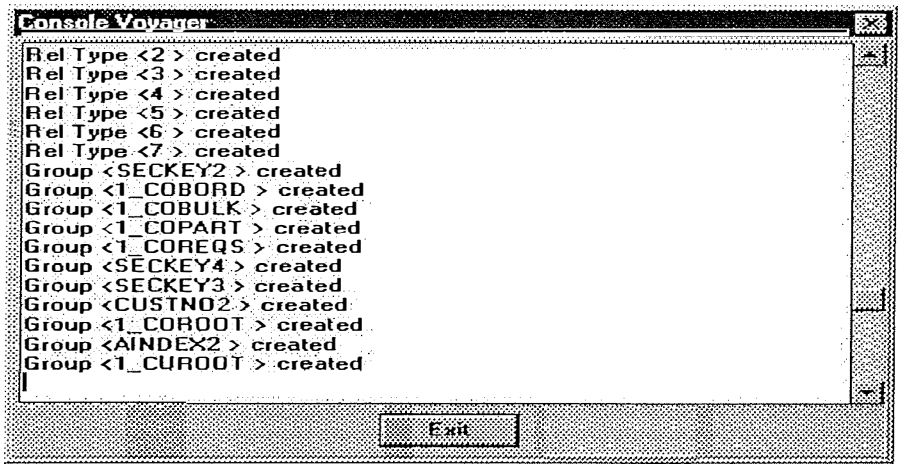


Figure 14-16: Console d'intégration

L'intégration fournit un nouveau schéma. Ce nouveau schéma n'a pas la forme d'une hiérarchie IMS. C'est de nouveau l'utilisateur qui doit réorganiser le schéma.
Après réorganisation du schéma, nous obtenons le résultat de le schéma 14-17.

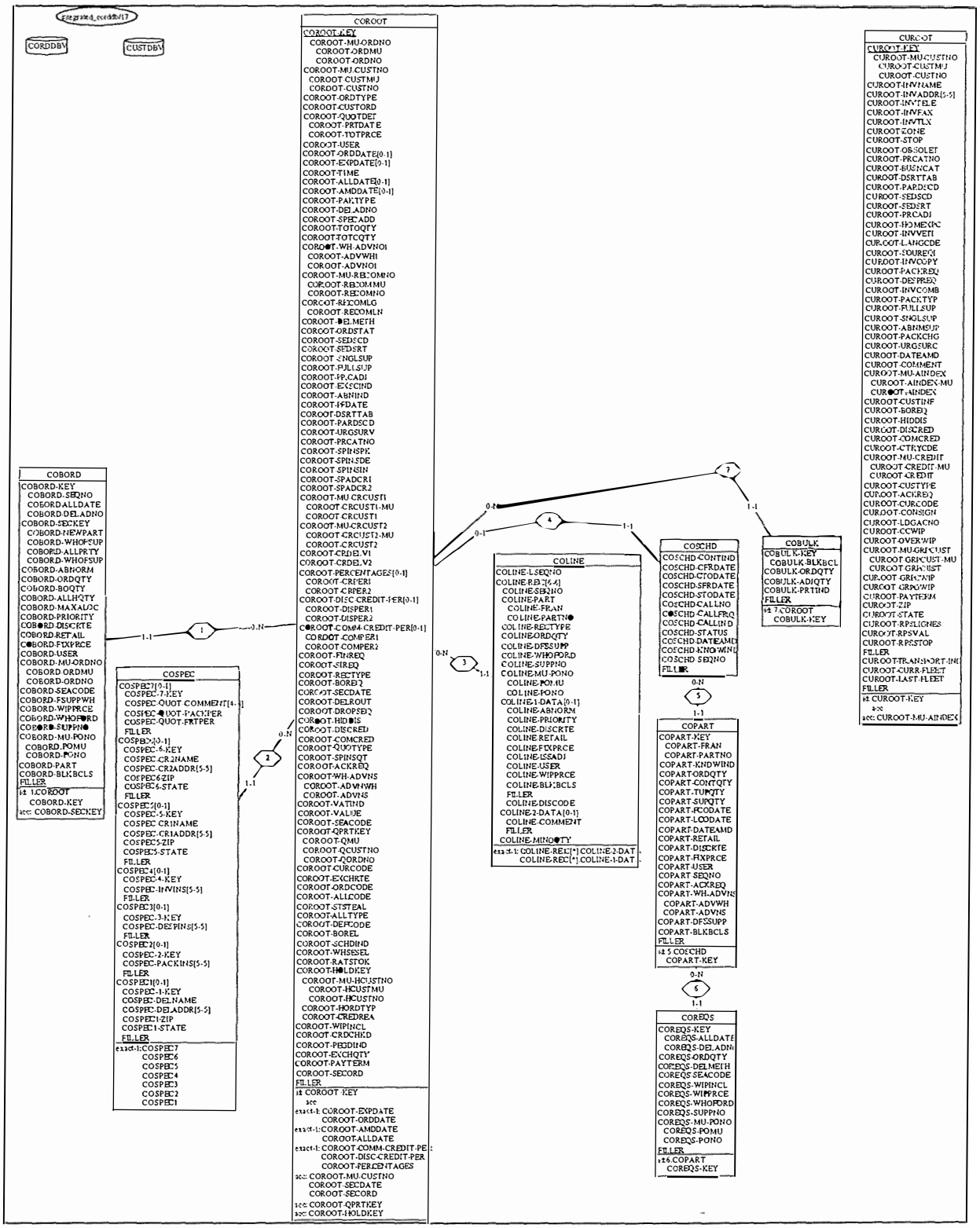


Figure 14-17: Schéma résultat de l'intégration

Le schéma précédent est donc la base de la suite du processus. Les éléments exposés aux chapitres huit, neuf et dix se déroulent la plupart du temps en parallèle. Durant notre stage, nous avons analysé les programmes et exploiter les sources diverses. Par manque de temps, nous n'avons pas examiné en profondeur l'analyse des données ; c'est pourquoi nous en parlerons peu dans cette étude de cas (dans le cadre de clés étrangères).

14.5. Analyse des programmes et des sources d'informations diverses par le rétro-ingénieur

Le rétro-ingénieur doit maintenant exploiter ces différentes sources d'information. Voici un rapport qu'il pourrait fournir à l'expert. Ce rapport contient un ensemble de suspicions qui ont été découvertes grâce à l'utilisation des différents outils développés au chapitre treize.

Suspicion de clé étrangère de COROOT_MU_CUSTNO vers CUROOT_KEY

Dans les programmes: (program slicing)
Programme PC35, ligne 469 et suivantes:

```
MOVE COROOT-MU-CUSTNO TO CUROOTQ-KEY
CALL 'LS3DLI' USING DFHEIBLK DB-USER-AREA FUNC-GU LS-PCB4  CUROOT CUROOTQ.
```

Les noms:
CUROOT_KEY est décomposable en

```
CUROOT_KEY
        CUROOT_MU_CUSTNO
                CUROOT_CUSTMU
                CUROOT_CUSTNO
```

On remarque donc une certaine similitude entre les deux noms. Cela constitue un indice supplémentaire.

Les types et les longueurs:
Les types et les longueurs de la clé étrangère et du champ référencé par celle-ci sont identiques.
Cette construction est également proposée par l'**assistant de clés étrangères**.

Suspicion de clé étrangère de COROOT_PRTKEY_QCUSTNO et de COROOT_QPRTKEY_QMU et vers CUROOT_KEY

Les noms:

```
CUROOT_KEY
        CUROOT_MU_CUSTNO
                CUROOT_CUSTMU
                CUROOT_CUSTNO
```

Il existe une certaine similitude entre QMU et CUSTMU, entre QCUSTNO et CUSTNO. Cela constitue un indice.

Les types:
Les types des clés étrangères présumées et des champs référencés par celles-ci sont identiques.

Suspicion de clé étrangère de HOLDKEY_MU_HCUSTNO vers CUROOT_CUSTNO

Les noms:
Les noms possèdent une racine commune.

Les types:
Les types de la clé étrangère et du champ référencé par celles-ci sont identiques.

Suspicion de cardinalité minimale de: (analyse des données)

COLINE-REC	: 1
CUROOT-INVADDR	: 1

14.6. L'avis de l'expert

Durant cette étape, l'expert examine attentivement le rapport du rétro-ingénieur et détermine les hypothèses qui sont réalistes et qui dès lors peuvent être intégrées au schéma. Si l'expert n'est pas absolument sûr de pouvoir valider les hypothèses du rétro-ingénieur, on choisira de ne pas intégrer ces structures litigieuses. Une autre possibilité pour l'expert consiste à demander au rétro-ingénieur de poursuivre son analyse de manière plus approfondie au sujet de ces structures. Ainsi, de nouveaux éléments peuvent être découverts.
L'expert fixe une pondération qui évalue le degré de certitude de l'élément. Elle a un sens purement indicatif.

	Pondération	Commentaires	Ajouter	Ne pas ajouter
Suspicion de clé étrangère de COROOT_MU_CUSTNO vers CUROOT_KEY	9/10		X	
Suspicion de clé étrangère de COROOT_QPRTKEY_QMU et COROOT_QPRTKEY_QCUSTNO vers CUROOT_KEY	6/10 6/10			X X
Suspicion de clé étrangère de COROOT_HOLDKEY_MU_HCUSTNO vers CUROOT_KEY	8/10		X	
Suspicion de cardinalité minimale de: COLINE-REC CUROOT-INVADDR	 8/10 8/10		 X X	

Dans le schéma suivant, les éléments que l'expert juge adéquats sont ajoutés.

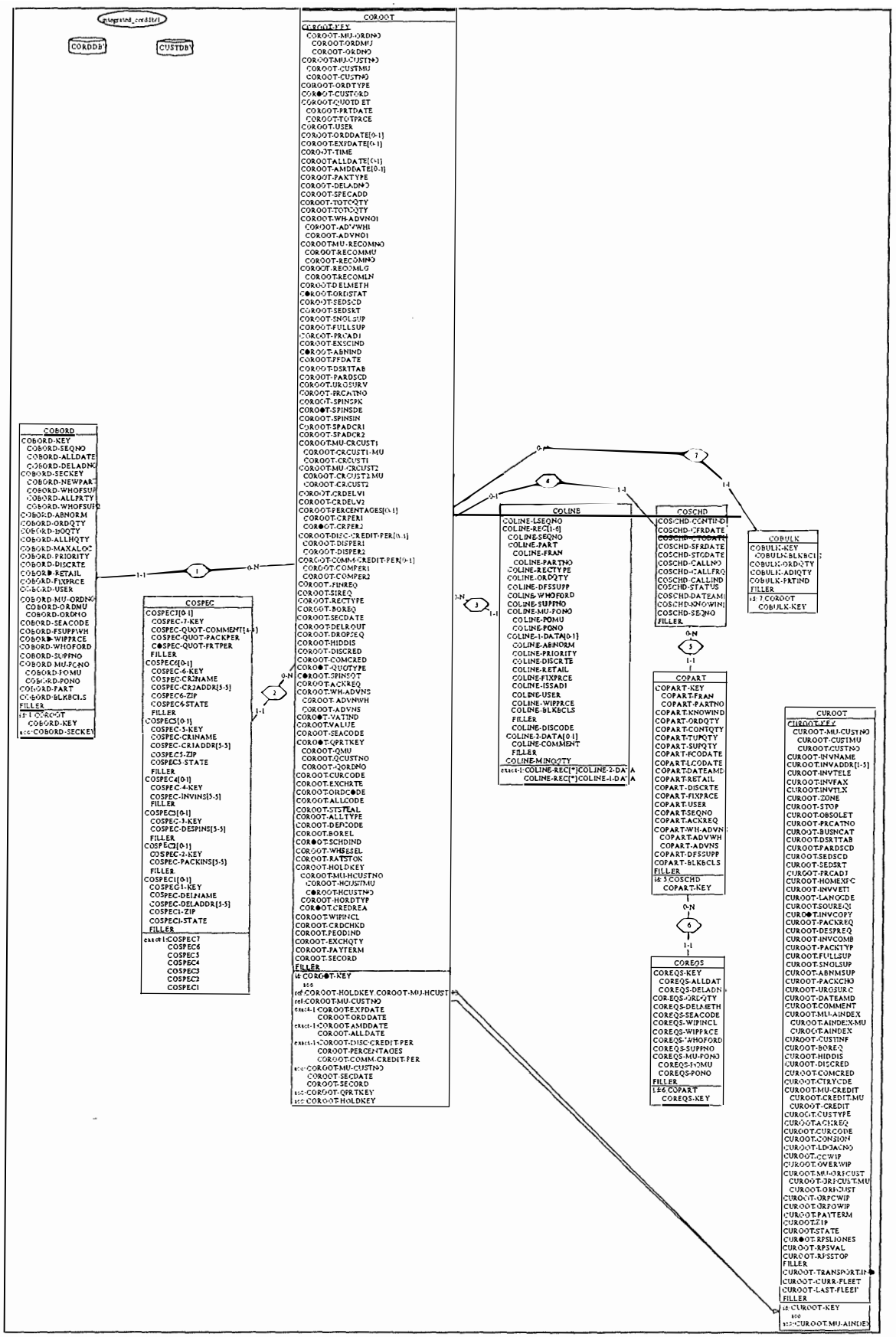


Figure 14-18: Schéma avec ajout des structures implicites

14.7. La conceptualisation

La première étape de la phase de conceptualisation consiste en la modification des noms des types d'entités et des types d'attributs. Le rétro-ingénieur doit veiller à donner à ces divers éléments des noms caractéristiques, simples et facilement compréhensibles. Il est à remarquer que l'information disponible pour retrouver des noms d'attributs adéquats n'est pas toujours disponible. Dans cette étude de cas, il existe certains types d'attributs pour lesquels nous n'avons pas trouvé de noms significatifs dans la documentation disponible. Nous avons alors choisi de les laisser avec leurs noms d'origine.

Les caractéristiques physiques telles que les collections, les clés d'accès ont été enlevées.

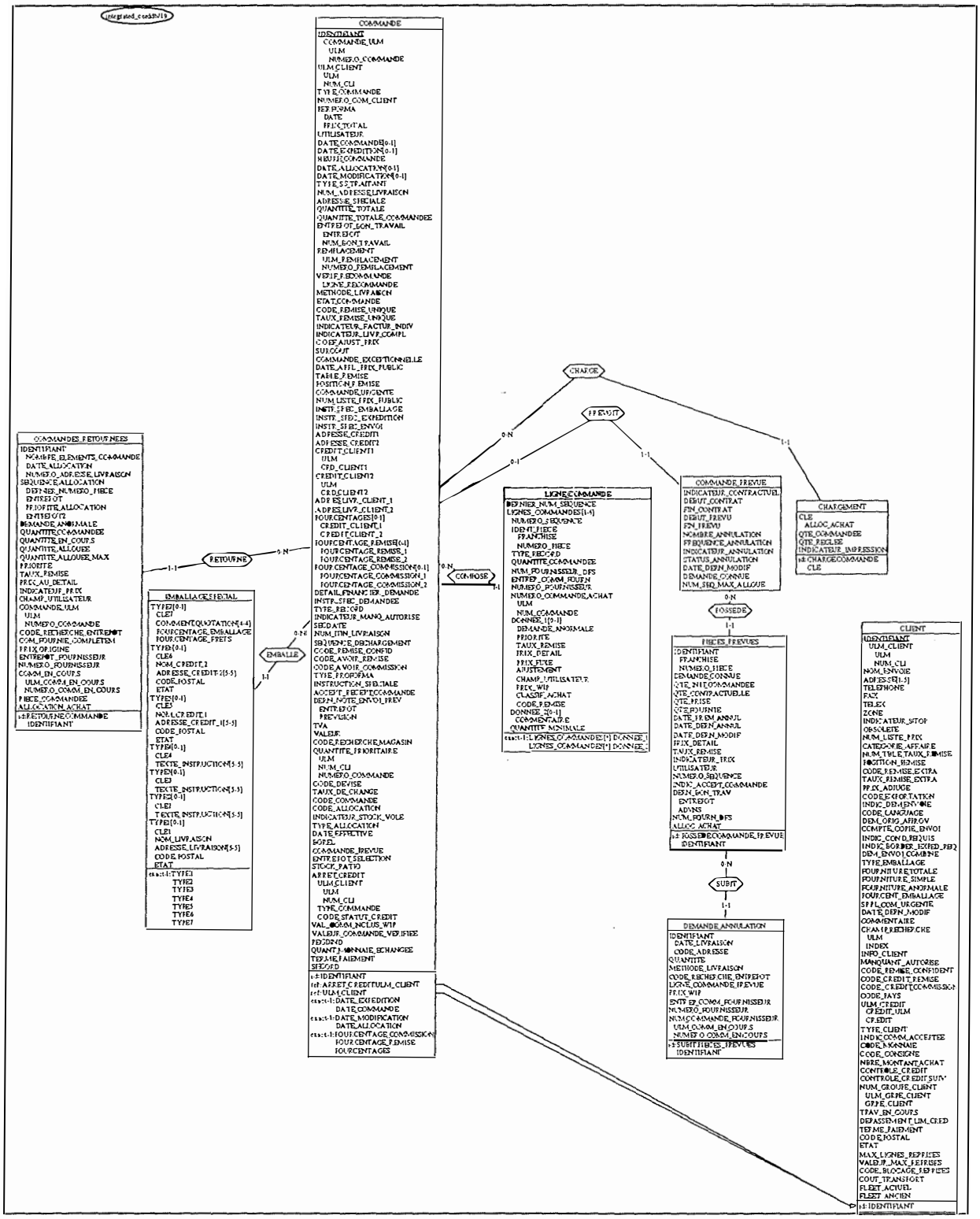


Figure 14-19: Schéma avec modification et ajout de noms

L'étape suivante consiste en la modification du dernier schéma afin qu'il devienne un schéma conceptuel. Il faut donc enlever toutes les constructions propres au SGBD IMS.

Les principales modifications sont les transformations des contraintes d'exactement-un en des types de relations IS-A. Cette transformation n'a pas été présentée dans le chapitre douze car elle appartient au processus de normalisation conceptuelle.

Les contraintes d'exactement-un sont transformées en relations IS-A.

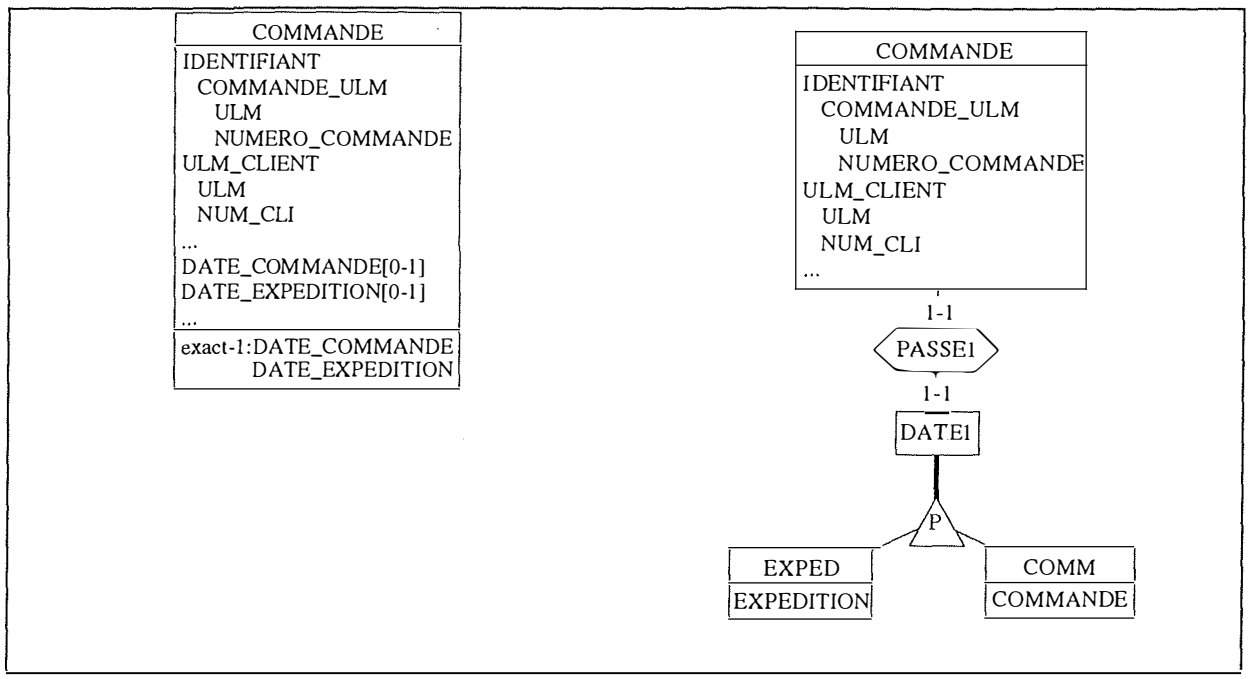


Figure 14-20: Transformation d'une contrainte d'exactement-un

Le symbole P dans la relation IS-A signifie que les deux sous-types forment une partition. Cela signifie que la COMMANDE possède *toujours* une DATE_EXPEDITION *ou (exclusif)* une DATE_COMMANDE.
Nous introduisons un nouveau type d'entité DATE1 qui est utilisé pour permettre de différencier les types de relations IS-A.

Dans le type d'entité CLIENT, nous avons transformé l'attribut multivalué ADRESSE en un type d'entité. La cardinalité de la relation est évidemment [1-5].

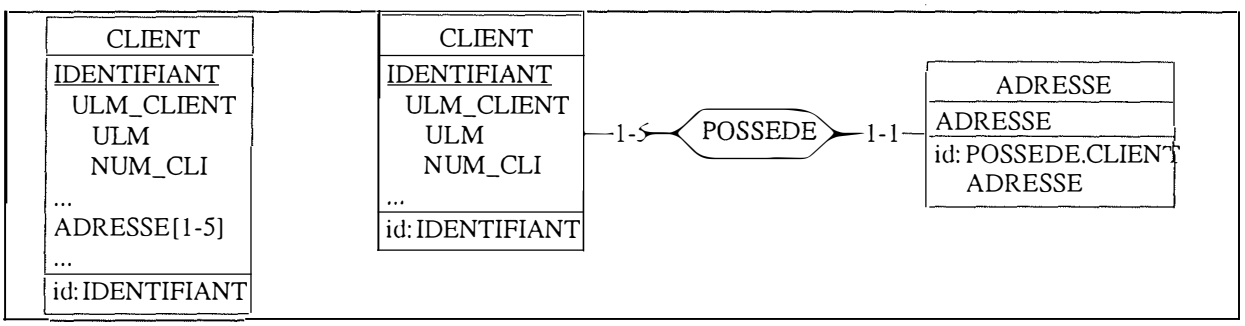


Figure 14-21: Transformation d'un attribut multivalué

Dans le type d'entité LIGNE_COMMANDE, l'attribut LIGNES_COMMANDES est également transformé en type d'entité. Ensuite, la contrainte d'exactement-un est transformée en relation IS-A.

Chaque clé étrangère est transformée en type d'association *one-to-many*.

Dans ce schéma, il existe de nombreux attributs décomposables qui peuvent être transformés en types d'entités. Nous avons choisi de ne pas les créer pour que le schéma reste lisible.

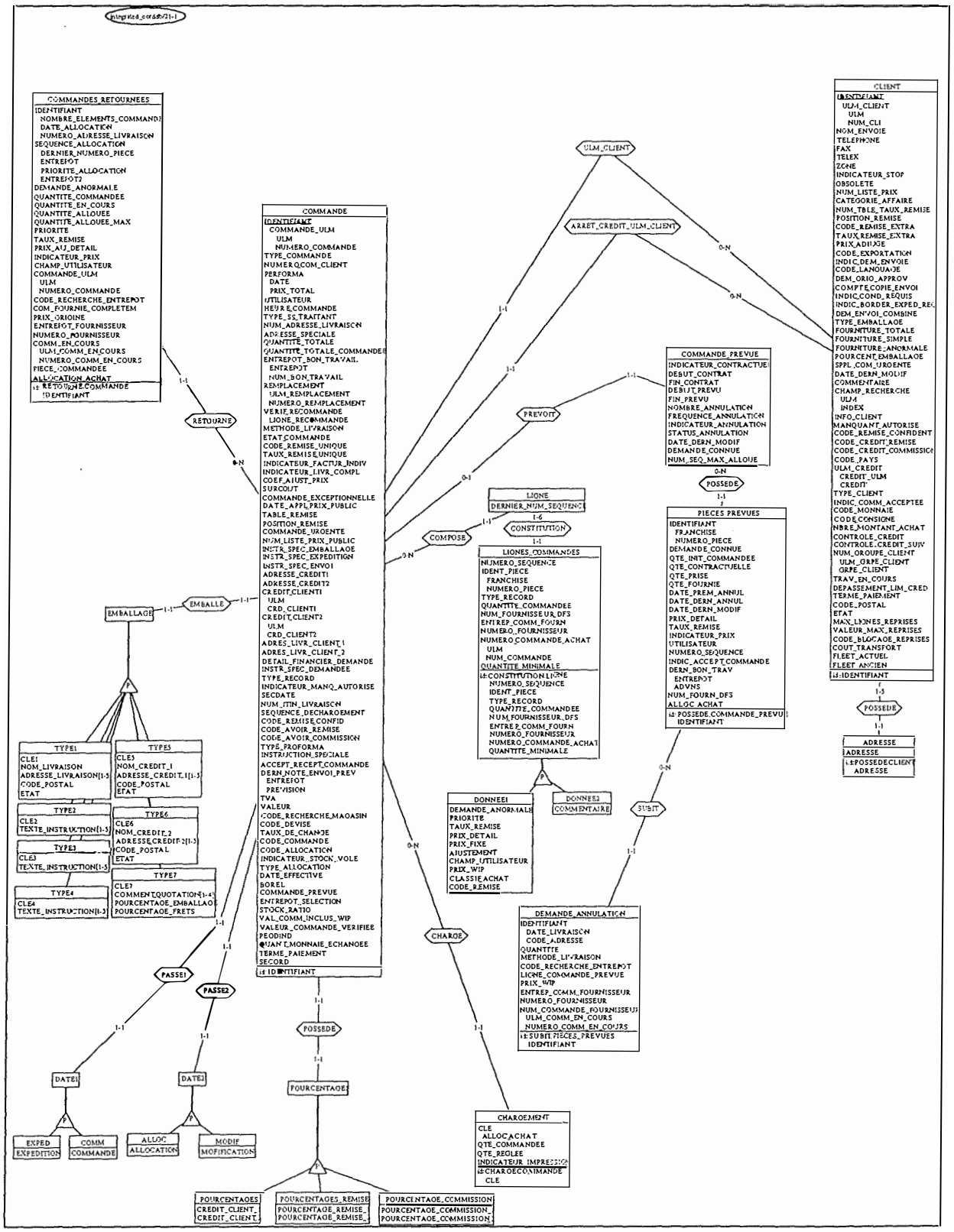


Figure 14-21: Schéma final

14.8. Conclusion

Cette étude de cas nous a permis de mettre en pratique la méthodologie proposée dans la partie trois de ce chapitre. Elle a également permis de montrer l'utilité des programmes développés dans ce mémoire ainsi que de l'atelier DB-MAIN.
Cependant, le lecteur doit savoir que la présentation d'une étude de cas est toujours idéalisée. En effet, tout peut y paraître aisé, automatique, même pour les éléments implicites trouvés dans les programmes; il faut être attentif au fait que la recherche est très ardue à réaliser. Il est rare qu'un projet de rétro-ingénierie trouve 100% du schéma conceptuel initial.

PARTIE 5: CONCLUSION

CHAPITRE 15: Conclusion

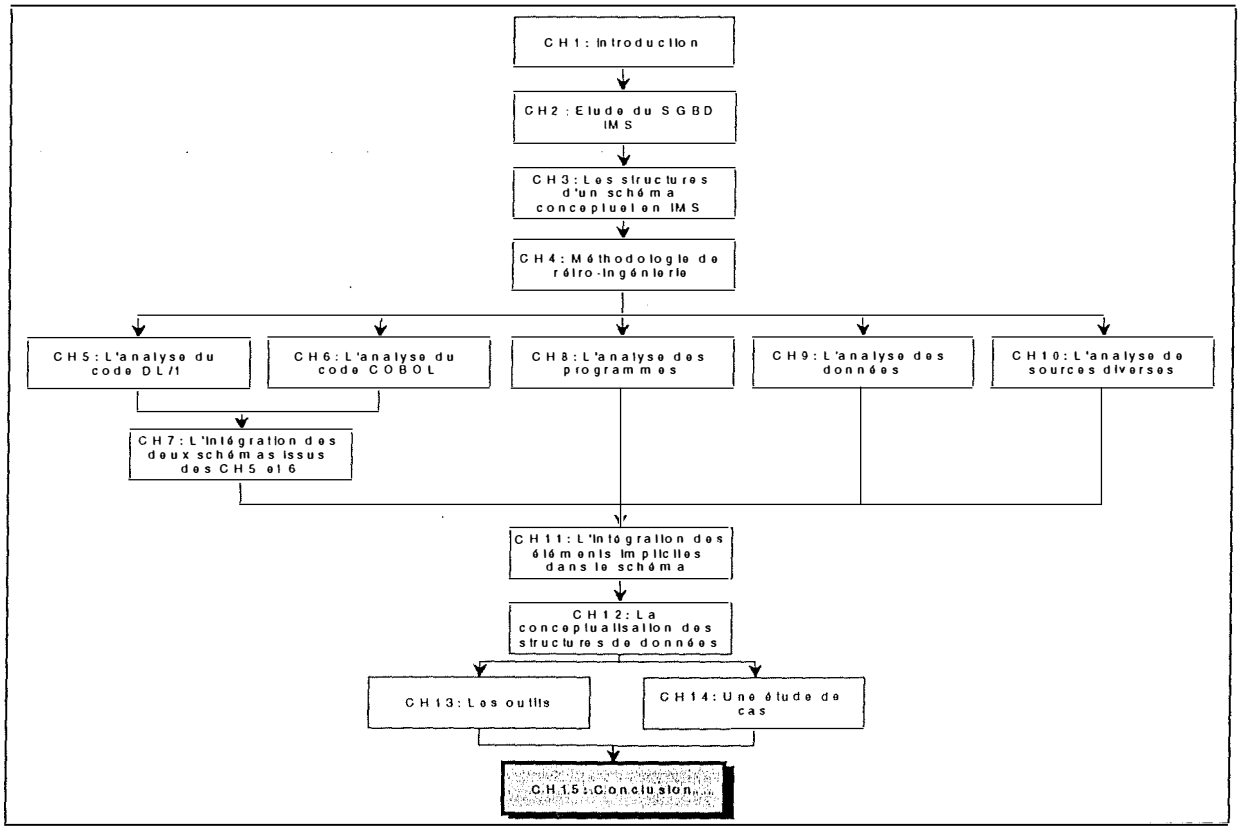


Figure 15-1: Le chapitre 15

Notre but dans ce mémoire était double:

⇒ D'une part, étudier complètement le processus de rétro-ingénierie de bases de données IMS. Pour cela, nous nous sommes basés sur le mémoire de PH. RICHARD **[RICHARD, 95]**, qui avait fait un travail considérable d'analyse du DL/1. Il nous a fallu compléter ce travail; il nous restait notamment à étudier LE problème des processus de rétro-ingénierie, et particulièrement dans les rétro-ingénieries IMS, à savoir la découverte des structures ou contraintes implicites.

⇒ D'autre part, ce mémoire avait aussi pour but de faire évoluer l'atelier DB-MAIN dans la mesure de nos faibles moyens. Nous avons donc réalisé six applications, qui, nous l'espérons, seront intégrées à DB-MAIN. Nous avons parfois fait évoluer les outils existant, qui ne prenaient pas en compte certains cas rencontrés dans notre analyse de l'application de d'Ieteren. Ainsi, l'outil REFERENTIAL KEY a été modifié par J.HENRARD pour prendre en compte les clés étrangères hiérarchiques; le PROGRAM SLICING a lui aussi été modifié pour prendre en compte l'instruction d'appel à la BD (CALL); enfin, le fait d'avoir programmé en VOYAGER 2 pendant quelques mois, nous a permis de découvrir quelques petits problèmes rapidement solutionnés par V. ENGLEBERT.

Quand nous avons commencé l'analyse de l'application chez d'Ieteren, nous nous sommes vite rendu compte que le processus de rétro-ingénierie de BD IMS serait plus complexe que d'autres. En effet, trois difficultés principales sont apparues:

1. La première difficulté fut la découverte des COPYBOOKS. Alors que dans **[RICHARD, 95]**, toute l'information se trouvait dans le DL/1, ici nous avions des codes DL/1 réduits à leur plus simple expression. Les COPYBOOKS ont bien sûr amené leur lot de difficultés, notamment le problème des redéfinitions de types de segments. Ils ont également posé le problème du parallélisme avec l'information du DL/1 (intégration). Cependant, les COPYBOOKS nous ont également facilité la vie; en effet, ils ont permis la découverte des premières structures et contraintes implicites, comme les champs décomposables, répétitifs ou encore les redéfinitions (conflictuelles ou non) de champs.

2. Le problème principal fut néanmoins l'analyse des programmes, et dans une moindre mesure des données, pour trouver des éléments implicites. Alors que jusque là, on savait où trouver l'information (DL/1, COPYBOOKS), nous étions confrontés à une masse imposante de code en COBOL (60 MB). Notre but était principalement de trouver des patterns qui pourraient aider le rétro-ingénieur dans sa tâche. Dans un premier temps, nous avons décidé d'analyser les programmes à la recherche d'indices de clés étrangères. Après plusieurs jours d'analyses laborieuses, nous avions trouvé quelques patterns intéressants. Cependant, il nous apparut très vite que nous ne pourrions pas analyser 60 MB de code pour chaque construction implicite possible. Finalement, nous nous sommes concentrés uniquement sur la découverte d'indices concernant les clés étrangères, les redondances et les attributs décomposables. La plupart des patterns du chapitre huit sont donc 'théoriques', dans le sens où ils ne sont pas issus de notre expérience personnelle.

3. Enfin, le processus de conceptualisation, c'est-à-dire la transformation des constructions IMS en des constructions conceptuelles ne fut pas des plus simple non plus. Bien sûr, l'atelier DB-MAIN facilita la tâche mais certaines transformations sont loin d'être directes, comme on a pu le voir au chapitre trois.

Ce mémoire est donc un ouvrage de recherche encore incomplet, mais qui a l'avantage de synthétiser le processus de rétro-ingénierie de bases de données IMS. Le résultat de ce mémoire nous semble aussi assez vraisemblable du fait que nous avons adopté une vue la plus pratique possible du processus. Nous avons retiré beaucoup de cette étude, tant au niveau théorique que pratique.

En ce qui concerne l'atelier DB-MAIN, notre environnement de référence, nous avons quelques remarques à faire. Nous avons tout d'abord été assez étonné de la vitesse de son évolution. En effet, sur la période d'un an pendant laquelle nous l'avons utilisé, il n'a cessé de se bonifier. Nous voudrions cependant insister sur quelques petites lacunes.

Premièrement, lors de notre stage chez d'Ieteren, nous avons été confrontés à des bases de données assez volumineuses. Dès que l'on dépasse une vingtaine de types d'entités et de types d'associations, le graphique résultant de l'extraction est pour le moins désorganisé. Il serait intéressant de définir des règles d'emplacement graphique qui permettrait d'obtenir automatiquement, dans le cadre d'une base de données hiérarchique, une hiérarchie dans le bon ordre.

Deuxièmement, il nous semble qu'un effort particulier devrait être fait au niveau du PDL, le langage de définition de patterns, qui, nous en avons fait l'expérience, est encore fort limité pour l'expression de patterns 'complexes'. Ce serait un outil formidable, qui ferait de DB-MAIN un logiciel qui deviendrait exceptionnel au niveau de l'analyse des programmes, avec ce langage PDL amélioré, le program slicing et le graphe de dépendance.

Ces deux petites remarques n'enlèvent rien à la qualité d'ensemble de DB-MAIN, tant au niveau des outils transformationnels, que des outils d'analyse de textes, sans oublier le langage VOYAGER 2, dont nous voudrions souligner la facilité d'utilisation en même temps que la puissance d'expressivité.

En guise de conclusion, nous voudrions répéter que la rédaction de ce mémoire a été très enrichissante pour nous. Nous espérons qu'il aura permis de faire avancer la recherche en matière de rétro-ingénierie.

BIBLIOGRAPHIE

[ANDERSSON, 96] Martin Andersson Reverse Engineering of Legacy Systems from Value-based to Object-based Models. Ecole polytechnique fédérale de Lausanne. Lausanne 1996.

[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.

[BODART,93] F. BODART et Y. PIGNEUR, Conception assistée des systèmes d'information, méthodes-modèles-outils, Masson 1993, ISBN 2-225-81807-X

[CLARINVAL, 91]: André CLARINVAL, Comprendre et connaître le COBOL 85, Presses universitaires de Namur, 1991.

[DATE, 90]: C.J. DATE, An Introduction to Database Systems, Appendix B: A Hierarchic System: IMS, p753-789, Volume 1, Fifth edition, Addison-Weslsey,1990.

[ELMASRI, 89]: R.EMASRI, 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.

[ENGLEBERT, 95] : Voyager2 Référence Manual Version 1.0, 20 octobre 1995, FUNDP Namur

[ENGLEBERT 96] : DB-MAIN Voyager2, 9 septembre 1996 DB-MAIN FUNDP Namur

[GALACSI, 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.

[HAINAUT, 86]: J.L. HAINAUT, Conception des applications informatiques Tome 2. Conception de la base de données, Masson, 1986.

[HAINAUT, 95a] : Jean-Luc HAINAUT Database Reverse Engineering, Problems, Methods and Tools. FUNDP juin 1995.

[HAINAUT, 95b] : Jean-Luc HAINAUT Transformation-Based Database Enginnering. VLDB'95 Zürich (Switzerland)- septembre 1995.

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

[HAINAUT, 95d] : J-L. Hainaut, Cours de bases de données de 1° Licence, Transformations de schémas, 1995.

[HAINAUT, 96a] : Jean-Luc HAINAUT Specification preservation in schema transformations-Application to semantics and statistics, reprinted from Data & Knowledge Engineering 19 (1996) 99-134.

[HAINAUT, 96b] : J-L. HAINAUT, J. HENRARD, D. ROLAND, V. ENGLEBERT, J-M HICK, Structure Elicitation in Database Reverse Engineering, 1996.

[HENNEBERT]: H. HENNEBERT, IMS/360, rapport pour l'institut d'informatique des FUNDP.

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

[NAVATHE] : S.B. Navathe, Awong A.M., Abstracting Relationnal and Hierarchical Data with a Semantic Data Model, Database Systems Research and development Center University of Florida Gainesville, FL 32611.

[PETIT] : PETIT J-M, KOULOUMDJIAN J, BOULICAUT J-F, TOUMANI F, Using Queries to Improve Database Reverse Engineering, Laboratoire d'Ingénierie des Systèmes d'Information INSA Lyon, F-69621 Villeurbanne Cedex, France.

[RICHARD, 95] : Philippe Richard : La Rétro-Ingénierie des Bases de Données, Application à IMS, Mémoire présenté pour l'obtention du grade de Licencié et Maître en informatique. FUNDP Institut d'informatique 1995.

[WEISER, 84] : M. Weiser. Program slicing. IEEE Transaction on Software Enginnering, SE-10(4): 352-357, July 1984.

[WINANS] : John WINANS, Kathi HOGSHEAD DAVIS, Software Reverse Engineering from a Currently IMS Database to an Entity-Relationship Model, Department of Computer Science, Northerm Illinois University Dekalb, IL 60115.

ANNEXES

Annexe A: La clause USAGE en COBOL (apport au niveau des longueurs physiques).

Tableau 1: données entières

Clause PICTURE	Clause USAGE	Longueur physique
PIC S9(n) [n <= 18]	USAGE IS DISPLAY	n
PIC S9(n) [n <= 18]	USAGE IS DISPLAY SIGN IS TRAILING	n
PIC S9(n) [n <= 18]	USAGE IS DISPLAY SIGN IS LEADING	n
PIC S9(n) [n <= 18]	USAGE IS DISPLAY SIGN IS TRAILING SEPARATE	n+1
PIC S9(n) [n <= 18]	USAGE IS DISPLAY SIGN IS LEADING SEPARATE	n+1
PIC 9(n) [n <= 18]	USAGE IS DISPLAY	n
PIC 9(n) [n <= 4]	USAGE IS COMP	2
PIC 9(n) [5 <= n <= 9]	USAGE IS COMP	4
PIC 9(n) [10 <= n <= 18]	USAGE IS COMP	8
PIC S9(n) [n <= 4]	USAGE IS COMP	2
PIC S9(n) [5 <= n <= 9]	USAGE IS COMP	4
PIC S9(n) [10 <= n <= 18]	USAGE IS COMP	8
	USAGE IS INDEX	4
	USAGE IS POINTER	4
	USAGE IS COMP-1	4
	USAGE IS COMP-2	8
PIC S9(n) [n <= 18]	USAGE IS COMP-3	(n+1) / 2 rounded up
PIC 9(n) [n <= 18]	USAGE IS COMP-3	(n+1) / 2 rounded up

Tableau 1: Les données entières

Tableau 2: données non entières

Clause PICTURE	Clause USAGE	Longueur physique
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY	n+s
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY SIGN IS TRAILING	n+s
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY SIGN IS LEADING	n+s
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY SIGN IS TRAILING SEPARATE	n+s+1
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY SIGN IS LEADING SEPARATE	n+s+1
PIC 9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY	n+s
PIC 9(n)V9(s) [(n+s) <= 4]	USAGE IS COMP	2
PIC 9(n)V9(s) [5 <= (n+s) <= 9]	USAGE IS COMP	4
PIC 9(n)V9(s) [10 <= (n+s) <= 18]	USAGE IS COMP	8
PIC S9(n)V9(s) [(n+s) <= 4]	USAGE IS COMP	2
PIC S9(n)V9(s) [5 <= (n+s) <= 9]	USAGE IS COMP	4
PIC S9(n)V9(s) [10 <= (n+s) <= 18]	USAGE IS COMP	8
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS COMP-3	(n+s+1) / 2 rounded up
PIC 9(n)V9(s) [(n+s) <= 18]	USAGE IS COMP-3	(n+s+1) / 2 rounded up

Tableau 2: Les données non entières

Annexe B: Syntaxe du langage PDL

<pattern>: <pattern_name>::= *;

: <terminal_seg>
| <pattern_name>
| <variable>
| <range>
| <optional_seg>
| <repeat_seg>
| <group_seg>
| <choice_seg>
| <regular_expr>

<variable>: @<pattern_name>
Le symbole '@' indique que le segment est une variable. Une variable ne peut pas apparaître dans une structure répétitive.

<range>: range(c1-c2)
Tout caractère entre les caractères c1 et c2.

<optional_seg>: []

<repeat_seg>: *

<group_seg>: (*)

<choice_seg>: { | ... | }

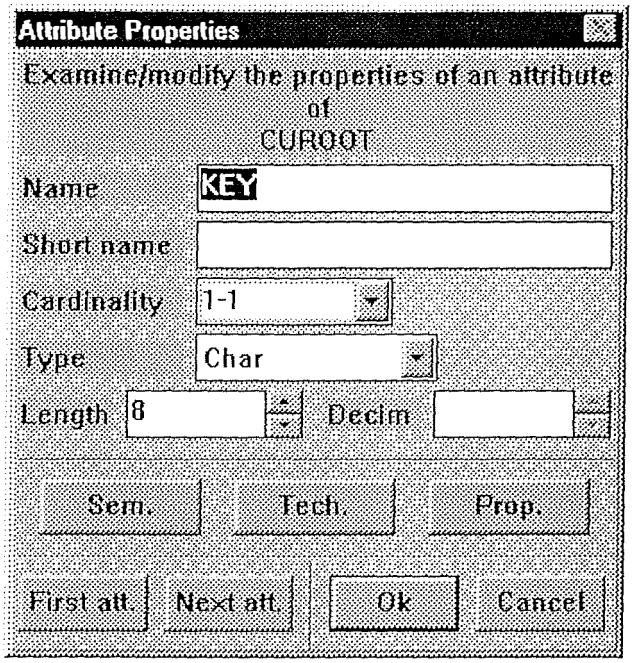
Le caractère | indique la possibilité d'un choix parmi les segments..

<regular_exp>:/g"a regular expression"

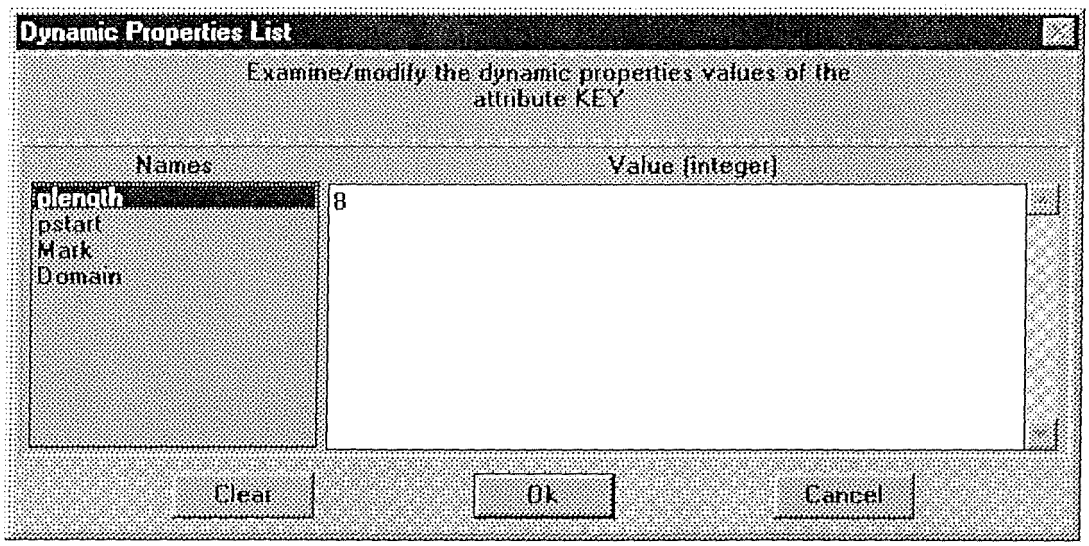
<terminal_seg>: "a string"
/t = tabulation; /n = new line

<pattern_name>: [A-Za-z0-9][A-Za-z0-9]0-29

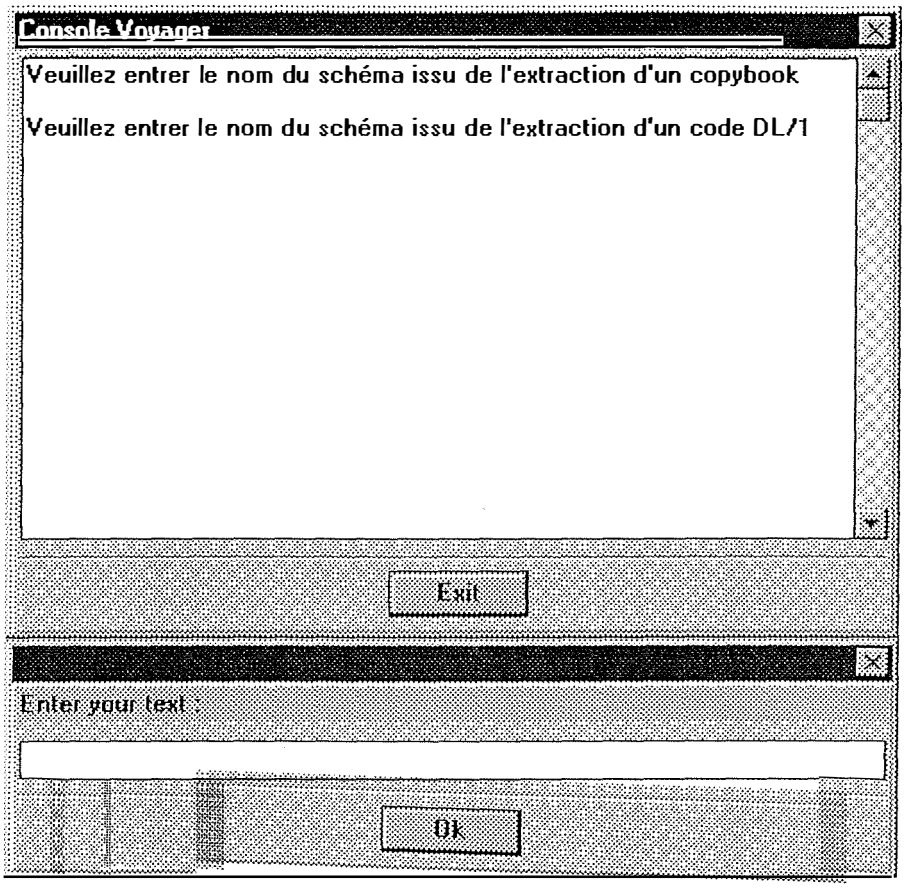
Annexe C: Les fenêtres des applications présentées dans le chapitre treize



Fenêtre 1: Propriétés d'un attribut



Fenêtre 2: Liste des méta-propriétés.



Fenêtre 3: Console Voyager.

DB-MAIN 3.alpha - [client-c.cob/1]

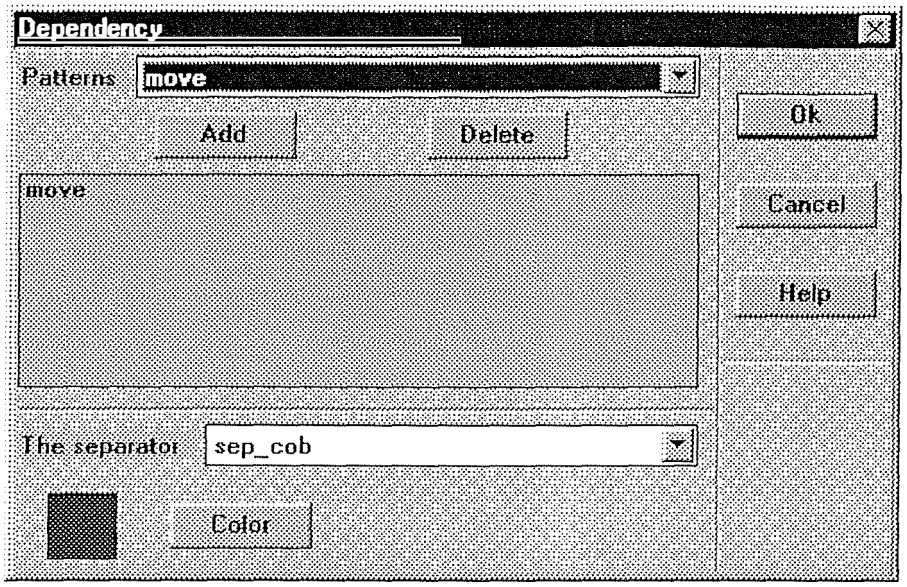
File Edit Product New Transform Assist QuickDB Log View Window

Schema analysis...
Integration
Text analysis
Load pattern
Search
Search next F3
Execute
Dependency
Prg slicing
Goto

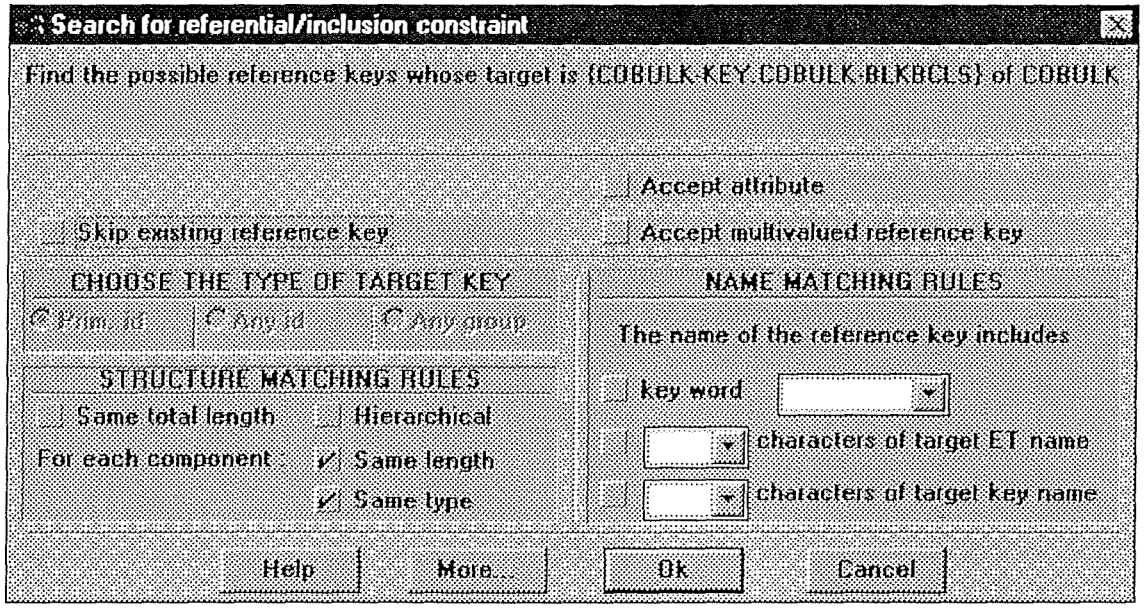
```
1   IDENTIFICATION DIVISION.
2   PROGRAM-ID. COMMANDE.
3
4   ENVIRONMENT DIVISION.
5   INPUT-OUTPUT SECTION.
6   FILE-CONTROL.
7     SELECT CLIENT ASSIGN TO "C
8       ORGANIZATION IS INDEXED
9       ACCESS MODE IS DYNAMIC
10      RECORD KEY IS NCLI.
11
12    SELECT COMMANDE ASSIGN TO "COMMANDE.DAT"
13      ORGANIZATION IS INDEXED
14      ACCESS MODE IS DYNAMIC
15      RECORD KEY IS NCOM.
16
17    SELECT DéTAIL ASSIGN TO "DéTAIL.DAT"
18      ORGANIZATION IS INDEXED
19      ACCESS MODE IS DYNAMIC
20      RECORD KEY IS IDDéTAIL.
21
22    SELECT PRODUIT ASSIGN TO "PRODUIT.DAT"
23      ORGANIZATION IS INDEXED
24      ACCESS MODE IS DYNAMIC
25      RECORD KEY IS NPRO.
26
27  DATA DIVISION.
```

computes the program slice with respect of the current line

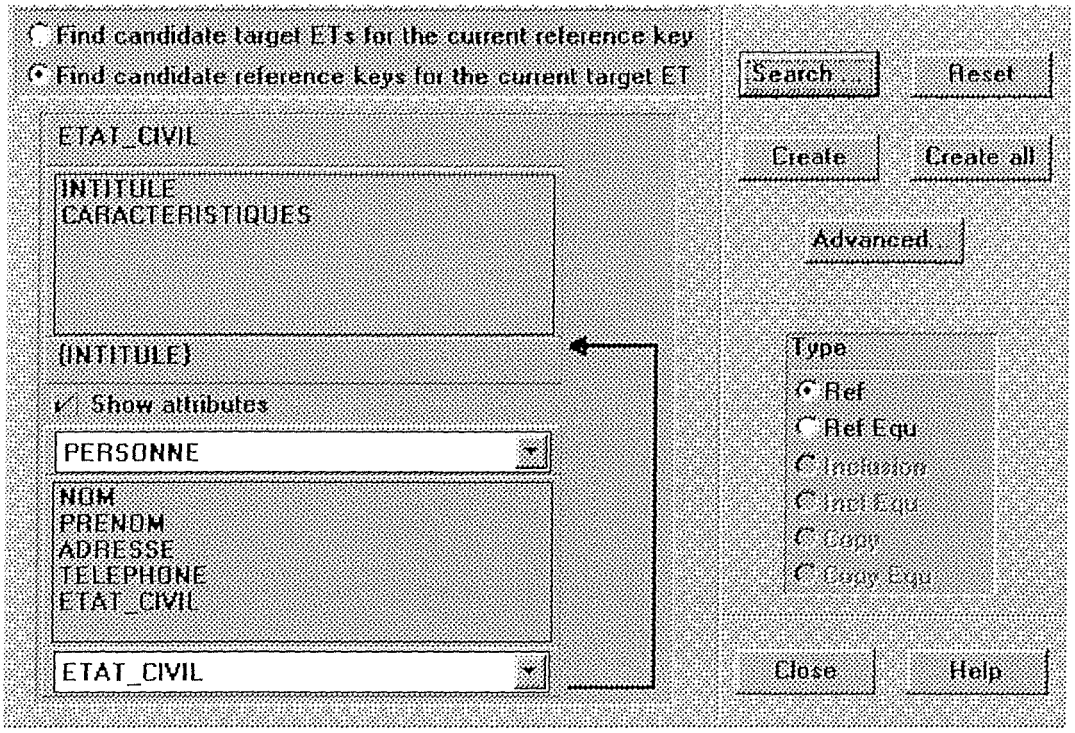
Fenêtre 4



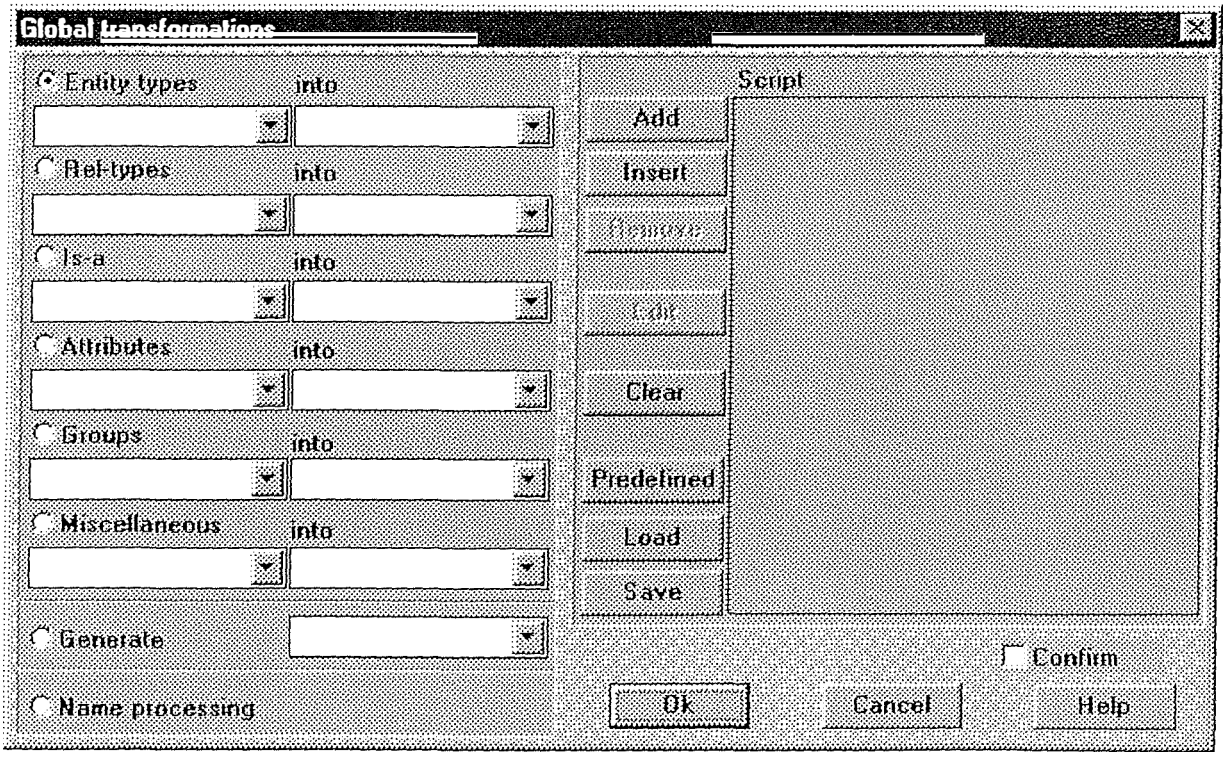
Fenêtre 5: Graphe de dépendance.



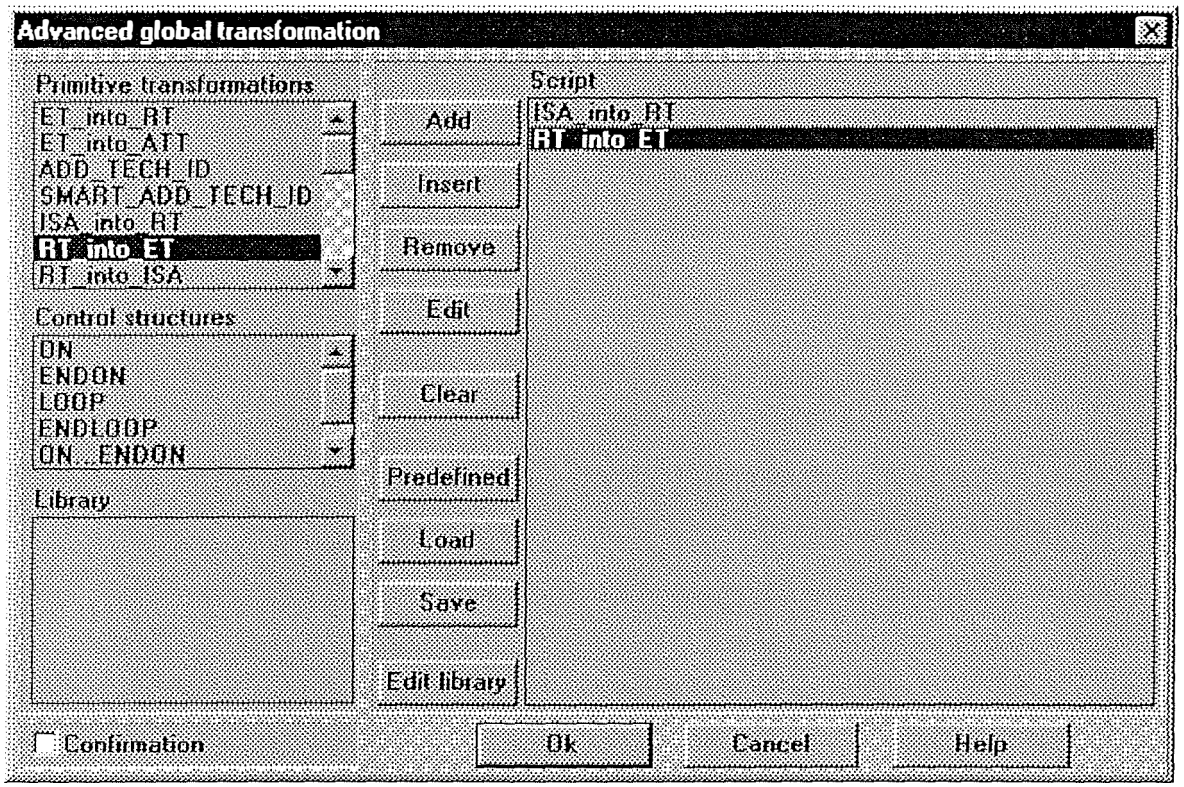
Fenêtre 6: Options de recherche de clés étrangères.



Fenêtre 7: résultat de la recherche.



Fenêtre 8: Boîte de dialogue des transformations globales.



Fenêtre 9: Boîte de dialogue des transformations globales avancées.

Annexe D: Le code du fichier input de *GENE.V2*

```
IDENTIFICATION DIVISION.
    PROGRAM-ID.   RAETUDPB.
      SKIP1
    AUTHOR.          TAIBI.
      SKIP1
    INSTALLATION.  DIETEREN.
      SKIP1
    DATE-WRITTEN.  05/12/96.
      SKIP1
    DATE-COMPILED.
      SKIP1
   ***************************************************************
   *----PROGRAMME DESTINE AUX ETUDIANTS DE NAMUR------------------*
   *-------------------------------------------------------------*
      EJECT
    ENVIRONMENT DIVISION.
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
    DATA DIVISION.
   ****************
    FILE SECTION.

   *
   *
    WORKING-STORAGE SECTION.
   *------------------------*
   *

   *ZONE RECEPTION DU GN

    01  WORK-TOTAL.
      04 ARGUMENT OCCURS 800 TIMES PIC X(01).

   *ZONE RECEPTION DU GU

    01  WK-TOTAL          PIC X(800).

    01  WORK-KEY.
      04 ARGKEY   OCCURS 100 TIMES PIC X(01).

    01  WORK-SEGMENT-GN     PIC X(08)  VALUE 'XXXXXXXX'.
    01  WORK-SEGMENT-GU     PIC X(08)  VALUE 'YYYYYYYY'.

   *
   *
    01  VARIABLE.
   *
      05  INDGN            PIC  9(10).
      05  INDGU            PIC  9(10).
      05  X              PIC S9(04) COMP.
      05  Y              PIC S9(04) COMP.
      05  IND1             PIC S9(04) COMP.
      05  IND2             PIC S9(04) COMP.
```

```
*

 01  SSA-QUALIFIE.
    05  ZONEQ-SEG          PIC X(8).
    05  FILLER             PIC X     VALUE '('.
    05  ZONEQ-KEY-NAME     PIC X(8)  VALUE 'KEY'.
    05  ZONEQ-OPERATOR     PIC XX    VALUE ' ='.
    05  ZONEQ-KEY          PIC X(21).
    05  FILLER             PIC X     VALUE ')'.

 01  SSA-UNQUALIFIE.
    05  ZONEU-SEG          PIC X(8).
    05  FILLER             PIC X     VALUE SPACE.

    EJECT
*********************************************
***   DEFINITION DES CODES FONCTION DL/1  ***
*********************************************
*
 01  DL-FUNCTIONS.
    03  DL-GU          PIC X(4)      VALUE 'GU  '.
    03  DL-GN          PIC X(4)      VALUE 'GN  '.
*

* FIN DE DB
***********

 01  FIN-DE-DB      PIC X.
   88  EOF                VALUE 'Y'.

*
*

    EJECT
 LINKAGE SECTION.
*----------------*

    COPY   DLIPCB.

    EJECT
*
 PROCEDURE DIVISION.
*********************

    COPY   DLITCBL.

    PERFORM INITIALISATION      THRU  EXIT-INITIALISATION.
    PERFORM LECTURE-DB          THRU  EXIT-LECTURE-DB.
    PERFORM TRAITEMENT          THRU  EXIT-TRAITEMENT
                          UNTIL EOF.
    MOVE ZERO  TO RETURN-CODE.
    PERFORM FIN-TRAITEMENT      THRU  EXIT-FIN-TRAITEMENT.
```

```
 INITIALISATION.
****************
*

    MOVE ZERO TO INDGN.
    MOVE ZERO TO INDGU.
    MOVE ZERO TO X.
    MOVE ZERO TO Y.
* INDICE DEBUT
    MOVE %%   TO IND1.
* INDICE FIN
    MOVE ££   TO IND2.

 EXIT-INITIALISATION.
*********************
    EXIT.
*
*******************************************************************
*                  TRAITEMENT                                     *
*******************************************************************

 TRAITEMENT.
************
*
      PERFORM CHARGEMENT      THRU EXIT-CHARGEMENT.

      PERFORM LECTURE-DB      THRU EXIT-LECTURE-DB.
                                        03410000
 EXIT-TRAITEMENT.
*****************
    EXIT.


*******************************************************************
*      TRAITEMENT LECTURE DB                                      *
*******************************************************************

 LECTURE-DB.
************
*

    MOVE  WORK-SEGMENT-GN   TO   ZONEU-SEG.

    CALL 'CBLTDLI' USING DL-GN PCB01 WORK-TOTAL SSA-UNQUALIFIE.

    IF PCB01-STATUS-CODE01 NOT EQUAL SPACES AND 'GA' AND 'GK'
      IF   PCB01-STATUS-CODE01 EQUAL 'GB'
          MOVE 'Y'    TO FIN-DE-DB
      ELSE MOVE 'Y'    TO FIN-DE-DB
          DISPLAY 'DERNIER RECORD = '
          DISPLAY 'STATUS-CODE    = ' , PCB01-STATUS-CODE01
          CALL 'CANCEL'.

 EXIT-LECTURE-DB.
*****************
    EXIT.
```

```
 CHARGEMENT.
************

    PERFORM FILL-KEY THRU EXIT-FILL-KEY.

 EXIT-CHARGEMENT.
*****************
    EXIT.

 FILL-KEY.
**********

    PERFORM TRAIT-TEST-SEG THRU EXIT-TRAIT-TEST-SEG.
    MOVE   WORK-SEGMENT-GU  TO  ZONEQ-SEG.
    MOVE   WORK-KEY       TO  ZONEQ-KEY.
    CALL 'CBLTDLI' USING DL-GU PCB13 WK-TOTAL SSA-QUALIFIE.
      IF  PCB13-STATUS-CODE13 EQUAL SPACES
          ADD  1  TO  INDGU
      ELSE
         IF  PCB13-STATUS-CODE13 EQUAL 'GE'
            DISPLAY 'KEY GU NOT FOUND : ' , WORK-KEY
         ELSE DISPLAY 'DERNIER RECORD = ' , WORK-KEY
            DISPLAY 'STATUS-CODE =' , PCB13-STATUS-CODE13
            CALL 'CANCEL'.

 EXIT-FILL-KEY.
***************
    EXIT.


 TRAIT-TEST-SEG.
****************

    IF PCB01-SEG-NAME01 NOT = WORK-SEGMENT-GN
       CALL  'CANCEL'.

       ADD  1     TO  INDGN.
       MOVE ZERO  TO  Y.
       PERFORM TRAIT-POS  THRU  EXIT-TRAIT-POS
                  VARYING X
                  FROM IND1 BY 1
                  UNTIL X > IND2.

 EXIT-TRAIT-TEST-SEG.
*********************
    EXIT.

 TRAIT-POS.
***********

    ADD 1  TO Y .
    MOVE   ARGUMENT (X)  TO  ARGKEY (Y).

 EXIT-TRAIT-POS.
***************
    EXIT.
```

```
*****************************************************************
*      TRAITEMENT FIN DU TRAVAIL                                *
*****************************************************************

FIN-TRAITEMENT.
****************

      DISPLAY 'COMPTEUR GN : ' , INDGN.
      DISPLAY 'COMPTEUR GU : ' , INDGU.

      DISPLAY 'FIN-TRAITEMENT'.
      GOBACK.

    EXIT-FIN-TRAITEMENT.
   **********************
      EXIT.
```

Annexe E: Les codes sources de l'étude de cas

Database Description CORDDB

```
1          DBD   NAME=CORDDB,
2              ACCESS=HDAM,RMNAME=(DFSHDC40,3,20)
3          DATASET DD1=CORDDBV,
4              DEVICE=3380,SIZE=2048
5          SEGM  NAME=COROOT,
6              BYTES=265,
7              PTR=T,
8              PARENT=((0))
9          FIELD NAME=(KEY,SEQ,U),
10             BYTES=9,
11             START=1
12         FIELD NAME=CUSTNO,
13             BYTES=8,
14             START=10
15         FIELD NAME=ORDTYPE,
16             BYTES=2,
17             START=18
18         FIELD NAME=ORDDATE,
19             BYTES=4,
20             START=64
21         FIELD NAME=SECORD,
22             BYTES=7,
23             START=252
24         FIELD NAME=SECDATE,
25             BYTES=7,
26             START=164
27         FIELD NAME=QPRTKEY,
28             BYTES=15,
29             START=198
30         FIELD NAME=HOLDKEY,
31             BYTES=12,
32             START=230
33         FIELD NAME=/SXROOT
34         LCHILD NAME=(COSEC1,CORDS1),
35             POINTER=INDX
36         XDFLD NAME=CUSTNO2,
37             SRCH=(CUSTNO,SECDATE,SECORD
38         LCHILD NAME=(COSEC3,CORDS3),
39             POINTER=INDX
40         XDFLD NAME=SECKEY3,
41             SRCH=(QPRTKEY),
42             NULLVAL=X'FF'
43         LCHILD NAME=(COSEC4,CORDS4),
44             POINTER=INDX
45         XDFLD NAME=SECKEY4,
46             SRCH=(HOLDKEY),
47             SUBSEQ=KEY,
48             NULLVAL=X'FF'
49         SEGM  NAME=COBORD,
50             BYTES=159,
51             PTR=(TB),
52             PARENT=((COROOT,SNGL))
53         FIELD NAME=(KEY,SEQ,U),
```

```
54          BYTES=14,
55          START=1
56      FIELD NAME=ALLDATE,
57          BYTES=7,
58          START=6
59      FIELD NAME=SECKEY,
60          BYTES=27,
61          START=15
62      FIELD NAME=/SXBORD
63      SPACE 1
64      LCHILD NAME=(COSEC2,CORDS2),
65          POINTER=INDX
66      XDFLD NAME=SECKEY2,
67          SEGMENT=COBORD,
68          SRCH=SECKEY,
69          NULLVAL=X'FF',
70          SUBSEQ=(ALLDATE,/SXBORD)
71      SEGM  NAME=COSPEC,
72          BYTES=205,
73          PTR=T,
74          RULES=(,LAST),
75          PARENT=((COROOT,DBLE))
76      FIELD NAME=RECCODE,
77          BYTES=1,
78          START=1
79      SEGM  NAME=COLINE,
80          BYTES=635,
81          RULES=(,LAST),
82          PTR=T,
83          PARENT=((COROOT,DBLE))
84      FIELD NAME=LSEQNO,
85          BYTES=5,
86          START=1
87      SEGM  NAME=COSCHD,
88          BYTES=80,
89          PTR=NT,
90          PARENT=((COROOT,SNGL))
91      SEGM  NAME=COPART,
92          BYTES=120,
93          PTR=TB,
94          PARENT=((COSCHD,SNGL))
95      FIELD NAME=(KEY,SEQ,U),
96          BYTES=21,
97          START=1
98      SEGM  NAME=COREQS,
99          BYTES=50,
100         PTR=TB,
101         PARENT=((COPART,SNGL))
102     FIELD NAME=(KEY,SEQ,U),
103         BYTES=9,
104         START=1
105     SEGM  NAME=COBULK,
106         BYTES=24,
107         PTR=T,
108         PARENT=((COROOT,SNGL))
109     FIELD NAME=(KEY,SEQ,U),
110         BYTES=5,
111         START=1
```

```
112          DBDGEN
113          END
```

Database Description CUSTDB

```
1            DBD  NAME=CUSTDB,
2                ACCESS=HDAM,RMNAME=(DFSHDC40,5,5000)
3            DATASET DD1=CUSTDBV,
4                DEVICE=3380,SIZE=4096
5            SEGM  NAME=CUROOT,
6                BYTES=412,
7                PTR=T,
8                COMPRTN=INFCMPPF,
9                PARENT=((0))
10           FIELD NAME=(KEY,SEQ,U),
11               BYTES=8,
12               START=1
13           FIELD NAME=/SXROOT
14           FIELD NAME=(AINDEX),
15               BYTES=12,
16               START=297
17           LCHILD NAME=(CUSEC1,CUSTS1),
18               POINTER=INDX
19           XDFLD NAME=AINDEX2,
20               NULLVAL=X'40',
21               SUBSEQ=/SXROOT,
22               SRCH=AINDEX
23           SEGM  NAME=CUDELV,
24               BYTES=265,
25               PTR=T,
26               COMPRTN=INFCMPPF,
27               PARENT=((CUROOT,SNGL))
28           FIELD NAME=(KEY,SEQ,U),
29               BYTES=2,
30               START=1
31           SEGM  NAME=CUWHSE,
32               BYTES=20,
33               PTR=T,
34               PARENT=((CUDELV,SNGL))
35           FIELD NAME=(KEY,SEQ,U),
36               BYTES=2,
37               START=1
38           SEGM  NAME=CUTEXT,
39               BYTES=67,
40               PTR=TB,
41               PARENT=((CUROOT,SNGL))
42           FIELD NAME=(KEY,SEQ,U),
43               BYTES=7,
44               START=1
45           SEGM  NAME=CUSTAT,
46               BYTES=100,
47               PTR=T,
48               COMPRTN=INFCMPPF,
49               PARENT=((CUROOT,SNGL))
50           FIELD NAME=(KEY,SEQ,U),
51               BYTES=3,
52               START=1
53           SEGM  NAME=CUSALE,
```

```
54              BYTES=204,
55              PTR=T,
56              COMPRTN=INFCMPPF,
57              PARENT=((CUROOT,SNGL))
58          FIELD NAME=(KEY,SEQ,U),
59              BYTES=9,
60              START=1
61          DBDGEN
62          END
```

COPYBOOKS

COROOT

```
        C O R O O T - CUSTOMER ORDER DB ORDER HEADER SEGMENT

1       05 COROOT-KEY.
2              10 COROOT-MU-ORDNO.
3                     15 COROOT-ORDMU    PIC XX.
4                     15 COROOT-ORDNO    PIC 9(7).
5       05 COROOT-MU-CUSTNO.
6              10 COROOT-CUSTMU      PIC XX.
7              10 COROOT-CUSTNO      PIC X(6).
8       05 COROOT-ORDTYPE       PIC 99.
9       05 COROOT-CUSTORD       PIC X(13).
10      05 COROOT-QUOTDET.
11             10 COROOT-PRTDATE     PIC S9(7) COMP-3.
12             10 COROOT-TOTPRCE     PIC S9(13) COMP-3.
13      05 COROOT-USER          PIC X(20).
14      05 COROOT-ORDDATE       PIC S9(7) COMP-3.
15      05 COROOT-EXPDATE REDEFINES COROOT-ORDDATE PIC S9(7) COMP-3.
16      05 COROOT-TIME          PIC S999V99 COMP-3.
17      05 COROOT-ALLDATE       PIC S9(7) COMP-3.
18      05 COROOT-AMDDATE REDEFINES COROOT-ALLDATE PIC S9(7) COMP-3.
19      05 COROOT-PAKTYPE       PIC X.
20      05 COROOT-DELADNO       PIC 99.
21      05 COROOT-SPECADD       PIC X.
22             88 COROOT-SPEC-DEL-ADDR-EXISTS    VALUE 'Y'.
23      05 COROOT-TOTOQTY       PIC S9(7) COMP-3.
24      05 COROOT-TOTCQTY       PIC S9(7) COMP-3.
25      05 COROOT-WH-ADVNO1.
26             10 COROOT-ADVWH1      PIC XX.
27             10 COROOT-ADVNO1      PIC S9(7) COMP-3.
28      05 COROOT-MU-RECOMNO.
29             10 COROOT-RECOMMU     PIC XX.
30             10 COROOT-RECOMNO     PIC S9(7) COMP-3.
31      05 COROOT-RECOMLG.
32             10 COROOT-RECOMLN     PIC S9(5) COMP-3.
33      05 COROOT-DELMETH       PIC X.
34      05 COROOT-ORDSTAT       PIC 9.
35             88 COROOT-NOT-YET-CONFIRMED   VALUE ZERO.
36             88 COROOT-BEING-CONFIRMED     VALUE 1.
37             88 COROOT-CONFIRMED           VALUE 2.
38             88 COROOT-PROCESSED           VALUE 4.
39             88 COROOT-BEING-DELETED       VALUE 8.
```

```
40          88 COROOT-QUOT-NOT-PRINTED   VALUE ZERO.
41          88 COROOT-QUOT-TO-BE-PRINTED VALUE 1.
42          88 COROOT-QUOT-PRINTED       VALUE 2.
43          88 COROOT-QUOT-DELETED       VALUE 9.
44     05 COROOT-SEDSCD        PIC X.
45     05 COROOT-SEDSRT        PIC S999V99 COMP-3.
46     05 COROOT-SNGLSUP       PIC X.
47          88 COROOT-SINGLE-SUPPLY-ORDER   VALUE 'Y'.
48     05 COROOT-FULLSUP       PIC X.
49          88 COROOT-FULL-SUPPLY-ORDER     VALUE 'Y'.
50     05 COROOT-PRCADJ        PIC S9V9999 COMP-3.
51     05 COROOT-EXSCIND       PIC X.
52          88 COROOT-EXPRESS-SURCHARGE     VALUE 'Y'.
53     05 COROOT-ABNIND        PIC X.
54          88 COROOT-ABNORMAL-ORDER        VALUE 'Y'.
55     05 COROOT-PFDATE        PIC S9(7) COMP-3.
56     05 COROOT-DSRTTAB       PIC X.
57     05 COROOT-PARDSCD       PIC 9.
58     05 COROOT-URGSURV       PIC S9(11) COMP-3.
59     05 COROOT-PRCATNO       PIC S999 COMP-3.
60     05 COROOT-SPINSPK       PIC X.
61          88 COROOT-SPEC-INSTR-PACKING    VALUE 'Y'.
62     05 COROOT-SPINSDE       PIC X.
63          88 COROOT-SPEC-INSTR-DESPATCH   VALUE 'Y'.
64     05 COROOT-SPINSIN       PIC X.
65          88 COROOT-SPEC-INSTR-INVOICING  VALUE 'Y'.
66     05 COROOT-SPADCR1       PIC X.
67          88 COROOT-SPEC-CR1-ADDR-EXISTS  VALUE 'Y'.
68     05 COROOT-SPADCR2       PIC X.
69          88 COROOT-SPEC-CR2-ADDR-EXISTS  VALUE 'Y'.
70     05 COROOT-MU-CRCUST1.
71          10 COROOT-CRCUST1-MU   PIC XX.
72          10 COROOT-CRCUST1      PIC X(6).
73     05 COROOT-MU-CRCUST2.
74          10 COROOT-CRCUST2-MU   PIC XX.
75          10 COROOT-CRCUST2      PIC X(6).
76     05 COROOT-CRDELV1       PIC 99.
77     05 COROOT-CRDELV2       PIC 99.
78     05 COROOT-PERCENTAGES.
79          10 COROOT-CRPER1      PIC S9(3)V99 COMP-3.
80          10 COROOT-CRPER2      PIC S9(3)V99 COMP-3.
81     05 COROOT-DISC-CREDIT-PER REDEFINES COROOT-PERCENTAGES.
82          10 COROOT-DISPER1     PIC S9(3)V99 COMP-3.
83          10 COROOT-DISPER2     PIC S9(3)V99 COMP-3.
84     05 COROOT-COMM-CREDIT-PER REDEFINES COROOT-DISC-CREDIT-PER
85          10 COROOT-COMPER1     PIC S9(3)V99 COMP-3.
86          10 COROOT-COMPER2     PIC S9(3)V99 COMP-3.
87     05 COROOT-FINREQ        PIC X.
88          88 COROOT-FINANCIAL-SCREEN-REQ  VALUE 'Y'.
89     05 COROOT-SIREQ         PIC X.
90          88 COROOT-SPEC-INSTR-SCREEN-REQ  VALUE 'Y'.
91     05 COROOT-RECTYPE       PIC X.
92          88 COROOT-ORDER          VALUE SPACE.
93          88 COROOT-QUOTATN        VALUE 'Q'.
94          88 COROOT-MANUAL-ORDER   VALUE 'M'.
95     05 COROOT-BOREQ         PIC X.
96          88 COROOT-BOREQ-ALLOWED  VALUE 'Y'.
97     05 COROOT-SECDATE       PIC 9(7).
```

```
98      05 COROOT-DELROUT        PIC S999 COMP-3.
99      05 COROOT-DROPSEQ        PIC S999 COMP-3.
100     05 COROOT-HIDDIS         PIC X.
101             88 COROOT-NO-HIDDEN-DISCOUNT    VALUE 'N'.
102             88 COROOT-INCLUDE-DISCOUNT      VALUE 'Y'.
103     05 COROOT-DISCRED        PIC 9.
104             88 COROOT-NO-DISCRED          VALUE 0.
105             88 COROOT-1-DISCRED-CUST       VALUE 1.
106             88 COROOT-2-DISCRED-CUST       VALUE 2.
107     05 COROOT-COMCRED        PIC 9.
108             88 COROOT-NO-COMCRED          VALUE 0.
109             88 COROOT-1-COMCRED-CUST       VALUE 1.
110             88 COROOT-2-COMCRED-CUST       VALUE 2.
111     05 COROOT-QUOTYPE        PIC X.
112     05 COROOT-SPINSQT        PIC X.
113             88 COROOT-SPEC-INSTR-QUOTATN    VALUE 'Y'.
114     05 COROOT-ACKREQ         PIC X.
115             88 COROOT-ACK-REQUIRED         VALUE 'Y'.
116     05 COROOT-WH-ADVNS.
117             10 COROOT-ADVNWH       PIC XX.
118             10 COROOT-ADVNS        PIC S9(7) COMP-3.
119     05 COROOT-VATIND         PIC X.
120             88 COROOT-VAT-CHARGEABLE    VALUE 'Y'.
121     05 COROOT-VALUE          PIC S9(13)   COMP-3.
122     05 COROOT-SEACODE        PIC XXX.
123     05 COROOT-QPRTKEY.
124             10 COROOT-QMU         PIC XX.
125             10 COROOT-QCUSTNO     PIC X(6).
126             10 COROOT-QORDNO      PIC 9(7).
127     05 COROOT-CURCODE        PIC X(3).
128     05 COROOT-EXCHRTE        PIC S9(5)V9(4)  COMP-3.
129     05 COROOT-ORDCODE        PIC X.
130     05 COROOT-ALLCODE        PIC X.
131             88 COROOT-ALLOCATE-ONLINE          VALUE '1'.
132             88 COROOT-ALLOCATE-TODAY           VALUE '2'.
133             88 COROOT-ALLOCATE-FOR-DELIVERY    VALUE '3'.
134     05 COROOT-STSTEAL        PIC X.
135             88 COROOT-NO-STEALING              VALUE '0'.
136             88 COROOT-STEAL-NOR                VALUE '1'.
137     05 COROOT-ALLTYPE        PIC X.
138             88 COROOT-URGENT                   VALUE '1'.
139             88 COROOT-NORMAL                   VALUE '2'.
140     05 COROOT-DEFCODE        PIC X.
141             88 COROOT-CURRENT-TERMS            VALUE '0'.
142             88 COROOT-PRICE-AT-ORDER           VALUE '1'.
143             88 COROOT-PRICE-AT-ALLOCATION      VALUE '2'.
144             88 COROOT-PRICE-ON-DELIVERY        VALUE '3'.
145     05 COROOT-BOREL          PIC X.
146             88 COROOT-RELEASE-FOR-DELIVERY     VALUE '0'.
147             88 COROOT-RELEASE-IMMEDIATLY       VALUE '1'.
148             88 COROOT-RELEASE-TODAY            VALUE '2'.
149     05 COROOT-SCHDIND        PIC X.
150             88 COROOT-SCHEDULED-ORDER          VALUE '1'.
151     05 COROOT-WHSESEL        PIC X.
152             88 COROOT-WHOUSE-SELECTION         VALUE '1'.
153     05 COROOT-RATSTOK        PIC X.
154             88 COROOT-RATION-STOCK             VALUE '1'.
155     05 COROOT-HOLDKEY.
```

```
156            88 COROOT-CREDIT-OK            VALUE HIGH-VALUES.
157            10 COROOT-MU-HCUSTNO.
158                   15 COROOT-HCUSTMU PIC XX.
159                   15 COROOT-HCUSTNO PIC X(6).
160            10 COROOT-HORDTYP    PIC XX.
161            10 COROOT-CREDREA    PIC XX.
162     05 COROOT-WIPINCL       PIC X.
163            88 COROOT-ORDER-IN-WIP          VALUE '1'.
164     05 COROOT-CRDCHKD       PIC X.
165            88 COROOT-CREDIT-TO-BE-CHKD     VALUE ' '.
166            88 COROOT-CREDIT-CHKD           VALUE '1'.
167            88 COROOT-CREDIT-OVERRIDDEN     VALUE '2'.
168     05 COROOT-PEGDIND       PIC X.
169            88 COROOT-ORDER-HAS-PEGGED-LINES VALUE '1'.
170     05 COROOT-EXCHQTY       PIC S9(5)V9(4) COMP-3.
171     05 COROOT-PAYTERM       PIC XX.
172     05 COROOT-SECORD        PIC 9(7).
173     05 FILLER               PIC X(7).
```

COBULK

```
        *       C O B U L K - CUSTOMERS ORDERS DB              *
1       05 COBULK-KEY.
2       10 COBULK-BLKBCLS     PIC X(5).
3       05 COBULK-ORDQTY       PIC S9(7) COMP-3.
4       05 COBULK-ADJQTY       PIC S9(7) COMP-3.
5       05 COBULK-PRTIND       PIC X.
6       05 FILLER              PIC X(10).
```

COLINE

```
        * C O L I N E - CUSTOMER ORDERS DB NEW ORDER LINES SEGMENT
1       05 COLINE-LSEQNO       PIC 99999.
2       05 COLINE-REC          OCCURS 6.
3              10 COLINE-SEQNO      PIC 99999.
4              10 COLINE-PART.
5                     15 COLINE-FRAN    PIC X.
6                     15 COLINE-PARTNO  PIC X(20).
7              10 COLINE-RECTYPE     PIC 9.
8                     88 COLINE-VALID-ORDER-LINE     VALUE 1.
9                     88 COLINE-INVALID-ORDER-LINE   VALUE 2.
10                    88 COLINE-LOST-ORDER-LINE      VALUE 3.
11             10 COLINE-ORDQTY      PIC S9(7) COMP-3.
12             10 COLINE-DFSSUPP     PIC X(6).
13             10 COLINE-WHOFORD     PIC XX.
14             10 COLINE-SUPPNO      PIC X(6).
15             10 COLINE-MU-PONO.
16                    15 COLINE-POMU     PIC XX.
17                    15 COLINE-PONO     PIC S9(7) COMP-3.
18             10 COLINE-1-DATA.
19                    15 COLINE-ABNORM   PIC X.
20                           88 COLINE-ABNORMAL-DEMAND   VALUE 'Y'.
21                    15 COLINE-PRIORITY PIC X.
22                           88 COLINE-PRIORITY-ORDER    VALUE 'Y'.
23                    15 COLINE-DISCRTE  PIC S999V99 COMP-3.
```

```
24                  15 COLINE-RETAIL   PIC S9(11) COMP-3.
25                  15 COLINE-FIXPRCE  PIC X.
26                         88 COLINE-FIXED-PRICE-LINE   VALUE 'Y'.
27                  15 COLINE-ISSADJ   PIC X.
28                         88 COLINE-QTY-ORDERED-ADJUSTED  VALUE 'Y'.
29                  15 COLINE-USER     PIC X(20).
30                  15 COLINE-WIPPRCE  PIC S9(13)   COMP-3.
31                  15 COLINE-BLKBCLS  PIC X(5).
32                  15 FILLER          PIC X(3).
33                  15 COLINE-DISCODE  PIC X(2).
34            10 COLINE-2-DATA     REDEFINES COLINE-1-DATA.
35                  15 COLINE-COMMENT  PIC X(40).
36                  15 FILLER          PIC X(10).
37            10 COLINE-MINOQTY     PIC S9(7) COMP-3.
```

COPART

```
      * C O P A R T - SCHEDULE PART DETAILS
1     05 COPART-KEY.
2           10 COPART-FRAN       PIC X.
3           10 COPART-PARTNO     PIC X(20).
4     05 COPART-KNOWIND        PIC X.
5           88 COPART-KNOWN-DEMAND    VALUE 'Y'.
6     05 COPART-ORDQTY         PIC S9(7) COMP-3.
7     05 COPART-CONTQTY        PIC S9(7) COMP-3.
8     05 COPART-TUPQTY         PIC S9(7) COMP-3.
9     05 COPART-SUPQTY         PIC S9(7) COMP-3.
10    05 COPART-FCODATE        PIC S9(7) COMP-3.
11    05 COPART-LCODATE        PIC S9(7) COMP-3.
12    05 COPART-DATEAMD        PIC S9(7) COMP-3.
13    05 COPART-RETAIL         PIC S9(11) COMP-3.
14    05 COPART-DISCRTE        PIC S9(7)V99 COMP-3.
15    05 COPART-FIXPRCE        PIC X.
16    05 COPART-USER           PIC X(20).
17    05 COPART-SEQNO          PIC 9(5).
18    05 COPART-ACKREQ         PIC X.
19          88 COPART-ACK-REQUIRED        VALUE 'Y'.
20    05 COPART-WH-ADVNS.
21          10 COPART-ADVWH      PIC XX.
22          10 COPART-ADVNS      PIC S9(7) COMP-3.
23    05 COPART-DFSSUPP        PIC X(6).
24    05 COPART-BLKBCLS        PIC X(5).
25    05 FILLER                PIC X(15).
```

COREQS

```
      * C O R E Q S - CALL OFF REQUIREMENTS
1     05 COREQS-KEY.
2           10 COREQS-ALLDATE    PIC 9(7).
3           10 COREQS-DELADNO    PIC 99.
4     05 COREQS-ORDQTY         PIC S9(7) COMP-3.
5     05 COREQS-DELMETH        PIC X.
6     05 COREQS-SEACODE        PIC XXX.
7     05 COREQS-WIPINCL        PIC X.
```

```
8                88 COREQS-LINE-IN-WIP        VALUE '1'.
9        05 COREQS-WIPPRCE        PIC S9(13) COMP-3.
10       05 COREQS-WHOFORD        PIC XX.
11       05 COREQS-SUPPNO         PIC X(6).
12       05 COREQS-MU-PONO.
13               10 COREQS-POMU        PIC XX.
14               10 COREQS-PONO        PIC S9(7) COMP-3.
15       05 FILLER                PIC X(11).
```

COSCHD

```
*  C O S C H D - CUSTOMER ORDER SCHEDULES HEADER SEGMENT
1        05 COSCHD-CONTIND        PIC X.
2                88 COSCHD-CONTRACTED-IN        VALUE 'Y'.
3        05 COSCHD-CFRDATE        PIC S9(7) COMP-3.
4        05 COSCHD-CTODATE        PIC S9(7) COMP-3.
5        05 COSCHD-SFRDATE        PIC S9(7) COMP-3.
6        05 COSCHD-STODATE        PIC S9(7) COMP-3.
7        05 COSCHD-CALLNO         PIC S9(3) COMP-3.
8        05 COSCHD-CALLFRQ        PIC S9(3) COMP-3.
9        05 COSCHD-CALLIND        PIC X.
10       05 COSCHD-STATUS         PIC X.
11               88 COSCHD-CANCELLED        VALUE '9'.
12       05 COSCHD-DATEAMD        PIC S9(7) COMP-3.
13       05 COSCHD-KNOWIND        PIC X.
14               88 COSCHD-KNOWN-DEMAND        VALUE 'Y'.
15       05 COSCHD-SEQNO          PIC 9(5).
16       05 FILLER                PIC X(47).
```

COSPEC1

```
         *  C O S P E C - KEY 1  CUSTOMER ORDERS DB
1        05 COSPEC-1-KEY          PIC 9.
2                88 COSPEC-DELV-ADDR-SEGMENT     VALUE 1."
3        05 COSPEC-DELNAME        PIC X(30).
4        05 COSPEC-DELADDR        PIC X(30) OCCURS 5.
5        05 COSPEC1-ZIP           PIC X(10).
6        05 COSPEC1-STATE         PIC X(04).
7        05 FILLER                PIC X(10).
```

COSPEC2

```
         *  C O S P E C - KEY 2  CUSTOMER ORDERS DB
1        05 COSPEC-2-KEY          PIC 9.
2                88 COSPEC-PACK-INSTR-SEGMENT     VALUE 2."
3        05 COSPEC-PACKINS        PIC X(30) OCCURS 5.
4        05 FILLER                PIC X(54).
```

COSPEC3

```
         *  C O S P E C - KEY 3  CUSTOMER ORDERS DB
1        05 COSPEC-3-KEY          PIC 9.
2                88 COSPEC-DESP-INSTR-SEGMENT     VALUE 3."
```

```
3       05 COSPEC-DESPINS        PIC X(30) OCCURS 5.
4       05 FILLER                PIC X(54).
```

COSPEC4

```
        *  C O S P E C  -  KEY 4  CUSTOMER ORDERS DB
1       05 COSPEC-4-KEY          PIC 9.
2               88 COSPEC-INV-INSTR-SEGMENT     VALUE 4."
3       05 COSPEC-INVINS          PIC X(30) OCCURS 5.
4       05 FILLER                PIC X(54).
```

COSPEC5

```
        *  C O S P E C  -  KEY 5  CUSTOMER ORDERS DB
1       05 COSPEC-5-KEY          PIC 9.
2               88 COSPEC-CR1-SPEC-ADDR-SEGMENT  VALUE 5."
3       05 COSPEC-CR1NAME         PIC X(30).
4       05 COSPEC-CR1ADDR         PIC X(30) OCCURS 5.
5       05 COSPEC5-ZIP           PIC X(10).
6       05 COSPEC5-STATE          PIC X(04).
7       05 FILLER                PIC X(10).
```

COSPEC6

```
        *  C O S P E C  -  KEY 6  CUSTOMER ORDERS DB
1       05 COSPEC-6-KEY          PIC 9.
2               88 COSPEC-CR2-SPEC-ADDR-SEGMENT  VALUE 6."
3       05 COSPEC-CR2NAME         PIC X(30).
4       05 COSPEC-CR2ADDR         PIC X(30) OCCURS 5.
5       05 COSPEC6-ZIP           PIC X(10).
6       05 COSPEC6-STATE          PIC X(04).
7       05 FILLER                PIC X(10).
```

COSPEC7

```
        *  C O S P E C  -  KEY 7  CUSTOMER ORDERS DB
1       05 COSPEC-7-KEY          PIC 9.
2               88 COSPEC-QUOT-COMMENT-SEGMENT   VALUE 7.
3       05 COSPEC-QUOT-COMMENT    PIC X(30) OCCURS 4.
4       05 COSPEC-QUOT-PACKPER    PIC S9(3)V99 COMP-3.
5       05 COSPEC-QUOT-FRTPER     PIC S9(3)V99 COMP-3.
6       05 FILLER                PIC X(78).
```

CUROOT

```
        *  C U R O O T  -  CUSTOMER DB ROOT SEGMENT     *000300
1       05 CUROOT-KEY.
2               10 CUROOT-MU-CUSTNO.
3                       15 CUROOT-CUSTMU   PIC XX.
4                       15 CUROOT-CUSTNO   PIC X(6).
5       05 CUROOT-INVNAME         PIC X(30).
```

```
6     05 CUROOT-INVADDR      PIC X(30) OCCURS 5.
7     05 CUROOT-INVTELE      PIC X(20).
8     05 CUROOT-INVFAX       PIC X(20).
9     05 CUROOT-INVTLX       PIC X(10).
10    05 CUROOT-ZONE         PIC XX.
11    05 CUROOT-STOP         PIC X.
12    05 CUROOT-OBSOLET      PIC X.
13          88 CUROOT-OBSOLETE          VALUE 'Y'.
14    05 CUROOT-PRCATNO      PIC S999     COMP-3.
15    05 CUROOT-BUSNCAT      PIC X.
16    05 CUROOT-DSRTTAB      PIC X.
17    05 CUROOT-PARDSCD      PIC 9.
18    05 CUROOT-SEDSCD       PIC X.
19    05 CUROOT-SEDSRT       PIC S999V99    COMP-3.
20    05 CUROOT-PRCADJ       PIC S9V9999    COMP-3.
21    05 CUROOT-HOMEXPC      PIC X.
22    05 CUROOT-INVVETI      PIC X.
23          88 CUROOT-INVOICE-VET-REQ          VALUE 'Y'.
24    05 CUROOT-LANGCDE      PIC XX.
25    05 CUROOT-SOUREQI      PIC X.
26          88 CUROOT-SOURCE-DESC-REQ          VALUE 'Y'.
27    05 CUROOT-INVCOPY      PIC S999     COMP-3.
28    05 CUROOT-PACKREQ      PIC X.
29          88 CUROOT-PACK-NOTE-REQ            VALUE 'Y'.
30    05 CUROOT-DESPREQ      PIC X.
31          88 CUROOT-DESP-NOTE-REQ            VALUE 'Y'.
32    05 CUROOT-INVCOMB      PIC X.
33          88 CUROOT-COMBINED-INV-REQ         VALUE 'Y'.
34    05 CUROOT-PACKTYP      PIC X.
35    05 CUROOT-FULLSUP      PIC X.
36          88 CUROOT-FULL-SUPP-CUST           VALUE 'Y'.
37    05 CUROOT-SNGLSUP      PIC X.
38          88 CUROOT-SINGLE-SUPP-CUST         VALUE 'Y'.
39    05 CUROOT-ABNMSUP      PIC X.
40          88 CUROOT-ABNORM-SUPP-CUST         VALUE 'Y'.
41    05 CUROOT-PACKCHG      PIC S999V99    COMP-3
42    05 CUROOT-URGSURC      PIC X.
43          88 CUROOT-SURCHARGE-APPLIES        VALUE 'Y'.
44    05 CUROOT-DATEAMD      PIC S9(7)    COMP-3.
45    05 CUROOT-COMMENT      PIC X(20).
46    05 CUROOT-MU-AINDEX.
47          10 CUROOT-AINDEX-MU    PIC XX.
48          10 CUROOT-AINDEX       PIC X(10).
49    05 CUROOT-CUSTINF      PIC X.
50          88 CUROOT-NO-CUST-INFO-REQ     VALUE SPACE.
51          88 CUROOT-PRINT-PARTNO-ONLY    VALUE '1'.
52          88 CUROOT-PRINT-BINLOC-ONLY    VALUE '2'.
53          88 CUROOT-PRINT-CPARTNO-BINLOC VALUE '3'.
54    05 CUROOT-BOREQ        PIC X.
55    05 CUROOT-HIDDIS       PIC X.
56          88 CUROOT-NO-HIDDEN-DISCOUNT   VALUE 'N'.
57          88 CUROOT-INCLUDE-DISCOUNT     VALUE 'Y'.
58    05 CUROOT-DISCRED      PIC 9.
59          88 CUROOT-NO-DISCRED           VALUE 0.
60          88 CUROOT-1-DISCRED-CUST       VALUE 1.
61          88 CUROOT-2-DISCRED-CUST       VALUE 2.
62    05 CUROOT-COMCRED      PIC 9.
63          88 CUROOT-NO-COMCRED           VALUE 0.
```

```
64              88 CUROOT-1-COMCRED-CUST        VALUE 1.
65              88 CUROOT-2-COMCRED-CUST        VALUE 2.
66      05 CUROOT-CTRYCDE        PIC X(3).
67      05 CUROOT-MU-CREDIT.
68              10 CUROOT-CREDIT-MU    PIC XX.
69              10 CUROOT-CREDIT       PIC S9(7) COMP-3.
70      05 CUROOT-CUSTYPE        PIC XX.
71      05 CUROOT-ACKREQ         PIC X.
72              88 CUROOT-ACK-REQUIRED    VALUE 'Y'.
73      05 CUROOT-CURCODE        PIC X(3).
74      05 CUROOT-CONSIGN        PIC X(4).
75      05 CUROOT-LDGACNO        PIC X(8).
76      05 CUROOT-CCWIP          PIC S9(13) COMP-3.
77      05 CUROOT-OVERWIP        PIC S9(13) COMP-3.
78      05 CUROOT-MU-GRPCUST.
79              10 CUROOT-GRPCUST-MU   PIC XX.
80              10 CUROOT-GRPCUST      PIC X(6).
81      05 CUROOT-GRPCWIP        PIC S9(13) COMP-3.
82      05 CUROOT-GRPOWIP        PIC S9(13) COMP-3.
83      05 CUROOT-PAYTERM        PIC XX.
84      05 CUROOT-ZIP            PIC X(10).
85      05 CUROOT-STATE          PIC X(4).
86      05 CUROOT-RPSLIGNES      PIC 9(3).
87      05 CUROOT-RPSVAL         PIC S9(9) COMP-3.
88      05 CUROOT-RPSSTOP        PIC X.
89              88 CUROOT-RPS-STOPPED            VALUE 'Y'.
90      05 FILLER                PIC X(1).
91      05 CUROOT-TRANSPORT-IND   PIC X.
92              88 CUROOT-TRANSPORT-CHARGED        VALUE 'Y'.
93      05 CUROOT-CURR-FLEET     PIC S9(7) COMP-3.
94      05 CUROOT-LAST-FLEET     PIC S9(7) COMP-3.
95      05 FILLER                PIC X.
```

L'intégration de features pour SMV : Un pas vers la programmation orientée feature

par
Christophe **Evrard**

Mémoire présenté en vue de l'obtention du diplôme
de Maître en Informatique

Année académique 1996 - 1997

Promoteur : Pierre-Yves **Schobbens**
Maître de stage : Mark **Ryan**

Résumé

Nous allons dans ce document explorer ce que pourrait être une approche « orientée feature » de la programmation. Pour ce faire, nous nous baserons sur un langage, SMV, permettant de décrire un certain type de systèmes, des machines à états finis, et de vérifier que ces machines possèdent un certain type de propriétés, celles que l'on peut écrire dans une logique temporelle. Les deux premiers chapitres décriront la technique de la vérification de modèles pour la logique temporelle en temps arborescent. Le troisième chapitre exposera la syntaxe et la sémantique du langage SMV, et sera suivi d'un chapitre définissant la notion de feature en SMV. La cinquième et dernière partie de ce document proposera et critiquera plusieurs définitions de l'interaction entre features.

Abstract

In this thesis, we will explore what could be called a « feature-oriented » approach of programming. In order to do this, we will study the SMV language, which allows to describe a particular kind of system, finite state machines, and to check them against particular properties, those one can write using temporal logic. The first two chapters describe the model checking technique for branching time temporal logic. The third chapter explains the syntax and semantics of the SMV language, and is followed by a chapter defining the feature concept in SMV. The fifth and last part of this paper presents and discusses several definitions of feature interaction.

Avant-propos

Je tiens à remercier ceux qui, de près ou de loin, m'ont aidé à mener à bien cette fastidieuse tâche que fut la réalisation de ce mémoire. Merci à mon promoteur, Pierre-Yves Schobbens, pour son suivi lors de la rédaction de ce mémoire, et ses conseils judicieux. Merci à Mark Ryan pour tout ce qu'il m'a appris lors de mon stage ainsi que pour sa disponibilité face à mes nombreuses questions. Merci à Jean-François Raskin pour ses articles et ses éclaircissements. Enfin, je tiens à remercier également mes proches pour leur patience et leur compréhension, au long de ce travail accaparant.

Introduction

Ce mémoire a été rédigé suite à une recherche menée lors d'un stage effectué en automne 1996 au Département d'Informatique de l'Université de Birmingham, sous la direction du Dr Mark Ryan.

Le langage SMV, *Symbolic Model Verifier*, a été défini au début des années 90 par K.L. McMillan, lors de sa thèse de doctorat à l'Université Carnegie Mellon. SMV permet de modéliser des automates finis, ainsi que leurs spécifications, et de contrôler si ces systèmes vérifient leurs spécifications. La technique mise en œuvre dans SMV est le *Symbolic Model Checking*.

La notion de *feature* est d'abord apparue il y a plus de 15 ans dans les milieux de la téléphonie, et a été développée de plusieurs façons. Une des approches, dans le contexte particulier du langage SMV, est due au Dr Mark Ryan. Sous sa supervision, nous avons approfondi ces idées, en étudiant une manière d'une part de décrire la notion de feature, et d'autre part d'intégrer un feature à un système.

La recherche conduite lors du stage a consisté en une approche opérationnelle de l'intégration de features à des systèmes d'ascenseurs, écrits en SMV par Mark Berry [Be96]. Cette approche nous a permis de trouver une syntaxe raisonnable pour les features, et de définir de façon informelle le rôle d'un intégrateur de features (implémenté par ailleurs, consulter à ce sujet [Byr97]).

Le stage nous a aussi permis d'aborder d'une façon originale la notion d'interaction entre features, *via* une approche basée sur les spécifications.

Chapitre premier

Vérification de modèles*

Les erreurs de logique dans le design de circuits et de protocoles sont un problème important pour les concepteurs de hardware. La technique communément utilisée pour vérifier de tels systèmes est basée sur la simulation, mais elle n'est plus satisfaisante lorsque le nombre d'états du système à vérifier devient important. Malgré une recherche importante sur l'usage de vérificateurs de preuves, ces techniques se sont montrées lentes et se passaient difficilement d'interventions manuelles.

1. Introduction

La technique de vérification de modèles fait partie du champ de la vérification automatique de systèmes formels, et est apparue pour la première fois dans une publication de Edmund Clarke et Allen Emerson [CE81b], en 1981. Selon leur approche, les systèmes sont modélisés comme des systèmes état-transition, et les spécifications à vérifier sont exprimées dans une logique temporelle propositionnelle.

La motivation qui sous-tend cette recherche dans le domaine de la vérification est la demande d'outils qui permettent la conception de systèmes corrects, et ce dès leur première mise en service. En effet, les erreurs de design constatées après la mise en production peuvent être très coûteuses, en termes de coût de correction et d'impact sur les délais. De plus, il n'existe pas de méthode mathématique permettant de vérifier des systèmes complexes, comme des systèmes de cache ou des protocoles de réseaux. Le concepteur devrait premièrement créer une quantité de tests suffisante pour mettre en évidence les erreurs de logique, puis en second lieu interpréter les données résultantes pour décider de la correction de son système. Et ce, souvent en l'absence de spécifications précises.

---

* Rédigé d'après [McMi92]

Les techniques de vérification formelle appréhendent ces deux aspects, d'une part en tenant compte de tous les comportements possibles du système, et d'autre part en permettant de décrire les spécifications dans un langage formel.

« Vérification formelle » signifie avoir un modèle mathématique d'un système, un langage pour spécifier ses propriétés souhaitées, d'une manière non ambiguë, et une méthode de preuves qui vérifie que ces propriétés sont satisfaites. On parle de vérification automatique quand c'est une machine qui se charge principalement de la preuve.

Dans cette approche, la modélisation des systèmes en tant que *machines à états finis* a comme désavantage ce que l'on appelle le problème de l'explosion des états. Il s'agit de la relation exponentielle entre le nombre d'états du système et le nombre de composants déterminant ces états. Ainsi la technique de vérification de modèles s'accommode-t-elle mieux de très grandes machines à états que de systèmes constitués d'un grand nombre de petites machines à états, ou que de systèmes manipulant des données. Ceci fait que cette technique s'adresse à des systèmes appelés systèmes réactifs, en ce sens qu'ils reçoivent des inputs et produisent des outputs en une interaction continue avec leur environnement, plutôt qu'à des programmes qui produisent un résultat puis s'arrêtent.

Pour éviter ce problème d'explosion, il est possible de représenter certains types de structures par une formule booléenne, et ce dans le but de pouvoir utiliser de puissants algorithmes manipulant des algèbres booléennes[1], combinés avec les algorithmes de point fixe de la vérification de modèles. Ceci donne lieu à la vérification symbolique de modèles, qui permet de considérer des systèmes difficiles à manipuler avec de purs démonstrateurs de théorèmes ou via l'approche classique de la vérification.

2. Logique temporelle

La logique temporelle utilise les opérateurs de la logique propositionnelle, augmentés d'opérateurs temporels. Ils sont utilisés pour décrire comment des situations changent dans le temps.

Dans cette logique, la formule Fq est vraie au temps présent si q est vrai à un moment dans le futur. De la même façon, la formule Pq est vraie au temps présent si q est vrai à un moment dans le passé. Le dual de l'opérateur F est G, et Gq est équivalent à $\neg F \neg q$, qui signifie que q est vrai à tout moment du futur. De même, le dual de P est H, et Hq est équivalent à $\neg P \neg q$, qui signifie que q est vrai à tout moment dans le passé.

Quand on travaille dans une logique formelle, on souhaite avoir une *théorie des modèles* pour cette logique, qui nous indique l'ensemble des univers imaginaires dans lesquels une formule donnée est vraie. En logique temporelle, cette sémantique est appelée la *sémantique des mondes possibles.* Une *structure* dans cette sémantique consiste en une classe S d'états entre lesquels le système évolue, et une relation $<$ représentant un ordre temporel. Intuitivement, si s et t sont deux états, $s < t$ signifie que s se produit avant t. Un *modèle* est une *structure* avec une valuation L, qui attribue pour chaque état une valeur de vérité à chaque proposition atomique. Ces structures sont souvent appelées structure et modèle de Kripke.

[1] Algorithmes basés sur les Binary Decision Diagrams, consulter à ce sujet [Bry86]

La valeur de vérité d'une formule temporelle est relative à l'état présent. Par exemple, la formule Fq est vraie dans l'état s si et seulement s'il existe un état t tel que q est vrai dans cet état, et si $s < t$. Notons qu'une formule temporelle est une phrase ouverte, avec un paramètre libre s représentant l'état courant. Une formule décrit donc une classe d'états dans lesquels elle est vraie. De même, un état définit une classe de formules qui sont vraies dans cet état.

Etant donnée cette sémantique des modèles, le choix des axiomes dans la logique caractérise efficacement la relation d'ordre temporel <. Par exemple, les axiomes suivants, ajoutés aux tautologies propositionnelles, sont vrais pour les structures dont la relation < est un ordre partiel[2] :

$$G(p \Rightarrow q) \Rightarrow (Gp \Rightarrow Gq) \qquad (1.1)$$
$$H(p \Rightarrow q) \Rightarrow (Hp \Rightarrow Hq) \qquad (1.2)$$
$$p \Rightarrow GPp \qquad (1.3)$$
$$p \Rightarrow HFp \qquad (1.4)$$

Une nouvelle règle d'inférence est nécessaire : la *généralisation temporelle*, exprimant que les tautologies doivent être vraies à tout moment, ou encore, que si α est prouvable, alors on peut inférer $G\alpha$ et $H\alpha$.

En spécialisant ce système, on peut obtenir des logiques caractérisant plusieurs modèles du temps, par exemple temps linéaire, temps discret, et temps arborescent.

2.1. Temps linéaire

Dans ce modèle de logique temporelle, le temps est considéré comme étant ordonné linéairement, et mesuré par des réels ou des entiers. Une structure est linéaire quand la relation d'ordre temporel est totale, càd quand pour tout état s, t, on a $s < t$, $s = t$ ou $s > t$. Pour caractériser ce type de structure, on ajoute les axiomes suivants :

$$(FPq) \Rightarrow (Pq \vee q \vee Fq) \qquad (1.5)$$
$$(PFq) \Rightarrow (Pq \vee q \vee Fq) \qquad (1.6)$$

On étend généralement cette logique avec les opérateurs *until* et *since*. Intuitivement, $p\mathcal{U}q$ signifie que q sera vrai à un moment du futur, et p sera vrai à tout moment jusqu'à l'instant où q sera vrai. $p\mathcal{S}q$, de manière similaire, signifie que q a été vrai à un moment du passé, et que p a été vrai dès l'instant où q n'était plus vrai. De façon plus précise, $p\mathcal{U}q$ est vrai à l'état s s'il existe un état t tel que $s < t$, et que q est vrai à l'état t, et que pour tout $s < u < t$, p est vrai à l'état u. De façon analogue, $p\mathcal{S}q$ est vrai à l'état s s'il existe un état t tel que $t < s$, et que q est vrai à l'état t, et que pour tout $t < u < s$, p est vrai à l'état u.

[2] càd transitive et antisymétrique

2.2. Temps discret

Une structure discrète est caractérisée par le fait que chaque état a un prédécesseur et un successeur immédiats. Une structure linéaire et discrète peut être caractérisée en ajoutant les deux axiomes suivants à ceux de la logique en temps linéaire :

$$p \wedge Hp \Rightarrow FHp \qquad (1.7)$$
$$p \wedge Gp \Rightarrow PGp \qquad (1.8)$$

Il est utile dans cette logique de définir un opérateur temporel *next*. La formule Xq est vraie à l'état s quand il existe un successeur immédiat de s pour lequel q est vrai. L'état t est un successeur immédiat de s s'il n'existe pas d'état u tel que s < u < t. Xq est donc exactement équivalent à false$\mathcal{U}$q, donc l'ajout de cet opérateur n'augmente pas l'expressivité de la logique.

2.3. Temps arborescent

Une structure arborescente est telle que la relation d'ordre temporel définit un arbre qui branche vers le futur. Chaque instant a donc un passé unique, mais plusieurs futurs possibles. Ce modèle non déterministe du temps est donc par exemple bien approprié à la définition de la sémantique de programmes non déterministes. Une structure est arborescente quand pour tout état s, t, u, si t < s et u < s alors t < u ou t = u ou t > u. Autrement dit, le passé de chaque état est ordonné linéairement. Ces cadres peuvent être caractérisés en supprimant simplement l'axiome (1.6) de la logique linéaire.

On voudrait interpréter la structure branchante de la façon suivante : chaque instant a plusieurs futurs possibles, et avec le temps qui passe, ces possibilités se réduisent. Il a donc existé, dans le passé, des futurs possibles qui ne sont plus envisageables. Cette interprétation mène aux notions de nécessité (inévitabilité) et de possibilité en logique temporelle. Intuitivement, la valeur de vérité des formules est relative à une certaine branche de l'arborescence, qui est une évolution possible du temps dans le futur. Une branche est définie comme un ensemble maximal d'états ordonné linéairement. On écrira q[s,b] si q est vrai dans l'état s de la branche b. On a Fq[s,b] si et seulement s'il existe un état t de b tel que s < t et q[t,b]. De façon analogue, on a Pq[s,b] si et seulement s'il existe un état t de b tel que t < s et q[t,b].

La notion « q est nécessairement vrai » est représentée par Aq. On dira que Aq[s,b] si et seulement si pour toutes les branches b′ contenant s, on a q[s,b′]. La notion « q peut être vrai » est représentée par la formule Eq. On dira que Eq[s,b] si et seulement s'il existe une branche b′ contenant s, telle que l'on ait q[s,b′]. Les opérateurs A et E sont une sorte de quantification de second ordre sur les sous-ensembles maximaux ordonnés linéairement. Ces opérateurs sont aussi représentés, dans la littérature, sous leur forme classique respective : $\Box$ et $\Diamond$.

Selon cette sémantique de la logique arborescente, il pourrait exister des possibilités dans le passé qui ne sont plus envisageables dans le présent. Par exemple, q $\Rightarrow$ HAFq n'est pas valide, en effet, le fait de q dans le présent n'implique pas la nécessité de q dans le passé. Cette logique pourrait dès lors être qualifiée de logique du regret. La logique peut aussi exprimer des propriétés sémantiques utiles de programmes non déterministes. Par exemple, si q représente le fait qu'un programme se termine, alors AFq exprime la

terminaison inévitable, et Efq la terminaison possible. Notons que Pq, APq, EPq sont logiquement équivalents, car le passé d'un état est identique pour toute branche. A et E sont duaux, car Aq est équivalent à ¬E¬q

L'interprétation de formules sur des sous-ensembles maximaux ordonnés linéairement permet de caractériser les structures arborescentes d'une autre façon : aux axiomes de la logique temporelle linéaire, on ajoute ceux-ci :

$$Ep \Rightarrow HEFp \qquad (1.9)$$

$$PAGp \Rightarrow Ap \qquad (1.10)$$

3. La logique temporelle CTL[3]

Cette logique est un sous-ensemble assez simple de la logique modale temporelle définie par Clarke et Emerson [CE81b]. Les opérateurs de cette logique sont caractérisés de façon simple comme des points fixes, caractérisation qui permet de calculer de façon efficace si une formule est satisfaite dans un certain état d'un certain modèle fini.

En CTL, les opérateurs temporels apparaissent par paires, premièrement un opérateur arborescent A ou E, suivi d'un opérateur temporel F, G, U ou X. Les opérateurs faisant référence au passé ne sont pas admis. Précisément :

1. Chaque proposition atomique est une formule CTL,
2. Si f et g sont deux formules CTL, il en va de même pour
   $\neg f$, $(f \wedge g)$, AXf, EXf, $A(f\,\mathcal{U}g)$, $E(f\,\mathcal{U}g)$

Notons qu'il est nécessaire de citer ici l'opérateur temporel X, car comme en CTL le futur contient le présent, il n'est plus possible de définir Xp avec $false\,\mathcal{U}g$.

Les autres opérateurs sont dérivés de ceux ci-dessus en suivant les règles suivantes :

$$f \vee g = \neg(\neg f \wedge \neg g)$$
$$AFg = A(true\,\mathcal{U}g)$$
$$EFg = E(true\,\mathcal{U}g)$$
$$AGf = \neg E(true\,\mathcal{U}\neg f)$$
$$EGf = \neg A(true\,\mathcal{U}\neg f)$$

Notons que comme tous les opérateurs sont préfixés par A ou E, la valeur de vérité d'une formule dépend seulement de l'état donné s, et pas de la branche particulière b.

Dans le but de vérifier des systèmes, on voudrait déterminer la valeur de vérité d'une formule de cette logique par rapport à un certain état d'un certain modèle fini. Ceci signifie qu'il faut être capable de représenter des systèmes réactifs, avec des comportement qui ne terminent pas, en utilisant un modèle fini. Pour réaliser cela, un modèle non standard est introduit : il s'agit d'un triplet (S, R, L), où S représente un ensemble d'états, R est une

[3] pour Computation Tree Logic

relation de transition entre ces états, et L est une valuation. La relation de transition est l'ensemble de toutes les paires (s, t) où t est un successeur immédiat de s. Un modèle arborescent standard (càd un *arbre d'exécution*) peut être obtenu en commençant à un certain état et en développant le graphe (S, R) en un arbre infini (si chaque état a au moins un successeur). Il est possible de donner une sémantique des formules CTL relative au modèle non standard, qui est équivalente à la sémantique standard relative à l'arbre infini correspondant[4].

Un *chemin* d'un modèle standard $K = (S, R, L)$ est une séquence infinie d'états $(s_0, s_1, s_2, ...) \in S^\omega$ telle que chaque paire d'états (s_i, s_{i+1}) est un élément de R. Chaque chemin est un sous-ensemble maximal ordonné linéairement, du modèle arborescent développé à partir de s_0.

La notation $K,s \models f$ signifie que la formule f est vraie dans l'état s du modèle K. Cette formule s'écrit $s \models f$ quand le modèle n'est pas ambigu. La sémantique des formules CTL par rapport au modèle (S, R, L) est donnée ci-dessous :

$s \models p$	ssi	$s \in L(p)$, où p est une proposition atomique
$s \models \neg f$	ssi	$s \not\models f$
$s \models f \wedge g$	ssi	$s \models f$ et $s \models g$
$s_0 \models AXf$	ssi	pour tout chemin $(s_0, s_1, ...)$, $s_1 \models f$
$s_0 \models EXf$	ssi	il existe un chemin $(s_0, s_1, ...)$ tel que $s_1 \models f$
$s_0 \models A(f\,\mathcal{U}g)$	ssi	pour tout chemin $(s_0, s_1, ...)$, il existe un i tel que $s_i \models g$ et pour tout $0 <= j < i$, $s_j \models f$
$s_0 \models E(f\,\mathcal{U}g)$	ssi	il existe un chemin $(s_0, s_1, ...)$, il existe un i tel que $s_i \models g$ et pour tout $0 <= j < i$, $s_j \models f$

4. Points fixes

Emerson et Clarke [CE81a] ont montré que diverses propriétés arborescentes de programmes peuvent être caractérisées comme des points fixes de fonctions monotones appropriées. Ils ont par après introduit la logique CTL, et montré que ses opérateurs peuvent être caractérisés d'une manière identique [CE81b]. Cette caractérisation a conduit à un algorithme plus efficace pour le problème de la vérification.

Pour obtenir cette caractérisation en termes de points fixes, rappelons d'abord quelques points concernant les algèbres booléennes. Identifions chaque formule logique avec son sous-ensemble caractéristique : le sous-ensemble de l'ensemble S où elle est vraie. La constante `false` est identifiée par l'ensemble vide, et `true` est identifiée par S. La formule $p \wedge q$ est identifiée par l'intersection de p et de q, $p \vee q$ est identifiée par l'union de p et q, et $\neg p$ par le complément de p dans S. Il n'est pas difficile de montrer que cette représentation des formules satisfait les axiomes des algèbres booléennes.

[4] Avec une différence : en CTL, le futur contient le présent. Donc, si p est vrai au présent, alors Fp est également vrai.

Une fonctionnelle est dénotée par $\lambda y.f$, où f est la formule, et y la variable. Appliquée à un paramètre p, la fonctionnelle fournit f avec y remplacé par p. Par exemple, si $\tau = \lambda y.(x \wedge y)$, alors $\tau(\mathtt{true}) = (x \wedge \mathtt{true}) = x$.

Définition : Un *point fixe* d'une fonctionnelle τ est tout p tel que $\tau(p) = p$.

Par exemple, si $\tau = \lambda y.(x \wedge y)$, alors $x \wedge y$ est un point fixe pour τ, car $\tau(x \wedge y) = x \wedge (x \wedge y) = (x \wedge y)$.

Définition : Une fonctionnelle τ est *monotone* quand $p \subseteq q$ implique $\tau(p) \subseteq \tau(q)$.

Par exemple, $\lambda y.(x \wedge y)$ est monotone, car $p \subseteq q$ implique $x \wedge p \subseteq x \wedge q$.

A contrario, $\lambda y.(x \wedge \neg y)$ ne l'est pas, car bien que $\mathtt{false} \subseteq \mathtt{true}$, on n'a pas $x \wedge \mathtt{true} \subseteq x \wedge \mathtt{false}$.

Les points fixes de fonctionnelles monotones ont quelques propriétés utiles. Tarski [Tar55] a montré qu'une fonctionnelle monotone a un plus petit point fixe, qui est l'intersection de tous les points fixes. Elle a aussi un plus grand point fixe, qui est l'union de tous les points fixes. On note le plus petit et le plus grand point fixe respectivement $\mu y.f$ et $\nu y.f$.

Les deux points fixes extrémaux peuvent être caractérisés comme la limite d'une série obtenue en itérant sur la fonctionnelle, à condition que celle-ci soit continue.

Définition : Une fonctionnelle est *union-continue* quand pour toute suite infinie non décroissante d'ensembles $p_1 \subseteq p_2 \subseteq p_3 \subseteq \ldots$, on a que $\cup_i \tau(p_i) = \tau(\cup_i p_i)$.

Définition : Une fonctionnelle est *intersection-continue* quand pour toute suite infinie non décroissante d'ensembles $p_1 \subseteq p_2 \subseteq p_3 \subseteq \ldots$, on a que $\cap_i \tau(p_i) = \tau(\cap_i p_i)$.

Tarski a montré que si τ est monotone et union-continue, alors le plus petit point fixe de τ est $\cup_i \tau^i(\mathtt{false})$, ou encore l'union de la suite obtenue en itérant sur τ à partir de la valeur $\mathtt{false}$.

Un résultat similaire vaut aussi pour le plus grand point fixe : si τ est monotone et intersection-continue, alors le plus grand point fixe de τ est $\cap_i \tau^i(\mathtt{true})$, ou encore l'intersection de la suite obtenue en itérant sur τ à partir de la valeur $\mathtt{true}$.

Notons que si l'ensemble S des états est fini, alors toute fonctionnelle τ monotone est nécessairement union-continue et intersection-continue. On montre ceci en constatant que toute suite de sous-ensembles $p_1 \subseteq p_2 \subseteq p_3 \subseteq \ldots$ d'un ensemble fini S doit avoir un élément maximum p_m [5] qui est $\cup_i p_i$. Si τ est monotone alors $\cup_i \tau(p_i) = \tau(p_m)$, donc $\cup_i \tau(p_i) = \tau(\cup_i p_i)$. On peut montrer de manière similaire que $\cap_i \tau(p_i) = \tau(\cap_i p_i)$.

[5] $\forall j \geq m : p_j = p_m \Rightarrow \forall j : p_j \subseteq p_m$

5. Vérification CTL de modèles

Identifions maintenant chaque formule CTL f avec $\{s \mid s \models f\}$, càd l'ensemble des états dans lesquels la formule est vraie. Notons que la formule EFp est logiquement équivalente à $p \vee EX\ EFp$, càd que EFp est vraie à l'état courant s quand p est vrai en s ou que EFp est vraie pour un des successeurs de s. Vu que deux formules logiquement équivalentes sont satisfaites dans le même ensemble d'états, on a que $EFp = p \vee EX\ EFp$, et ceci fait que EFp est un point fixe de la fonctionnelle $\tau = \lambda y.p \vee EXy$. On peut facilement montrer que cette fonction est monotone.

La fonctionnelle $\tau = \lambda y.p \vee EXy$ a un plus petit point fixe qui est exactement EFp. Montrons-le en supposant que y est un point fixe de τ. Il suit que $p \subseteq y$ et que $EXy \subseteq y$. De cette dernière expression, on constate que y contient tous ses prédécesseurs. Donc, il n'existe pas de chemin qui commence dans un état n'appartenant pas à y et qui atteint un état de y. Comme $p \subseteq y$, il suit que $EFp \subseteq y$. Donc, EFp est contenu dans tous les points fixes de τ, ce qui en fait le plus petit point fixe de τ.

Des caractérisations similaires peuvent être obtenues pour les autres opérateurs CTL. En particulier,

Théorème (Clarke-Emerson) : *Pour tout modèle* (S, R, L),

$$\begin{aligned} EFp &= \mu y.(p \vee EXy) \\ EGp &= \nu y.(p \wedge EXy) \\ E(q\,\mathcal{U}p) &= \mu y.(p \vee (q \wedge EXy)) \end{aligned}$$

Comme dans le cas de la vérification de modèles, on ne considère que des modèles basés sur un ensemble S fini, chacun des points fixes ci-dessus peut être caractérisé comme la limite d'une série obtenue en itérant sur la fonctionnelle correspondante.

Considérant donc que S est fini, on a donc :

$$EFp = \cup_i(\lambda y.(p \vee EXy))^i(false) \qquad (1.11)$$
$$EGp = \cap_i(\lambda y.(p \wedge EXy))^i(true) \qquad (1.12)$$
$$E(q\,\mathcal{U}p) = \cup_i(\lambda y.(p \vee (q \wedge EXy)))^i(false) \qquad (1.13)$$

Comme S est fini, ces séries atteignent leur limite en un nombre fini d'itérations, qui vaut au maximum $|S|$. Ceci à cause du fait qu'une fonctionnelle monotone appliquée à $false$ donne lieu à une suite non décroissante, et qu'il n'existe pas de suite strictement croissante d'ensembles qui aurait une taille supérieure à $|S|$. Toute suite non décroissante de longueur supérieure à $|S|$ doit avoir un élément qui se répète, et qui est nécessairement la limite.

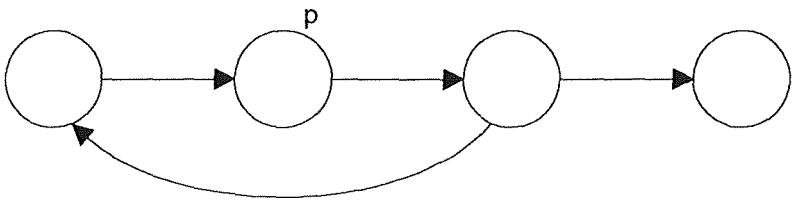
Une fonctionnelle monotone appliquée à $true$ produit une suite non croissante, avec le même résultat.

Ceci fournit une procédure efficace pour évaluer le plus petit {plus grand} point fixe d'une fonctionnelle τ :

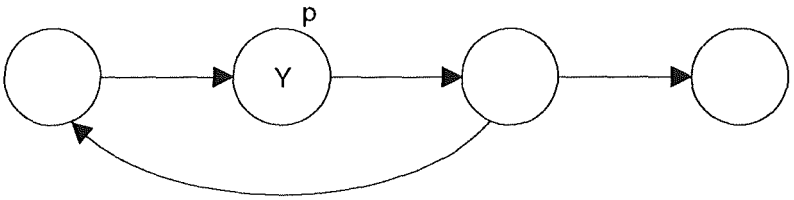
```
let Y = false {true}
do
  let Y' = Y, Y = τ(Y)
until Y'=Y;
return Y
```

6. Exemples

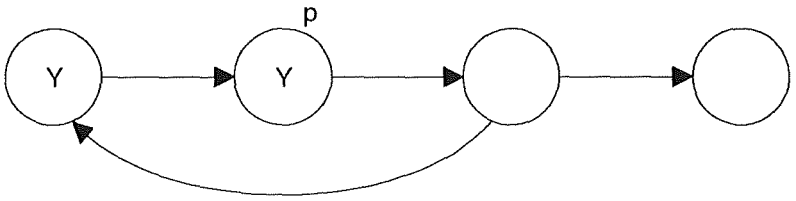
1. Comme première illustration, considérons le calcul de EFp dans le modèle de Kripke[6] suivant :



Comme |S| = 4, le nombre d'itérations nécessaires pour trouver le point fixe vaudra au plus 4. Calculons donc τ^i(false) pour i variant de 1 à 4, où $\tau = \lambda y.p \vee EXy$. Après la première itération, on a que τ^1(false) = p ∨ EXfalse = p :

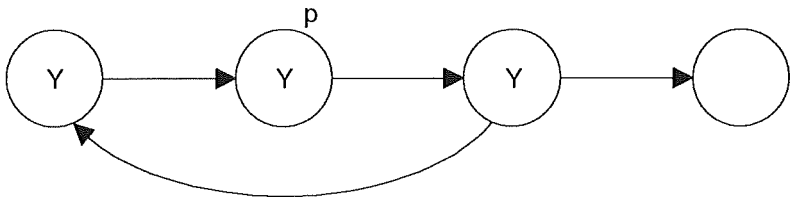


Après la seconde itération, on a τ^2(false) = p ∨ EXp :



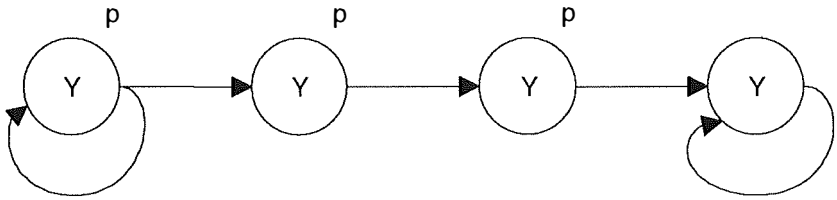
Après la troisième itération, on a τ^3(false) = p ∨ EX(p ∨ EXp) :

[6] On représente ici un modèle de Kripke par un graphe (S, R) en étiquetant chaque état par les propositions atomiques qui sont vraies dans cet état.

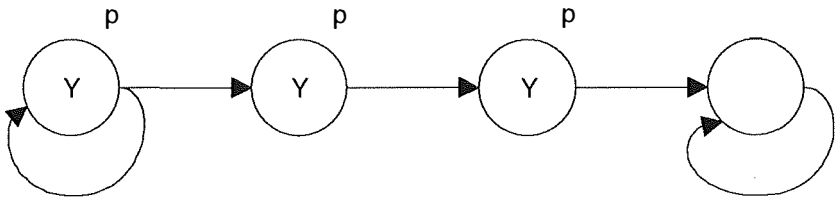


Il s'agit donc du point fixe, puisque l'itération suivante, $\tau^4(\texttt{false})$, produit le même résultat. Notons qu'à chaque itération `i`, nous avons l'ensemble d'états s_0 tel qu'il existe un chemin (s_0, s_1, s_2, ...) où `p` est vrai à un des états inférieur à `i`. En effet, chaque itération propage la formule `EFp` d'un pas en arrière dans le graphe. Quand ce processus atteint un point fixe, l'ensemble des états étiquetés contient exactement ceux qui se trouvent sur un chemin menant vers un état étiqueté par p.

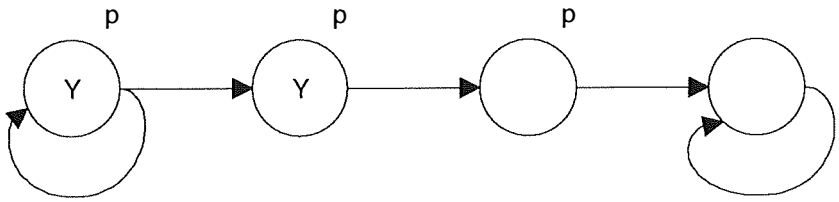
2. Comme second cas, nous allons calculer `EGp` dans le modèle de Kripke suivant :



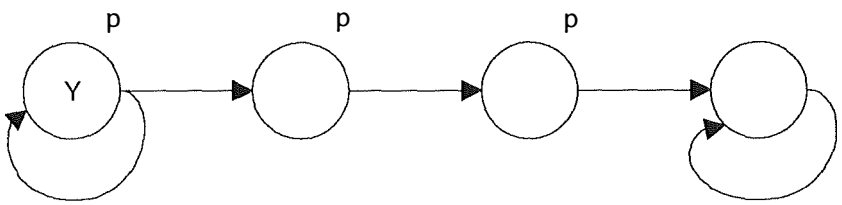
Après la première itération, on a que $\tau^1(\texttt{true}) = \texttt{p} \wedge \texttt{EXtrue} = \texttt{p}$:



Après la seconde itération, on a $\tau^2(\texttt{true}) = \texttt{p} \wedge \texttt{EXp}$:



Après la troisième itération, $\tau^3(\mathtt{true}) = \mathtt{p} \wedge \mathtt{EX}(\mathtt{p} \wedge \mathtt{EXp})$:



Ceci est le plus grand point fixe, puisque l'itération suivante fournit le même résultat. Notons qu'à chaque itération i, nous avons l'ensemble d'états tel qu'il existe un chemin de longueur i où chaque état satisfait p. Quand un point fixe est atteint, chaque état de l'ensemble a un successeur de cet ensemble qui satisfait p, donc pour tout état de l'ensemble, il existe un chemin infini où p est toujours vrai.

Les opérateurs EX, E($\mathcal{U}$) et EG sont en fait suffisants pour caractériser la logique entière, les autres opérateurs pouvant être dérivés de ceux-ci par les règles suivantes :

$$\mathtt{EFp} = \mathtt{E}(\mathtt{true}\,\mathcal{U}\mathtt{p})$$
$$\mathtt{AXp} = \neg\mathtt{EX}\neg\mathtt{p}$$
$$\mathtt{AGp} = \neg\mathtt{EF}\neg\mathtt{p}$$
$$\mathtt{A}(\mathtt{q}\,\mathcal{U}\mathtt{p}) = \neg(\mathtt{E}(\neg\mathtt{p}\,\mathcal{U}\neg\mathtt{q} \wedge \neg\mathtt{p}) \vee \mathtt{EG}\neg\mathtt{p})$$

C'est pour cette raison que dans la suite, on ne considérera que les opérateurs EX, E($\mathcal{U}$) et EG. Nous citerons tout de même les caractérisations en termes de points fixes des opérateurs suivants :

$$\mathtt{AGp} = \nu\mathtt{Y}.(\mathtt{p} \wedge \mathtt{AXY})$$
$$\mathtt{A}(\mathtt{q}\,\mathcal{U}\mathtt{p}) = \mu\mathtt{Y}.\mathtt{p} \vee (\mathtt{p} \wedge \mathtt{AXY})$$

La caractérisation en termes de points fixes fournit un algorithme efficace pour le problème de la vérification de modèles. Un algorithme plus efficace existe cependant, basé sur une recherche en largeur d'abord, et sur le calcul de composantes fortement connexes dans le graphe (S, R) [CES86].

Chapitre 2

Vérification symbolique de modèles*

Dans le précédent chapitre, nous avons identifié une formule CTL avec l'ensemble des états dans lesquelles cette formule est vraie. Nous avons vu que les opérateurs CTL peuvent être caractérisés comme des points fixes de certaines fonctionnelles continues dans le treillis des sous-ensembles, et nous avons montré que ces points fixes peuvent être calculés de façon itérative. Ceci nous a fourni un algorithme de vérification pour la logique CTL, mais nécessite de construire un modèle de Kripke fini de notre système et conduit au problème de l'explosion des états. Nous allons, dans ce chapitre, explorer une méthode qui évite dans certains cas ce problème, en représentant de manière implicite le modèle de Kripke dans une formule booléenne. Ceci permet d'implémenter la technique de vérification de modèles en utilisant les techniques déjà connues de manipulation de formules booléennes. Comme le modèle de Kripke est représenté symboliquement, il n'est pas nécessaire de le développer en une structure de données explicite. Le problème de l'explosion des états peut donc être évité.

1. Représentations booléennes

Tout d'abord, il s'agit de développer une façon d'identifier les formules booléennes avec les ensembles d'états et les relations sur les états. Soit donc S un ensemble de variables, et considérons l'algèbre booléenne 2^{2^S}. Comme précédemment, nous identifions chaque formule par un ensemble, et dans ce cas-ci un ensemble de sous-ensembles de S. On interprète chaque sous-ensemble de S comme une valuation vraie, où les variables dans l'ensemble sont vraies, et celles en dehors sont fausses. Donc, chaque variable $v \in S$ est identifiée par l'ensemble des ensembles qui contiennent v, soit $\{a \in 2^S \mid v \in a\}$.

* Rédigé d'après [McMi92]

Maintenant , considérons S comme l'ensemble des vecteurs booléens {true, false}^{n}. On peut de cette façon identifier une fonctionnelle avec un ensemble d'états. Considérons par exemple la fonctionnelle vectorielle $\lambda(v_1, ..., v_n).f$. Appliquée à un vecteur $(a_1, ..., a_n)$, noté a, la fonctionnelle fournit f avec les v_i remplacés par les a_i. Les fonctionnelles vectorielles obéissent aux axiomes usuels utilisés avec la notation λ :

$$(\lambda v.v_i)(a) = a_i \quad (2.1)$$

$$(\lambda v.f)(a) = f \text{ si } f \text{ ne contient aucun } v_i \quad (2.2)$$

$$(\lambda v.f(g))(a) = (\lambda v.f)(a)\ ((\lambda v.g)(a)) \quad (2.3)$$

Supposons que nous ayons une fonctionnelle vectorielle $\tau = \lambda(v_1, ..., v_n).f$ qui soit fermée, dans le sens qu'elle donne une valeur (soit true soit false) à tout vecteur booléen $a \in$ {true, false}^{n}. Une formule f qui contient seulement les variables v_1, ..., v_n est une telle fonctionnelle. Dans ce cas la fonctionnelle τ caractérise de façon univoque un ensemble de vecteurs booléens, exactement les vecteurs (ou états) a tels que $\tau(a)$ = true. Pour tout ensemble d'états p, nous avons une représentation univoque **p** telle que

$$p = \lambda(v_1, ..., v_n).\mathbf{p} \quad (2.4)$$

Notons que p est un ensemble de vecteurs booléens, et que **p** est un ensemble de sous-ensembles de S. La représentation est fixée par le choix particulier des variables v_1, ..., v_n.

Passons maintenant à la représentation des relations. Considérons la fonctionnelle vectorielle $\tau = \lambda((v_1, ..., v_n),(v_1', ..., v_n')).f$, qui prend la paire de vecteurs $(v_1, ..., v_n)$ et $(v_1', ..., v_n')$ comme arguments. Si cette fonctionnelle est fermée, au sens vu ci-dessus, alors elle caractérise un ensemble de paires d'états (a, a') tel que τ(a, a') = true. Donc, pour toute relation R sur les états, nous avons une représentation **R** univoque telle que

$$R = \lambda((v_1, ..., v_n),(v_1', ..., v_n')).\mathbf{R} \quad (2.5)$$

Notons que R est un ensemble de paires de vecteurs, et que **R** est un ensemble de sous-ensembles de S. La représentation est fixée par le choix particulier des variables v_1, ..., v_n et v_1', ..., v_n'.

2. Modèles symboliques

2.1. Représentation symbolique des états, transitions, opérateurs

Supposons que nous ayons un modèle K = (S, R, L) tel que l'ensemble des états S est l'ensemble des vecteurs booléens {true, false}^{n} et que la relation de transition R est caractérisée par l'équation (2.5). Supposons avoir un vecteur de propositions atomiques $l = l_1, ..., l_n$, dont les valeurs de vérité correspondent aux éléments du vecteur d'état. Donc la valuation L fait correspondre chaque proposition l_i à l'ensemble de vecteurs $(a_1, ..., a_n)$ tel que a_i = true. En d'autres mots,

$$L(l_i) = \lambda(v_1, ..., v_n).v_i \tag{2.6}$$

Ce qui veut dire, selon notre représentation symbolique des ensembles, que $\mathbf{l}_i = v_i$. On admettra par la suite, sans perte de généralité, que les vecteurs l et v sont égaux : $\mathbf{l} = v$.

Pour caractériser les opérateurs CTL par rapport à notre modèle, il faut introduire la notion de quantification booléenne. Par la formule $\exists(v_1, ..., v_n).f$, on entend qu'il existe un vecteur booléen $(v_1, ..., v_n)$ qui rend f vrai. En d'autres mots,

$$\exists(v_1, ..., v_n).f = \bigvee_{a \in \{true, false\}^n} (\lambda(v_1, ..., v_n).f)(a) \tag{2.7}$$

De façon similaire,

$$\forall(v_1, ..., v_n).f = \bigwedge_{a \in \{true, false\}^n} (\lambda(v_1, ..., v_n).f)(a) \tag{2.8}$$

Supposons que l'ensemble p des états est caractérisé par l'équation (2.4). La formule EXp identifie l'ensemble des états $(v_1, ..., v_n)$ tels qu'il existe un état $(v_1', ..., v_n')$ tel que la paire $((v_1, ..., v_n),(v_1', ..., v_n'))$ est dans R et que $(v_1', ..., v_n')$ est dans p. Donc,

$$EXp = \lambda v.\exists v'.(R(v, v') \wedge p(v')) \tag{2.9}$$

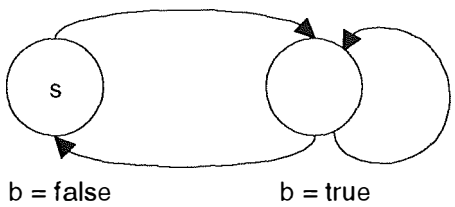
où $v = (v_1, ..., v_n)$ et $v' = (v_1', ..., v_n')$. Selon l'équation (2.5), $R(v, v') = \mathbf{R}$ et selon (2.4), $p(v') = \mathbf{p'}$, où $\mathbf{p'}$ est obtenu en remplaçant v_i' par v_i dans $\mathbf{p}$.
Ceci donne $EXp = \lambda v.\exists v'.(\mathbf{R} \wedge \mathbf{p'})$. Donc, selon la représentation booléenne,

$$EXp = \exists v'.(\mathbf{R} \wedge \mathbf{p'}) \tag{2.10}$$

En d'autres mots, étant donnée la représentation de p, on peut donner une représentation de EXp en remplaçant chaque variable v_i par v_i', puis en prenant la conjonction avec $\mathbf{R}$ et en quantifiant les variables $v_1', ..., v_n'$ existentiellement.

2.2. Exemples

1. Considérons un modèle dans lequel $v = (b)$, $v' = (b')$, et $\mathbf{R} = b \vee b'$. Le modèle de Kripke (S, R, L) est celui ci-dessous :

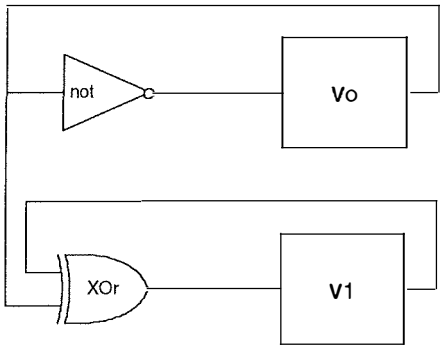


Si nous voulons déterminer l'ensemble d'états satisfaisant à $EX\neg b$, on peut écrire, suivant l'équation (2.10) :

$$\begin{aligned} EX\neg \mathbf{b} &= \exists(b').(\mathbf{R} \wedge (\neg b)') \\ &= \exists(b').((b \vee b') \wedge (\neg b')) \\ &= \exists(b').(b \wedge \neg b') \\ &= (b \wedge \neg false) \vee (b \wedge \neg true) \\ &= b \end{aligned}$$

Par quelques manipulations, on arrive à une formule qui caractérise exactement les états satisfaisant EXb. Pour trouver sa valeur de vérité pour un état particulier, il suffit d'affecter la valeur des composants du vecteur d'état aux variables appropriées. Par exemple la valeur de EXb à l'état s est $(\lambda(b).b)(false) = false$.

2. Considérons un modèle d'un compteur binaire à deux bits, comme ci-dessous :



Sa relation de transition est caractérisée par

$$\mathbf{R} = (v_0' \Leftrightarrow \neg v_0) \wedge (v_1' \Leftrightarrow v_0 \oplus v_1)$$

où $v = (v_0, v_1)$ et $v' = (v_0', v_1')$. Calculons symboliquement l'ensemble des états qui satisfont $EX(v_0 \wedge v_1)$. Selon l'équation (2.10) :

$$\begin{aligned} EX\, \mathbf{v_0} \wedge \mathbf{v_1} &= \exists(v_0', v_1').(\mathbf{R} \wedge (v_0 \wedge v_1)') \\ &= \exists(v_0', v_1').((\neg v_0 \wedge v_1 \wedge v_0' \wedge v_1') \\ &= \exists(v_1').(\neg v_0 \wedge v_1 \wedge false \wedge v_1' \ \vee \ \neg v_0 \wedge v_1 \wedge true \wedge v_1') \\ &= (\neg v_0 \wedge v_1 \wedge false) \vee (\neg v_0 \wedge v_1 \wedge true) \\ &= \neg v_0 \wedge v_1 \end{aligned}$$

Il n'est pas surprenant que l'état (1,1) soit précédé par l'état (0,1) uniquement, et c'est exactement le résultat que nous obtenons. Notons qu'un grand nombre de manipulations algébriques ont eu lieu dans la première étape. Ces calculs pourraient être effectués automatiquement en transformant toutes les formules en une forme canonique, par

exemple la forme canonique en mintermes[7]. Les questions concernant l'automatisation de ce processus seront abordées dans la prochaine section.

2.3. L'algorithme

Pour le moment, considérons les opérateurs CTL restants, qui ont une caractérisation en termes de EX. Par exemple, supposons comme dans le premier exemple que $\mathbf{R} = b \vee b'$. On voudrait calculer l'ensemble des états qui satisfont à EFb. Cette formule est le plus petit point fixe de $\tau = \lambda y.\ b \vee EXy$, et qui peut être obtenue comme la limite de la série $\cup_i \tau^i(\mathtt{false})$. Si on évalue les quelques premières itérations de cette série dans notre représentation booléenne, on obtient :

$$\begin{aligned}
\tau^1(\mathtt{false}) &= b \vee EX\mathtt{false} \\
&\quad\ b \\
\tau^2(\mathtt{false}) &= b \vee EXb \\
&\quad\ b \vee \exists b'.((b \vee b') \wedge b') \\
&\quad\ \mathtt{true} \\
\tau^3(\mathtt{false}) &= b \vee EX\mathtt{true} \\
&\quad\ \mathtt{true}
\end{aligned}$$

Un point fixe a été atteint après deux itérations. L'ensemble des états qui satisfont EFb est $\lambda(b).\mathtt{true} = S$.

Dans le cas du second exemple, pour évaluer $EF(v_0 \wedge v_1)$, on a successivement :

$$\begin{aligned}
\tau^1(\mathtt{false}) &= v_0 \wedge v_1 \\
\tau^2(\mathtt{false}) &= v_1 \\
\tau^3(\mathtt{false}) &= v_0 \vee v_1 \\
\tau^4(\mathtt{false}) &= \mathtt{true}
\end{aligned}$$

Les ensembles d'états caractérisés par ces formules sont exactement ceux que l'on aurait obtenus en utilisant la procédure de vérification basée sur les points fixes, dans le modèle explicite ; ce qui revenait à propager la formule $EF(v_0 \wedge v_1)$ en arrière dans le graphe.

Voici la procédure complète pour la vérification symbolique de modèles :

```
function eval(f)
  case
      f = proposition atomique : return f
      f = ¬p : return ¬eval(p)
      f = p ∨ q : return eval(p) ∨ eval(q)
      f = EXp : return evalEX(eval(p))
      f = E(p 𝒰 q) : return evalEU(eval(p), eval(q), false)
      f = EGp : return evalEG(eval(p), true)
```

[7] La forme canonique en mintermes est une disjonction de mintermes. Par exemple, si on a deux propositions, $a \vee b$ se transforme en $(a \wedge \neg b) \vee (a \wedge b) \vee (\neg a \wedge b)$.

```
    end case
end function

function evalEX(p) = ∃v'.(R ∧ p')

function evalEU(p, q, y)
    y' = q ∨ (p ∧ evalEX(y))
    if y' = y then return y
    else return evalEU(p, q, y')
end function

function evalEG(p, y)
    y' = (p ∧ evalEX(y))
    if y' = y then return y
    else return evalEG(p, y')
end function
```

Notons que pour déterminer que la limite est atteinte dans une série de points fixes, il faut savoir déterminer quand deux représentations symboliques d'ensembles sont égales. Ceci montre l'avantage d'une forme canonique pour la représentation d'ensembles.

3. Diagrammes de décision binaires (BDD)

Pour rendre utilisable la technique de la vérification symbolique, nous avons donc besoin d'une méthode automatique pour manipuler des formules booléennes. Nous devons notamment pouvoir réduire à une représentation adéquate les formules utilisant des quantificateurs existentiels, pour pouvoir tester l'égalité de formules. La représentation la plus appropriée pour réaliser cela est sans conteste la forme de *Ordered Binary Decision Diagram* (OBDD), développée par Bryant [Bry86]. Cette représentation est canonique, et la conjonction ainsi que la disjonction de deux OBDD's peut être calculée en temps quadratique.

Les OBDD's peuvent représenter des fonctions booléennes (ainsi que des ensembles de vecteurs booléens, ensembles de paires de vecteurs, etc.) de la même façon que des formules booléennes ordinaires, en associant les éléments d'un vecteur en entrée avec des variables d'un ensemble S. Dans les cas des OBDD's, une représentation canonique est obtenue en imposant un ordre total sur les variables de S. La représentation canonique, sous forme d'OBDD, d'une fonction booléenne peut être obtenue par réduction en une structure appelée *arbre de décision ordonné*. La valeur de vérité d'un arbre de décision ordonné est obtenue en parcourant l'arbre de la racine à une des feuilles. A chaque noeud le long du chemin, on descend d'un côté ou de l'autre suivant la valeur de la variable étiquetant le noeud. Chaque feuille de l'arbre est étiquetée par une valeur `true` ou `false`, qui donne le résultat. L'arbre doit respecter l'ordre sur S, en ce sens que les variables apparaissent toujours en ordre croissant sur n'importe quel chemin de la racine vers une feuille. En lisant les feuilles de gauche à droite, on obtient la table de vérité de la fonction représentée.

Considérons par exemple un arbre de décision ordonné représentant a ∧ b ∨ c ∧ d. L'ordre utilisé est a < b < c < d, et nous utilisons 0 pour représenter `false` et 1 pour `true` :

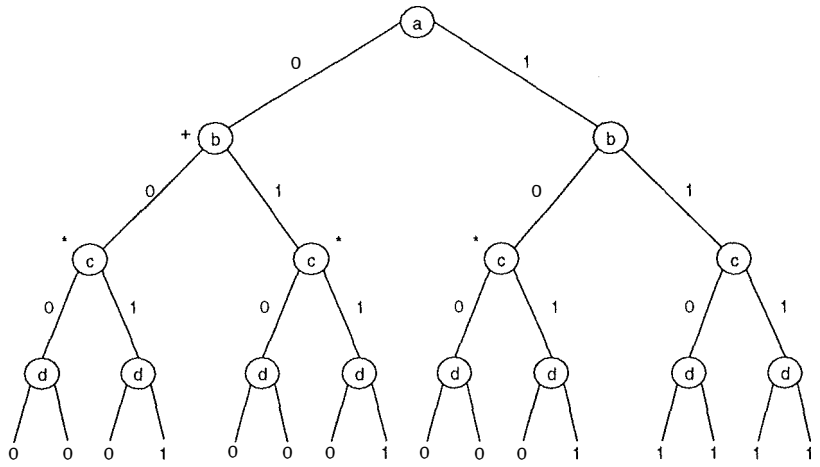


Figure 1. Un arbre de décision ordonné pour a ∧ b ∨ c ∧ d

La forme canonique OBDD est un graphe orienté sans cycle, et peut être obtenue à partir de l'arbre de décision ordonné en deux pas, appliqués de façon bottom-up :

- rassembler les arbres isomorphes en un arbre unique,
- éliminer tous les noeuds dont les deux fils sont isomorphes.

La transformation d'un arbre de décision ordonné en OBDD, appelée *Reduce* par Bryant, peut se faire en temps linéaire. La taille du graphe résultant dépend fortement de l'ordre choisi pour les variables, qui est la clé pour obtenir cette forme canonique.

Pour illustrer la réduction, considérons l'arbre de la figure Figure 1. Le noeuds marqués « * » sont les racines de sous-arbres isomorphes et peuvent donc être rassemblés un en seul sous-arbre. D'autre part, le noeud marqué « + » est le père de deux sous-arbres isomorphes, et peut donc être supprimé, puisqu'il n'affecte pas la valeur de la fonction. L'application de la réduction à cet arbre donne la Figure 2 :

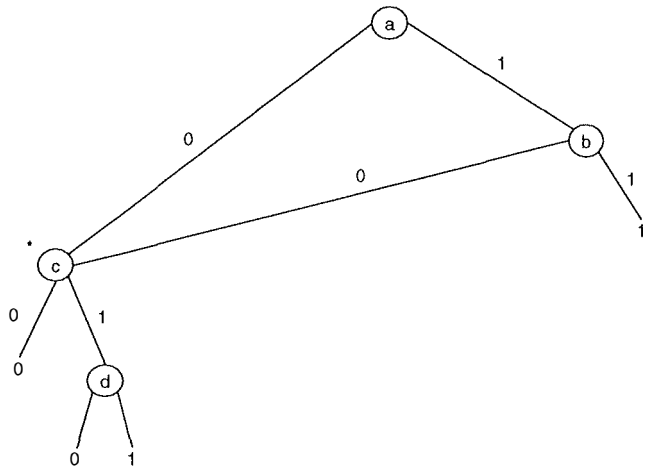


Figure 2. OBDD associé.

Le fait de rassembler les sous-arbres isomorphes fait que les OBDD's sont des graphes orientés sans cycle plutôt que des arbres. Les feuilles de ces graphes sont appelées terminaisons. Chaque noeud interne est un triplet (v, l, h) où $v \in S$ est la variable étiquetant le noeud, l (*low*) est le fils situé à gauche et h (*high*) est le fils de droite. L'ordre sur les variables induit un préordre sur les noeuds :

Définition : Pour tous noeuds $n_1 = (v_1, l_1, h_1)$ et $n_2 = (v_2, l_2, h_2)$, on a $n_1 < n_2$ quand $v_1 < v_2$. Pour tout noeud n, $n < 0$ et $n < 1$.

On peut définir par induction un ensemble de noeuds qui représente canoniquement les éléments de l'algèbre booléenne 2^{2^S}.

Définition : Soit S un ensemble totalement ordonné : OBDD(S) est le plus petit ensemble tel que

- $\{0, 1\} \subseteq$ OBDD(S) et
- pour tout noeud $l \neq h \in$ OBDD(S), pour tout $v \in S$, si $(v, l, h) < l$ et $(v, l, h) < h$ alors $(v, l, h) \in$ OBDD(S).

Ayant défini l'ensemble des noeud OBDD, on peut associer chaque noeud n à une formule booléenne f_n. Il est pour cela pratique d'introduire un opérateur booléen $p \rightarrow (x, y)$, qui est équivalent à x quand p est vrai, et à y quand p est faux. La formule $p \rightarrow (x, y)$ est équivalente à $(p \wedge x) \vee (\neg p \wedge y)$.

Définition :
1. $f_0 = \text{true}$
2. $f_1 = \text{false}$
3. si $n = (v, l, h) \in$ OBDD(S) alors $f_n = (v \rightarrow (f_h, f_l))$

Les théorèmes suivants montrent d'abord qu'il n'existe pas deux OBDD's différents représentant la même formule, et ensuite qu'il existe toujours un OBDD pour représenter une formule[8].

[8] Pour les preuves, on se rapportera utilement à [McMi 92], pp. 34 et 35.

Théorème 1 : Si n et $n' \in$ OBDD(S), alors $f_n = f_{n'} \Rightarrow n = n'$.

Théorème 2 : Pour tout $f \in 2^{2^S}$, il existe $n \in$ OBDD(S) tel que $f_n = f$.

Puisque deux formules égales sont représentées par le même noeud OBDD, vérifier l'égalité fonctionnelle de deux OBDD's peut se faire en temps constant. Cette propriété des OBDD's est utile pour déterminer quand la limite d'une série vers un point fixe a été atteinte, dans l'algorithme de vérification symbolique. Comme les noeuds OBDD sont en bijection avec les éléments de l'algèbre booléenne, on traitera les OBDD's de la même façon que les formules booléennes.

3.1. L'algorithme Apply

Bryant a décrit un algorithme appelé *Apply*, qui applique une opération booléenne $f : \{0, 1\}^2 \rightarrow \{0, 1\}$ à deux OBDD's. L'opération f peut être une des 16 fonctions booléennes de deux variables. Les opérations $\wedge$, $\vee$ et $\neg$ sont celles qui nous intéressent le plus, mais d'autres opérations comme le 'ou exclusif' sont également possibles.

Cet algorithme procède à une descente récursive sur les deux OBDD's. Il utilise une table de hashage pour enregistrer le résultat produit pour chaque paire de noeuds, de façon à calculer une seule fois le résultat d'une certaine paire. *Apply* a une complexité quadratique.

3.2. L'algorithme AndExists

L'algorithme *AndExists* prend comme arguments un vecteur v de variables et une paire d'OBDD's (p, q), et renvoie $\exists v.(p \wedge q)$. Il s'agit d'un cas particulier de *Apply*, avec $f(p, q) = p \wedge q$. Néanmoins, avant de renvoyer un résultat $r = (w, l, h)$, on teste la variable w pour voir si elle apparaît dans le vecteur v. Si c'est le cas, on appelle *Apply* pour calculer $l \vee h$, car $\exists v.(v \rightarrow (h, l)) = h \vee l$. Sinon, on renvoie r.

La motivation à la base de cet algorithme est d'éviter de produire un OBDD entier pour $p \wedge q$, qui aurait $2n$ variables, où n est le nombre de variables d'état du modèle. Ceci est réalisé en effectuant une quantification existentielle sur les résultats des sous-problèmes dès qu'ils sont disponibles, avec comme conséquence de fournir un résultat de seulement n variables. Ceci fournit donc des gains substantiels d'espace.

Chapitre 3

SMV : un vérificateur de modèles*

Pour mettre en pratique la technique de vérification de modèles et l'appliquer à des cas réels, on a besoin d'un langage tel que l'on puisse décrire notre modèle à un suffisamment haut niveau. Le langage doit fournir des opérations sur des type de données de haut niveau, comme les types énumérés, et permettre de décrire facilement des choix non déterministes, pour pouvoir décrire des protocoles sans se soucier de détails d'implémentation. Le langage doit avoir une sémantique mathématique précise qui définit la traduction d'un programme du langage vers une forme adéquate pour la vérification symbolique (càd une formule booléenne représentant la relation de transition). Idéalement, il devrait aussi être possible d'utiliser au moins un sous-ensemble utile du langage pour réaliser la description de systèmes réels, pour leur réalisation pratique.

SMV est un outil permettant de vérifier des systèmes finis par rapport à leurs propriétés exprimées en logique temporelle CTL. Le langage permet la description de systèmes finis aussi différents que les systèmes complètement synchrones et complètement asynchrones, fortement détaillés ou abstraits. Le langage permet des descriptions hiérarchiques et modulaires, et la définition de composants réutilisables. Comme il est supposé décrire des machines à états finis, les types de données de base sont les tableaux bornés d'entiers et les types énumérés symboliques. Des types structurés statiques peuvent aussi être construits à partir de ces composants de base. La logique CTL permet de définir de façon concise un grand nombre de propriétés temporelles, comme la sécurité, la vivacité et l'absence d'interblocage. SMV utilise l'algorithme de vérification de modèles basé sur les OBDD's pour déterminer si les spécifications en CTL sont satisfaites.

Ce chapitre décrit la syntaxe et la sémantique du langage SMV en des termes appropriés pour la vérification de modèles.

---

* Inspiré de [McMi92]

1. Introduction informelle

Commençons par illustrer les concepts de base du langage, avec le programme suivant :

```
MODULE main
VAR
  request : boolean ;
  state : {ready, busy} ;
ASSIGN
  init(state) := ready ;
  next(state) := case
                   state = ready & request : busy ;
                   1 : {ready, busy} ;
                 esac ;
SPEC
  AG(request -> AF state = busy)
```

Ce programme décrit à la fois le modèle et la spécification. Le modèle est un modèle de Kripke, dont l'état est défini par un ensemble de variables d'état, qui peuvent être de type booléen, tableau fini d'entiers, ou énuméré. La variable `request` ci-dessus est déclarée booléenne, et la variable `state` peut seulement prendre les valeurs symboliques `ready` ou `busy`. La valeur d'une variable de type énuméré est encodée par le compilateur avec un certain nombre de variables booléennes, de façon à pouvoir représenter la relation de transition par un OBDD.

Le comportement de transition du programme et son ou ses états initiaux, sont déterminés par une série d'assignations parallèles. Ils sont introduits par le mot-clé `ASSIGN`. Dans le programme ci-avant, la valeur initiale de la variable `state` est positionnée à `ready`. A tout moment, la valeur suivante (une unité de temps dans le futur) de `state` est donnée par l'expression

```
case
  state = ready & request : busy ;
  1 : {ready, busy} ;
esac ;
```

La valeur d'une expression `case` est déterminée par la première expression à droite d'un (:) telle que la condition du côté gauche est vraie. Donc, si `state = ready & request` est vraie, alors le résultat de l'expression est `busy`, sinon c'est l'ensemble `{ready, busy}`. Quand on assigne un ensemble à une variable, le résultat est un choix non déterministe parmi les valeurs de l'ensemble. Donc, si la valeur de `state` n'est pas `ready`, ou que `request` est `false`, dans l'état présent, alors la valeur suivante de `state` est soit `ready` soit `busy`. Ces assignations non déterministes sont utiles quand il s'agit de décrire des systèmes qui ne sont pas complètement spécifiés, càd quand il suffit de savoir simplement qu'une variable peut changer d'état, sans avoir à se soucier des circonstances non modélisées qui amèneraient ce changement d'état.

On peut noter que la variable `request` n'est pas assignée dans ce programme, laissant SMV libre de choisir n'importe quelle valeur pour cette variable.

La spécification du système apparaît sous forme d'une formule CTL sous le mot-clé `SPEC`. Le vérificateur contrôle que tous les états possibles satisfont à la

spécification. Dans notre exemple, la spécification est que si `request` est vrai, alors on a inévitablement que `state` est à `busy`.

Le programme suivant illustre la définition de modules et d'expressions réutilisables. Il s'agit du modèle d'un circuit représentant un compteur binaire 3 bits. Notons que le module appelé main a une signification spéciale en SMV, de la même façon que par exemple dans le langage C.

```
MODULE main
VAR
  bit0 : counter-cell(1);
  bit1 : counter-cell(bit0.carry_out);
  bit2 : counter-cell(bit1.carry_out);
SPEC
  AG AF bit2.carry_out

MODULE counter-cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value) := value + carry_in mod 2;
DEFINE
  carry_out := value & carry_in;
```

Dans cet exemple, on voit qu'une variable peut aussi être une instance d'un module. Le module est dans ce cas `counter_cell`, qui est instancié 3 fois. Ce module a un paramètre formel `carry_in`. Dans le cas de `bit1`, on assigne à `carry_in` la valeur de l'expression `bit0.carry_out`. Cette expression est évaluée dans le contexte du module principal. Néanmoins, une expression de la forme `a.b` dénote le composant `b` du module `a`, comme si le module a était une structure de données d'un langage de programmation classique. Donc, le composant `carry_in` du module `bit1` est le `carry_out` du module `bit0`. Le mot-clé `DEFINE` est utilisé pour assigner l'expression `value & carry_in` au symbole `carry_out`. Ce genre d'expression est analogue à la définition de macros, et un symbole ainsi défini peut être référencé avant d'être déclaré.

L'effet du mot-clé `DEFINE` aurait pu être obtenu en déclarant une variable et en lui affectant une valeur, comme suit :

```
VAR
  carry_out : boolean ;
ASSIGN
  carry_out := value & carry_in ;
```

Notons que dans ce cas, c'est la valeur courante de la variable qui est assignée, plutôt que la valeur suivante. Les symboles définis sont parfois préférables aux variables, dans la mesure où ils ne nécessitent pas d'introduire une nouvelle variable dans la représentation du système sous forme d'OBDD. Néanmoins, la faiblesse des symbole définis est de ne pas pouvoir recevoir de valeurs non déterministes. Une autre caractéristique qui distingue les symboles des variables est que celles-ci sont typées statiquement, alors que les définitions ne le sont pas.

Dans les langages à assignation parallèle, on peut se demander ce qui se produit quand on assigne deux fois une valeur à une variable, en parallèle, ou encore ce qui se passe quand on écrit `a := a+ 1`, au lieu de `next(a) = a + 1`. Dans le cas de SMV, le compilateur détecte les assignations multiples et les dépendances circulaires, et les considère comme des erreurs, même dans le cas où le système d'équations correspondant a une solution unique. Une autre façon d'exprimer ceci est qu'il doit toujours exister un ordre total dans lequel les assignations doivent être effectuées, et qui respecte toutes les dépendances de variables. La même logique vaut aussi pour les symboles. Un résultat de ceci est que tous les programmes SMV peuvent être réalisés sous forme de circuits séquentiels.

Par défaut, toutes les instructions d'assignation d'un programme SMV sont exécutées en parallèle et simultanément. Il est néanmoins possible de définir un ensemble de processus parallèles, qui tournent à des intervalles de temps arbitraires. Ceci est pratique pour définir des protocoles de communication, des circuits asynchrones, ou d'autres systèmes dont les actions ne sont pas synchronisées par une horloge globale. Cette technique est illustrée par le programme suivant, qui représente un anneau de trois portes oscillantes.

```
MODULE main
VAR
  gate1 : process inverter(gate3.output) ;
  gate2 : process inverter(gate1.output) ;
  gate3 : process inverter(gate2.output) ;
SPEC
  (AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR
  output : boolean ;
ASSIGN
  init(output) := 0 ;
  next(output) := !input ;
```

Un processus est une instance d'un module qui est introduite par le mot-clé `process`. A chaque instant, le processus est soit actif soit inactif. Une assignation à la valeur suivante d'une variable se produit seulement quand le processus est actif. Sinon, la valeur de la variable au moment suivant est identique. L'intervalle séparant deux états actifs d'un processus est arbitraire, autrement dit non déterministe.

Dans l'exemple ci-dessus, on utilise un processus pour modéliser une porte dans un circuit asynchrone, parce qu'on ne souhaite faire aucune hypothèse concernant le temps qui est nécessaire à la sortie d'une porte pour changer d'état. Au plus un seul des trois processus de ce programme est autorisé à être actif à un moment donné. Ceci nous donne un modèle entrelacé de l'exécution. Le vérificateur peut utiliser ce fait pour structurer l'arbre de décision de telle façon qu'il représente une relation de transition disjonctive, avec pour effet d'économiser l'espace occupé par cette représentation.

La spécification de ce programme est que la sortie de la porte 1 doit osciller (i.e. que sa valeur doit être infiniment souvent 0 et infiniment souvent 1). Le lecteur remarquera que cette spécification est fausse, car dans ce programme rien n'oblige un processus à

s'exécuter infiniment souvent. La sortie d'une porte peut donc rester constante, même si la valeur de son entrée change.

Pour forcer un processus à s'exécuter infiniment souvent, on peut utiliser une contrainte d'équité. Une telle contrainte limite l'attention du vérificateur aux seuls chemins d'exécution le long desquels une formule CTL donnée est vraie infiniment souvent. Chaque processus a une variable spéciale `running`, qui st vraie si et seulement si ce processus est en train de s'exécuter. En ajoutant la déclaration

```
FAIRNESS
  running
```

au module `inverter`, on peut forcer chaque instance de ce module à s'exécuter infiniment souvent, rendant la spécification vraie.

L'alternative à l'utilisation de processus pour modéliser un circuit asynchrone pourrait être de permettre à toutes les portes, mais en laissant à chaque porte le choix non déterministe de changer ou non sa sortie. Ceci est alors appelé modèle simultané, avec l'utilisation d'une relation de transition conjonctive. Un tel modèle de l'anneau oscillant pourrait ressembler à ceci :

```
MODULE main
VAR
  gate1 : inverter(gate3.output) ;
  gate2 : inverter(gate1.output) ;
  gate3 : inverter(gate2.output) ;
SPEC
  (AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR
  output : boolean ;
ASSIGN
  init(output) := 0 ;
  next(output) := !input union output;
```

L'opérateur `union` permet d'exprimer un choix non déterministe entre deux expressions. La prochaine sortie de chaque porte peut donc être soit sa sortie actuelle, soit l'inverse de son entrée actuelle ; chaque porte peut alors choisir de façon non déterministe de commuter ou non. Il apparaît donc que le nombre de transitions possibles à partir d'un certain état est 2^n, où `n` est le nombre de portes. Ceci peut parfois rendre plus coûteuse la représentation de la relation de transition.

Le mécanisme d'assignation parallèle possède l'avantage que les programmes qui l'utilisent ont la garantie d'être implémentables. Il est néanmoins possible en SMV de décrire directement la relation de transition en utilisant une formule booléenne en termes des valeurs courantes et suivantes des variables d'états. De façon similaire, il est aussi possible de décrire une condition initiale en termes des variables d'état courantes. Ces deux fonctions sont réalisées par les instruction `TRANS` et `INIT` respectivement. Voici en guise d'exemple une description de l'anneau oscillant utilisant exclusivement `TRANS` et `INIT` pour décrire la relation de transition :

```
MODULE main
```

```
VAR
  gate1 : inverter(gate3.output) ;
  gate2 : inverter(gate1.output) ;
  gate3 : inverter(gate2.output) ;
SPEC
  (AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR
  output : boolean ;
INIT
  output = 0
TRANS
  next(output) = !input | next(output) = output
```

Selon la déclaration `TRANS`, pour chaque inverseur, la prochaine valeur de `output` est soit la négation de son entrée, soit la valeur inchangée de sa sortie actuelle. L'utilisation de `TRANS` et `INIT` n'est toutefois pas recommandée, vu que des absurdités logiques dans ces déclarations pourraient conduire à des descriptions non implémentables. Par exemple, on pourrait déclarer que la constante 0 (`false`) représente la relation de transition, décrivant alors un système sans transitions. Néanmoins, cette flexibilité peut être pratique pour réaliser des traducteurs d'autres langages vers SMV.

En résumé, le langage SMV a été construit pour être flexible, en termes des types de modèles qu'il peut décrire. Le langage permet de décrire de façon relativement concise des systèmes synchrones et asynchrones, de décrire des modèles déterministes détaillés ou des modèles non déterministes abstraits, et d'exploiter la structure modulaire d'un système pour rendre sa description plus courte. Il est également possible d'écrire des absurdités logiques si on désire le faire, en utilisant les déclarations `TRANS` et `INIT`. Ce problème peut toutefois être évité en recourant au mécanisme d'assignation parallèle. Le langage est prévu pour exploiter les possibilités de la technique de vérification symbolique de modèles, avec comme effet que les type de données sont finis et statiques. Le langage ne prévoit pas de modèle de communication particulier entre processus concurrents (par exemple l'envoi synchrone ou asynchrone de messages). De plus, il n'y a pas de support explicite pour certaines caractéristiques des modèles de processus communiquants, telles que l'exécution séquentielle. Comme l'intégralité de la technique du vérification symbolique est disponible dans SMV, il est possible de créer des traducteurs opérant à partir de nombreux langages, de modèles de processus ou de formats intermédiaires.

2. Syntaxe

2.1. Conventions lexicales

Un **atome** peut être n'importe quelle suite de caractères de l'ensemble {A-Z, a-z, 0-9, -, _}, commençant par un caractère alphabétique. Tous les caractères d'un nom sont significatifs, ainsi que la casse. Les caractères d'espacement sont l'espace, la tabulation et le retour à la ligne. Toute chaîne commençant par deux tirets (« -- ») et se terminant par un retour à la ligne est un commentaire. Un nombre est constitué d'une suite de chiffres. Tout autre symbole reconnu par l'analyseur syntaxique sera signalé entre guillemets dans les expressions ci-dessous.

2.2. Expressions

Les expressions sont construites à partir de variables, de constantes, et d'un ensemble d'opérateurs, y compris les connecteurs booléens, les opérateurs arithmétiques entiers et les expressions **case**. La syntaxe des expressions est comme suit.

```
expr ::
        atom                          ;; une constante symbolique
        | number                      ;; une constante numérique
        | id                          ;; un identifiant de variable
        | «!» expr                    ;; non logique
        | expr1 «&» expr2             ;; et logique
        | expr1 «|» expr2             ;; ou logique
        | expr1 «->» expr2            ;; implication logique
        | expr1 «<->» expr2           ;; équivalence logique
        | expr1 «=»expr2              ;; égalité
        | expr1 «<» expr2             ;; inférieur
        | expr1 «>» expr2             ;; supérieur
        | expr1 «<=» expr2            ;; inférieur ou égal
        | expr1 «>=» expr2            ;; supérieur ou égal
        | expr1 «+» expr2             ;; addition entière
        | expr1 «-» expr2             ;; soustraction entière
        | expr1 «*» expr2             ;; multiplication entière
        | expr1 «/» expr2             ;; division entière
        | expr1 «mod» expr2           ;; reste entier
        | «next» «(» id «)»           ;; valeur suivante
        | set_expr                    ;; une expression d'ensemble
        | case_expr                   ;; une expression case
```

Un `id`, ou identifiant, est un symbole ou une expression qui identifie un objet, tel qu'une variable ou un symbole défini. Comme un `id` peut être un atome, il y a une possibilité d'ambiguïté si une variable ou un symbole défini porte le même nom qu'une constante symbolique. Une telle ambiguïté est considérée comme une erreur par le compilateur. L'expression `next(id)` réfère à la valeur de `id` dans le prochain état. L'ordre d'évaluation, du plus prioritaire au moins prioritaire est

*, /
+, -
mod
=, <, >, <=, >=
!
&
|
->, <->

Les opérateurs de même priorité s'associent à gauche. Des parenthèses peuvent être utilisées pour grouper des expressions.

Une expression `case` a la syntaxe suivante :

```
case_expr ::
        «case»
                expr_a1 «:» expr_b1 «;»
                expr_a2 «:» expr_b2 «;»
                ...
        «esac»
```

Une expression `case` renvoie la valeur de la première expression du côté droit, telle que la condition correspondante du côté gauche est vraie. Donc, si `expr_a1` est vraie, alors le résultat est `expr_b1`. Sinon, si `expr_a2` est vraie, alors le résultat est `expr_b2`, etc. Si aucune des expressions du côté gauche n'est vraie, alors le résultat de l'expression `case` est 1. Il se produit une erreur si une expression du côté gauche renvoie autre chose que les valeurs de vérité 0 et 1.

Une expression d'ensemble a la syntaxe

```
set_expr ::
       «{» val1 «,» val2 «,» ... «}»
       | expr1 «in» expr2                    ;; prédicat d'inclusion
       | expr1 «union» expr2                 ;; union d'ensembles
```

Un ensemble peut être défini en énumérant ses éléments entre accolades. Les éléments d'un ensemble peuvent être des nombres ou des constantes symboliques. L'opérateur d'inclusion teste l'appartenance d'une valeur à un ensemble. L'opérateur d'union réalise l'union de deux ensembles. Si un des deux arguments est un nombre ou une valeur symbolique plutôt qu'un ensemble, il est considéré comme un singleton.

2.3. Déclarations

2.3.1. La déclaration VAR

Un état du modèle est une assignation de valeurs à un ensemble de variables d'état. Ces variables, ainsi que les instances de modules, sont déclarées par la notation

```
decl :: «VAR»
          atom1 «:» type 1 «;»
          atom2 «:» type 2 «;»
          ...
```

Le type associé à une déclaration de variable peut être soit booléen, ou énuméré, ou un module utilisateur. Un type a la syntaxe

```
type :: boolean
        | «{» val1 «,» val2 «,» ... «}»
        | atom [ «(» expr1 «,» expr2 «,» ... «)» ]
        | «process» atom [ «(» expr1 «,» expr2 «,» ... «)» ]

val :: atom | number
```

Une variable de type booléen peut prendre seulement les valeurs numériques 0 et 1, représentant faux et vrai, respectivement. Dans le cas d'une liste de valeurs entre accolades (où les atomes sont considérés comme des constantes symboliques), la variable peut prendre une quelconque de ces valeurs. Finalement, un atome éventuellement suivi d'une liste d'expressions entre parenthèses indique une instance du module de nom `atom`. Le mot-clé `process` a pour effet d'instancier le module en tant que processus asynchrone.

2.3.2. La déclaration ASSIGN

Une déclaration d'assignation a la forme

```
decl :: «ASSIGN»
        dest1 «:=» expr1 «;»
        dest2 «:=» expr2 «;»
        ...

dest :: atom
      | «init» «(» atom «)»
      | «next» «(» atom «)»
```

Du côté gauche d'une assignation, `atom` dénote la valeur courante d'une variable, `init(atom)` dénote sa valeur initiale, et `next(atom)` sa valeur dans l'état suivant. Si l'évaluation de l'expression du côté droit donne un entier ou une constante symbolique, l'assignation signifie simplement que le côté gauche est égal au côté droit. Au contraire, si cette évaluation fournit un ensemble, alors l'assignation signifie que le côté gauche appartient à cet ensemble. Le fait d'avoir que la valeur d'une expression n'appartient pas à l'ensemble des valeurs possibles pour le côté gauche produit une erreur.

Dans le but de rendre un programme implémentable, il doit y avoir un ordre dans lequel les assignations peuvent être réalisées, tel qu'aucune variable n'est référencée avant d'avoir été assignée. Ce n'est par exemple pas le cas lorsqu'il y a une dépendance circulaire parmi les assignations dans un processus donné. Donc, une dépendance circulaire produit une erreur. De plus, le fait d'assigner plusieurs fois une variable constitue également une erreur.

2.3.3. La déclaration TRANS

Le comportement de transition d'un modèle est défini par une formule booléenne, fonction des variables représentant les valeurs courante et suivante des variables du programme. Cette formule peut être spécifiée directement en utilisant la déclaration TRANS. La syntaxe en est :

```
decl :: «TRANS» expr
```

La valeur courante d'une variable `x` est simplement représentée par `x`, et sa valeur suivante est représentée par `next(x)`. Une expression ne peut fournir aucune autre valeur que 0 ou 1. S'il y a plusieurs déclarations TRANS, la relation de transition résultante est la conjonction de toutes celles-ci.

2.3.4. La déclaration INIT

La situation initiale du modèle est déterminée par une expression booléenne précédée du mot-clé INIT. La syntaxe suit :

```
decl :: «INIT» expr
```

L'expression ne peut contenir l'opérateur «`next`» et doit fournir une valeur 0 ou 1. Dans le cas de plusieurs déclarations INIT, la résultante est la conjonction de celles-ci.

2.3.5. La déclaration SPEC

La spécification du système est donnée par une formule de la logique temporelle CTL, introduite par le mot-clé SPEC. La syntaxe de cette déclaration est

```
decl :: «SPEC» cltform
```

Une formule CTL a la syntaxe

```
ctlform ::
        expr                          ;; une expression booléenne
        | «!» cltform                 ;; non logique
        | cltform1 «&» cltform2       ;; et logique
        | cltform1 «|» cltform2       ;; ou logique
        | cltform1 «->» cltform2      ;; implication logique
        | cltform1 «<->» cltform2     ;; équivalence logique
        | «E» pathform                ;; quantification existentielle de
                                         chemin
        | «A» pathform                ;; quantification universelle de
                                         chemin
```

La syntaxe d'une formule de chemin est

```
pathform ::
        «X» cltform                   ;; prochaine fois
        | «F» cltform                 ;; dans le futur
        | «G» cltform                 ;; globalement
        | cltform1 «U» cltform2       ;; jusqu'à ce que
```

L'ordre de priorité des opérateurs est, dans l'ordre décroissant :

E, A, X, F, G, U
!
&
|
->, <->

On associe à gauche les opérateurs de même priorité, et des parenthèses peuvent être utilisées pour grouper des expressions. Une formule CTL ne peut contenir l'opérateur **next()** ou renvoyer une valeur différente de 0 et de 1. S'il y a plusieurs déclarations SPEC, leur résultante est produite par la conjonction de ces dernières.

2.3.6. La déclaration FAIR

Une contrainte d'équité est une formule CTL qui est supposée être vraie infiniment souvent sur tous les chemins d'exécution équitables. Quand une spécification est évaluée, le vérificateur de modèles considère les quantificateurs de chemin comme ne s'appliquant qu'aux chemins équitables. Ces contraintes sont déclarées en utilisant la syntaxe suivante :

```
decl :: «FAIRNESS» cltform
```

Un chemin est considéré équitable si et seulement si toutes les contraintes déclarées de cette façon sont vraies infiniment souvent le long de ce chemin.

2.3.7. La déclaration DEFINE

Pour rendre les descriptions plus concises, on peut utiliser des symboles pour remplacer des expressions couramment utilisées. La syntaxe de ce type de déclaration est

```
decl :: «DEFINE»
        atom1 «:=» expr1 «;»
        atom2 «:=» expr2 «;»
        ...
```

Quand un identifiant référençant le symbole du côté gauche apparaît dans une expression, il est remplacé par la **valeur** de l'expression du côté droit (et pas par l'expression elle-même : en effet, dans le cas de l'instanciation des modules, les arguments sont évalués dans le contexte du module appelant, et le fait de remplacer un symbole défini par l'expression associée reviendrait à introduire des valeurs qui ne sont pas connues dans le module appelé). On permet des références en avant à des symboles de cette sorte, mais des définitions circulaires entraînent une erreur.

2.4. Modules

Un module est un ensemble de déclarations, telles que définies ci-avant. Une fois défini, un module peut être réutilisé autant de fois que nécessaire. On peut aussi paramétrer un module, de façon à ce qu'un module puisse adresser différentes valeurs de données. Un module peut contenir des instances d'autres modules, ce qui permet d'en construire une structure hiérarchique. La syntaxe suit :

```
module ::
        «MODULE» atom [ «(» atom1 «,» atom2 «,» ... «)» ]
             decl1
             decl2
             ...
```

L'atome qui suit immédiatement le mot-clé MODULE est le nom associé au module. La liste optionnelle d'atomes entre parenthèses représente les paramètres formels du module. Quand ces paramètres se retrouvent dans des expressions du module, ils sont remplacés par les paramètres effectifs au moment où le module est instancié.

Une instance de module peut être créée en utilisant une déclaration VAR. Cette déclaration permet de nommer une instance ainsi qu'une liste éventuelle de paramètres effectifs, qui sont assignés aux paramètres formels de la définition du module. Un paramètre effectif peut être une expression autorisée quelconque. Il se produit une erreur si le nombre de paramètres effectifs n'est pas le même que celui de paramètres formels. La sémantique de l'instanciation de modules relève de celle de l'appel par référence.

2.5. Identifiants

Un identifiant est une expression qui fait référence à un objet. Un objet est une instance de module, une variable ou un symbole défini. Sa syntaxe est

```
id :: atom
    | id.atom
```

Le point est utilisé pour former des identifiants qui correspondent à la position d'un objet dans la hiérarchie de modules d'un programme. Par exemple, si une instance de module est créée avec le nom `a`, alors toute occurrence d'un identifiant `x` dans le module (sauf comme paramètre effectif) devient `a.x` lors de l'instanciation. Les paramètres effectifs

ne sont pas sujets à ce préfixage, pour permettre l'effet d'appel par référence décrit ci-dessus.

2.6. Processus

Les processus sont utilisés pour modéliser une forme entrelacée de la concurrence. Une instance d'un module est spécialisée en processus à l'aide du mot-clé `process`. Chaque processus a une variable booléenne spéciale appelée `running`, qui est vraie quand le processus est actif. Un processus peut être actif seulement quand son parent est actif. De plus, deux processus issus du même parent ne peuvent être actifs simultanément. La valeur de `running` est non déterministe. Une assignation du type `ASSIGN next(v) := expr;` dans un processus s'applique uniquement quand `running` est vrai. Dans le cas contraire, on a `next(v) = v`. Il y a cependant une exception à cette règle, qui rend possible pour plus d'un processus de modifier une variable : une assignation de la forme

```
ASSIGN if expr1 then next(v) := expr2
```

s'applique seulement quand `expr1` est vraie. Dans le cas contraire, la valeur de `next(v)` est indéterminée. Une variable peut être conditionnellement assignée à plusieurs endroits, pourvu que les conditions soient mutuellement exclusives.

2.7. Programmes

La syntaxe d'un programme SMV est

```
program ::
        module1
        module2
        ...
```

Il doit exister un module sans paramètres formels appelé `main`. Ce module est instancié par le compilateur.

3. Sémantique formelle

C'est dans cette section que l'on assigne une sémantique aux programmes SMV. Comme il s'agit d'appliquer l'algorithme de vérification symbolique à ces programmes, il est naturel de décrire leur sémantique en terme de leur traduction en une formule booléenne représentant leur relation de transition. Ceci a l'avantage supplémentaire de fournir quelque intuition à propos du rôle du compilateur dans le système de vérification SMV.

Il est pratique d'introduire quelques notations. Premièrement, la notation λ sera utile pour définir la sémantique des modules. Si `x` est une variable, et `f` est une formule, alors λ`x.f` est une fonctionnelle. Cette expression est en effet une fonction d'un paramètre, qui remplacera `x` dans `f`. Si `g` est une autre formule, alors (λ`x.f`)(`g`) est équivalente au terme obtenu en remplaçant `x` par `g` dans `f`. L'effet de λ peut être résumé à travers les axiomes suivants :

$$(\lambda x.f)(g) = f \text{ si } x \text{ n'apparaît pas dans } f \qquad (3.1)$$

$$(\lambda x.x)(f) = f \tag{3.2}$$
$$(\lambda x.(\neg f))(g) = \neg\lambda x.f(g) \tag{3.3}$$
$$\lambda x.(f \vee g)(h) = (\lambda x.f(h)) \vee (\lambda x.g(h)) \tag{3.4}$$

Ces axiomes peuvent être utilisés pour éliminer les λ des formules, en les faisant passer à l'intérieur de celles-ci jusqu'à ce qu'ils rencontrent des variables.

De plus, pour manipuler des valeurs non booléennes, on introduit une notation $a \rightarrow (b,c)$, qui signifie que si a, alors b, sinon c, où b et c peuvent être des entiers ou des constantes symboliques.

Notre sémantique des programmes SMV est simplement une fonction qui fait correspondre un fragment f d'un programme SMV à une formule $[\![f]\!]$ appelée sa dénotation. La sémantique est « orientée syntaxe », en ce sens que la dénotation d'un programme est définie comme une fonction des dénotations de ses composants syntaxiques. En pratique, un programme SMV a plusieurs dénotations pour différentes utilisations. Par exemple, à partir d'un même programme, on peut dériver des formules pour sa relation de transition, ses conditions initiales, ses contraintes d'équité et sa spécification. C'est ainsi que l'on distingue $[\![f]\!]_R$ pour la relation de transition, $[\![f]\!]_I$ pour les conditions initiales, $[\![f]\!]_F$ pour les contraintes d'équité, et $[\![f]\!]_S$ pour la spécification.

3.1. Sémantique des déclarations de variables

La première catégorie syntaxique que nous considérons dans notre sémantique est la déclaration de variables. Si n est un nom, et l est une liste de decl, alors

$$[\![\text{VAR n : boolean; l}]\!] = [\![\text{l}]\!] \tag{3.5}$$

En d'autres termes, déclarer une variable comme étant de type booléen est sémantiquement neutre, et ne produit que de l'information pragmatique pour le compilateur. Dans le cas des types énumérés, nous devons utiliser des variables booléennes pour encoder chaque valeur. Ceci peut être réalisé par une fonction récursive qui divise les valeurs possibles en deux sous-ensembles de taille plus ou moins égale, crée une variable pour les distinguer, et poursuit la récursion sur les deux sous-ensembles. Par exemple, si n est un nom, i un naturel, v une constante symbolique, et L et R sont deux listes de constantes symboliques telles que $0 \leq |L| - |R| \leq 1$, alors

$$\text{encode}(n.i, v) = v$$
$$\text{encode}(n.i, (L, R)) = n.i \rightarrow (\text{encode}(n.(i+1), L), \text{encode}(n.(i+1), R))$$

Par exemple,

$$\text{encode}(a.0,(\text{bon, moyen, mauvais})) = a.0 \rightarrow (a.1 \rightarrow (\text{bon, moyen}), \text{mauvais})$$

En utilisant cette méthode, la sémantique d'une déclaration de type énuméré est

$$[\![\text{VAR n : \{L\}; l}]\!] = (n = \text{encode}(n.0, L)) \wedge [\![\text{l}]\!] \tag{3.6}$$

Donc, pour toute occurrence du nom de variable n, on peut remplacer l'expression fournissant la valeur de n par les variables d'encodage n.0, n.1, ...

Les variables de type tableaux d'entiers sont manipulées de la même façon, en transformant le vecteur spécifié en une liste de valeurs. Si n est un nom, x et y sont des entiers, et l est une liste de decl, alors

$$[\![\texttt{VAR n : x..y; l}]\!] = (\texttt{n} = \texttt{encode}(\texttt{n.0}, (\texttt{x}, \texttt{x+1}, ..., \texttt{y}))) \wedge [\![\texttt{l}]\!] \qquad (3.7)$$

3.2. Sémantique des constantes et expressions

Dans la plupart des langages, une constante est une constante, mais il n'en n'est pas de même en SMV. Une constante est une expression, et en SMV, les expressions sont non déterministes, dans le sens où elles peuvent renvoyer une valeur choisie dans un ensemble. Sémantiquement, on considère qu'une expression renvoie un ensemble de valeurs, desquelles une peut être choisie par une instruction ASSIGN. Une constante c est donc considérée comme une abréviation syntaxique de {c}. Comme une expression est une formule, le moyen le plus simple pour manipuler la sémantique des expressions est de considérer qu'une expression se dénote elle-même, et donc, que si e est une expr, alors

$$[\![\texttt{e}]\!] = \texttt{e} \qquad (3.8)$$

Dans ce cas, nous avons besoin d'un ensemble d'axiomes permettant d'éliminer les nombreux opérateurs non booléens des formules. Une façon particulièrement utile de réaliser cela correspond exactement à la façon dont la représentation sous forme d'OBDD est construite. Si o est un opérateur de l'ensemble +, -, *, /, mod, >, >=, <, <=, &, |, ->, <->, alors

$$\texttt{n} = \texttt{n} \rightarrow (1, 0) \text{ si n est une variable} \qquad (3.9)$$

$$\texttt{n} \rightarrow (\texttt{x}, \texttt{x}) = \texttt{x} \qquad (3.10)$$

$$(\texttt{x} \rightarrow (\texttt{y}, \texttt{z})) \circ \texttt{a} = \texttt{x} \rightarrow (\lambda\texttt{x}.(\texttt{y} \circ \texttt{a})\ 1, \lambda\texttt{x}.(\texttt{z} \circ \texttt{a})\ 0) \qquad (3.11)$$

$$\texttt{a} \circ (\texttt{x} \rightarrow (\texttt{y}, \texttt{z})) = \texttt{x} \rightarrow (\lambda\texttt{x}.(\texttt{a} \circ \texttt{y})\ 1, \lambda\texttt{x}.(\texttt{a} \circ \texttt{z})\ 0) \qquad (3.12)$$

$$(\texttt{x} \rightarrow (\texttt{y}, \texttt{z})) \circ (\texttt{x} \rightarrow (\texttt{a}, \texttt{b})) = \texttt{x} \rightarrow (\lambda\texttt{x}.(\texttt{y} \circ \texttt{a})\ 1, \lambda\texttt{x}.(\texttt{z} \circ \texttt{b})\ 0) \qquad (3.13)$$

Le fait de décider d'un ordre total sur les variables, et de choisir lequel des 3 derniers axiomes on va appliquer, selon l'ordre des variables se présentant de part et d'autre de l'opérateur o, produit exactement la forme OBDD.

Les axiomes ci-dessus peuvent être utilisés pour réduire une expression à une forme dans laquelle les opérateurs SMV se présentent systématiquement dans des sous-expressions constantes, qui peuvent être réduites à des constantes en évaluant les fonctions de base. Ceci nous donne un moyen efficace pour définir la sémantique des expressions en termes de formules booléennes.

3.3. Les fonctions de base

Les fonctions de base sont +, -, *, /, et sont les fonctions usuelles en arithmétique modulo 2^{32}. La fonction de base dénotée par mod est le reste positif d'une

division modulo 2^{32}. Les fonctions de base dénotées par les opérateurs >, >=, < et <= sont les opérateurs arithmétiques de comparaison classiques, renvoient 0 quand la relation est fausse et un quand elle est vraie. Toutes ces fonctions sont exclusivement définies pour des arguments numériques. Quand elles reçoivent des arguments non numériques, elles renvoient {0, 1}. L'opérateur d'égalité est défini pour tous les types d'arguments, et renvoie 0 (1) en cas d'inégalité (d'égalité). Les fonctions dénotées par les opérateurs booléens sont &, |, !, ->, <-> (respectivement : et, ou, non, implication, équivalence), elles sont définies pour les seuls arguments 0 et 1, ont leurs tables de vérité habituelles et renvoient {0, 1} pour les autres types d'argument.

Quand les fonctions de base ci-dessus reçoivent comme arguments des ensembles, elles fournissent l'image de leur produit cartésien par la fonction. C'est-à-dire que si S_1 et S_2 sont des ensembles de valeurs, alors $f(S_1, S_2) = \{f(x, y) \mid (x, y) \in S_1 \times S_2\}$.

La fonction `union` prend deux ensembles quelconques et renvoie leur union. La fonction `in` prend deux ensembles quelconques et renvoie 1 quand l'argument de gauche est inclus dans celui de droite, et 0 sinon.

3.4. Sémantique de ASSIGN

Assigner la valeur d'une expression à une variable est différent en SMV par rapport à d'autres langages, car les expressions renvoient des ensembles de valeurs possibles, et non une valeur unique. Par exemple, si on écrit `v := e` en SMV, on veut en fait dire que $v \in e$, c'est-à-dire que la valeur assignée à `v` est choisie dans l'ensemble renvoyé par `e`, mais n'est pas autrement déterminée. Il y a trois types d'assignations, selon que c'est la valeur initiale, courante ou suivante d'une variable qui est assignée. Dans le premier cas, si `n` est un nom, `e` est une expression, et `l` une liste de `decl`, alors

$$[\![\texttt{ASSIGN init(n) := e; l}]\!]_I = n \in [\![e]\!] \wedge [\![l]\!] \tag{3.14}$$

On note que l'on définit la dénotation d'une liste de déclarations par la conjonction des dénotations des éléments de la liste. On pourrait omettre d'expliciter le fait que les assignations de valeurs initiales sont neutres par rapport à la relation de transition, les contraintes d'équité et la spécification :

$$[\![\texttt{ASSIGN init(n) := e; l}]\!]_Z = [\![l]\!] \tag{3.15}$$

où `Z` est `R`, `F` ou `S`.

Dans le cas de l'assignation de la valeur courante d'une variable, on a :

$$[\![\texttt{ASSIGN n := e; l}]\!]_R = n \in [\![e]\!] \wedge [\![l]\!] \tag{3.16}$$

Le cas de l'assignation de la valeur suivante d'une variable est compliqué par le fait que cette assignation ne peut se produire que quand le processus qui la contient est actif. On a donc :

$$[\![\texttt{ASSIGN next(n) := e; l}]\!]_R = (n' \in (\texttt{running} \rightarrow ([\![e]\!], n))) \wedge [\![l]\!] \tag{3.17}$$

Une assignation conditionnelle se traite comme suit :

$$\begin{gathered}[\![\texttt{ASSIGN if c then next(n) := e; l}]\!]_R \\ = \\ (c \Rightarrow n' \in (\texttt{running} \rightarrow ([\![e]\!], n))) \wedge [\![l]\!]\end{gathered} \tag{3.18}$$

Comme signalé au point 2.6, n' reste indéfini quand un processus p est actif, permettant à p de modifier la valeur de n.

On peut aussi préciser que

$$[\![\texttt{ASSIGN n := e; l}]\!]_Z = [\![l]\!] \tag{3.19}$$

$$[\![\texttt{ASSIGN next(n) := e; l}]\!]_Z = [\![l]\!] \tag{3.20}$$

où Z est I, S ou F.

3.5. Sémantique de DEFINE

L'instruction DEFINE est semblable à ASSIGN, sauf qu'elle assigne une valeur de façon déterministe. Quand on écrit v := e, on veut vraiment dire que v = e. Donc,

$$[\![\texttt{DEFINE n := e; l}]\!]_R = (n = [\![e]\!]) \wedge [\![l]\!] \tag{3.21}$$

Ceci permet au compilateur d'utiliser l'axiome de substitution :

$$(v = e) \wedge f \Rightarrow \lambda v.f(e) \tag{3.22}$$

pour éliminer la variable v et la traiter comme une abréviation syntaxique de l'expression e.

3.6. Sémantique de MODULE

Une déclaration de module consiste en un nom et une liste paramétrée de déclarations. Si n, p_1, p_2, ..., p_i sont des noms, l est une liste de decl et m une liste de module, alors

$$[\![\texttt{MODULE } n(p_1, p_2, ..., p_i)\ l\ m]\!] = (n = \lambda p_1.\lambda p_2. \ldots .\lambda p_i.l) \wedge [\![m]\!] \tag{3.23}$$

Pour traiter l'instanciation des modules, il est utile d'introduire quelques notations pour pouvoir construire les noms dans le style de SMV, utilisant la notation « dot ». On réalisera cela en introduisant des axiomes permettant de rapprocher le « point » des noms de variables :

$$a.(\neg b) = \neg(a.b) \tag{3.24}$$

$$a.(b \vee c) = (a.b) \vee (a.c) \tag{3.25}$$

$$a.c = c \text{ si } c \text{ est une constante} \tag{3.26}$$

Il est aussi nécessaire de disposer de parenthèses particulières pour protéger certaines expressions de l'opérateur « point » :

$$a.\langle b\rangle = b \tag{3.27}$$

Un module est instancié par une forme spéciale de déclaration VAR. Si `k` et `n` sont des noms, si $e_1, e_2, ..., e_i$ est une liste de `expr`, et si `l` est une liste de `decl`, alors

$$[\![\text{VAR } k : n(e_1, e_2, ..., e_i);\ l]\!] = k.\lambda \texttt{running}.n\langle e_1\rangle\langle e_2\rangle ... \langle e_i\rangle\langle \texttt{running}\rangle \wedge [\![l]\!] \tag{3.28}$$

En d'autres mots, on obtient une instanciation en remplaçant les paramètres formels $p_1, p_2, ..., p_i$ par les paramètres effectifs $e_1, e_2, ..., e_i$, et en préfixant le résultat par le nom de l'instance `k`. Notons l'usage des ⟨⟩ pour éviter que les arguments soient préfixés par `k`. En termes de sémantique, ceci signifie que les arguments sont évalués dans le contexte du module appelant plutôt qu'appelé, rendant l'effet d'un passage par référence. Notons que la variable `running` est implicitement un paramètre de l'instance et que l'instance hérite de la variable `running` de son parent, protégée par ⟨⟩. Une instance est donc active si et seulement si son parent est actif.

3.7. Sémantique de `process`

Un processus, par contre, est une instance de processus qui n'hérite pas de la variable `running` de son parent, bien qu'il hérite de certaines caractéristiques de cette variable. Un processus peut être actif seulement quand son parent est actif, et pas plus d'un processus d'un certain parent ne peut être actif à un instant donné. Pour transposer cet effet dans la sémantique, on peut introduire une autre variable appelée `blocked`. Un processus actif se sert de cette variable pour empêcher d'autres processus du même parent de devenir actifs. On a :

$$\begin{aligned}[\![\text{VAR } \texttt{process}\ k : n(e_1, e_2, ..., e_i);\ l]\!] = {} & k.n\langle e_1\rangle\langle e_2\rangle ... \langle e_i\rangle \\ & \wedge (k.\texttt{running} \Rightarrow \neg\texttt{blocked} \wedge \texttt{running}) \\ & \wedge \lambda\texttt{blocked}.[\![l]\!](\texttt{blocked} \vee k.\texttt{running})\end{aligned} \tag{3.29}$$

Par cette définition, un processus cède la priorité aux processus définis au-dessus de lui, et est prioritaire par rapport à ceux définis sous lui. Même si le choix d'être actif ou non est non déterministe, tous les processus d'un parent donné ont la possibilité d'être actifs à un moment donné. Néanmoins, il est également possible qu'à un moment donné, aucun processus ne soit actif.

3.8. TRANS, INIT, FAIR et SPEC

Ces quatre déclarations correspondent aux quatre dénotations d'un programme SMV : la relation de transition, les conditions initiales, les contraintes d'équité, et les spécifications. Leur sémantique est définie ci-dessous, où `e` est une expression et `l` une liste de `decl` :

$$[\![\text{TRANS } e;\ l]\!]_R = [\![e]\!] \wedge [\![l]\!] \tag{3.30}$$

$$[\![\text{INIT } e;\ l]\!]_I = [\![e]\!] \wedge [\![l]\!] \tag{3.31}$$

$$[\![\text{FAIRNESS } e;\ l]\!]_F = (\mathtt{GF}[\![e]\!]) \wedge [\![l]\!] \tag{3.32}$$

$$[\![\text{SPEC } e;\ l]\!]_S = [\![e]\!] \wedge [\![l]\!] \tag{3.33}$$

Notons que dans le cas de FAIRNESS, la déclaration est légèrement différente, car la contrainte d'équité est une conjonction de formules de la forme `GFf`, signifiant que `f` est vraie infiniment souvent. Pour les catégories dénotationnelles non définies ci-dessus, ces déclarations sont sémantiquement neutres.

3.9. Sémantique des programmes

Un programme est simplement une liste de modules suivie d'un signe End-Of-File. Sa dénotation est celle d'un module sans paramètres appelé `main`. Donc, si `m` est une liste de `module`, alors

$$[\![m\ \mathbf{EOF}]\!] = [\![m]\!] \wedge \mathtt{main} \wedge \mathtt{running} \tag{3.34}$$

La dernière conjonction suggère l'effet d'un univers clos, en exigeant que le programme soit toujours actif, et donc jamais interrompu par un processus extérieur.

La condition de correction d'un programme SMV `p` peut être écrite succinctement comme suit :

$$[\![p]\!]_R \models [\![p]\!]_I \wedge [\![p]\!]_F \Rightarrow [\![p]\!]_S \tag{3.35}$$

par laquelle on veut dire que $[\![p]\!]_I \wedge [\![p]\!]_F \Rightarrow [\![p]\!]_S$ est vraie dans le modèle symbolique dont la relation de transition est $[\![p]\!]_R$. Cette condition est exactement ce que le vérificateur de modèles SMV calcule.

4. Utilisation de SMV

L'exécution du programme SMV sur du code SMV `p` syntaxiquement correct produit un fichier qui contient, pour chaque spécification de `p`, sa valeur de vérité. De plus, pour chaque spécification fausse, le fichier de sortie contient également un scénario qui, à partir des conditions initiales de `p`, montre une succession de transitions qui contredit cette spécification. On trouvera en annexe 2 un exemple de fichier de sortie.

Chapitre 4

Features

Les attentes que l'on a des services offerts par un logiciel sont susceptibles d'évoluer avec le temps, par exemple à cause de nouveaux desiderata de la part de l'utilisateur, ou par le souci de son concepteur d'offrir à chaque profil d'utilisateur des fonctionnalités adaptées.

En programmation classique, ce fait résulte principalement en un processus de maintenance adaptive, d'une part, et de maintenance évolutive, d'autre part. Dans le premier cas, ou entend plutôt une transformation du produit suite à un changement dans la façon d'utiliser les services existants, et dans le second cas, on suggère plutôt la création de nouveaux services. Dans les deux cas, le comportement du système résultant est susceptible de ne pas être celui attendu, et le doute qui règne à ce propos ne peut pas être intégralement levé, même éventuellement suite à une séquence de tests qui par nature ne peut être exhaustive.

Il serait d'autre part intéressant de pouvoir parler de programmes en termes d'un composant de base, bien défini, et de diverses fonctionnalités que l'on pourrait lui ajouter, et qui auraient pour effet d'étendre, de modifier ou de restreindre le comportement du composant de base. On devrait facilement pouvoir ajouter ou supprimer ces extensions, manoeuvres qui résulteraient toujours en un produit bien défini, et dont la spécification serait une fonction de la spécification du produit de base et de celles de ses features. Cette façon d'envisager le développement et l'évolution de logiciels pourrait s'appeler « approche orientée feature[9] ».

La conjonction d'une approche orientée feature et de la technique de vérification de modèles pourrait fournir une solution au problème de l'évolution de certains

[9] Aucune des traductions de ce terme anglais ne s'avère satisfaisante : il peut signifier service, fonctionnalité caractéristique ou parfois même propriété. Nous nous garderons donc de tenter une traduction, forcément réductrice.

types de programmes, et nous allons dans ce chapitre présenter la notion de feature pour un langage particulier, SMV. Au cours de ce chapitre, nous allons donner la définition de ce que pourrait être un feature, et quelques pistes seront fournies quant à la façon d'ajouter un feature à un système de base. Nous tenterons, dans le chapitre suivant, une approche de l'interaction entre features.

1. Approche intuitive

Un feature est donc vu comme un complément, une option que l'on voudrait ajouter à un système. Comme exemple dans le domaine de la téléphonie, on pourrait avoir le transfert d'appel inconditionnel, le transfert d'appel sur ligne occupée, la notification de correspondant libre, etc. qui sont des features potentiels pour un système téléphonique. En matière d'ascenseurs, on peut citer : l'absence de service en cas de surcharge ou au contraire si l'ascenseur est vide, la priorité pour les services demandés de l'intérieur de l'ascenseur, le parcage de l'ascenseur en cas d'inactivité, etc.

Nous avons vu qu'en SMV, on décrit un système par sa relation de transition et ses états initiaux, et sa spécification par des formules de la logique temporelle CTL. SMV permet aussi une structuration modulaire des systèmes.

Il est donc naturel d'envisager un feature de façon similaire* : il doit être possible d'y décrire de nouvelles variables, en vue de disposer de nouveaux états, et de pouvoir établir les transitions possibles entre ces états. Un feature a, au même titre qu'un module SMV, une spécification.

Dans le but de pouvoir lier le feature à un système existant, il doit exister une clause spéciale déclarant quelle variable ou définition du système de base va voir son utilisation modifiée, et à quelle condition.

Une autre clause spéciale déclarera quelles sont les variables que le système de base devra au minimum contenir, et quels doivent être leurs types.

On peut trouver ci-après le code d'un feature modifiant le comportement d'un ascenseur desservant 5 étages : on souhaite qu'il ne se produise pas de réponse suite à un appel issu de la cabine (`car`) de l'ascenseur, quand celle-ci est vide.

La clause REQUIRE indique quelles sont les caractéristiques minimales qu'un système de base doit avoir, en vue de pouvoir y greffer ce feature. En l'occurrence, ce système devra comporter un module nommé `lift`, dans lequel sont déclarées ou définies la variable `car_call`, nécessairement de type énuméré et de domaine {0..5} (qui indique le prochain appel interne que la cabine doit servir, ou 0 s'il n'y en a pas), les variables `floorn.pressed`[10], booléennes (donnant l'état de chaque bouton d'appel dans la cage de l'ascenseur), la variable `floor`, de domaine {1..5} (indiquant l'étage courant de l'ascenseur), et la variable `door`, à valeurs dans {`open`, `close`} (représentant l'état d'ouverture de la porte).

---

* La définition syntaxique des features telle que présentée ici se base sur une idée de Mark Ryan.

[10] Notons que SMV n'accepte pas cette notation « indexée » des noms de variable, mais nous l'utiliserons néanmoins dans un but de concision.

```
FEATURE car-empty
  -- When a car is empty, car calls are not answered.

REQUIRE
  MODULE lift
    VAR car_call : {0,1,2,3,4,5} ;
        floorn.pressed : boolean ;
        door : {open, close} ;
        floor : {1,2,3,4,5} ;

INTRODUCE
  MODULE lift
    VAR car-empty : boolean;
    ASSIGN car-empty := case
                          door=closed : car-empty;
                                    1 : {0,1};
                        esac;

  -- PRESSING A LIFT BUTTON ***DOESN'T*** GUARANTEE SERVICE,
  -- if the lift is empty
  SPEC !AG (floorn.pressed & car-empty -> AF (floor=n & door=open))

  -- PRESSING A LIFT BUTTON GUARANTEES SERVICE,
  -- if the lift is not empty
  SPEC AG (floorn.pressed & !car-empty -> AF (floor=n & door=open))

  -- a lift can't become empty with doors closed.
  SPEC AG ((car-empty & (AF(door=open))) -> A [car-empty U
door=open])

CHANGE
  MODULE lift
    IF car-empty
    THEN TREAT car_call = 0
```

La clause INTRODUCE décrit les expressions de type `decl`[11] qui devront être ajoutées au module `lift`. On veut premièrement y ajouter une variable `car_empty`, de type booléen. En second lieu, on veut assigner à la valeur courante de `car_empty` l'expression

```
case
  door=closed : car-empty;
            1 : {0,1};
esac;
```

qui signifie que si `door=closed`, alors la valeur de `car_empty` est inchangée (on souhaite par là que le contenu de la cabine ne change pas quand ses portes sont fermées), et que dans les autres cas la variable prend une valeur de l'ensemble {0, 1}, représentant le contenu non déterministe de la cabine. On introduit aussi plusieurs « spécifications du feature » : le premier groupe de spécifications signifie que dans le cas où l'ascenseur est vide, aucune demande de service ne sera satisfaite. Le second groupe de spécifications exprime que si l'ascenseur n'est pas vide, alors toutes les demandes finiront par être satisfaites, et le dernier groupe représente le fait que si l'ascenseur est vide, il le reste au moins jusqu'à l'ouverture des portes. La clause CHANGE déclare un changement conditionnel de comportement, et il s'agit ici d'assigner 0 à la valeur suivante de la variable

[11] dans le sens « expression de déclaration », comme vu dans le chapitre précédent

car_call, dans le cas où car_empty est true, ou en d'autres mots, de faire « comme si » il n'y avait pas d'appel issu de la cage lorsqu'elle est vide.

On notera que les exigences en matière de structure du système de base et de noms de variables sont très fortes, en ce sens qu'il est indispensable que ce système possède des variables de type et de nom voulus, dans un module déterminé. Il est également indispensable, dans ce cas, de savoir comment une absence d'appel peut être décrite dans le système de base ; il s'agit, pour l'ascenseur de base visé[12], de la valeur 0 pour la variable car_call.

Il est à signaler que les déclarations de la clause INTRODUCE respectent rigoureusement la syntaxe SMV, ce qui suggère leur insertion directe dans le code SMV du système de base.

On notera également que rien dans le feature ne précise exactement la valeur que la variable car_empty doit prendre, on déclare simplement quand cette variable peut changer, en fonction d'éléments modélisés dans le système. On profite donc de cette caractéristique du langage SMV qui permet d'éviter de décrire un système de façon trop détaillée, en utilisant le non déterminisme.

2. Syntaxe

Les conventions lexicales s'appliquant pour les features sont identiques à celles de SMV : les atomes, les nombres, les commentaires sont similaires à la description donnée au chapitre précédent. Il en est de même pour les types, les expressions, les formules de la logique temporelle CTL et les déclarations : VAR, ASSIGN, DEFINE, TRANS, INIT, SPEC et FAIR.

On trouve ci-dessous un schéma de la syntaxe d'un feature, syntaxe qui sera détaillée immédiatement après ce schéma, de façon plus précise.

```
FEATURE atom
     REQUIRE
        {MODULE atom [(atom {,atom}*)]
          VAR {atom : type ;}*
        }*
     INTRODUCE
        {MODULE atom
          [VAR {atom : type ;}*]
          [DEFINE {atom := expr ;}*]
          [ASSIGN {[init(atom) | next(atom) | atom] := expr ;}*]
          [{TRANS expr}*]
          [{INIT expr}*]
          [{SPEC ctlform}*]
          [{FAIRNESS ctlform}*]
        }*
     CHANGE
        {MODULE atom
          {IF expr THEN
             [TREAT {atom = expr [IN atom {,atom}* ] ;}*    |
             IMPOSE {atom := expr ;}*]
          }*
        }*
```

[12] dont la description se trouve en annexe, page ? ?

où {}* signifie une répétition, [] signifie optionnel, [| |] signifie un choix

2.1. La clause REQUIRE

Pour qu'un système de base puisse recevoir un feature, il faut qu'il possède dans sa description certaines caractéristiques, et plus précisément le système doit disposer de certaines déclarations ou définitions de variables d'un type déterminé, au sein de certains modules. Ces exigences sont déclarées comme suit :

```
REQUIRE_clause :: «REQUIRE»
          «MODULE» atom [ «(» atom1 «,» atom2 «,» ... «)» ]
                VAR_decl
          «MODULE» ...
          ...
```

où `VAR_decl` a la syntaxe d'une déclaration de variable telle que vue au point 2.1 du chapitre 3.

2.2. La clause INTRODUCE

Un feature introduit en général de nouvelles variables, et un nouveau comportement de transition. De même, de nouvelles spécifications seront ajoutées à différents modules, ainsi que des contraintes d'équité (FAIR). L'ajout de ces éléments est décrit par

```
INTRODUCE_clause :: «INTRODUCE»
          «MODULE» atom
                decl
                ...
          «MODULE» ...
          ...
```

où `decl` peut être une déclaration de variable (VAR), d'assignation (ASSIGN), de définition (DEFINE), de relation de transition (TRANS), de conditions initiales (INIT), de spécification (SPEC) ou de contrainte d'équité (FAIR).

2.3. La clause CHANGE

Dans le but de modifier le comportement d'un système de base, il faut préciser les conditions de modification de ce comportement, et de quelle façon ce changement va se produire. Deux mécanismes sont proposés : dans le premier cas, on souhaite que des variables du système de base soient remplacées par des expressions, à certains endroits, et cela est réalisé par la clause TREAT. Dans le second cas, et il s'agit de la clause IMPOSE, on souhaite qu'une variable prenne, à l'instant suivant et sous conditions, la valeur d'une certaine expression. Ces transformations sont définies par :

```
CHANGE_clause :: «CHANGE»
                        «MODULE» atom
                              condition
                              ...
                        ...

condition :: «IF» cond_expr «THEN» change
```

```
change :: «TREAT» treat_block    |
          «IMPOSE» impose_block
```

où `cond_expr` est une expression conditionnelle.

```
treat_block :: treat_sentence
               ...
treat_sentence :: atom «=» expr [«IN» atom «,» atom «,» ...] «;»
```

et

```
impose_block :: impose_sentence
                ...
impose_sentence :: atom «:=» expr «;»
```

2.4. Feature

Un feature est constitué de la succession des clauses REQUIRE, INTRODUCE et CHANGE. Il est à noter que la clause INTRODUCE est facultative, dans le sens où il n'est pas indispensable d'inclure de nouvelles variables pour produire un nouveau comportement. La clause CHANGE, en effet, peut à elle seule transformer le comportement de transition d'un système. Toutefois, un feature ne comportant pas de spécification est plutôt sans intérêt.

Voici la syntaxe d'un feature :

```
feature :: «FEATURE» atom
              REQUIRE_clause
              [ INTRODUCE_clause ]
              CHANGE_clause
```

3. Sémantique opérationnelle

Nous allons présenter cet aspect d'un feature en montrant l'effet de son ajout à un système de base, qui peut être décrit simplement sous forme de manipulations syntaxiques sur le code SMV représentant ce système.

Cette sémantique suggère un certain nombre de pistes quant à la réalisation d'un « intégrateur » de features, décrivant l'effet de chacune des clauses REQUIRE, INTRODUCE et CHANGE sur un code SMV.

Soit un système S. Soit **S** le code SMV représentant S. Nous allons décrire l'effet d'un feature F sur **S**.

3.1. La clause REQUIRE

Cette clause n'apporte aucune modification à **S**. Elle est néanmoins indispensable à l'intégrateur pour déterminer si **S** dispose au minimum des déclarations mentionnées dans F, pour permettre l'ajout. A chaque module cité dans cette clause de F, doit correspondre un module de même nom dans **S**, muni éventuellement de la même liste de paramètres formels. Chaque variable typée citée sous un module requis dans F doit être

déclarée et de même type dans le module correspondant de **S**. Si tel n'est pas le cas, il y a lieu de ne pas poursuivre l'intégration du feature.

3.2. La clause INTRODUCE

C'est par cette clause facultative que l'on va pouvoir ajouter à **S**, dans chaque module visé, un certain nombre de déclarations. Il n'est pas permis de citer dans une clause INTRODUCE de `F` des modules n'apparaissant pas dans sa clause REQUIRE. En conséquence de ce choix, il n'est pas possible d'introduire dans **S** de nouveaux modules.

Dans chaque module de la clause INTRODUCE de `F` :

- Les déclarations, définitions et assignations d'atomes cités dans une déclaration VAR, DEFINE ou ASSIGN seront recopiés de la même façon dans la déclaration correspondante du module correspondant de **S**. Si cela donne lieu à une redéclaration, l'intégration se termine en produisant une erreur.
- De même, les déclarations INIT ou TRANS seront reproduites dans le module correspondant de **S**, pour autant qu'il ne se produise pas de redéclaration.
- Les déclarations SPEC seront également reproduites dans le module correspondant de **S**, sans autre contrainte.

3.3. La clause CHANGE

Tous les modules cités dans cette clause doivent apparaître dans la clause REQUIRE. Pour chaque module de **S** cité dans la clause CHANGE :

3.3.1. Cas de TREAT

On utilisera une clause TREAT lorsque l'on voudra, dans certaines circonstances, faire « comme si » une certaine variable avait une certaine valeur, mais en gardant une partie du comportement « habituel » du système. L'effet de la construction

```
IF cond THEN TREAT atom = expr IN a1, a2, ...
```

est le suivant : dans les déclarations ASSIGN, DEFINE, INIT et TRANS du module, dans la déclaration de chaque atome a_i, on remplace chaque occurrence droite de `atom` par l'expression

```
case
    cond  : expr ;
    1     : atom ;
esac
```

Cette expression, rappelons-le, vaut `expr` quand `cond` est vraie, et `atom` sinon.

Si le mot-clé « IN » n'est pas présent, ce remplacement est alors opéré dans toutes les déclarations d'atomes des déclarations d'atomes des déclarations ASSIGN, DEFINE, INIT et TRANS du module.

Il se produit une erreur quand aucun remplacement n'a été effectué, et l'intégration du feature se termine.

La syntaxe permet d'écrire plusieurs `treat_sentence` sous une même condition :

```
IF cond THEN TREAT atom1 = expr1 ; atom2 = expr2 ;
```

Dans ce cas, les remplacements de `atom1` et `atom2` par les expressions `case` associées se feront en parallèle, de sorte que l'ordre des `treat_sentence` n'est pas significatif. Pour décrire cela formellement, introduisons la notation suivante, désignant le remplacement parallèle des a_i par les e_i dans une expression φ : $\varphi[a_1 = e_1 \parallel \ldots \parallel a_n = e_n]$, et définie par

$$a_i[a_1 = e_1 \parallel \ldots \parallel a_n = e_n] = e_i$$
$$f(e_1, \ldots, e_k)\,[a_1 = e_1 \parallel \ldots \parallel a_n = e_n] = f(e_1[a_1 = e_1 \parallel \ldots \parallel a_n = e_n], \ldots, e_k[a_1 = e_1 \parallel \ldots \parallel a_n = e_n])$$

EXEMPLE

```
MODULE main
VAR
    x : boolean ;
    y : boolean ;
    z : boolean ;
ASSIGN
    init(x) := 0 ;
    init(y) := 1 ;
    init(z) := 0 ;
    next(x) := !x ;
    next(y) := x ;
    next(z) := !y ;
```

Ce module modélise un système où les variables `x`, `y` et `z` prennent successivement les valeurs suivantes :

x	0	1	0	1	0	1	0	1
y	1	0	1	0	1	0	1	0
z	0	0	1	0	1	0	1	0

Soit le feature :

```
FEATURE simple_treat
REQUIRE
    MODULE main
        VAR x, y, z : boolean ;
CHANGE
    MODULE main
        IF x=0 then TREAT y = z
```

Ce feature `simple_treat` ajouté au module `main` donne le module suivant :

```
MODULE main
VAR
```

```
    x : boolean ;
    y : boolean ;
    z : boolean ;
ASSIGN
    init(x) := 0 ;
    init(y) := 1 ;
    init(z) := 0 ;
    next(x) := !x ;
    next(y) := x ;
    next(z) := !case
                  x=0 : z ;
                  1   : y ;
                esac ;
```

où on voit que chaque valeur droite de l'atome `y` a été remplacée par une expression `case` appropriée, dans la déclaration ASSIGN. Ceci produit les valeurs :

x	0	1	0	1	0	1	0	1
y	1	0	1	0	1	0	1	0
z	0	1	1	0	1	0	1	0

3.3.2. Cas de IMPOSE

On utilisera la clause IMPOSE quand dans certaines circonstances, on souhaite « catégoriquement » que la valeur suivante d'une variable soit fixée. Le comportement du système sera souvent, dans ces circonstances, assez différent du comportement « habituel ».

L'effet de la construction

```
IF cond THEN IMPOSE atom := expr
```

est comme suit : il s'agit, dans la déclaration ASSIGN du module, de remplacer l'expression qui suit `next(atom)` par l'expression

```
case
    cond  : expr ;
    1     : [ancienne définition de atom] ;
esac
```

Si l'expression `next(atom)` n'a pas été trouvée (par exemple quand l'implémentation d'origine a voulu le non déterminisme pour `atom`), celle-ci est insérée, suivie également de l'expression `case` appropriée.

Les remplacements des atomes de plusieurs `impose_sentence` par les expressions `case` associées se feront également en parallèle. On notera que la syntaxe ne permet pas de mélanger TREAT et IMPOSE sous une même condition.

La syntaxe autorise néanmoins plusieurs conditions au sein du même module, par exemple :

```
IF c1 THEN TREAT x=expr1 ;
IF c2 THEN TREAT y=expr2 ;
```

Dans ce cas, si x est différent de y, on peut considérer indifféremment un remplacement parallèle ou séquentiel. Si ce n'est pas le cas (x et y représentent la même variable), et que les conditions c1 et c2 sont exclusives, on peut considérer les conditions dans un ordre arbitraire. On remarquera qu'il est absurde que les conditions ne soient pas mutuellement exclusives, avec néanmoins x et y identiques : ceci reviendrait à exiger qu'une variable soit à remplacer sous certaines conditions par une expression case fonction d'une certaine expression, et que sous les mêmes conditions elle soit à remplacer par une expression case fonction d'une expression différente. Le fait que les conditions ne soient pas mutuellement exclusives représente donc une erreur qui doit être détectée et signalée lors de l'intégration. SMV pourrait déterminer si les conditions peuvent être vraies simultanément, en prouvant en logique propositionnelle $\vdash_{PL} (c_1 \wedge c_2) \Leftrightarrow$ false.

EXEMPLE

```
FEATURE simple_impose
REQUIRE
    MODULE main
        VAR x, y, z : boolean ;
CHANGE
    MODULE main
        IF x=0 then impose y := z
```

Si on ajoute ce feature simple_impose au module main décrit précédemment, on obtient le module

```
MODULE main
VAR
    x : boolean ;
    y : boolean ;
    z : boolean ;
ASSIGN
    init(x) := 0 ;
    init(y) := 1 ;
    init(z) := 0 ;
    next(x) := !x ;
    next(y) := case
                 x=0 : z ;
                 1   : x ;
               esac ;
    next(z) := !y ;
```

x	0	1	0	1	0	1	0	1	0	1	0
y	1	0	1	1	1	0	1	1	1	0	1
z	0	0	1	0	0	0	1	0	0	0	1

On peut noter que les effets des deux cas de la clause change sont très différents l'un de l'autre.

4. Intégration de features

Soit s un système SMV, et f un feature.

L'intégration du feature `f` au système `s` donne un nouveau système SMV, `s'`, si :

- l'intégration n'a pas produit d'erreurs,
- les spécifications codées dans le feature sont vraies.

On dit alors que le feature `f` est intégrable au système `s`.

Les erreurs possibles lors de l'intégration sont

- l'absence dans `s` des variables citées dans la clause REQUIRE de `f`,
- la redéclaration d'atomes par la clause INTRODUCE de `f`, dans les déclarations VAR, ASSIGN, DEFINE, INIT et TRANS,
- des conditions non mutuellement exclusives dans les `condition` de la clause CHANGE d'un même module, dans le cas où on vise le même atome après un TREAT ou ASSIGN.

Même si l'intégration du feature n'a pas causé d'erreur, il se peut que ses spécifications soient fausses, et on considère alors que l'intégration n'a pas été réussie. D'autre part, il se peut que l'intégration ait rendu fausses certaines spécifications du système de base, phénomène que nous étudierons dans le prochain chapitre « interaction de features ».

5. Exemples d'intégration

On trouvera dans l'annexe 1 un exemple de programme SMV, « SBCC.SMV », modélisant un ascenseur, ainsi qu'un feature (annexe 3), « CAR_PREFERENCE », ajoutant la propriété suivante : les appels issus de l'intérieur de l'ascenseur ont priorité sur ceux provenant des étages. On trouvera également le système combiné avec ce feature, « SBCC5_CP.SMV » (annexe 4), et le fichier de sortie SMV correspondant : « SBCC5_CP.OUT » (annexe 5).

6. Conclusion

A ce stade des investigations, la notion de feature n'est pas très satisfaisante. Un feature n'a pas cet aspect de généricité que l'on pourrait attendre de lui, dans le sens où il est fortement dépendant de la façon dont le système auquel on veut l'ajouter est décrit, ou autrement dit, si on a deux implémentations différentes d'un même système, un feature écrit d'un façon donnée ne pourra pas être ajouté indifféremment à l'une ou à l'autre des implémentations. Pour illustrer cette dépendance, un feature correct pour un système (d'ascenseurs par exemple) ne s'intégrera pas dans un système similaire où l'on aurait simplement renommé des variables (*elevator* à la place de *lift*, par exemple !). Ceci fait que l'écriture d'un feature passe par une bonne connaissance du système de base visé, non seulement au niveau fonctionnel, mais également au niveau des détails d'implémentation.

Néanmoins, l'expérience a montré qu'à ce stade déjà, il est possible de décrire un certain nombre de fonctionnalités sous forme de feature, de les ajouter isolément ou successivement, et ce avec succès.

Chapitre 5

Interaction de features

Le téléphone représente un bon exemple de système en évolution permanente dans le monde des télécommunications. Le système de base (POTS, pour *Plain Old Telephone service*) a connu de nombreuses améliorations apportant de nouvelles fonctionnalités aux utilisateurs (déviation d'appel, par exemple) et au fournisseur de service. Le fait d'ajouter un nouveau feature à ce système peut donc influencer les autres features déjà présents, particulièrement de manière négative.

La notion qui est au centre de l'interaction de features est celle de « comportement » d'un feature. En effet, pour qu'il se produise une interaction, il faut que le comportement d'un feature soit changé. De plus, lorsqu'il faut décider si un comportement a changé, on peut envisager plusieurs aspects du comportement. Il y a les aspects fonctionnels, représentés par la relation de transition, et ceux non fonctionnels, notamment les aspects de performance ou de temps réel. Nous n'envisagerons pas ces derniers aspects, et réduirons la notion d'interaction de features aux seuls aspects fonctionnels.

Nous avons au cours du chapitre précédent défini formellement la notion de feature, et nous allons, dans cette partie, exposer formellement plusieurs définitions d'interaction entre features. Nous allons également montrer comment le vérificateur SMV peut être utile pour mettre en évidence de quelle façon l'ajout d'un feature peut avoir une influence sur d'autres features précédemment ajoutés.

Il y a plusieurs manières d'envisager une interaction entre features, qui peuvent chacune correspondre à une idée intuitive que l'on en a. La première approche envisagée ici se base sur les spécifications du système de base et des features pour décider si ces features interagissent, et offre plusieurs définitions formelles de ce qu'est l'interaction. La seconde approche étudie le comportement de systèmes au niveau de leur relation de transition, et propose en ces termes une définition d'interaction.

1. L'approche orientée spécifications

Soit $\oplus$: Systèmes X Features $\rightarrow$ Systèmes la fonction réalisée par l'intégrateur de features, qui à un système et à un feature fait correspondre un nouveau système,

Soit S : Systèmes $\cup$ Features $\rightarrow$ Spécifications la fonction qui à un système ou un feature fait correspondre l'ensemble de ses spécifications,

Soit V : Spécifications X Systèmes $\rightarrow$ {true, false}* la fonction qui pour un système donné, associe à sa suite de spécifications la suite de leurs valeurs de vérité, et c'est la fonction que réalise un vérificateur de modèles,

Soit s un système, et f_1 et f_2 deux features séparément intégrables au système s, c'est-à-dire dont l'intégration n'a pas provoqué d'erreur et dont les spécifications respectives sont vraies, ou encore que $V[S(f_1), s \oplus f_1]$ et $V[S(f_2), s \oplus f_2]$ ne contiennent pas `false` :

Définition 1 : on dit que f_1 et f_2 interagissent si l'un des trois cas suivants se présente :

$$V[S(s), (s \oplus f_1) \oplus f_2] \neq V[S(s), (s \oplus f_2) \oplus f_1] \quad (5.1)$$

$$V[S(f_1) \cup S(f_2), (s \oplus f_1) \oplus f_2] \text{ contient } \texttt{false} \quad (5.2)$$

$$V[S(f_1) \cup S(f_2), (s \oplus f_2) \oplus f_1] \text{ contient } \texttt{false} \quad (5.3)$$

Dans le cas contraire, on dit que les deux features n'interagissent pas.

Informellement, il y a interaction quand l'ordre d'intégration des features a une incidence sur les spécifications du système de base, ou si dans un des ordres d'intégration considéré, au moins une des spécifications des features ajoutés n'est pas vérifiée.

Cette définition de l'interaction a la particularité qu'elle ne tient compte que des spécifications qui ont été explicitement citées tant dans le système de base que dans les features, et elle ignore les différences entre les systèmes $(s \oplus f_1) \oplus f_2$ et $(s \oplus f_2) \oplus f_1$, au niveau de leur relation de transition. Malheureusement, si le système de base et les features contiennent peu ou pas de spécifications, il y a beaucoup de chances que les features soient déclarés « sans interaction », alors que les deux systèmes $(s \oplus f_1) \oplus f_2$ et $(s \oplus f_2) \oplus f_1$ peuvent différer sensiblement. Néanmoins, ce procédé a d'une part l'avantage d'être entièrement automatique. D'autre part, si on estime que les spécifications explicites données dans le système et les features sont suffisantes pour les caractériser, alors cette définition est satisfaisante.

On peut rendre la définition 1 moins exigeante en affaiblissant les conditions d'interaction, en disant que :

Définition 2 : les systèmes n'interagissent pas dès que

$$V[S(s), (s \oplus f_1) \oplus f_2] = V[S(s), (s \oplus f_2) \oplus f_1] \quad (5.4)$$

et que l'on a trouvé un ordre d'ajout des features tel que l'un des cas suivants se présente :

$$V[S(f_1) \cup S(f_2), (s \oplus f_1) \oplus f_2] \in \{true\}^* \text{ ou} \quad (5.5)$$
$$V[S(f_1) \cup S(f_2), (s \oplus f_2) \oplus f_1] \in \{true\}^* \quad (5.6)$$

Ceci revient à dire que dans les mêmes conditions que précédemment concernant les spécifications du système de base fixées par (5.1), on n'exige plus que les spécifications des features soient vraies pour tout ordre d'ajout, il suffit de trouver un seul ordre qui les rende vraies pour déclarer qu'il n'y a pas d'interaction.

On peut modifier à nouveau la définition 1 de façon à pouvoir envisager, dans le système de base, deux types de spécifications.

Soit S_B : Systèmes $\rightarrow$ Spécifications, une fonction qui à un système fait correspondre l'ensemble des spécifications qui lui sont « fondamentales », et S_O une fonction similaire qui à un système fait correspondre les spécifications qui lui sont optionnelles, utiles mais pas indispensables, et telles que $S_B(s)$ et $S_O(s)$ forment une partition de $S(s)$, pour tout système s. On dira que

Définition 3 : f_1 et f_2 interagissent si une des suites $V[S_B(s),(s \oplus f_1) \oplus f_2]$, $V[S_B(s),(s \oplus f_2) \oplus f_1)]$ contient la valeur de vérité `false`, ou si un des cas (5.2) ou (5.3) se présente.

Cette définition nécessite encore que le système et ses features soient spécifiés de façon assez précise, mais présente l'avantage de ne détecter une interaction que lorsque l'ajout viole une des propriétés essentielles du système.

On peut, de la même manière que pour la définition 2, assouplir la définition 3 de façon à ce que

Définition 4 : on ne déclare pas d'interaction lorsque

$$V[S(f_1) \cup S(f_2), (s \oplus f_1) \oplus f_2] \in \{true\}^* \text{ ou} \quad (5.7)$$
$$V[S(f_1) \cup S(f_2), (s \oplus f_2) \oplus f_1] \in \{true\}^* \quad (5.8)$$

2. L'approche orientée relation de transition*

Dans l'approche orientée relation de transition, on ne définit plus l'interaction en termes de la validité de certaines spécifications, mais bien en termes d'effets sur la relation de transition d'un système.

Soit $A = (S^g, T, s_0)$, l'automate global décrivant le système de base, où S^g est un espace d'états, T un ensemble de transitions et s_0 un état initial.

* Sur base de [Bre95]

Une exécution de l'automate global est une séquence d'états qui peut être construite à partir de l'ensemble des transitions, et démarrant à l'état initial.

Dans cette approche, on considère qu'il doit être possible d'associer chaque transition avec un seul feature[13], de telle sorte que chaque transition contienne une fonctionnalité d'un seul feature. Avec pour conséquence que l'on ne modifie jamais une transition existante, mais qu'on ne fait qu'ajouter des transitions entièrement nouvelles.

La seule façon d'étendre un automate global, dans ce formalisme, est d'ajouter de nouvelles exécutions possibles, et donc le système étendu contient toutes les exécutions possibles du système de base.

On définit un feature $f = (\phi^g, T^e, d^0)$ pour un automate global par une fonction ϕ^g d'extension des états, un ensemble de nouvelles transitions T^e et un nouvel état initial possible d^0. Un feature est donc bien un incrément sur les comportements possibles.

Considérons différents cas d'interaction. S'il existe une exécution c, dont les transitions sont partiellement associées à un feature f, et partiellement avec un feature g, alors cette exécution est possible seulement si les deux features f et g sont présents dans l'automate global. On ne peut pas ajouter les deux features indépendamment à l'automate sans influencer le comportement de l'autre feature. Donc, les deux features interagissent.

La condition qu'une exécution est associée à deux features différents (et différents du système de base B), implique une interaction de features, mais le contraire n'est pas vrai. Il peut exister deux exécutions différentes appartenant à deux features différents, mais ces exécutions ont un préfixe commun appartenant au système de base, préfixe tel qu'il n'est pas décidé quelle exécution va se produire ensuite. S'il y a un tel choix non déterministe, alors il est possible que le comportement d'origine du premier feature ne se produise plus. Si ce choix est seulement causé par l'addition du second feature, alors il s'agit d'une interaction.

Définition 5 : Un feature f de l'automate global A interagit avec un feature g dans A si et seulement si $f \neq g$ et qu'il existe une exécution c^f qui comprend au moins une transition de f, et une exécution c^g qui comprend au moins une transition de g, et un indice i naturel tel que les préfixes de c^f et c^g jusqu'au i^e élément sont égaux (autrement dit que $\langle c_0{}^f, ..., c_i{}^f\rangle = \langle c_0{}^g, ..., c_i{}^g\rangle$), et que la transition $(c_i{}^f, c_{i+1}{}^f)$ appartient au feature f.

On définit aussi l'interaction entre plusieurs features de la façon suivante, qui se base sur la définition 5 :

Définition 6 : Une interaction entre les features d'un automate global se A se produit si et seulement si il existe deux features f et g dans A tels que f interfère avec g et où f et g ne sont pas le système de base B.

[13] Dans ce sens, le système de base est aussi considéré comme un feature.

3. Conclusion

Dans la première approche, la notion d'interaction s'exprime en termes de spécifications, qui sont les propriétés que l'on attend des systèmes et des features. Ces définitions présentent donc l'avantage de définir une interférence entre features en se basant sur des concepts qui sont exprimables de façon concrète, ou autrement dit dans un langage proche de l'utilisateur. Mais cette approche ne détecte pas d'interaction entre features si peu de spécifications sont explicitées.

L'approche par la relation de transition déclarera si une interaction s'est produite en se basant sur la description détaillée d'un système, et dès lors sera plus sera plus sensible à des interférences entre features. D'autre part, elle explique l'interaction en des termes très bas niveau, qui ne font pas partie du vocabulaire de l'utilisateur.

A défaut d'un consensus sur la notion d'interaction entre features (même sur la notion intuitive), chacun pourra, selon ses besoins, privilégier une des définitions : haut niveau, orientée spécification, ou au contraire bas niveau, orientée transition. A moins de préférer une formalisation intermédiaire ...

Conclusion

Au long de ce mémoire, nous avons montré l'intérêt et la faisabilité d'une approche « orientée feature » de la programmation. Nous avons envisagé cette approche dans un contexte restreint, qui est celui de la modélisation de systèmes sous forme de machines à états finis, et dont on peut décrire les spécifications par des formules de la logique temporelle propositionnelle.

Il a été montré de quelle façon il est possible de contrôler qu'un système, sous la forme de diagramme état-transition, vérifie sa spécification, grâce à un algorithme de vérification de modèles, le *model checking*.

Nous avons ensuite exposé comment rendre cet algorithme plus efficace et applicable à des systèmes de taille réaliste, en transformant la structure du système en une formule booléenne, pour arriver à la vérification symbolique de modèles, le *symbolic model checking*.

Ensuite, nous avons décrit la syntaxe et la sémantique d'un langage, SMV, implémentant cette technique efficace de vérification de modèles. SMV permet de décrire des systèmes finis tant synchrones qu'asynchrones, de façon modulaire, et il dispose de types de données structurés. Ce langage permet de décrire ces systèmes à un suffisamment haut niveau, et permet d'exprimer des choix non déterministes.

Par la suite, nous avons introduit le concept de *feature* pour le langage SMV, et exprimé sa syntaxe et sa sémantique. La manière de le décrire et de l'intégrer permet l'ajout, la suppression ou la réorganisation d'une collection de features pour un système.

Une autre notion accompagne naturellement celle de feature, il s'agit de l'interaction entre features. Nous en avons donné plusieurs définitions, tout en les critiquant. Nous avons finalement, pour ces définitions, montré comment un vérificateur de modèles tel que SMV et un intégrateur de feature tel que celui suggéré peuvent détecter les certaines interférences entre features.

Bibliographie

[Be96] M. Berry. Proving properties of the lift system. Master's thesis, School of computer Science, University of Birmingham, 1996.

[Bre95] J. Bredeke. Formal criteria for feature interactions in telecommunication systems. Technical report, University of Kaiserslautern, Germany, 1995.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE transactions on computers*, C-35(8), 1986.

[Byr97] N. J. Byrnes. Investigating features in software systems. Technical report, School of Computer Science, University of Birmingham, 1997

[CE81a] E. M. Clarke and E. A. Emerson. Characterizing properties of parallel programs as fixpoints. In *Seventh International Colloqium on Automata, Languages, and Programming*, volume 85 of *LNCS*, 1981.

[CE81b] E. M. Clarke and E. A. Emerson. Synthesis of synchronisation skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logic of Programs : Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-verlag.

[CES86] E. M. Clarke, E. A. Emerson and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming languages and Systems*, 8(2) :244-263, 1986.

[KiVe95] K. Kimbler and H. Velthuijsen. Feature interaction benchmark. Technical report, Lund University, Sweden, 1995.

[McMi92] K. L. McMillan. Symbolic model checking. PhD Thesis, Carnegie Mellon University, june 92.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285-309, 1955.

Table des matières

Annexe 1 : Exemple de code SMV

Ce fichier contient la description d'un système d'ascenseurs, sbcc_5.smv.

```
-------------------------------------------------------
--                     sbcc_5.smv                    --
--                                                   --
--   SINGLE BUTTON COLLECTIVE CONTROL FOR 5 FLOORS   --
--                                                   --
--  Author: Mark Berry                               --
--  Date  : 18 June 1996                             --
--  Lifts:  1                                        --
--  Floors: 5                                        --
--                                                   --
--                  SBCC PRINCIPLES                  --
--                                                   --
--  The collective control principle states that a   --
--  lift should answer any calls in its current      --
--  direction of travel, in floor sequence. In the   --
--  single button case, there is only one landing call--
--  button at each floor. An SBCC lift travels upwards--
--  answering all landing and car calls in its path  --
--  until there are no more calls. It then reverses  --
--  direction and travels down, again answering any  --
--  calls in its path until there are no more, when it--
--  again reverses and travels back up. If at any    --
--  point there are no calls at all, the lift remains --
--  stationary with its doors closed.                --
--  It can easily be seen that, unlike SCAC, this    --
--  algorithm guarantees service to any registered   --
--  call. However, when a landing call is registered --
--  it may very well be answered by a lift going in  --
--  the opposite direction to that required by the   --
--  user. The user can either get in the lift anyway, --
--  or wait and catch the lift on its return journey. --
--  Down Collective Control (DCC) is very similar,   --
--  except that all landing calls are assumed to be  --
--  down calls, so the lift never stops for landing  --
--  calls on its way up.                             --
--                                                   --
--                      CONTENTS                     --
--                                                   --
--  1.  Module "main"                                --
--  2.  Module "SBCC5_lift"                          --
--  3.  Module "button"                              --
--                                                   --
-------------------------------------------------------


-------------------------------------------------------
--                  1. Module "main"                 --
--                                                   --
--  VAR                                              --
--  Declares a series of 5 landing buttons with reset --
--  conditions that the lift is at their floor with  --
--  its doors open. The 5 floors are served by one   --
--  SBCC lift.                                       --
--                                                   --
--  DEFINE                                           --
--  "main" passes to the lift (car1) the floor number --
--  of the next landing call in its current direction --
--  of travel. If there is no such call, it passes 0. --
--  In the case where 0 is passed, there may or may  --
--  not be other landing calls behind the lift, and  --
--  the lift needs to know this in order to decide   --
--  whether to park or reverse direction. Therefore, --
--  a further parameter no_call is also passed, which --
--  is true when no buttons are pressed.             --
--                                                   --
--  SPEC                                             --
--  When any landing button is pressed, SBCC will    --
--  eventually answer the call (the lift will arrive --
--  at the relevant floor and open its doors). There --
--  is no guarantee that the lift will be travelling --
--  downwards, however (which distinguishes SBCC from --
--  DCC), and the SPECs that require this extra      --
```

```
--  condition fail for SBCC.                          --
--  The final SPEC, stating that if the lift is at the--
--  top floor it will always eventually return to the --
--  bottom floor, provides trace output demonstrating --
--  the operation  of the algorithm.                  --
--                                                    --
-------------------------------------------------------

MODULE main
VAR
  landing1 : button ((car1.floor=1) & (car1.door=open),0);
  landing2 : button ((car1.floor=2) & (car1.door=open),0);
  landing3 : button ((car1.floor=3) & (car1.door=open),0);
  landing4 : button ((car1.floor=4) & (car1.door=open),0);
  landing5 : button ((car1.floor=5) & (car1.door=open),0);

  car1     : SBCC5_lift (landing_call, no_call);

DEFINE
 landing_call:=case
                car1.direction = down : case
                                         landing5.pressed & car1.floor>4 : 5;
                                         landing4.pressed & car1.floor>3 : 4;
                                         landing3.pressed & car1.floor>2 : 3;
                                         landing2.pressed & car1.floor>1 : 2;
                                         landing1.pressed                : 1;
                                         1                               : 0;
                                        esac;

                car1.direction = up   : case
                                         landing1.pressed & car1.floor<2 : 1;
                                         landing2.pressed & car1.floor<3 : 2;
                                         landing3.pressed & car1.floor<4 : 3;
                                         landing4.pressed & car1.floor<5 : 4;
                                         landing5.pressed                : 5;
                                         1                               : 0;
                                        esac;
               esac;

  no_call := (!landing1.pressed &
              !landing2.pressed &
              !landing3.pressed &
              !landing4.pressed &
              !landing5.pressed);


    -- PRESSING A LANDING BUTTON GUARANTEES SERVICE:
SPEC
  AG (landing1.pressed -> AF (car1.floor=1 & car1.door=open))
SPEC
  AG (landing2.pressed -> AF (car1.floor=2 & car1.door=open))
SPEC
  AG (landing3.pressed -> AF (car1.floor=3 & car1.door=open))
SPEC
  AG (landing4.pressed -> AF (car1.floor=4 & car1.door=open))
SPEC
  AG (landing5.pressed -> AF (car1.floor=5 & car1.door=open))

    -- PRESSING A LANDING BUTTON DOES NOT GUARANTEE SERVICE IN A
    -- DOWNWARD DIRECTION:
SPEC
  !AG (landing1.pressed ->
        AF (car1.floor=1 & car1.door=open & car1.direction=down))
SPEC
  !AG (landing2.pressed ->
        AF (car1.floor=2 & car1.door=open & car1.direction=down))
SPEC
  !AG (landing3.pressed ->
        AF (car1.floor=3 & car1.door=open & car1.direction=down))
SPEC
  !AG (landing4.pressed ->
        AF (car1.floor=4 & car1.door=open & car1.direction=down))
SPEC
  !AG (landing5.pressed ->
        AF (car1.floor=5 & car1.door=open & car1.direction=down))

    -- TRACE OUTPUT:
SPEC
 AG (car1.floor=5 -> AF (car1.floor=1))
```

```
-------------------------------------------------------
--                  2. Module "SBCC5_lift"           --
--                                                   --
--  VAR                                              --
--  The lift's current floor (1-5) is held in the    --
--  variable 'floor'. The lift's current direction of --
--  travel and the status of the doors are held in   --
--  'direction' and 'doors'.                         --
--  The lift has 5 internal buttons, denoted by      --
--  floor1, floor2, ...   These buttons are reset    --
--  when the lift is at the relevant floor with its  --
--  doors open.                                      --
--                                                   --
--  DEFINE                                           --
--  The 'idle' flag is set when the no_call parameter --
--  indicates that no landing buttons are pressed and --
--  there are also no lift buttons pressed.          --
--  'car_call' finds the next call in the lift's     --
--  current direction of travel, in the same way as  --
--  the landing call parameter was calculated. It is --
--  0 if there are no such calls.                    --
--                                                   --
--  ASSIGN                                           --
--  The doors open when the lift is at the landing   --
--  call or car call floor.                          --
--  The lift's movement is modelled by 'floor'. The  --
--  lift only moves if the doors are closed and there --
--  is a landing call or car call to be answered in  --
--  the current direction of travel: otherwise, it   --
--  stays where it is.                               --
--  The lift changes direction if it is at the top or --
--  bottom floors, or if there are no further calls  --
--  in its current direction of travel. The exception --
--  to this is when there are no calls at all to     --
--  answer, and the 'idle' flag is set, in which case --
--  the lift's direction stays the same and the lift --
--  will remain stationary, with its doors closed.   --
--                                                   --
--  SPEC                                             --
--  The first specification shows that the lift door --
--  will not necessarily re-open once it has closed. --
--  This is because of the 'idle' feature.           --
--  The second group of specification checks that the --
--  pressing of each of the lift buttons guarantees  --
--  service.                                         --
--  To demonstrate that, when there is a call ahead  --
--  of the lift, it does not change direction until it--
--  has answered the call, would require many separate--
--  specifications. Only two are given here.         --
--  The specifications showing that a lift can remain --
--  idle at any floor come in pairs: it is necessary --
--  to show that the precondition can be satisfied   --
--  for the proof to be sound.                       --
--  The next group of specifications check that the  --
--  lift does not park at floor 1.                   --
--  The final group of specifications show that a car --
--  travelling upwards may stop for landing calls.   --
--  This is not the case in a DCC system, and is one --
--  of the major differences between SBCC and DCC.   --
--                                                   --
-------------------------------------------------------

MODULE SBCC5_lift (landing_call, no_call)
VAR
  floor         : {1,2,3,4,5};
  door          : {open,closed};
  direction     : {up,down};

  floor5        : button (floor=5 & door=open,0);
  floor4        : button (floor=4 & door=open,0);
  floor3        : button (floor=3 & door=open,0);
  floor2        : button (floor=2 & door=open,0);
  floor1        : button (floor=1 & door=open,0);


DEFINE
  idle         := (no_call &
                   !floor1.pressed &
                   !floor2.pressed &
                   !floor3.pressed &
                   !floor4.pressed &
```

```
                  !floor5.pressed);

  car_call := case
               direction = down : case
                                    floor5.pressed & floor>4 : 5;
                                    floor4.pressed & floor>3 : 4;
                                    floor3.pressed & floor>2 : 3;
                                    floor2.pressed & floor>1 : 2;
                                    floor1.pressed           : 1;
                                    1                        : 0;
                                  esac;

               direction = up   : case
                                    floor1.pressed & floor<2 : 1;
                                    floor2.pressed & floor<3 : 2;
                                    floor3.pressed & floor<4 : 3;
                                    floor4.pressed & floor<5 : 4;
                                    floor5.pressed           : 5;
                                    1                        : 0;
                                  esac;
             esac;

ASSIGN
  door          := case
                     floor=car_call      : open;
                     floor=landing_call  : open;
                     1                   : closed;
                   esac;

  init (floor) := 1;
  next (floor) := case
                     door=open                   : floor;
                     car_call=0 & landing_call=0 : floor;
                     direction=up & floor<5      : floor +1;
                     direction=down & floor>1    : floor -1;
                     1                           : floor;
                   esac;


  init (direction) := down;
  next (direction) := case
                        idle          : direction;
                        floor = 5     : down;
                        floor = 1     : up;

                        car_call=0 & landing_call=0 & direction=down : up;
                        car_call=0 & landing_call=0 & direction=up   : down;

                        1             : direction;
                      esac;


    -- ONCE A DOOR CLOSES, IT MAY REMAIN CLOSED (IDLE):
SPEC
  !AG (door=closed -> AF door=open)

    -- PRESSING A LIFT BUTTON GUARANTEES SERVICE:
SPEC
  AG (floor1.pressed -> AF (floor=1 & door=open))
SPEC
  AG (floor2.pressed -> AF (floor=2 & door=open))
SPEC
  AG (floor3.pressed -> AF (floor=3 & door=open))
SPEC
  AG (floor4.pressed -> AF (floor=4 & door=open))
SPEC
  AG (floor5.pressed -> AF (floor=5 & door=open))

    -- WHEN THERE IS A CALL AHEAD OF THE LIFT, IT DOES NOT
    -- CHANGE DIRECTION UNTIL IT HAS ANSWERED THE CALL
    -- (ONLY 2 OF THE CASES ARE SPECIFIED):
SPEC
  AG (floor=2 & floor5.pressed & direction=up
                  -> A[direction=up U floor=5])
SPEC
  AG (floor=5 & floor1.pressed & direction=down
                  -> A[direction=down U floor=1])

    -- THE LIFT CAN REMAIN IDLE WITH ITS DOORS CLOSED:
SPEC
  EF(floor=1 & door=closed & idle)
```

```
SPEC
  AG (floor=1 & idle & door=closed -> EG (floor=1 & door=closed))
SPEC
  EF(floor=2 & door=closed & idle)
SPEC
  AG (floor=2 & idle & door=closed -> EG (floor=2 & door=closed))
SPEC
  EF(floor=3 & door=closed & idle)
SPEC
  AG (floor=3 & idle & door=closed -> EG (floor=3 & door=closed))
SPEC
  EF(floor=4 & door=closed & idle)
SPEC
  AG (floor=4 & idle & door=closed -> EG (floor=4 & door=closed))
SPEC
  EF(floor=5 & door=closed & idle)
SPEC
  AG (floor=5 & idle & door=closed -> EG (floor=5 & door=closed))

    -- THE LIFT DOES NOT PARK AT FLOOR 1:
SPEC
  !AG(floor=2 & idle -> E [idle U floor=1])
SPEC
  !AG(floor=3 & idle -> E [idle U floor=1])
SPEC
  !AG(floor=4 & idle -> E [idle U floor=1])
SPEC
  !AG(floor=5 & idle -> E [idle U floor=1])

    -- A CAR TRAVELLING UPWARDS MAY STOP FOR LANDING CALLS:
SPEC
  !AG ((floor=1 & !floor1.pressed & direction=up)
        -> !(door=open))
SPEC
  !AG ((floor=2 & !floor2.pressed & direction=up)
        -> !(door=open))
SPEC
  !AG ((floor=3 & !floor3.pressed & direction=up)
        -> !(door=open))
SPEC
  !AG ((floor=4 & !floor4.pressed & direction=up)
        -> !(door=open))
SPEC
  !AG ((floor=5 & !floor5.pressed & direction=up)
        -> !(door=open))


-------------------------------------------------------
--                    3. Module "button"               --
--                                                     --
--  The generic button definition, used for both car  --
--  buttons and landing buttons.                       --
--                                                     --
-------------------------------------------------------

MODULE button (reset, suppress)
VAR
  pressed : boolean;
ASSIGN
  init (pressed) := 0;
  next (pressed) := case
                      reset     : 0;
                      pressed   : 1;
                      suppress  : 0;
                      1         : {0,1};
                    esac;
```

Annexe 2 : Fichier de sortie SMV

Ce fichier représente la sortie produite par SMV lorsqu'on lui donne le fichier sbcc_5.smv en entrée.
Il comprend une ligne pour chaque spécification du fichier d'entrée, suivie de son évaluation. Dans le cas où une spécification est fausse, elle est immédiatement suivie par un scénario montrant la succession d'états qui, à partir d'une situation initiale, a permis de l'invalider.
Le fichier se termine par quelques chiffres concernant l'utilisation des ressources pour ce système.

```
-- specification !AG (door = closed -> AF door = open) (in module car1) is true
-- specification AG (floor1.pressed -> AF (floor = 1 & do... (in module car1) is true
-- specification AG (floor2.pressed -> AF (floor = 2 & do... (in module car1) is true
-- specification AG (floor3.pressed -> AF (floor = 3 & do... (in module car1) is true
-- specification AG (floor4.pressed -> AF (floor = 4 & do... (in module car1) is true
-- specification AG (floor5.pressed -> AF (floor = 5 & do... (in module car1) is true
-- specification AG (floor = 2 & floor5.pressed & directi... (in module car1) is true
-- specification AG (floor = 5 & floor1.pressed & directi... (in module car1) is true
-- specification EF (floor = 1 & door = closed & idle) (in module car1) is true
-- specification AG (floor = 1 & idle & door = closed -> ... (in module car1) is true
-- specification EF (floor = 2 & door = closed & idle) (in module car1) is true
-- specification AG (floor = 2 & idle & door = closed -> ... (in module car1) is true
-- specification EF (floor = 3 & door = closed & idle) (in module car1) is true
-- specification AG (floor = 3 & idle & door = closed -> ... (in module car1) is true
-- specification EF (floor = 4 & door = closed & idle) (in module car1) is true
-- specification AG (floor = 4 & idle & door = closed -> ... (in module car1) is true
-- specification EF (floor = 5 & door = closed & idle) (in module car1) is true
-- specification AG (floor = 5 & idle & door = closed -> ... (in module car1) is true
-- specification !AG (floor = 2 & idle -> E(idle U floor ... (in module car1) is true
-- specification !AG (floor = 3 & idle -> E(idle U floor ... (in module car1) is true
-- specification !AG (floor = 4 & idle -> E(idle U floor ... (in module car1) is true
-- specification !AG (floor = 5 & idle -> E(idle U floor ... (in module car1) is true
-- specification !AG (floor = 1 & !floor1.pressed & direc... (in module car1) is true
-- specification !AG (floor = 2 & !floor2.pressed & direc... (in module car1) is true
-- specification !AG (floor = 3 & !floor3.pressed & direc... (in module car1) is true
-- specification !AG (floor = 4 & !floor4.pressed & direc... (in module car1) is true
-- specification !AG (floor = 5 & !floor5.pressed & direc... (in module car1) is true
-- specification AG (landing1.pressed -> AF (car1.floor =... is true
-- specification AG (landing2.pressed -> AF (car1.floor =... is true
-- specification AG (landing3.pressed -> AF (car1.floor =... is true
-- specification AG (landing4.pressed -> AF (car1.floor =... is true
-- specification AG (landing5.pressed -> AF (car1.floor =... is true
-- specification !AG (landing1.pressed -> AF (car1.floor ... is true
-- specification !AG (landing2.pressed -> AF (car1.floor ... is true
-- specification !AG (landing3.pressed -> AF (car1.floor ... is true
-- specification !AG (landing4.pressed -> AF (car1.floor ... is true
-- specification !AG (landing5.pressed -> AF (car1.floor ... is true
-- specification AG (car1.floor = 5 -> AF car1.floor = 1) is false
-- as demonstrated by the following execution sequence
state 1.1:
no_call = 1
landing_call = 0
landing1.pressed = 0
landing2.pressed = 0
landing3.pressed = 0
landing4.pressed = 0
landing5.pressed = 0
car1.car_call = 0
car1.idle = 1
car1.floor = 1
car1.door = closed
car1.direction = down
car1.floor5.pressed = 0
car1.floor4.pressed = 0
car1.floor3.pressed = 0
car1.floor2.pressed = 0
car1.floor1.pressed = 0

state 1.2:
car1.idle = 0
car1.floor5.pressed = 1

state 1.3:
```

```
car1.car_call = 5
car1.direction = up

state 1.4:
car1.floor = 2

state 1.5:
car1.floor = 3

state 1.6:
car1.floor = 4

-- loop starts here --
state 1.7:
car1.floor = 5
car1.door = open

state 1.8:
car1.car_call = 4
car1.door = closed
car1.direction = down
car1.floor5.pressed = 0
car1.floor4.pressed = 1

state 1.9:
car1.floor = 4
car1.door = open

state 1.10:
car1.car_call = 0
car1.door = closed
car1.floor5.pressed = 1
car1.floor4.pressed = 0

state 1.11:
car1.car_call = 5
car1.direction = up

state 1.12:
car1.floor = 5
car1.door = open


resources used:
user time: 20.9667 s, system time: 0.316667 s
BDD nodes allocated: 12708
Bytes allocated: 1048576
BDD nodes representing transition relation: 988 + 93
```

Annexe 3 : Le feature car preference

On souhaite par l'introduction de ce feature que dans certaines circonstances, les appels issus de l'intérieur de l'ascenseur soient prioritaires sur ceux issus dans les étages.

```
FEATURE car_preference
REQUIRE
  MODULE main
    VAR car1 : SBCC5_lift ;
        landing1.pressed : boolean ;
        landing2.pressed : boolean ;
        landing3.pressed : boolean ;
        landing4.pressed : boolean ;
        landing5.pressed : boolean ;

  MODULE SBCC5_lift
    VAR floor1.pressed : boolean ;
        floor2.pressed : boolean ;
        floor3.pressed : boolean ;
        floor4.pressed : boolean ;
        floor5.pressed : boolean ;

INTRODUCE
  MODULE SBCC5_lift
    VAR cp : boolean ;
    SPEC - car calls have the preference
      AG (cp & floor2.pressed & !floor3.pressed ->
             A [ !(floor=3 & door=open) U ((floor=2 & door=open) | !cp)])

CHANGE
  MODULE main
    IF car1.cp &
             (car1.floor1.pressed |
              car1.floor2.pressed |
              car1.floor3.pressed |
              car1.floor4.pressed |
              car1.floor5.pressed)
    THEN TREAT landing1.pressed = 0 ;
               landing2.pressed = 0 ;
               landing3.pressed = 0 ;
               landing4.pressed = 0 ;
               landing5.pressed = 0 ;
```

L'effet de la clause change de ce feature est de remplacer toute occurrence droite de `landingi.pressed` du module `main`, par l'expression

```
case cp &
      (car1.floor1.pressed |
       car1.floor2.pressed |
       car1.floor3.pressed |
       car1.floor4.pressed |
       car1.floor5.pressed) : 0 ;
     1                      : landingi.pressed ;
esac
```

Annexe 4 : SBCC5_CP.SMV

Ce fichier contient la description d'un système

```
MODULE main
VAR
  landing1 : button ((car1.floor=1) & (car1.door=open),0);
  landing2 : button ((car1.floor=2) & (car1.door=open),0);
  landing3 : button ((car1.floor=3) & (car1.door=open),0);
  landing4 : button ((car1.floor=4) & (car1.door=open),0);
  landing5 : button ((car1.floor=5) & (car1.door=open),0);

  car1     : SBCC5_lift (landing_call, no_call);

DEFINE
 landing_call:=case
                car1.direction = down : case
                                        case car1.cp & (car1.floor1.pressed |
               car1.floor2.pressed |
               car1.floor3.pressed |
               car1.floor4.pressed |
               car1.floor5.pressed) : 0;
     1 : landing5.pressed;
esac & car1.floor>4 : 5;
                                        case car1.cp & (car1.floor1.pressed |
               car1.floor2.pressed |
               car1.floor3.pressed |
               car1.floor4.pressed |
               car1.floor5.pressed) : 0;
     1 : landing4.pressed;
esac & car1.floor>3 : 4;
                                        case car1.cp & (car1.floor1.pressed |
               car1.floor2.pressed |
               car1.floor3.pressed |
               car1.floor4.pressed |
               car1.floor5.pressed) : 0;
     1 : landing3.pressed;
esac & car1.floor>2 : 3;
                                        case car1.cp & (car1.floor1.pressed |
               car1.floor2.pressed |
               car1.floor3.pressed |
               car1.floor4.pressed |
               car1.floor5.pressed) : 0;
     1 : landing2.pressed;
esac & car1.floor>1 : 2;
                                        case car1.cp & (car1.floor1.pressed |
               car1.floor2.pressed |
               car1.floor3.pressed |
               car1.floor4.pressed |
               car1.floor5.pressed) : 0;
     1 : landing1.pressed;
esac                : 1;
                                        1                              : 0;
                                       esac;

                car1.direction = up   : case
                                        case car1.cp & (car1.floor1.pressed |
               car1.floor2.pressed |
               car1.floor3.pressed |
               car1.floor4.pressed |
               car1.floor5.pressed) : 0;
     1 : landing1.pressed;
esac & car1.floor<2 : 1;
                                        case car1.cp & (car1.floor1.pressed |
               car1.floor2.pressed |
               car1.floor3.pressed |
               car1.floor4.pressed |
               car1.floor5.pressed) : 0;
     1 : landing2.pressed;
esac & car1.floor<3 : 2;
                                        case car1.cp & (car1.floor1.pressed |
               car1.floor2.pressed |
               car1.floor3.pressed |
               car1.floor4.pressed |
               car1.floor5.pressed) : 0;
     1 : landing3.pressed;
```

```
esac & car1.floor<4 : 3;
                                        case car1.cp & (car1.floor1.pressed |
                car1.floor2.pressed |
                car1.floor3.pressed |
                car1.floor4.pressed |
                car1.floor5.pressed) : 0;
     1 : landing4.pressed;
esac & car1.floor<5 : 4;
                                        case car1.cp & (car1.floor1.pressed |
                car1.floor2.pressed |
                car1.floor3.pressed |
                car1.floor4.pressed |
                car1.floor5.pressed) : 0;
     1 : landing5.pressed;
esac                : 5;
                                        1                                : 0;
                                       esac;
               esac;

  no_call := (!case car1.cp & (car1.floor1.pressed |
                car1.floor2.pressed |
                car1.floor3.pressed |
                car1.floor4.pressed |
                car1.floor5.pressed) : 0;
     1 : landing1.pressed;
esac &
               !case car1.cp & (car1.floor1.pressed |
                car1.floor2.pressed |
                car1.floor3.pressed |
                car1.floor4.pressed |
                car1.floor5.pressed) : 0;
     1 : landing2.pressed;
esac &
               !case car1.cp & (car1.floor1.pressed |
                car1.floor2.pressed |
                car1.floor3.pressed |
                car1.floor4.pressed |
                car1.floor5.pressed) : 0;
     1 : landing3.pressed;
esac &
               !case car1.cp & (car1.floor1.pressed |
                car1.floor2.pressed |
                car1.floor3.pressed |
                car1.floor4.pressed |
                car1.floor5.pressed) : 0;
     1 : landing4.pressed;
esac &
               !case car1.cp & (car1.floor1.pressed |
                car1.floor2.pressed |
                car1.floor3.pressed |
                car1.floor4.pressed |
                car1.floor5.pressed) : 0;
     1 : landing5.pressed;
esac);


    -- PRESSING A LANDING BUTTON GUARANTEES SERVICE:
SPEC
  AG (landing1.pressed -> AF (car1.floor=1 & car1.door=open))
SPEC
  AG (landing2.pressed -> AF (car1.floor=2 & car1.door=open))
SPEC
  AG (landing3.pressed -> AF (car1.floor=3 & car1.door=open))
SPEC
  AG (landing4.pressed -> AF (car1.floor=4 & car1.door=open))
SPEC
  AG (landing5.pressed -> AF (car1.floor=5 & car1.door=open))

    -- PRESSING A LANDING BUTTON ***DOES NOT*** GUARANTEE SERVICE:
SPEC
  !AG (landing1.pressed -> AF (car1.floor=1 & car1.door=open))
SPEC
  !AG (landing2.pressed -> AF (car1.floor=2 & car1.door=open))
SPEC
  !AG (landing3.pressed -> AF (car1.floor=3 & car1.door=open))
SPEC
  !AG (landing4.pressed -> AF (car1.floor=4 & car1.door=open))
SPEC
  !AG (landing5.pressed -> AF (car1.floor=5 & car1.door=open))
```

```
    -- PRESSING A LANDING BUTTON DOES NOT GUARANTEE SERVICE IN A
    -- DOWNWARD DIRECTION:
SPEC
  !AG (landing1.pressed ->
        AF (car1.floor=1 & car1.door=open & car1.direction=down))
SPEC
  !AG (landing2.pressed ->
        AF (car1.floor=2 & car1.door=open & car1.direction=down))
SPEC
  !AG (landing3.pressed ->
        AF (car1.floor=3 & car1.door=open & car1.direction=down))
SPEC
  !AG (landing4.pressed ->
        AF (car1.floor=4 & car1.door=open & car1.direction=down))
SPEC
  !AG (landing5.pressed ->
        AF (car1.floor=5 & car1.door=open & car1.direction=down))

    -- TRACE OUTPUT:
SPEC
 AG (car1.floor=5 -> AF (car1.floor=1))


MODULE SBCC5_lift (landing_call, no_call)
VAR
  floor          : {1,2,3,4,5};
  door           : {open,closed};
  direction      : {up,down};

  floor5         : button (floor=5 & door=open,0);
  floor4         : button (floor=4 & door=open,0);
  floor3         : button (floor=3 & door=open,0);
  floor2         : button (floor=2 & door=open,0);
  floor1         : button (floor=1 & door=open,0);

  cp            : boolean;

DEFINE
  idle          := (no_call &
                     !floor1.pressed &
                     !floor2.pressed &
                     !floor3.pressed &
                     !floor4.pressed &
                     !floor5.pressed);

  car_call := case
                direction = down : case
                                     floor5.pressed & floor>4 : 5;
                                     floor4.pressed & floor>3 : 4;
                                     floor3.pressed & floor>2 : 3;
                                     floor2.pressed & floor>1 : 2;
                                     floor1.pressed           : 1;
                                     1                        : 0;
                                   esac;

                direction = up   : case
                                     floor1.pressed & floor<2 : 1;
                                     floor2.pressed & floor<3 : 2;
                                     floor3.pressed & floor<4 : 3;
                                     floor4.pressed & floor<5 : 4;
                                     floor5.pressed           : 5;
                                     1                        : 0;
                                   esac;
              esac;

ASSIGN
  door          := case
                     floor=car_call      : open;
                     floor=landing_call  : open;
                     1                   : closed;
                   esac;

  init (floor) := 1;
  next (floor) := case
                     door=open                   : floor;
                     car_call=0 & landing_call=0 : floor;
                     direction=up & floor<5      : floor +1;
                     direction=down & floor>1    : floor -1;
                     1                           : floor;
                   esac;
```

```
  init (direction) := down;
  next (direction) := case
                        idle          : direction;
                        floor = 5     : down;
                        floor = 1     : up;

                        car_call=0 & landing_call=0 & direction=down : up;
                        car_call=0 & landing_call=0 & direction=up   : down;

                        1             : direction;
                     esac;


    -- ONCE A DOOR CLOSES, IT MAY REMAIN CLOSED (IDLE):
SPEC
  !AG (door=closed -> AF door=open)

    -- PRESSING A LIFT BUTTON GUARANTEES SERVICE:
SPEC
  AG (floor1.pressed -> AF (floor=1 & door=open))
SPEC
  AG (floor2.pressed -> AF (floor=2 & door=open))
SPEC
  AG (floor3.pressed -> AF (floor=3 & door=open))
SPEC
  AG (floor4.pressed -> AF (floor=4 & door=open))
SPEC
  AG (floor5.pressed -> AF (floor=5 & door=open))

    -- WHEN THERE IS A CALL AHEAD OF THE LIFT, IT DOES NOT
    -- CHANGE DIRECTION UNTIL IT HAS ANSWERED THE CALL
    -- (ONLY 2 OF THE CASES ARE SPECIFIED):
SPEC
  AG (floor=2 & floor5.pressed & direction=up
                    -> A[direction=up U floor=5])
SPEC
  AG (floor=5 & floor1.pressed & direction=down
                    -> A[direction=down U floor=1])

    -- THE LIFT CAN REMAIN IDLE WITH ITS DOORS CLOSED:
SPEC
  EF(floor=1 & door=closed & idle)
SPEC
  AG (floor=1 & idle & door=closed -> EG (floor=1 & door=closed))
SPEC
  EF(floor=2 & door=closed & idle)
SPEC
  AG (floor=2 & idle & door=closed -> EG (floor=2 & door=closed))
SPEC
  EF(floor=3 & door=closed & idle)
SPEC
  AG (floor=3 & idle & door=closed -> EG (floor=3 & door=closed))
SPEC
  EF(floor=4 & door=closed & idle)
SPEC
  AG (floor=4 & idle & door=closed -> EG (floor=4 & door=closed))
SPEC
  EF(floor=5 & door=closed & idle)
SPEC
  AG (floor=5 & idle & door=closed -> EG (floor=5 & door=closed))

    -- THE LIFT DOES NOT PARK AT FLOOR 1:
SPEC
  !AG(floor=2 & idle -> E [idle U floor=1])
SPEC
  !AG(floor=3 & idle -> E [idle U floor=1])
SPEC
  !AG(floor=4 & idle -> E [idle U floor=1])
SPEC
  !AG(floor=5 & idle -> E [idle U floor=1])

    -- A CAR TRAVELLING UPWARDS MAY STOP FOR LANDING CALLS:
SPEC
  !AG ((floor=1 & !floor1.pressed & direction=up)
        -> !(door=open))
SPEC
  !AG ((floor=2 & !floor2.pressed & direction=up)
        -> !(door=open))
SPEC
  !AG ((floor=3 & !floor3.pressed & direction=up)
```

```
        -> !(door=open))
SPEC
  !AG ((floor=4 & !floor4.pressed & direction=up)
        -> !(door=open))
SPEC
  !AG ((floor=5 & !floor5.pressed & direction=up)
        -> !(door=open))


MODULE button (reset, suppress)
VAR
  pressed : boolean;
ASSIGN
  init (pressed) := 0;
  next (pressed) := case
                      reset     : 0;
                      pressed   : 1;
                      suppress  : 0;
                      1         : {0,1};
                    esac;
```

Annexe 5 : SBCC5_CP.OUT

Ce fichier contient la sortie produite par SMV lorsqu'on lui soumet en entrée le fichier sbcc5_cp.smv.

```
-- specification !AG (door = closed -> AF door = open) (in module car1) is true
-- specification AG (floor1.pressed -> AF (floor = 1 & do... (in module car1) is true
-- specification AG (floor2.pressed -> AF (floor = 2 & do... (in module car1) is true
-- specification AG (floor3.pressed -> AF (floor = 3 & do... (in module car1) is true
-- specification AG (floor4.pressed -> AF (floor = 4 & do... (in module car1) is true
-- specification AG (floor5.pressed -> AF (floor = 5 & do... (in module car1) is true
-- specification AG (floor = 2 & floor5.pressed & directi... (in module car1) is true
-- specification AG (floor = 5 & floor1.pressed & directi... (in module car1) is true
-- specification EF (floor = 1 & door = closed & idle) (in module car1) is true
-- specification AG (floor = 1 & idle & door = closed -> ... (in module car1) is true
-- specification EF (floor = 2 & door = closed & idle) (in module car1) is true
-- specification AG (floor = 2 & idle & door = closed -> ... (in module car1) is true
-- specification EF (floor = 3 & door = closed & idle) (in module car1) is true
-- specification AG (floor = 3 & idle & door = closed -> ... (in module car1) is true
-- specification EF (floor = 4 & door = closed & idle) (in module car1) is true
-- specification AG (floor = 4 & idle & door = closed -> ... (in module car1) is true
-- specification EF (floor = 5 & door = closed & idle) (in module car1) is true
-- specification AG (floor = 5 & idle & door = closed -> ... (in module car1) is true
-- specification !AG (floor = 2 & idle -> E(idle U floor ... (in module car1) is true
-- specification !AG (floor = 3 & idle -> E(idle U floor ... (in module car1) is true
-- specification !AG (floor = 4 & idle -> E(idle U floor ... (in module car1) is true
-- specification !AG (floor = 5 & idle -> E(idle U floor ... (in module car1) is true
-- specification !AG (floor = 1 & !floor1.pressed & direc... (in module car1) is true
-- specification !AG (floor = 2 & !floor2.pressed & direc... (in module car1) is true
-- specification !AG (floor = 3 & !floor3.pressed & direc... (in module car1) is true
-- specification !AG (floor = 4 & !floor4.pressed & direc... (in module car1) is true
-- specification !AG (floor = 5 & !floor5.pressed & direc... (in module car1) is true
-- specification AG (landing1.pressed -> AF (car1.floor =... is false
-- as demonstrated by the following execution sequence
state 1.1:
no_call = 1
landing_call = 0
landing1.pressed = 0
landing2.pressed = 0
landing3.pressed = 0
landing4.pressed = 0
landing5.pressed = 0
car1.car_call = 0
car1.idle = 1
car1.floor = 1
car1.door = closed
car1.direction = down
car1.floor5.pressed = 0
car1.floor4.pressed = 0
car1.floor3.pressed = 0
car1.floor2.pressed = 0
car1.floor1.pressed = 0
car1.cp = 0

-- loop starts here --
state 1.2:
landing1.pressed = 1
car1.idle = 0
car1.floor2.pressed = 1
car1.cp = 1

state 1.3:
car1.car_call = 2
car1.direction = up

state 1.4:
no_call = 0
car1.floor = 2
car1.door = open
car1.cp = 0

state 1.5:
car1.car_call = 0
car1.door = closed
car1.floor2.pressed = 0
```

```
state 1.6:
landing_call = 1
car1.direction = down

state 1.7:
no_call = 1
landing_call = 0
car1.floor = 1
car1.floor2.pressed = 1
car1.cp = 1

-- specification AG (landing2.pressed -> AF (car1.floor =... is false
-- as demonstrated by the following execution sequence
state 2.1:
no_call = 1
landing_call = 0
landing1.pressed = 0
landing2.pressed = 0
landing3.pressed = 0
landing4.pressed = 0
landing5.pressed = 0
car1.car_call = 0
car1.idle = 1
car1.floor = 1
car1.door = closed
car1.direction = down
car1.floor5.pressed = 0
car1.floor4.pressed = 0
car1.floor3.pressed = 0
car1.floor2.pressed = 0
car1.floor1.pressed = 0
car1.cp = 0

state 2.2:
no_call = 0
landing2.pressed = 1
car1.idle = 0

-- loop starts here --
state 2.3:
landing_call = 2
car1.direction = up

state 2.4:
no_call = 1
landing_call = 0
car1.floor = 2
car1.floor1.pressed = 1
car1.cp = 1

state 2.5:
car1.car_call = 1
car1.direction = down

state 2.6:
no_call = 0
car1.floor = 1
car1.door = open
car1.cp = 0

state 2.7:
landing_call = 2
car1.car_call = 0
car1.door = closed
car1.direction = up
car1.floor1.pressed = 0

-- specification AG (landing3.pressed -> AF (car1.floor =... is false
-- as demonstrated by the following execution sequence
state 3.1:
no_call = 1
landing_call = 0
landing1.pressed = 0
landing2.pressed = 0
landing3.pressed = 0
landing4.pressed = 0
landing5.pressed = 0
car1.car_call = 0
car1.idle = 1
car1.floor = 1
car1.door = closed
```

```
car1.direction = down
car1.floor5.pressed = 0
car1.floor4.pressed = 0
car1.floor3.pressed = 0
car1.floor2.pressed = 0
car1.floor1.pressed = 0
car1.cp = 0

state 3.2:
no_call = 0
landing3.pressed = 1
car1.idle = 0

-- loop starts here --
state 3.3:
landing_call = 3
car1.direction = up

state 3.4:
no_call = 1
landing_call = 0
car1.floor = 2
car1.floor1.pressed = 1
car1.cp = 1

state 3.5:
no_call = 0
car1.car_call = 1
car1.direction = down
car1.cp = 0

state 3.6:
car1.floor = 1
car1.door = open

state 3.7:
landing_call = 3
car1.car_call = 0
car1.door = closed
car1.direction = up
car1.floor1.pressed = 0

-- specification AG (landing4.pressed -> AF (car1.floor =... is false
-- as demonstrated by the following execution sequence
state 4.1:
no_call = 1
landing_call = 0
landing1.pressed = 0
landing2.pressed = 0
landing3.pressed = 0
landing4.pressed = 0
landing5.pressed = 0
car1.car_call = 0
car1.idle = 1
car1.floor = 1
car1.door = closed
car1.direction = down
car1.floor5.pressed = 0
car1.floor4.pressed = 0
car1.floor3.pressed = 0
car1.floor2.pressed = 0
car1.floor1.pressed = 0
car1.cp = 0

state 4.2:
no_call = 0
landing4.pressed = 1
car1.idle = 0

-- loop starts here --
state 4.3:
landing_call = 4
car1.direction = up

state 4.4:
no_call = 1
landing_call = 0
car1.floor = 2
car1.floor1.pressed = 1
car1.cp = 1
```

```
state 4.5:
no_call = 0
car1.car_call = 1
car1.direction = down
car1.cp = 0

state 4.6:
car1.floor = 1
car1.door = open

state 4.7:
landing_call = 4
car1.car_call = 0
car1.door = closed
car1.direction = up
car1.floor1.pressed = 0

-- specification AG (landing5.pressed -> AF (car1.floor =... is false
-- as demonstrated by the following execution sequence
state 5.1:
no_call = 1
landing_call = 0
landing1.pressed = 0
landing2.pressed = 0
landing3.pressed = 0
landing4.pressed = 0
landing5.pressed = 0
car1.car_call = 0
car1.idle = 1
car1.floor = 1
car1.door = closed
car1.direction = down
car1.floor5.pressed = 0
car1.floor4.pressed = 0
car1.floor3.pressed = 0
car1.floor2.pressed = 0
car1.floor1.pressed = 0
car1.cp = 0

state 5.2:
no_call = 0
landing5.pressed = 1
car1.idle = 0

-- loop starts here --
state 5.3:
landing_call = 5
car1.direction = up

state 5.4:
no_call = 1
landing_call = 0
car1.floor = 2
car1.floor1.pressed = 1
car1.cp = 1

state 5.5:
no_call = 0
car1.car_call = 1
car1.direction = down
car1.cp = 0

state 5.6:
car1.floor = 1
car1.door = open

state 5.7:
landing_call = 5
car1.car_call = 0
car1.door = closed
car1.direction = up
car1.floor1.pressed = 0

-- specification !AG (landing1.pressed -> AF (car1.floor ... is true
-- specification !AG (landing2.pressed -> AF (car1.floor ... is true
-- specification !AG (landing3.pressed -> AF (car1.floor ... is true
-- specification !AG (landing4.pressed -> AF (car1.floor ... is true
-- specification !AG (landing5.pressed -> AF (car1.floor ... is true
-- specification !AG (landing1.pressed -> AF (car1.floor ... is true
-- specification !AG (landing2.pressed -> AF (car1.floor ... is true
-- specification !AG (landing3.pressed -> AF (car1.floor ... is true
```

```
-- specification !AG (landing4.pressed -> AF (car1.floor ... is true
-- specification !AG (landing5.pressed -> AF (car1.floor ... is true
-- specification AG (car1.floor = 5 -> AF car1.floor = 1) is false
-- as demonstrated by the following execution sequence
state 6.1:
no_call = 1
landing_call = 0
landing1.pressed = 0
landing2.pressed = 0
landing3.pressed = 0
landing4.pressed = 0
landing5.pressed = 0
car1.car_call = 0
car1.idle = 1
car1.floor = 1
car1.door = closed
car1.direction = down
car1.floor5.pressed = 0
car1.floor4.pressed = 0
car1.floor3.pressed = 0
car1.floor2.pressed = 0
car1.floor1.pressed = 0
car1.cp = 0

state 6.2:
car1.idle = 0
car1.floor5.pressed = 1

state 6.3:
car1.car_call = 5
car1.direction = up

state 6.4:
car1.floor = 2

state 6.5:
car1.floor = 3

state 6.6:
car1.floor = 4

-- loop starts here --
state 6.7:
car1.floor = 5
car1.door = open

state 6.8:
car1.car_call = 4
car1.door = closed
car1.direction = down
car1.floor5.pressed = 0
car1.floor4.pressed = 1

state 6.9:
car1.floor = 4
car1.door = open

state 6.10:
car1.car_call = 0
car1.door = closed
car1.floor5.pressed = 1
car1.floor4.pressed = 0

state 6.11:
car1.car_call = 5
car1.direction = up

state 6.12:
car1.floor = 5
car1.door = open


resources used:
user time: 24.0167 s, system time: 0.316667 s
BDD nodes allocated: 13892
Bytes allocated: 1048576
BDD nodes representing transition relation: 1167 + 147
```