Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Contribution à l'atelier logiciel de conception de bases de données étude de transformations d'algorithmes

Lazzaroni, R.; Modart, G.

Award date:
1987

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 03. Jul. 2025

FACULTÉS
UNIVERSITAIRES
N. D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

Contribution à l'atelier logiciel de conception de bases de données : étude de transformations d'algorithmes.

R. Lazzaroni et G. Modart

Promoteur : J-L Hainaut

Mémoire présenté en vue de l'obtention du grade de licencié et maître en informatique

Année académique 1986 –1987

Adresse : rue Grandgagnage, 21, B-5000 NAMUR (Belgium) - ☎ 081 - 22 90 65 - Télex 59.222 FAC NAM-B

résumé

Ce mémoire s'inscrit dans un contexte de conception de logiciels assistée par ordinateur (CLAO). Il s'agit plus précisément de conception d'applications sur bases de données.

Ce mémoire aborde les problèmes de transformation d'algorithmes qui se posent dans le cadre d'une méthode de conception de bases de données.

Un schéma de base de données est transformé afin qu'il se conforme aux exigences d'un Système de Gestion de Données choisi. Les algorithmes qui ont été préalablement rédigés en tenant compte de la structure du schéma initial non conforme doivent être à leur tour transformés afin d'opérer sur le schéma transformé. Ces algorithmes doivent en outre n'utiliser que des primitives autorisées par le Système de Gestion de Données choisi.

C'est dans ce contexte que ce travail analyse la possibilité d'une transformation automatique des algorithmes, dans le but d'intégrer un outil réalisant cette transformation dans un atelier préexistant de CLAO.

abstract

This thesis is concerned with computer–aided software engineering (CASE) concepts. More precisely, it is about database applications development.

This thesis describes the problems of algorithms transformations in a database design methodology.

A database schema is transformed so that it complies with the claims of a particular Data Managment System. Algorithms previously written on the first schema must be transformed in order to work on the second schema. Moreover, these algorithms must use only Data Managment System primitives.

In this context, the thesis analyses the possibility of automatic algorithms transformation in order to provide an existing CASE workbench, with an extra tool implementing this transformation.

La réalisation de ce mémoire de fin d'études fut soutenue par les précieux conseils de M. Hainaut. Nous tenons par ces propos à lui exprimer notre gratitude.

Nous remercions également M. Pigneur pour son accueil et la possibilité qu'il nous a offerte d'effectuer un stage à l'université de Lausanne.

Certaines étapes de ce mémoire sont redevables aux informations et références que nous a communiquées M. Lecharlier. Nous lui sommes reconnaissants de cet apport.

Nous tenons d'autre part à remercier M. Cadelli dont l'expérience et la constante disponibilité furent appréciées.

Nous remercions enfin M^{elle} Charlot pour l'intérêt qu'elle a porté à notre travail.

Table des matières

Introduction 1

0.1. Le développement d'un système d'information. 3
0.2. La démarche de conception d'une base de données. 4
0.2.1. La conception logique. 4
0.2.2. La conception physique. 5
0.2.3. Aspects de la démarche. 5
0.3. L'à-propos du mémoire. 6
0.4. Définition du mémoire. 7

Première partie : analyse générale du problème.

Chapitre 1 : Un système de gestion de données virtuel : LDA/MAG. 10
1.1. Introduction. 11
1.2. MAG. 12
1.2.1. Les objets de base. 12
1.2.2. Les contraintes d'intégrité. 17
1.2.3. Les structures de données permises par le SGD LDA/MAG. 17
1.3. Les primitives. 23
1.3.1. Les primitives d'accès. 23
1.3.2. Primitives d'extraction de valeurs. 24
1.3.3. Primitives de modification de données. 24
1.4. Conclusion. 26

Chapitre 2 : Comparaison entre le SGD LDA/MAG et divers SGD commerciaux. 27
2.1. Introduction. 28
2.2. Restrictions de trois SGD commerciaux par rapport à LDA/MAG et primitives admises. 29
2.2.1. Restrictions quant au schéma d'une base de données. 29
2.2.2. Les primitives offertes par les SGD. 30

2.3. Conclusion. 31

Chapitre 3 : LDA, le langage offert par le SGD LDA/MAG. 32
3.1. Introduction. 33
3.2. La syntaxe. 36
3.2.1. Symboles de base, nombres, strings. 36
3.2.2. Variables. 38
3.2.3. Expressions arithmétiques et booléennes. 40
3.2.4. Structure d'un algorithme. 42
3.3 Conclusion. 57

Chapitre 4 : Transformation des algorithmes. 58
4.0. Introduction. 59
4.1. Les formes syntaxiques concernées par les transformations. 60
4.1.0. Introduction. 60
4.1.1. Les formes syntaxiques. 62
4.2. Transformation d'un algorithme suite à une transformation de schéma. 64
4.2.0. Introduction. 64
4.2.1. Transformations. 65
4.3. Des primitives autorisées par le SGD cible : une seconde étape de transformation. 134
4.3.0. Introduction. 134
4.3.1. Deuxième étape de la transformation des algorithmes. 134
4.4. Les formes syntaxiques non concernées par les transformations. 138
4.4.0. Introduction. 138
4.4.1. Exposé des formes syntaxiques. 138
4.5. Optimisation. 140
4.6. Problèmes liés à la transformation des algorithmes. 143
4.6.0. Introduction. 143
4.6.1. La gestion de la cohérence de la base de données. 143
4.6.2. La notion de transaction. 150

4.6.3. Conclusion. 153
4.7. Conclusion. 154

Seconde partie : intégration du mémoire à l'atelier logiciel.

Chapitre 5 : L'atelier logiciel de conception d'applications sur bases de données. 157
5.1. Objet de l'atelier. 158
5.2. Schéma de la base des spécifications de l'atelier logiciel. 159
5.3. Les outils logiciels de l'atelier. 165
5.4. Contribution du mémoire à l'atelier. 167

Chapitre 6 : Un type abstrait de données : l'arbre. 167
6.1. Introduction. 169
6.2. L'analyse suntaxique. 171
6.3. La notion d'arbre dans l'atelier. 174
6.4. Représentation d'un algorithme LDA sous la forme d'arbre. 175
6.4.1. Structure d'un nœud et table des symboles. 175
6.4.2. Règles de représentation. 177
6.4.3. Représentation de l'arbre selon le type abstrait de données de l'atelier. 188
6.4.4. Exemple de représentation en arbre. 188
6.5. Conclusion. 192

Chapitre 7 : Représentation des règles de transformation en termes d'arbre. 193
7.1. Introduction. 194
7.2. Représentation des règles de transformation de la première étape. 195
7.3. Représentation des règles de transformation de la seconde étape. 233
7.4. Gestion de la table des symboles. 235
7.5. Conclusion. 236

Chapitre 8 : Intégration de la notion de transformation dans la base des spécifications de l'atelier. 237
8.0. Introduction. 238
8.1. Représentation des transformations de schéma. 239
8.1.1. Règles générales de représentation. 239
8.1.2. Règles de représentation des transformations de schéma dans l'atelier. 240
8.2. Représentation des algorithmes. 263
8.3. Exemple d'une base des spécifications possible. 265
8.4. Conclusion. 267

Conclusion 268
1. Evaluation du travail effectué 269
2. Perspectives 271

Annexes

Introduction
Annexe 1 : La restriction de la notion d'identifiant dans un type de chemins.
Annexe 2 : Développement d'un exemple.

Introduction

Ce mémoire se situe dans le cadre d'une démarche de conception de bases de données, qui a été développée par M. J-L Hainaut, professeur à l'Institut d'Informatique de Namur.

Cette démarche se propose de concevoir une base de données et les traitements qui lui sont associés de manière rigoureuse et de façon à remplir deux objectifs communément reconnus : la base de données doit être une représentation fidèle du système réel et constituer avec les traitements qui lui sont associés, un serveur de données opérationnel et efficace.

L'application de cette démarche permet au concepteur de bases de données d'obtenir la base dont il est chargé d'assurer le développement. L'aide que l'on pourrait offrir au concepteur dans l'application de cette méthode serait sans nul doute accueillie favorablement. Un ensemble d'outils dénommé "atelier logiciel" lui est dès lors proposé pour l'aider et parfois même le remplacer dans le développement de la base de données. Le but de ce mémoire est d'étudier la conception d'un outil supplémentaire qui viendrait compléter l'atelier logiciel actuellement en développement à l'Institut d'Informatique de Namur.

Afin d'apprécier l'étude qui a été menée dans ce mémoire, il est utile de situer ce dernier dans la démarche dans laquelle il s'inscrit.

0.1. Le développement d'un système d'information

Le développement d'un système d'information est le processus qui conduit à la production d'un système opérationnel et de ses règles d'utilisation. Il est suivi, selon le modèle traditionnel du cycle de vie des systèmes informatiques, d'une phase d'utilisation et de maintenance.

Le processus de développement se décompose en quatre phases : l'étude d'opportunité, l'analyse conceptuelle, la conception de mise en œuvre, la réalisation et mise au point.

L'étude d'opportunité et l'analyse conceptuelle traitées dans [Bodart, Pigneur, 83] aboutissent notamment à la structuration des informations de la mémoire du système d'information selon le modèle entité/association, à la décomposition arborescente d'un traitement en ses composants, à la spécification des règles d'activation des composants. Pour chaque composant de traitement est produite également une spécification des informations en entrée, des informations en sortie, des règles de production de ces dernières à partir des premières, ainsi que des dialogues entre le composant et l'environnement du système d'information.

Après l'analyse conceptuelle viennent les phases de conception de mise en œuvre et réalisation-mise au point de la base de données du système d'information. Ces phases font l'objet de la démarche particulière définie par le professeur J-L Hainaut et traitée de façon détaillée dans [Hainaut 86a]. Cette démarche consiste à traduire les spécifications conceptuelles (indépendantes de la notion d'outil informatique) produites par les deux premières phases en une solution exécutable par un système matériel/logiciel.

L'analyse conceptuelle et les phases de conception de mise en œuvre et réalisation-mise au point constituent les étapes de développement d'une base de données.

La description qui est faite de la démarche dans les paragraphes suivants, omet certains aspects qui apparaissent non pertinents dans le cadre du mémoire.

0.2. La démarche de conception d'une base de données

La démarche de conception d'une base de données vise à produire de façon systématique à partir des spécifications conceptuelles d'un système d'information, un description qui respecte ces spécifications, qui soit efficace et exécutable par un Système de Gestion de Données (SGD) disponible sur le marché.

La démarche se scinde en deux phases : celle de conception logique produit, à partir de la solution conceptuelle, une solution correcte et efficace mais indépendante d'un SGD réel; celle de conception physique traduit la solution précédente en termes d'un SGD réel et de ses exigences. Cette découpe cherche avant tout à retarder le plus possible les décisions concernant les particularités d'un SGD commercial. Il s'ensuit un effort réduit à consentir pour reconcevoir la base de données en cas de changement de SGD.

0.2.1. La conception logique

la conception logique vise à produire une solution correcte, efficace et indépendante d'un SGD réel. Les algorithmes satisfont à leurs spécifications; le schéma de données exprime toute la sémantique du schéma conceptuel entité/association de la mémoire du système d'information; les algorithmes minimisent le nombre d'accès logiques aux données; enfin, la solution est exécutable par un SGD virtuel dénommé LDA/MAG. Ce SGD est constitué d'une part d'un modèle d'organisation des données et de primitives de manipulation de celles-ci, MAG (Modèle d'accès généralisé), et d'autre part d'un langage de programmation, LDA (Langage de Description d'Algorithmes).

La conception logique comporte diverses étapes.

La première d'entre elles est la production d'un schéma des accès possibles qui est une simple transcription en termes de LDA/MAG du schéma conceptuel de la mémoire du système d'information. Son objectif est d'exprimer les données dans un formalisme adapté aux processus de conception logique.

A partir des spécifications des traitements développées lors de l'analyse conceptuelle, une architecture de modules est ensuite conçue. Chaque module fait l'objet d'une rédaction d'algorithme qui accède éventuellement à la base de données perçue via le schéma des accès possibles. Ces algorithmes dits prédicatifs sont rédigés de manière telle que les données utilisées sont décrites par la condition de sélection qui les définit et non par la spécification des accès qui y conduisent.

Une troisième étape dans la conception logique est l'élaboration d'algorithmes effectifs conformes LDA/MAG. Les algorithmes prédicatifs sont d'abord traduits de façon à être évaluables par accès : une condition sur données est alors telle qu'il existe une primitive LDA/MAG qui fournit les données vérifiant cette condition, et elles seulement. Ces algorithmes cherchent enfin à minimiser leur nombre d'opérations logiques (accès à des articles).

A partir des algorithmes effectifs conformes LDA/MAG, on établit finalement le relevé des caractéristiques des accès aux données que ces algorithmes utilisent. Ces caractéristiques d'accès sont reportées sur le schéma des accès possibles, produisant ainsi le schéma des accès nécessaires.

La conception logique conduit donc à un schéma de base de données lié aux algorithmes qui opèrent sur cette base. Malgré les inconvénients qui peuvent résulter de cette dépendance lors de la prise en considération ultérieure éventuelle de nouveaux algorithmes, on retire l'avantage

majeur d'une configuration de base de données assurant l'efficacité globale des algorithmes.

0.2.2. La conception physique

A partir du schéma des accès nécessaires et des algorithmes effectifs conformes LDA/MAG, la conception physique est chargée de produire une solution qui soit correcte, efficace et cette fois exécutable par un SGD commercial (dit aussi cible).

Cette phase distingue deux couches physiques. L'une, dite abstraite, spécifie une solution conforme au SGD cible, mais encore exprimée en termes de LDA/MAG. La seconde, dite concrète, débouche sur un schéma de base de données rédigé selon le Langage de Description de Données (LDD) du SGD cible. les algorithmes sont quant à eux rédigés dans un langage de programmation et les instructions qui accèdent à la base de données y apparaissent dans les termes du Langage de Manipulation de Données (LMD) du SGD.

La couche abstraite débute avec la transformation du schéma des accès nécessaires hérité de la conception logique, en schéma conforme au SGD cible. Cette transformation consiste à obtenir, à partir d'un schéma de données LDA/MAG, un autre schéma LDA/MAG qui lui est équivalent (tant du point de vue sémantique que de celui des accès), et qui en outre respecte la spécification de ce SGD.

La seconde étape de la couche abstraite est la production à partir des algorithmes effectifs conformes LDA/MAG, d'algorithmes conformes au SGD commercial cible choisi. Pour être conforme au SGD cible, un algorithme se doit de vérifier trois conditions :

1. travailler sur le schéma conforme au SGD.
2. les expressions de désignation et de modification de données qu'il contient, doivent être totalement évaluables et exécutables par les primitives de ce SGD.
3. enfin, l'enchaînement de ces primitives doit être conforme à ce qu'autorise le SGD.

Cette seconde étape se préoccupe de produire des algorithmes satisfaisant à ces trois aspects.

La couche concrète de la phase de conception physique se consacre notamment à rédiger un texte LDD représentant le schéma de données conforme au SGD cible. Traduire les algorithmes dans un langage de programmation spécifique et les instructions qui accèdent à la base de données dans le LMD du SGD cible, constitue une autre tâche de la couche concrète.

0.2.3. Aspects de la démarche

La démarche qui a ainsi été définie comporte donc des transformations successives qui à partir d'une description de la solution, produisent une nouvelle description tenant compte d'un aspect particulier : efficacité, accès nécessaires, conformité par rapport au SGD cible.... Ces aspects sont, autant que possible, indépendants les uns des autres, ce qui limite la complexité des transformations.

Telle qu'elle est proposée, la démarche offre une modularité qui lui permet de s'adapter à des contingences particulières (choix d'un autre modèle de structuration de la mémoire du système d'information, que le modèle entité/association) et de maîtriser la complexité de la conception d'une base de données.

En particulier, la démarche prend en charge complètement la conception d'une base de données, tout en étant indépendante des SGD sur lesquels la base de données est implantée. Cette indépendance garantit à la solution élaborée une adaptabilité importante en cas de revirement dans le choix du SGD cible.

0.3. L'à–propos du mémoire

Ce mémoire présente une étude de transformation des algorithmes effectifs conformes LDA/MAG en algorithmes conformes à un SGD commercial cible. Son propos s'inscrit donc dans la phase de conception physique d'une base de données et plus précisément dans la couche abstraite de celle-ci.

Le but de l'étude qui a été menée ici est de déterminer un ensemble de règles de transformation afin qu'un algorithme satisfasse aux trois éléments constitutifs de la conformité à un SGD cible.

De cet ensemble de règles découle la possibilité d'automatiser la transformation. L'opportunité est donc offerte d'apporter un outil supplémentaire à l'atelier logiciel de conception de base de données qui soutient la démarche présentée auparavant.

Cet atelier est composé d'un ensemble d'outils logiciels qui permettent au concepteur de bases de données de cheminer dans la démarche, d'étape en étape. L'atelier apporte une aide au concepteur et parfois même lui fournit automatiquement le résultat d'une étape dans la conception de sa base de données.

L'atelier logiciel se pose comme un complément à la démarche de conception de bases de données, qui a été élaborée. Il allège en effet l'utilisation de la démarche qui si elle est hiérarchisée, n'en présente pas moins quelque complexité. En offrant au concepteur de bases de données une conception assistée, l'atelier logiciel allie la rapidité d'exécution de certaines étapes de conception avec le jugement qualitatif irremplaçable du concepteur. En automatisant totalement d'autres étapes, l'atelier apporte une gestion complète et rapide des problèmes qui peuvent se poser et une réduction du risque d'erreur de conception.

L'outil d'automatisation de transformation d'algorithmes effectifs conformes LDA/MAG en algorithmes conformes à un SGD cible, dont ce mémoire a pour but d'étudier la conception, trouve sa place dans l'atelier entre deux outils déjà existants : en amont, l'outil de transformation d'un schéma des accès nécessaires en schéma conforme à un SGD cible; en aval, un générateur de code Cobol à partir d'algorithmes rédigés dans les termes du langage de programmation offert par LDA/MAG. Cette séquence rend compte de l'aspect de la démarche de conception d'une base de données, qui consiste à obtenir un schéma conforme, à transformer les algorithmes qui travaillent sur ce schéma et à écrire le code correspondant à ces algorithmes.

0.4. Définition du mémoire

La première partie de ce travail a pour but d'introduire le cadre particulier dans lequel s'inscrit le mémoire ainsi que d'élaborer une analyse générale de la transformation d'algorithmes.

Ainsi, le premier chapitre présente le SGD virtuel LDA/MAG en termes duquel les algorithmes à transformer (effectifs conformes LDA/MAG) s'expriment.

Les particularités de trois SGD commerciaux cibles (Codasyl 71/73, SQL/DS et le système de gestion de fichiers Cobol Ansi-74) par rapport à LDA/MAG font l'objet du chapitre 2. Ce sont les caractéristiques particulières de ces SGD qui nécessitent la transformation d'algorithmes.

Le chapitre 3 présente le langage de programmation LDA offert par le SGD LDA/MAG et dans lequel les algorithmes sont rédigés. La syntaxe et la sémantique du langage y sont exposées.

La détermination de règles de transformation d'algorithmes effectifs conformes LDA/MAG en algorithmes conformes à un SGD cible, occupe pleinement le quatrième chapitre. Celui-ci circonscrit d'abord les formes syntaxiques susceptibles d'être transformées. Ses paragraphes 2 et 3 se consacrent respectivement à déterminer des règles de transformation d'un algorithme pour qu'il respecte les première et seconde composantes de la conformité à un SGD cible : le paragraphe 2 s'attache à faire en sorte que l'algorithme travaille sur le schéma conforme; le troisième paragraphe montre comment un algorithme peut n'utiliser que les primitives autorisées par le SGD cible.

Le chapitre 4 montre également les problèmes de gestion de la cohérence de la base de données, qui peuvent surgir lors de la transformation d'algorithmes. Une gestion des incidents et erreurs qui peuvent avoir lieu lors de l'exécution de l'algorithme transformé est aussi de mise. Enfin, les règles de transformation définies dans les paragraphes 2 et 3 ont un caractère systématique. Aussi quelques optimisations pour rendre un algorithme transformé plus efficace sont-elles proposées.

La seconde partie du mémoire étudie le problème de l'intégration dans l'atelier logiciel de l'outil de transformation d'algorithmes effectifs conformes LDA/MAG en algorithmes conformes à un SGD cible. Cet outil doit automatiser la transformation d'algorithmes en appliquant les règles exposées au chapitre 4.

Le cinquième chapitre présente dès lors l'atelier logiciel hôte du nouvel outil. L'objectif de l'atelier, le schéma de sa base des spécifications et les outils qu'il offre déjà, y sont décrits.

Le chapitre 6 expose la représentation en arbre qui est donnée à un algorithme. Un aperçu de l'analyseur syntaxique top-down qui doit produire cet arbre est également donné. Les outils de manipulation d'arbre syntaxique que l'atelier logiciel met à la disposition de son développeur font également l'objet de quelque description.

C'est sur la représentation en arbre présentée ci-dessus que l'outil "transformateur d'algorithmes" travaille. Dès lors, le chapitre 7 expose les règles de transformation développées au chapitre 4, en termes de cette représentation.

Comme cela a déjà été dit, c'est suite au travail effectué par le transformateur de schéma qu'opère l'outil étudié dans le présent mémoire. Cet outil a donc besoin d'information sur les transformations de schémas. La façon dont ces informations sont représentées dans l'atelier fait l'objet du chapitre 8. Ce chapitre décrit également la représentation qui est donnée à un

algorithme dans l'atelier.

Une évaluation du travail effectué et des perspectives d'extension concluent le mémoire.

Les annexes se proposent d'autre part de développer une illustration complète de l'étude effectuée.

Première partie :

analyse générale du problème

Chapitre 1 :

Un système de gestion de données virtuel :

LDA/MAG

1.1 Introduction

La phase de conception logique d'une base de données étant le développement d'une solution exécutable sur une machine abstraite indépendante de toute machine réelle, le but de ce chapitre est de décrire cette machine. Elle est entièrement définie par la définition d'un SGD abstrait (ou virtuel en ce sens qu'il n'est implémenté sur aucune machine existante) : LDA/MAG.

Comme tout SGD, LDA/MAG offre à l'utilisateur une série de primitives de manipulation de données, ainsi que la possibilité de définir l'organisation de ces données. Ces primitives et ces structures de données forment les parties constituantes du MAG (Modèle d'Accès Généralisé). Ce chapitre se propose de décrire le modèle cité. Les primitives qu'il inclut sont d'autres part supportées par le langage de programmation LDA qui permet d'écrire des applications; ce langage sera décrit au chapitre 3.

On peut trouver une description plus détaillée de MAG dans [Hainaut 86a] et [Hainaut 84b]

1.2. MAG

MAG est un modèle d'organisation de données qui ne peut être considéré comme un modèle conceptuel dans la mesure où il prend en compte non seulement la sémantique qu'expriment les données, mais aussi les accès dont elles peuvent faire l'objet; il se veut de plus indépendant de tout SGD. Il se situe donc en toute logique "à mi–chemin" entre l'analyse conceptuelle et la conception physique.

MAG regroupe différents concepts communs aux SGDs existants (en ce sens il peut être utilisé comme modèle de description de ces SGDs), principalement le type de chemins, le type d'articles et l'item. Ces objets de base sont décrits dans le premier point du paragraphe. Les contraintes d'intégrité font l'objet du second point; le troisième est consacré à la description des assemblages possibles des objets de base.

Le dernier paragraphe s'attache à décrire les primitives offertes par le MAG.

1.2.1 Les objets de base

Les objets de base sont les (types d')articles, les fichiers, la base de données, le type d'articles système, les (valeurs d')items, les (types de) chemins, les identifiants, les clés d'accès et les ordres.

1. ARTICLES ET TYPES D'ARTICLES

L'article est une entité d'information appartenant à la base de données. On peut le créer, y accéder, le modifier et le détruire. C'est l'unité de base sur laquelle travaillent le programmeur et l'utilisateur.

Tous les articles sont regroupés en classes qui en définissent les propriétés communes. Une classe est appelée "type d'articles". Chaque classe est identifiée par un nom et peut regrouper à un moment donné de l'existence de la base de données un nombre quelconque d'articles. Un article appartient à une et une seule classe.

Un type d'articles est représenté par un cartouche contenant le nom du type d'articles.



Fig 1.1 : représentation graphique des types d'articles

LE FICHIER

Un fichier est une collection dynamique d'articles.

LA BASE DE DONNEES

Une base de données est une collection d'articles d'un ensemble de fichiers. Certains de ces articles sont mis en relation les uns avec les autres. Une base de données porte un nom qui la distingue des autres bases de données dans un contexte déterminé.

Les notions de fichier et de base de données ne seront pas approfondies dans la mesure où elles n'ont que peu d'importance dans le cadre des transformations d'algorithmes envisagées.

LE TYPE D'ARTICLES SYSTEME

Chaque base de données contient un article spécial appelé SYSTEME. Il peut être origine de chemins. Il porte le nom de la base de données à laquelle il appartient.

LES ITEMS ET LES VALEURS D'ITEMS

Une valeur d'item est un élément d'un type de données (le domaine : p. ex. l'ensemble des entiers) associé à un article. Il est possible d'obtenir les valeurs d'items associées à un article en accédant à cet article.

A un type d'articles peuvent être associés un nombre quelconque de domaines. Les rôles joués par ces domaines dans ces associations sont appelés "items du type d'articles".

On reconnaît aux items les propriétés suivantes :

Items élémentaires et décomposables

Un item élémentaire est la relation qui existe entre un type d'articles et un seul domaine. Les valeurs d'items sont alors atomiques. Un item décomposable est le rôle joué par un ensemble de domaines dans la relation qui les lie au type d'articles. Outre le rôle global joué par l'ensemble des domaines, les domaines jouent chacun un ou plusieurs rôles "personnels" dans le rôle global; ces rôles sont les items composants de l'item décomposable. Ces items sont considérés comme des items à part entière; ils héritent donc de toutes leurs propriétés. Un item décomposable se décompose en une arborescence d'items, et on définit une valeur d'item décomposable comme l'ensemble des valeurs prises par les feuilles de cette arborescence.

Items simples et répétitifs

Un item est simple si à chaque article ne peut être associée qu'une seule valeur de cet item. Il est répétitif si plus d'une valeur peuvent y être associées. Dans ce dernier cas, la répétitivité peut être fixe (un même nombre de valeurs pour tout article), variable limitée (pour un article donné le nombre de valeurs est quelconque mais ne peut dépasser un maximum spécifié) ou illimitée.

Cette définition peut être formulée différemment en disant qu'un ENSEMBLE de valeurs peut être associé à l'article. De cette formulation ensembliste on peut déduire qu'il est erroné de parler de première, de seconde, de $i^{ème}$ valeur d'un item répétitif, ainsi que d'une valeur apparaissant plusieurs fois dans l'ensemble des valeurs d'un item répétitif.

Items obligatoires et facultatifs.

Un item est obligatoire si à chaque article est associée au moins une valeur de cet item, il est facultatif sinon.

Items identifiants et non identifiants.

Un item est identifiant si à chaque valeur de cet item est associé au plus un article. Cette notion est très importante dans le cadre du mémoire, elle sera affinée par la suite.

Graphiquement les items sont représentés de la façon suivante :

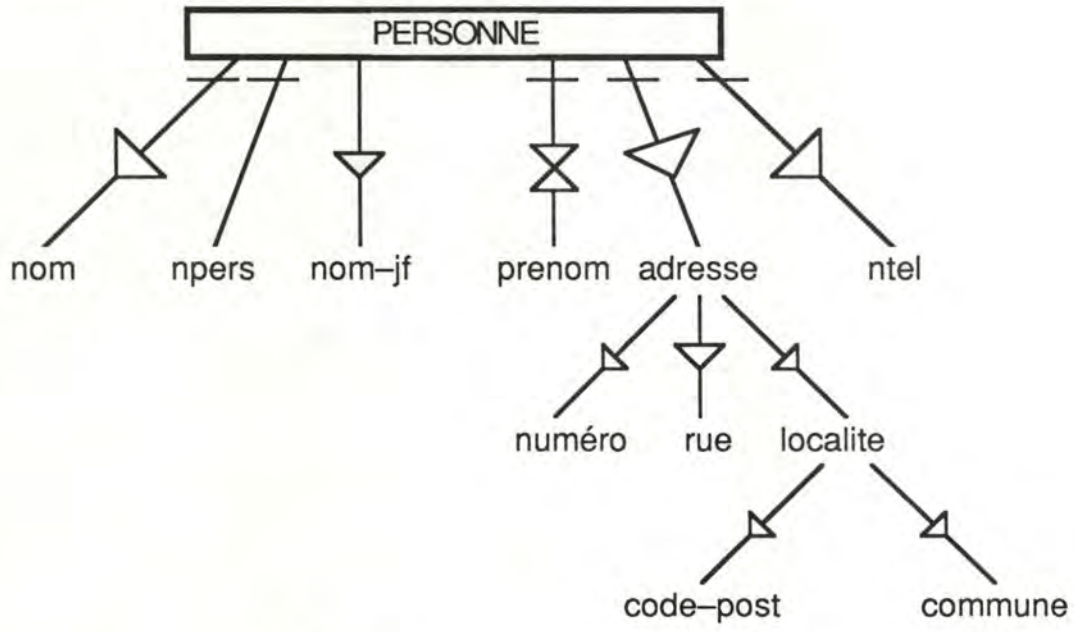


Fig 1.2 : représentation graphique des items

Sont décomposables : adresse, localité.
Sont répétitifs : prenom, ntel.
Est facultatif : nom–jf.
Sont identifiants : npers, ntel.

LES CHEMINS ET LES TYPES DE CHEMINS

Un chemin d'accès inter–articles est un mécanisme qui associe à un article origine du chemin une suite de 0, 1 ou plusieurs articles cibles du chemin. Il est possible d'autre part d'accéder de l'origine d'un chemin vers les cibles, mais pas le contraire.

Comme les articles, les chemins sont regroupés en classes (les types de chemins) qui en définissent les propriétés communes. Un type de chemins est caractérisé par son nom, et les types d'articles qui peuvent en être origine et cible.

Un type de chemins dont plusieurs types d'articles sont origines et/ou cibles est appelé multi–type, ou encore multi–origine et/ou multi–cible.

Si un même type d'articles est à la fois origine et cible d'un type de chemins, ce dernier est dit "récursif". Cette propriété n'est nullement incompatible avec celle énoncée précédemment.

Deux types de chemins sont inverses si les cibles de l'un sont les origines de l'autre et vice–versa. Ces types de chemins sont souvent appelés, à tort, un type de chemins bidirectionnel (en opposition à unidirectionnel). On peut justifier cette interprétation du fait de la représentation graphique usuelle (Cfr. fig. 1.3).

Deux autres propriétés importantes des types de chemins sont les classes fonctionnelles et

les contraintes d'existence. Leur définition est reprise de [Hainaut 86a].

La classe fonctionnelle d'un type de chemins détermine le nombre maximum d'articles d'un membre que l'on peut associer à chaque article de l'autre membre.

Cette classe est N–N si un chemin peut contenir un nombre quelconque de cibles et qu'une cible peut l'être d'un nombre quelconque de chemins.

Elle est 1–N si un chemin peut contenir un nombre quelconque de cibles mais qu'une cible ne peut l'être que d'un seul chemin.

Elle est N–1 si un chemin ne peut contenir qu'une seule cible et qu'une cible peut l'être d'un nombre quelconque de chemins.
Elle est 1–1 si un chemin ne peut contenir qu'une seule cible et qu'une cible ne peut l'être que d'un seul chemin. On nie donc les 2 propriétés énoncées pour N–N.

Une contrainte d'existence consiste à imposer à tout article d'un membre d'un type de chemins d'être associé à au moins un article de l'autre membre.

Graphiquement on représente les types de chemins de la façon suivante.

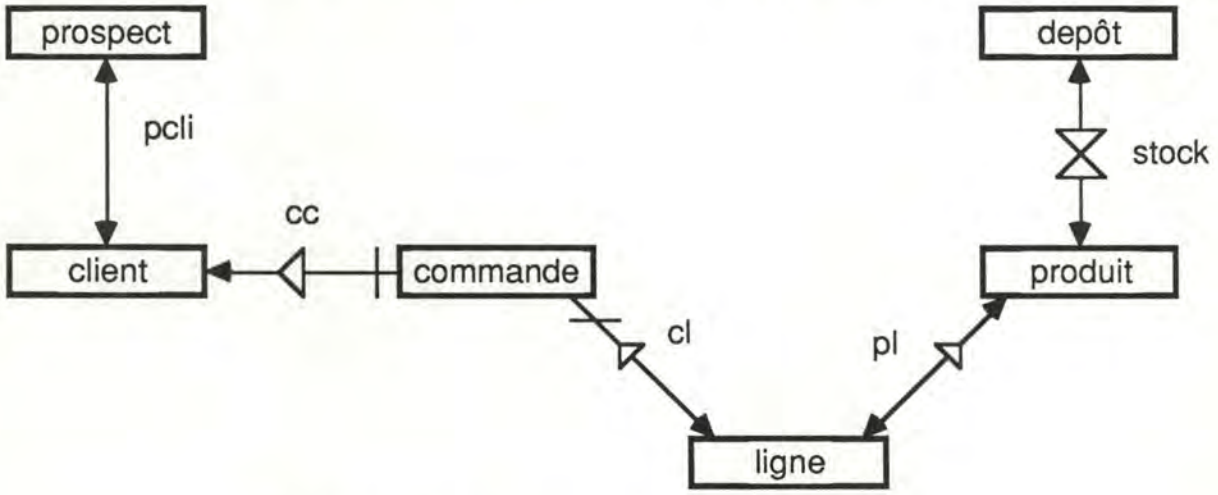


Fig. 1.3 : représentation graphique des types de chemins

On voit notamment que pcli est de classe fonctionnelle 1–1, cc est N–1 et qu'il est obligatoire pour commande, ou, en d'autres termes, qu'une contrainte d'existence est imposée à commande en ce qui concerne le type de chemins cc. On constate également que stock est de classe fonctionnelle N–N et qu'il a un synonyme (!) inverse de même classe fonctionnelle.

Cette représentation est très proche de celle des items dans la mesure où on peut considérer qu'il existe un type de chemins anonyme entre un item et un type d'articles.

LES IDENTIFIANTS COMPOSES

Un identifiant n'est plus seulement un item, mais il peut être composé de plusieurs items, et/ou d'un ou plusieurs types de chemins. Ces items et types de chemins sont les composants de l'identifiant. Dans le cas des types de chemins, on peut dire encore que l'identifiant contient un composant rôle.

Graphiquement, on représente des identifiants composés par un semi– parallélisme entre les arcs des composants.

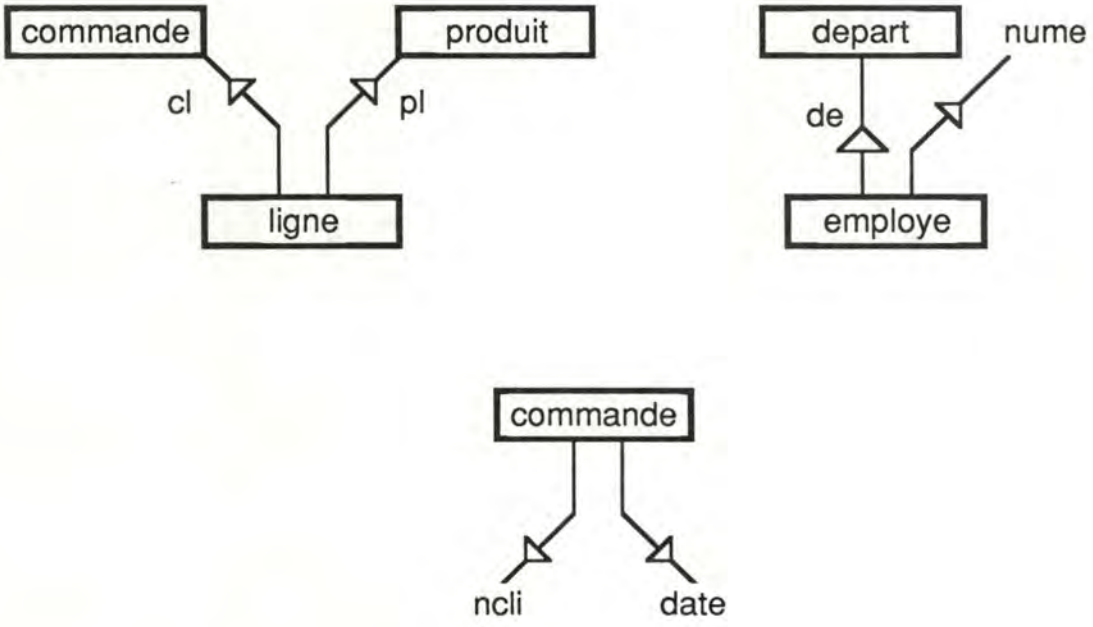


Fig. 1.4 : représentation graphique des identifiants composés.

LES CLES D'ACCES

Une clé d'accès est un (groupe d') item(s) d'un même type d'articles, tel qu'il existe un mécanisme permettant d'accéder aux articles répondant à une certaine condition exprimée sur la valeur de cette clé, et à eux seulement. Cette notion peut être locale à un chemin, on parle alors de clé d'accès dans un chemin. Si la clé est composée de plusieurs items, c'est une clé groupe.

Graphiquement, on représente une clé par une flèche du groupe d'items ou de l'item vers le type d'articles.



Fig. 1.5 : représentation graphique des clés d'accès

ORDRE D'UNE SEQUENCE D'ARTICLES

L'utilisateur peut déterminer des séquences d'articles (p. ex. par une valeur de clé ou par un chemin). L'accès séquentiel à ces articles peut se faire selon différents ordres : LIFO, FIFO,

aléatoire, ...

1.2.2. Les contraintes d'intégrité

Lors de l'analyse conceptuelle, le schéma conceptuel des données s'est vu précisé par une série de contraintes d'integrité. Le concepteur se doit de répercuter ces contraintes au niveau de MAG.

Excepté celles plus fréquentes évoquées au point précédent (identifiant, classe fonctionnelle et contrainte d'existence), ces contraintes ne font pas l'objet d'un développement ultérieur, dans la mesure où elles n'ont que peu d'importance dans le cadre du mémoire.

Le lecteur intéressé consultera [Hainaut 86a] pour plus de détails.

1.2.3. Les structures de données permises par le SGD LDA/MAG

Il s'agit dans ce paragraphe de spécifier comment le concepteur peut agencer les différents composants de base énoncés précédemment.

Certaines restrictions sont imposées arbitrairement uniquement dans le souci de limiter le travail, d'autres sont la conséquence des difficultés qui sont apparues lors de l'étude des transformations d'algorithmes (elles sont dès lors détaillées au chapitre 4); d'autres encore sont le fruit de réflexions diverses.

Ces restrictions sont classées selon les objets qu'elles concernent.

LES TYPES D'ARTICLES

Tout type d'articles doit avoir au moins un item identifiant simple, élémentaire et obligatoire.

Cette position assez restrictive peut être justifiée par le fait qu'elle rend les transformations de schéma beaucoup plus aisées.

LES ITEMS

Un item décomposable peut être répétitif mais aucun item composant d'item décomposable ne peut l'être.

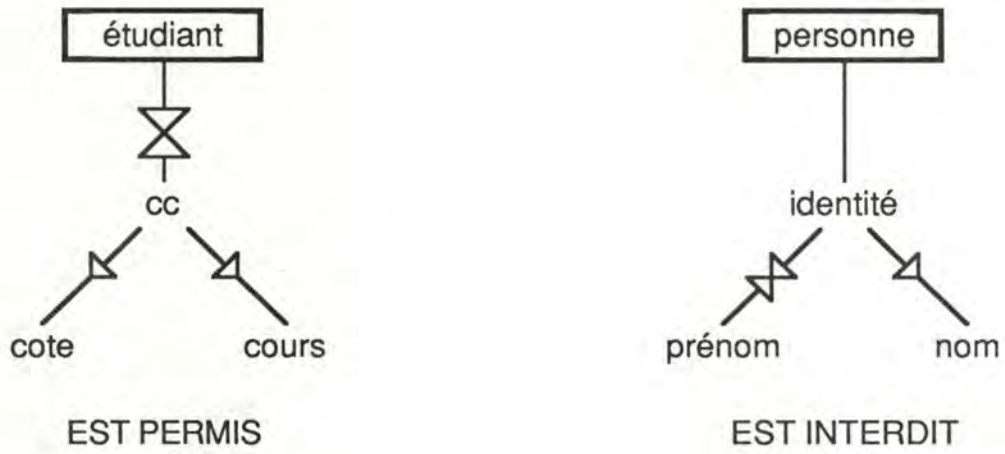


Fig. 1.6 : exemple de restriction sur des items répétitifs décomposables

Soit un utilisateur exploitant une base de données gérée par LDA/MAG. Comme cela a déjà été explique, il peut obtenir les valeurs d'items d'un article auquel il a accédé.

Un problème se pose si un des items est facultatif et qu'aucune valeur de cet item n'est associée à l'article auquel on a accédé.

Cette absence doit–elle entraîner l'obtention d'une valeur "parasite" due à la lecture d'un champ non initialisé, voire une erreur d'exécution pour la même raison ou pour la tentative d'accès à "quelque chose" d'inexistant?

Il paraît assez sévère de pénaliser l'utilisateur de la sorte, d'autant plus que l'absence de valeur est liée à un état de la base de données dont il n'est pas précisément responsable.

Une solution consiste à rendre tous les items obligatoires. Ceci pose cependant des difficultés au concepteur qui lors de sa modélisation du réel a été amené à définir des attributs facultatifs. Ce problème peut être résolu par l'utilisation d'une valeur "spéciale" (appelée NULL).

Dans ce cadre, un item "pseudo–facultatif" est un item pouvant prendre la valeur NULL et un item "obligatoire" (l'ambiguïté par rapport à sa définition précedente ne prête pas à conséquence) est un item ne pouvant pas prendre la valeur NULL.

NULL est utilisée comme "valeur absente". Par exemple, la valeur de l'item "nom de jeune fille" d'un article du type "personne" est NULL si la personne décrite est de sexe masculin. Il faudra donc offrir la possibilité à l'utilisateur de spécifier NULL comme valeur d'item d'un article.

La sémantique de NULL ne se limite cependant pas à cela. En effet, si l'on considère (par ex.) une entreprise manufacturière qui gère une base de données de ses ouvriers, on y trouve des articles "ouvriers" qui ont chacun une valeur d'item "groupe sanguin" afin de faciliter les soins lors d'accidents graves. Si un ouvrier récemment engagé est déjà enregistré dans la base de données mais n'a pas encore passé les examens médicaux, son groupe sanguin n'est pas encore connu. L'opérateur mettra donc NULL comme valeur de l'item "groupe sanguin" de cet ouvrier. Il faut toutefois remarquer que NULL n'a pas ici la signification de valeur absente (dans la mesure où tout être humain appartient à un groupe sanguin) mais bien celle de valeur INCONNUE (au moment de la création de l'article). Lorsque le groupe sanguin sera connu, la valeur d'item sera modifieé.

NULL peut encore signifier la valeur par défaut, c'est–à–dire celle qui "est là" à défaut d'y en avoir mis une autre. Lorsqu'un utilisateur crée un article, s'il ne spécifie pas toutes les valeurs d'items qui y sont associées, alors celles non spécifiées sont NULL.

Un item décomposable pseudo–facultatif est composé uniquement d'items pseudo–facultatifs. Inversément il est obligatoire si au moins un de ses composants l'est. Une valeur décomposable est NULL si chacun des composants est NULL.

La valeur par défaut, inconnue ou absente d'un item répétitif pseudo-facultatif est une et une seule fois NULL.

Si l'utilisateur veut accéder à la valeur d'un item pseudo–facultatif, il doit s'assurer que celle–ci n'est pas NULL avant d'en faire un usage "malheureux" (Cfr. exemple ci–dessous).

Soit dans un formalisme PASCAL l'instruction " x := 4 + (5 * V) ". Si V est NULL, le résultat de l'exécution de cette instruction est tout à fait inattendu dans la mesure où il dépend de la représentation interne de NULL qui n'est connue que de LDA/MAG.

Il faut garder à l'esprit que NULL est une valeur (d'item) qui se distingue des autres par le fait de lui avoir donné une sémantique conventionnelle particulière et par la possibilité qu'elle a d'appartenir à n'importe quel domaine. En cela, elle peut être considérée comme une valeur spéciale.

LES TYPES DE CHEMINS

Un type de chemins ne peut être ni multi–origine, ni multi–cible.

Un type de chemins ne peut être le siège de deux contraintes d'existence. Soit l'exemple ci–dessous.

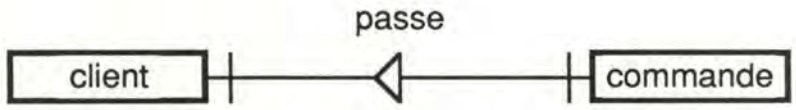


Fig. 1.7 : exemple de type de chemins obligatoire pour les deux membres.

Un client ne peut exister dans la base de données que s'il a passé au moins une commande, de même qu'une commande ne peut exister que si elle est passée par un client. Dans ce contexte, il est impossible de creér un article avant ou après l'autre; il faut les créer "en même temps". Cette dernière notion n'est cependant pas connue de LDA/MAG.

Un type de chemins récursif ne peut être le siège d'aucune contrainte d'existence. Soit l'exemple formel suivant.

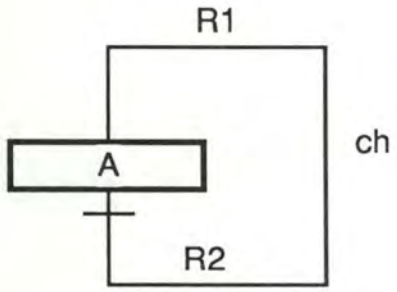


Fig. 1.8 : type de chemins récursif siège d'une contrainte d'existence.

R1 et R2 constituent les rôles joués par le type d'article A dans le type de chemins ch. La contrainte d'existence peut être formulée de la façon suivante : le rôle R2 est obligatoire pour A.

Si un utilisateur veut créer "a1", un article de type A, il doit lui faire jouer le rôle R2. En termes d'occurences, on a donc au moins :



Fig 1.9 : articles reliés par un chemin récursif.

Où a2 est de type A et doit donc également jouer un rôle R2, ... etc ... . Ceci provoque une chaîne infinie d'articles qui ne peut être évitée que par un circuit.

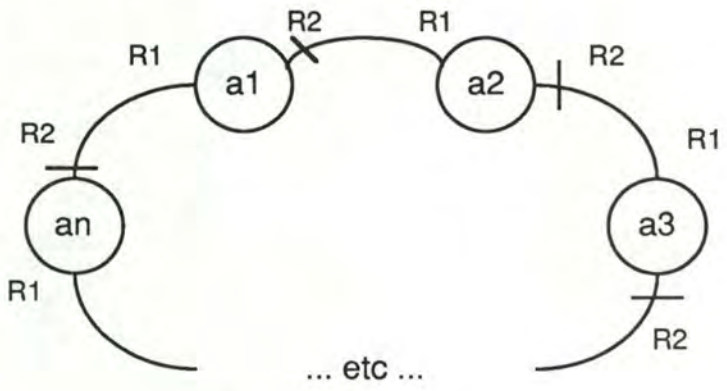


Fig 1.10 : circuit d'articles.

Cette structure entraîne cependant le problème évoqué précédemment à propos de types de chemins sièges de deux contraintes d'existence. L'article a1 ne peut en effet être créé qu'après a2, a2 après a3, ... , an après a1. Il faut donc créer tous les articles "en même temps". Comme cela a déjà été signalé, LDA/MAG n'accepte pas cette notion.

Si les deux rôles sont obligatoires, le raisonnement est analogue.

LES IDENTIFIANTS

Un (composant d') identifiant ne peut être "absent". En d'autres termes, si un type d'articles est associé à au moins un identifiant, tout article de ce type doit être identifié à chaque instant de son existence de la base de données. Un article de ce type doit donc toujours être associé à une valeur pour cet identifiant. Pour les composants items de l'identifiant, cela ne pose pas de problème étant donné qu'aucun item ne peut être facultatif, il peut être "au maximum" pseudo-facultatif. Pour les composants type de chemins, cette obligation d'identification se traduit par le fait que ces composants type de chemins sont obligatoires pour le type d'articles identifié.

Du conflit existant entre cette dernière affirmation et celle selon laquelle un type de chemins récursif ne peut être le siège d'aucune contrainte d'existence, on déduit qu'un type de chemins récursif ne peut faire partie d'un identifiant.

Il est d'autre part assez logique qu'aucun des composants d'un identifiant composé ne soit identifiant lui–même, car l'identifiant initial n'aurait dès lors plus de raison d'être (en particulier, cela veut dire que les type de chemins sont de classes 1–N ou N–N). Les identifiants respectant cette exigence sont dits stricts.

Si un identifiant est composé d'au moins un type de chemins et d'au moins un item, alors les types de chemins sont de classe fonctionnelle 1–N (la restriction énoncée au paragraphe précédent est donc encore renforcée dans la mesure où les types de chemins N–N ne sont plus admis) et les items sont simples. Cette restriction sera justifiée dans les annexes lors de l'exposé de ce qui a été écarté lors de l'analyse.

Si un identifiant est composé de plusieurs items, ceux-ci sont tous directement rattachés au type d'articles et aucun d'eux ne peut être répétitif. Cette restriction est expliquée au chapitre 4.

LES CLES D'ACCES

Une clé d'accès définie avec les composants d'un item répétitif décomposable doit les regrouper tous. Soit l'exemple formel décrit à la figure 1.11.

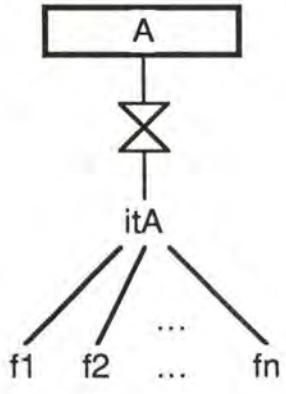


Fig. 1.11 : clé d'accès répétitive décomposable.

itA peut être clé d'accès. De même, f1, f2, ... , fn peuvent former une clé d'accès. Mais

une combinaison de fi qui ne le regroupe pas tous (p. ex. : f1 ou fj,fn) ne peut l'être. Cette restriction sera justifiée au chapitre 4.

Aucun composant d'une clé ne peut être un identifiant dans la mesure où une clé ainsi définie apporterait à peine plus en termes d'accès qu'une clé définie uniquement sur l'identifiant.

Similairement, une clé d'accès dans un chemin ne peut comporter ou être un (groupe d')item(s) identifiant.

Si une clé d'accès est composée de plusieurs items, ceux-ci sont tous directement rattachés au type d'articles. Cette restriction recouvre celle exposée ci-dessus à propos des items répétitifs décomposables. Elle est cependant imposée pour d'autres raisons qui sont décrites au chapitre 4.

Si une clé d'accès est composée de plusieurs items, aucun de ces items ne peut être répétitif. Cette restriction est expliquée au chapitre 4.

Une clé d'accès dans un chemin ne peut être un item répétitif ou un groupe d'items dont au moins un est répétitif. Cette restriction est justifiée au chapitre 4.

L'ORDRE

L'utilisateur ne peut déterminer l'ordre dans lequel il accède aux articles d'une séquence.

Il ne pourrait le faire, en effet, que par le biais d'une instruction spécifique offerte par le langage LDA. Or LDA n'offre pas cette instruction (Cfr. syntaxe de LDA ch.3 §2).

1.3. Les primitives

Un autre point important de MAG est l'ensemble des opérations permises sur les données exposées au paragraphe précédent (1.2.).

On distingue parmi ces opérations appelées primitives, les primitives d'accès aux données et celles de modification de données.

Seules sont reprises ici les primitives reconnues par LDA/MAG. Une description plus complète des primitives de MAG peut être trouvée dans [HainautT 86a] et [Hainaut 84b].

1.3.1. Les primitives d'accès

Un accès consiste à rendre disponible à l'utilisateur un ou plusieurs articles présents dans une base de données.

L'utilisateur détermine un ensemble d'articles en exprimant des conditions sur ces articles. Les primitives lui permettent d'accéder aux articles de cet ensemble. Ceci précède généralement l'extraction de valeurs d'items, la modification des articles ou de leur environnement.

LDA/MAG permet les accès suivants.

ACCES AUX ARTICLES D'UN TYPE

Cette primitive permet à l'utilisateur d'obtenir tous les articles d'un type donné. Elle est appelée aussi "accès séquentiel".

ACCES PAR CLE

Cette primitive permet d'accéder à des articles sur base d'une clé. Elle requiert de l'utilisateur qu'il spécifie le type d'articles auquel il veut accéder, la clé, et dans la mesure où l'accès par clé nécessite la comparaison entre une clé et une valeur déterminée, cette valeur et l'opérateur de comparaison sont également indispensables.

Si la clé est un item élémentaire, tous les opérateurs de comparaison sont permis sauf "<>" dans la mesure où on peut émettre des réserves quant à son utilité. Ils ont de plus une signification évidente si l'item est défini sur un domaine numérique.

Il convient cependant de préciser la sémantique de <, >, <=, >= dans le cas où le domaine d'un item est alphabétique (p. ex. à la figure 1.2, l'item "nom" est défini sur le domaine : chaîne de 30 caractères alphabétiques). La signification de ces opérateurs est alors liée à l'ordre alphabétique. Une expression comme "Alain < Bernard" signifie que la chaîne de caractères "Alain" précède la chaîne de caractères "Bernard" selon l'ordre alphabétique. C'est cette sémantique qui est utilisée lors de l'accès par clé.

Si la clé est composée de plusieurs items, l'utilisateur spécifie une valeur pour chacun des composants de cette clé, et les articles obtenus sont ceux dont la clé "correspond à" cet ensemble de valeurs; en d'autres termes ceux tels que la relation établie par l'opérateur de comparaison entre les composants et les valeurs de composants (2 à 2) est respectée pour tout composant.

Le seul opérateur permis est "=". Tous les autres opérateurs sont refusés dans la mesure

où on ne peut leur trouver aucune signification pratique.

En effet, "<" (p. ex.) est toujours utilisé comme "plus petit que" pour les clés numériques et comme "précède alphabétiquement" pour les clés alphabétiques. Pour étendre cette signification aux clés–groupes, la seule solution est d'établir un ordre sur les composants de la clé. Ce raisonnement est similaire à celui tenu lorsqu'on veut (p. ex.) établir un ordre alphabétique entre deux mots : on compare les premières lettres de chaque mot, si le résultat n'est pas discriminant, on compare les secondes, ... etc ... . Cette comparaison est impossible au niveau de la clé dans le mesure où l'utilisateur ne peut définir d'ordre sur les composants d'une clé.

Le fait qu'il soit impossible de définir une sémantique satisfaisante pour l'opérateur "<" dans le cas d'un groupe d'items clé d'accès justifie le refus de son utilisation.

Le raisonnement est analogue pour les opérateurs interdits suivant >, >=, <=, ainsi que pour les cas où la clé est un (groupe d') item(s) décomposable(s).

Comme précédemment "<>" est refusé dans la mesure où il n'est d'aucune utilité pratique.

Le fait de ne pas permettre à l'utilisateur de définir d'ordre sur les composants d'une clé est cependant tout à fait particulier à LDA/MAG. Les SGD commerciaux offrent toujours des clés dont les composants sont ordonnés. Généralement, cet ordre est celui dans lequel apparaissent les items dans la déclaration de la clé.

ACCES PAR CHEMIN

Cette fonction permet à l'utilisateur d'accéder aux articles cibles d'un chemin dont il a spécifié l'origine.

ACCES PAR CLE DANS UN CHEMIN

Cette fonction est une combinaison des deux précédentes au sens où elle permet d'accéder par clé aux articles cibles d'un chemin dont l'origine est spécifiée. La clé utilisée ici est une clé dans un chemin.

Le seul opérateur de comparaison permis est "=". Cette restriction sera justifiée au chapitre 4.

1.3.2. Primitives d'extraction de valeurs

OBTENTION DES VALEURS D'ITEMS

L'utilisation des primitives d'accès décrites au point 1.3.1. précéde souvent, comme cela a déjà été dit, l'extraction des valeurs d'items des articles auxquels on a accédé. Une primitive permet d'obtenir ces valeurs.

1.3.3. Primitives de modification de données

Une base de données est censée représenter un état du réel perçu ayant existé. Dans la mesure où le réel perçu évolue, la base de données doit pouvoir être modifiée afin de s'adapter à

cette évolution. Dans ce contexte, une série de primitives de modification de données est offerte à l'utilisateur.

On reconnaît à ces primitives un caractère atomique en ce sens qu'elles sont exécutées entièrement ou pas du tout. Elles laissent en outre la base de données cohérente. En d'autres termes elles ne sont exécutées que si leur exécution respecte les contraintes d'intégrité (d'identifiant, d'existence, de classe fonctionelle) définies sur le schéma de la base de données.

Ce point décrit les primitives qui concernent les articles, les items, les chemins.

CREATION D'UN ARTICLE

Le but de la primitive de création d'article est d'insérer un article dans la base de données. Un article est inséré si sa spécification (valeurs d'items auxquels il doit être associé, chemins dans lesquels il doit être inséré) est telle que l'article respectera toutes les contraintes d'intégrité de son type. Il est dans ce cas stocké avec ses valeurs d'items dans la base de données et relié via des chemins à tous les articles avec lesquels l'utilisateur a demandé qu'il le soit.

Si une contrainte n'est pas respectée (p. ex. violation d'identifiant), la création n'est pas du tout effectuée en accord avec le principe d'atomicité des primitives.

DESTRUCTION D'UN ARTICLE

Un article détruit est retiré de la base de données et devient donc inaccessible.

Si cette destruction entraîne la violation de contraintes d'intégrité, LDA/MAG détruit tous les articles qui sont responsables de cet état incohérent de la base de données.

MODIFICATION DES VALEURS D'ITEMS D'UN ARTICLE

Cette primitive permet à l'utilisateur de modifier certaines valeurs d'items d'un article auquel il a accédé.

La modification des valeurs d'items (composants d')identifiants ne peut évidemment se faire que s'il n'y a pas violation de la contrainte.

INSERTION D'UN ARTICLE DANS UN CHEMIN

Un article est inséré comme cible d'un chemin spécifié (et comme origine du chemin inverse s'il y en a un) pour autant qu'il n'en soit pas déjà membre et que, comme précédemment, aucune contrainte ne soit violée.

RETRAIT D'UN ARTICLE D'UN CHEMIN

Le but de cette primitive est de retirer un article d'un chemin spécifié. Si le type de chemins est doté d'un inverse, les cibles du chemin inverse dont l'origine est l'article concerné sont retirées également.

1.4. Conclusion

Ce chapitre a exposé une partie du SGD virtuel LDA/MAG. Cette partie a été définie à partir des concepts de base et des primitives du Modèle d'Accès Généralisé.

LDA/MAG sera complètement défini quand le langage qu'il offre au programmeur pour supporter les primitives de manipulation de données sera décrit. Ceci constitue l'objet du troisième chapitre.

Le second chapitre se propose de voir en quoi le SGD défini ici se distingue de certains SGDs commerciaux.

CHAPITRE 2 :

Comparaison entre le SGD LDA/MAG et divers SGD commerciaux

2.1. Introduction

Lors de la phase de conception logique d'une base de données, le schéma de la base de données et les traitements qui lui sont associés sont exprimés dans les termes du SGD LDA/MAG décrit au chapitre précédent. Au terme de cette phase, on dispose d'un schéma des accès nécessaires ainsi que d'algorithmes effectifs conformes à LDA/MAG.

L'étape de conception physique exige quant à elle que le schéma et les traitements exprimés de la sorte s'accordent à ce qu'autorise le SGD cible choisi.

"La production d'un schéma conforme [à un SGD cible] consiste à obtenir, à partir d'un schéma [LDA/]MAG, un autre schéma [LDA/]MAG qui lui soit équivalent et qui en outre respecte la spécification de ce SGD". [Hainaut 86a]

La production d'algorithmes conformes à un SGD cible consiste, quant à elle, "à donner des algorithmes opérant sur le schéma non conforme, des versions LDA équivalentes (...) qui opèrent [cette fois] sur le schéma conforme" en s'accordant à ce qu'autorise le SGD cible en termes d'accès. [Hainaut 86a]

Afin d'apprécier le fondement de ces deux étapes de conception physique d'une base de données, ce second chapitre expose les restrictions qu'apportent certains SGD par rapport à ce qu'autorise LDA/MAG.

L'atelier logiciel de conception de bases de données apporte une aide dans la conception d'une base de données gérée par l'un des trois Systèmes de Gestion de Données souvent considérés comme concurrents : le SGBD Codasyl 71/73, le SGBD relationnel SQL/DS et le système de gestion de fichiers Cobol Ansi-74.

Le lecteur intéressé par la spécification complète de ces SGD peut consulter la documentation qui est largement diffusée. Les propos qui suivent ne présentent que les particularités que ces SGD requièrent par rapport à LDA/MAG ainsi que les primitives qu'ils admettent.

2.2. Restrictions de trois SGD commerciaux par rapport à LDA/MAG et primitives admises

La liste donnée ci-dessous expose d'une façon structurée la plupart des contraintes qui peuvent être posées par un SGD sur un schéma des accès nécessaires et sur des algorithmes effectifs conformes à LDA/MAG. L'étude de transformation d'algorithmes doit tenir compte de ces contraintes afin qu'un algorithme opère sur un schéma conforme au SGD choisi en n'utilisant que des primitives admises par ce SGD.

2.2.1. Restrictions quant au schéma d'une base de données

Les contraintes majeures que les SGD Codasyl 71/73, SQL/DS et Cobol Ansi-74 font peser sur un schéma LDA/MAG d'une base de données sont les suivantes :

Les types d'articles

- Pas de type d'articles sans item (SQL/DS, Cobol).
- Pas de type d'articles système (SQL/DS, Cobol).

Les items

- Pas d'item facultatif (Codasyl, SQL/DS, Cobol).
- Pas d'item décomposable (SQL/DS).
- Pas plus de 99 niveaux de décomposition (Codasyl).
- Pas plus de 49 niveaux de décomposition (Cobol).
- Pas d'item répétitif (SQL/DS).
- Pas de répétitivité variable sans item compteur (Codasyl, Cobol).

Les types de chemins

- Pas de type de chemins (SQL/DS, Cobol).
- Tout type de chemins est de classe fonctionnelle 1-N ou N-1 et est doté d'un inverse (Codasyl).
- Les types de chemins de classe fonctionnelle 1-N sont mono-origines (Codasyl).
- Origine et cible d'un type de chemins sont des types d'articles distincts (Codasyl).
- Le type d'articles particulier système ne peut être cible d'un type de chemins (Codasyl).

Les identifiants

- Un seul item identifiant composé uniquement d'items par type d'articles (Codasyl).
- Pas d'identifiant composé de plusieurs items (Cobol).

Les clés d'accès

- Un seul item clé d'accès par type d'articles (Codasyl).
- Tout item identifiant est clé d'accès (Codasyl, Cobol).
- Pas de clé d'accès répétitive (Codasyl, Cobol).

- Plusieurs items peuvent être clés d'accès dans type de chemins de classe fonctionnelle 1-N pour les articles cibles (Codasyl).
- Si une ou plusieurs clés d'accès sont définies sur un type d'articles, une au moins doit être identifiante (Cobol).
- Pas de clé d'accès composée de plusieurs items (Cobol).
- Tout préfixe d'une clé d'accès est également une clé d'accès (Cobol).

Les contraintes d'intégrité

- Dans un type de chemins, une contrainte d'existence ne peut être posée que sur les cibles d'un type de chemins de classe fonctionnelle 1-N (Codasyl).
- Pas de contrainte d'intégrité autre que sur le format des valeurs d'un item (SQL/DS,Cobol).

2.2.2. Les primitives offertes par les SGD

Les principales primitives offertes par le SGD Codasyl 71/73 sont les suivantes :

- accès séquentiel aux articles d'un fichier.
- accès aux articles d'un type correspondant à une valeur de clé d'accès dans un fichier (premier, suivant).
- accès aux articles cibles d'un chemin 1-N, et à l'article cible d'un chemin N-1.
- accès par clé aux articles cibles d'un chemin 1-N (premier,suivant).
- obtention pour un article de ses valeurs d'items.
- création d'un article.
- suppression d'un article, avec suppression éventuelle d'articles qui lui seraient associés par des chemins (et ainsi récursivement).
- modification de valeurs d'items.
- insertion, retrait et transfert d'un article cible d'un chemin 1-N.

Les principales primitives offertes par le SGD SQL/DS sont :

- accès aux articles d'une séquence éventuellement filtrée (premier, suivant) et à leurs valeurs d'items.
- créer des articles dont les valeurs d'items sont une copie des valeurs d'items d'articles dont on donne un filtre.
- supprimer les articles qui vérifient un filtre.
- modifier les valeurs d'items des articles qui vérifient un filtre.
- créer, supprimer un article, modifier les valeurs d'items d'un article.

Les primitives essentielles du système de gestion de fichiers Cobol Ansi-74 sont :
- accès séquentiel aux articles.
- accès aux articles dont la valeur de clé est égale, supérieure, inférieure à une valeur donnée.
- création d'un article.
- suppression d'un article.
- modification des valeurs d'items d'un article.

2.3. Conclusion

L'exposé qui a été fait a montré les restrictions qu'imposent trois SGD de grande diffusion sur un schéma des accès nécessaires et sur des algorithmes effectifs conformes à LDA/MAG.

Cette description révèle la nécessité d'adapter ce schéma et ces algorithmes aux spécifications du SGD cible : le schéma doit être transformé pour répondre aux contraintes du SGD cible; les algorithmes doivent être modifiés à leur tour afin de s'exprimer en termes de schéma transformé mais aussi afin de n'utiliser que des primitives autorisées par le SGD choisi.

Seules les contraintes de schéma plus restrictives que celles imposées par LDA/MAG sont citées : les SGD qui ont été présentés permettent parfois des configurations plus laxistes que celles de LDA/MAG. Le SGD Cobol, par exemple, admet un item répétitif comme composant d'un item décomposable. Ceci n'est pas mentionné.

Les restrictions portant sur un schéma des accès nécessaires ont été répertoriées pour l'ensemble des trois SGD considérés. Cette structuration s'accorde au fait que la première phase de transformation d'un algorithme ne tient compte que de la transformation de schéma qui a été opérée, indépendamment de tout SGD particulier. Les algorithmes doivent être modifiés afin d'opérer sur le schéma transformé quelle que soit la raison de la transformation de celui-ci.

Les primitives offertes par les SGD font quant à elles l'objet d'un répertoire structuré par SGD. L'association qui est établie ainsi entre des primitives et un SGD reflète bien l'obligation qui est faite à la seconde étape de transformation d'un algorithme : modifier l'algorithme afin qu'il n'utilise que des primitives offertes par le SGD cible.

CHAPITRE 3 :

LDA, le langage offert par le SGD LDA/MAG

3.1.Introduction

Le premier chapitre était consacré au SGD LDA/MAG qui permet la définition de données ainsi que leur manipulation via un ensemble de primitives. Ces primitives sont supportées par le langage de programmation LDA (Langage de Description d'Algorithmes) décrit dans ce chapitre.

De même que LDA/MAG permet une expression des structures de données indépendamment d'un SGD particulier, le langage LDA permet la rédaction d'algorithmes indépendamment d'un langage de manipulation de données (LMD) spécifique offert par un SGD. Cette indépendance s'inscrit parfaitement dans la démarche hiérarchisée de conception d'applications sur bases de données qui est décrite en introduction.

L'apport d'une approche hiérarchisée dans la rédaction d'algorithmes se situe sur deux plans : celui de la réduction de la complexité, ensuite celui de la non-exposition à des perturbations ultérieures éventuelles. Le principe de réduction de la complexité consiste à offrir un langage de manipulation des données qui permet l'expression simple de requêtes complexes indépendamment de ce qu'autorise le LMD d'un SGD particulier tant en termes d'accès que de syntaxe. Le principe de non-exposition à des perturbations tient à ce que l'on puisse revenir sur le choix d'un SGD sans être contraint à réécrire un algorithme.

Le langage LDA répond parfaitement à ces deux principes en s'inscrivant ainsi dans la classe des langages que l'on qualifie communément de "pseudo-langages". Des pseudo-langages ont été définis (diagrammes de Nassi-Schneiderman, ordinogrammes) mais ils pèchent en particulier en ce qui concerne l'accès aux données à structure complexe et leur gestion.

Certains langages généraux d'accès aux données ont été définis tel UDL [Date 81]. Le Unified Database Language (UDL) exige que les données utilisées dans un algorithme soient désignées par les accès qui y conduisent. Ce langage ne s'insère donc pas idéalement dans une démarche hiérarchisée (par niveau d'abstraction) de conception d'applications sur bases de données. Au niveau le plus haut, on aimerait rédiger un programme de manière telle que les données utilisées soient désignées uniquement par la condition de sélection qui les définit. Le langage LDA évite cet écueil.

LDA tente de réaliser l'intégration de primitives d'accès aux données dans des structures algorithmiques traditionnelles. Travaillant sur des structures de données de LDA/MAG, il permet d'exprimer les accès les plus complexes comme les plus simples.

Le langage LDA comprend 2 niveaux : un niveau prédicatif et un niveau effectif. Le langage LDA prédicatif autorise l'expression d'algorithmes prédicatifs : les données utilisées y sont décrites par la condition de sélection qui les définit et non pas par le spécification des accès qui y conduisent. Le langage LDA effectif n'admet quant à lui que la rédaction d'algorithmes effectifs (coût minimum d'accès aux données) conformes LDA/MAG c'est à dire dont les conditions sont évaluables par accès dans le cadre de LDA/MAG (conditions telles qu'il existe une primitive de LDA/MAG qui fournit les articles vérifiant cette condition, et eux seulement).

Les propos qui suivent décrivent uniquement le langage LDA effectif, support des algorithmes effectifs conformes LDA/MAG qui seront transformés en algorithmes effectifs conformes à un des SGD cibles exposés au chapitre 2. Seul le niveau effectif du langage LDA intervient en effet dans l'étude menée dans ce mémoire. Dès lors, seul le terme "langage LDA" sera dorénavant utilisé.

L'unité d'exécution de base du langage LDA est l'algorithme. La structure d'un algorithme est résumée par le schéma de la figure 3.1.

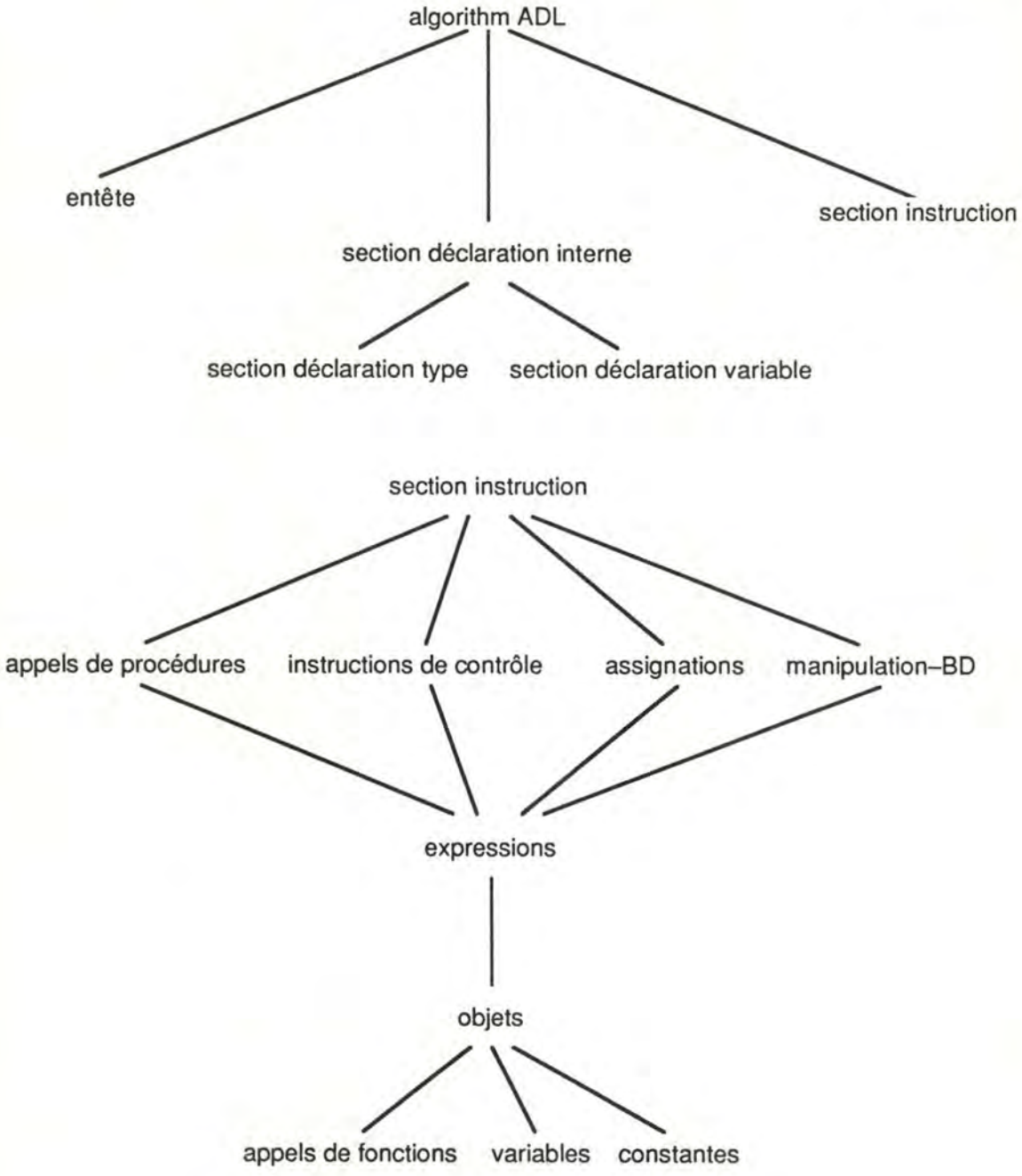


fig 3.1 : structure d'un algorithme LDA

On peut voir qu'un algorithme est composé d'un entête suivi d'une section de déclarations où l'on spécifie les types de données et les variables. Cette section est elle–même suivie d'une section d'instructions. La section des instructions peut contenir, en vrac, des appels de procédures, des instructions de contrôle (while, if, ...), des assignations et des instructions de manipulation de la base de données (création, mise à jour, suppression d'articles). Toutes ces instructions travaillent sur des expressions composées de noms d'objets pouvant être des variables, des appels de fonctions ou des constantes, des types d'articles de la base de données, des items, des types de chemins.

LDA étant un langage "Pascal-like", il offre diverses fonctionnalités propres aux langages dits évolués notamment, comme le montre la figure 3.1 ci–dessus, des instructions de contrôle

alternatives, itératives, des fonctions, des procédures, des variables structurées, ...

Il s'en distingue cependant par le fait qu'il est essentiellement orienté vers le développement d'applications sur bases de données. Cela se marque notamment par des instructions implémentant spécifiquement les primitives de manipulation de données offertes par LDA/MAG, et par la présence de variables de référence (voir ci–dessous).

Le programmeur voit la base de données sur laquelle il travaille comme un ensemble d'articles groupés en types d'article, ayant des valeurs d'items et reliés par des chemins. Dans ce contexte, une variable de référence est un mécanisme qui offre au programmeur la possibilité de désigner un article dans la base. Pour profiter de cette facilité, il définit un ensemble d'articles par le biais d'une série de conditions et fait désigner un de ces articles par une variable de référence. La référence ainsi établie peut être modifiée à souhait par l'utilisateur soit en détruisant l'article, soit en utilisant la variable pour référencer un autre article.

A un instant donné, une variable de référence désigne donc au plus un article; on peut toutefois noter qu'un article peut être référencé par un nombre quelconque de variables de référence.

Ces variables permettent en outre d'obtenir les valeurs d'items des articles qu'elles désignent.

Le paragraphe suivant s'attache, après cette brève description générale, à définir la syntaxe et la sémantique du langage LDA.

3.2. La syntaxe

Un programme écrit dans n'importe quel langage peut être vu comme une suite de caractères choisis dans un alphabet. Le problème est alors de déterminer les suites formant des programmes "valables"; il faut pour cela définir une série de règles appelée syntaxe du langage.

Le paragraphe précédent a abordé de façon générale le langage LDA. Celui–ci s'attache notamment à décrire la syntaxe de ce langage en utilisant la grammaire BNF (Backus–Naur–Form) supposée connue.

Cette syntaxe a été élaborée sur base du travail fait par M. Cadelli et D. Muller [Cadelli, Muller 85].

La définition syntaxique du langage LDA est cependant tout à fait insuffisante pour le circonscrire complètement. Il est essentiel de pouvoir connaître la signification d'un programme "valable". La définition sémantique du langage viendra dès lors compléter la description des formes syntaxiques.

L'exposé débute par la définition des symboles de base, nombres et strings, ensuite de variables et expressions, et se termine par la structure d'un algorithme.

3.2.1. Symboles de base, nombres, strings

1. SYMBOLES DE BASE

Un symbole de base peut être une <lettre>, un <chiffre>, un <symbole_spécial>, une <valeur_logique>, une <valeur_null>, ou un <mot_réservé>.

Lettres

<lettre> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

Chiffres

<chiffre> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

blancs

<blanc> ::= nombre quelconque non nul de caractères blancs, de tabulateurs ou de returns

Symboles spéciaux

<symbole_spécial> ::= + | - | / | * | ^ | < | > | = | (|) | [|] | { | } | ' | , | . | ; | : | _

Est considéré comme <symbole_spécial> tout caractère appartenant à l'alphabet du langage, mais qui n'est ni une <lettre> ni un <chiffre>.

Valeurs logiques

<valeur_logique> ::= true | false

Valeur null

<valeur_null> ::= NULL

LDA/MAG n'admettant pas les items facultatifs, la valeur conventionnelle null est utilisée pour représenter l'absence de valeur.

Elle représente également la valeur inconnue ou par défaut d'un item.

Mots réservés

<mot_réservé> ::= <valeur_logique> | <valeur_null> | and | or | not | for | while | if | next | exit | create | modify | delete | do | then | else | endfor | endif | endwhile | begin | end | algorithm | type | var | group | array | of | items | ref | boolean | string | real | integer | numeric

Avant tout développement ultérieur, il est bon de signaler que <mot_réservé> et <symbole_spécial> ne figurent dans aucune règle par la suite et sont donc donnés ici à titre documentaire.

vide

<vide> ::= quand cette clause apparaît dans les alternatives d'une règle, elle est choisie uniquement si aucune autre clause ne peut l'être.

Par exemple, dans les règles de définition d'un <nom> (Cfr. point 2 ci–dessous), <vide> est choisi dans <suite_nom> si <seq> ne peut l'être, c'est à dire si le symbole suivant lu lors de l'évaluation du nom n'est ni une <lettre>, ni un <chiffre>, ni un tiret (-), ni un underscore (_); ce symbole peut être par exemple un <blanc>, un point-virgule (;), un crochet ([]), ...

2. NOMS

<nom> ::= <lettre> <suite_nom>

<suite_nom> ::= <vide> | <seq>

<seq> ::= <lettre> <suite_nom> | <chiffre> <suite_nom> | -<seq> | _<seq>

Ex. i
CUS
num_PROD_3

Les noms sont les identificateurs définis par le programmeur. En opposition aux mots réservés qui sont des identificateurs dont la sémantique est prédéfinie, les noms n'ont de signification que celle que le programmeur veut bien leur donner. Il peut s'en servir pour identifier des variables, des fonctions, ...

3 NOMBRES

<entier_non_signé> ::= <chiffre> <suite_entier_non_signé>

<suite_entier_non_signé> ::= <entier_non_signé> | <vide>

<nombre_non_signé> ::= <entier_non_signé> <suite_nombre_non_signé>

<suite_nombre_non_signé> ::= <vide> | . <entier_non_signé> <puissance>

<signe> ::= + | -

<exposant> ::= e <valeur_exposant>

<valeur_exposant> ::= <signe> <entier_non_signé> | <entier_non_signé>

<puissance> ::= <vide> | <exposant>

Ex. 30
3.1
0.31e-3

L'<exposant> est implicitement en base 10, c'est–à–dire que 0.31e-3 doit se lire $0.31 * 10^{-3}$.

4. STRINGS

<string> ::= '<séquence de <lettre>, <chiffre> ou <symbole_spécial> ne contenant pas '> '

Ex. '3, Rue du stade'

Les strings sont des séquences de longueur déterminée de symboles arbitrairement choisis dans l'alphabet du langage.

3.2.2. Variables

<variable> ::= <class_var> | <var_refs>

<class_var> ::= <variable_non_hiérar> <hiérarchie>

<hiérarchie> ::= <vide> | . <class_var>

<variable_non_hiérar> ::= <nom> <suite_variable_non_hiérar>

<suite_variable_non_hiérar> ::= <vide> | [<indices>]

<indices> ::= <expression_arithmétique> <suite_indice_2>

<suite_indice_2> ::= <vide> | , <expression_arithmétique> <suite_indice_3>

<suite_indice_3> ::= <vide> | , <expression_arithmétique>

<var_refs> ::= (<nom>).<class_var>

Ex. CLIENT
CLIENT.NOM
(prod).prix
tab[I+J,20]

Une variable désigne un espace en mémoire centrale où une (séquence de) valeur(s) est conservée. Le contenu de cet espace à un instant donné est la valeur courante de la variable. Cette valeur peut être modifiée à souhait par l'utilisateur. Elle a en outre un type qui est le type de la variable (Cfr 3.2.4 point 2 : déclaration des données).

Un tableau est une collection d'éléments d'un type défini et disposés dans une structure à 1, 2 ou 3 dimensions. Une mesure de la distance le long d'une dimension est appelée un indice. Un indice est une valeur entière comprise entre les bornes inférieure et supérieure d'une dimension du tableau. Les éléments d'un tableau sont établis à des indices le long de chaque dimension du tableau.

Les variables hiérarchiques désignent des valeurs qui sont des composants de structures complexes de données. Une variable de ce type se présente comme une suite de champs séparés l'un de l'autre par un point. Le premier champ désigne la structure totale, le dernier est une variable simple ou indicée (qui bien sûr a un type). Chaque champ intermédiaire désigne une structure contenue dans celle désignée par le champ à sa gauche et contenant la structure désignée par le champ à sa droite.

Les variables de référence sont, comme cela a déjà été signalé, un mécanisme qui permet au programmeur de désigner un article de la base de données. Dès lors <var_refs> est défini comme étant un mécanisme qui permet au programmeur d'obtenir une valeur d'item d'un article désigné par une variable de référence.

Il s'en suit que le <nom> qui apparaît dans <var_refs> ne peut être qu'un nom de variable de référence; il ne peut de plus apparaître nulle part ailleurs dans l'ensemble des règles qui définissent la syntaxe des variables.

Une expression comme (prod).prix représente l'appel à une fonction qui accède à l'article désigné par la variable de référence "prod" et construit en mémoire centrale une structure de données contenant toutes les valeurs d'items de cet article. La partie <class_var> de l'expression (ici prix), désigne un des composants de cette structure selon le même principe que les variables hiérarchiques. Plus précisément on peut dire que cette structure est une copie (en termes de noms et de décomposition) de la structure des items de l'article désigné. Dès lors, les champs "intermédiaires" sont des noms d'items décomposables et le champ final (le plus à droite) est un nom d'item élémentaire.

Aucune variable indicée ne peut en outre apparaître dans <class_var>; en d'autres termes, des expressions du type (cli).nom[i], (étudiant).cotation[j].cours, ... ne sont pas admises.

Cette restriction paraît singulière dans la mesure où la notation indicée paraît intuitivement la plus appropriée pour la désignation d'une valeur d'item répétitif. Cependant, le refus signifié ici est justifié par le fait que ces formes posent des problèmes extrêmement délicats lorsqu'elles sont envisagées dans le contexte des transformations d'algorithmes. Ces problèmes ne sont pas exposés ici afin de ne pas anticiper sur la suite du travail. Ils seront détaillés au second paragraphe du chapitre 4.

Sans envisager le problème des transformations d'algorithmes, on peut déjà dire que ce genre de forme est incorrecte dans la mesure où, comme cela a déjà été dit, il est erroné de vouloir désigner une $i^{ème}$ ou $j^{ème}$ valeur dans l'ENSEMBLE de celles prises par un item répétitif.

3.2.3. Expressions arithmétiques et booléennes

1.EXPRESSIONS ARITHMETIQUES

<expression_arithmétique> ::= <addition> <blanc> <expression_arithmétique> |
<facteur> <suite_expression_arithmétique>

<suite_expression_arithmétique> ::= <vide> | <blanc> <addition> <blanc>
<expression_arithmétique>

<facteur> ::= <terme> <suite_facteur>

<suite_facteur> ::= <vide> | <blanc> <multiplication> <blanc> <facteur>

<terme> ::= <terme_primaire> <suite_terme>

<suite_terme> ::= <vide> | <blanc> ^ <blanc> <terme_primaire>

<terme_primaire> ::= <nombre_non_signé> | <variable> | <appel> |
(<expression_arithmétique>)

<addition> ::= + | -

<multiplication> ::= * | /

Ex. 3
C
(compte + TRANSFERT) * (pret).taux
(((a - 3) / b) ^ reste(31,4))

Une expression arithmétique est une construction à valeur numérique contenant des opérandes (<terme_primaire>) et des opérateurs. Sa valeur est définie par l'application des opérateurs aux opérandes, lesquelles doivent être à valeur unique et numérique.

Cela veut dire que les <variable> doivent être numériques, les <appel> doivent être des appels de fonctions à résultat numérique, les <var_refs> doivent rendre compte de valeurs d'items numériques.

Le <terme_primaire> d'une <suite_terme> doit être un entier en vertu du fait qu'on ne peut élever un nombre qu'à une puissance entière.

La valeur d'une <expression_arithmétique> n'est définie que si elle ne contient pas de division par zéro (0) et si la valeur de chaque <terme_primaire> est définie.

Cette valeur est réelle si au moins un des opérateurs est la division (/), ou si au moins une des opérandes a une partie décimale, ou si au moins un des <terme_primaire> de <suite_terme> est strictement négatif. La valeur est entière dans les autres cas.

L'évaluation d'une expression arithmétique commence par l'évaluation des <terme_primaire> (appels des fonctions et évaluation des variables) auxquels on applique les éventuels opérateurs ^ pour obtenir des <terme>. A ces mêmes <terme> on applique les éventuelles <multiplication> pour obtenir des <facteur> et l'évaluation se termine après application des éventuelles <addition>. La priorité des opérateurs peut cependant être contrée par l'utilisation de parenthèses qui provoquent l'évaluation immédiate de l'expression qu'elles entourent.

D'autre part, la présence de <blanc> entre les opérateurs et les opérandes est imposée afin d'éviter toute ambigüité. Si ce n'était le cas, l'expression "compte-3" (par exemple) pourrait être un <nom> aussi bien qu'une <expression_arithmétique>. Le cas présent, elle ne peut bien sûr être qu'un <nom> et "compte - 3" ne peut être, par définition de <nom>, qu'une <expression_ arithmétique>.

2. EXPRESSIONS BOOLEENNES

<expression_booléenne> ::= <facteur_booléen> <suite_expression_booléenne>

<suite_expression_booléenne> ::= <vide> | or <expression_booléenne>

<facteur_booléen> ::= <terme_booléen> <suite_facteur_booléen>

<suite_facteur_booléen> ::= <vide> | and <facteur_booléen>

<terme_booléen> ::= not <primaire_booléen> | <primaire_booléen>

<primaire_booléen> ::= <expression_de_test> | (<expression_booléenne>)

<opérateur_de_comp_exp_booléenne> ::= <opérateur_de_comparaison> | <>

<expression_de_test> ::= <opérande> <opérateur_de_comp_exp_booléenne> <opérande>

<opérande> ::= <exp> | <référence_nulle>

<référence_nulle> ::= ( )

Ex. (a + b < C)
found = true and i < max
(pro).q_stk >= 10

Une <expression_booléenne> est une construction pouvant prendre deux valeurs différentes, true ou false.

Pour évaluer une <expression_booléenne>, on évalue d'abord les <expression_de_test> qui la composent; on y applique les éventuels opérateurs unaires "not"; on a ainsi évalué les <terme_booléen>; on évalue leur conjonction en appliquant les éventuels opérateurs "and" et on obtient ainsi les <facteur_booléen>; on évalue leur éventuelle disjonction (or) pour obtenir la valeur de l'expression totale. Toutefois, l'ordre d'application des opérateurs peut être remanié par l'utilisation de parenthèses.

L'évaluation d'une <expression_de_test> se fait en évaluant d'abord ses 2 opérandes, qui

doivent être de même type, et en vérifiant ensuite si elles respectent la relation établie entre elles par l'<opérateur_de_comp_exp_booléenne> >.

Les <opérande> peuvent être des nombres, string, booléens ou la <référence_nulle>.

La <référence_nulle> désigne l'absence de référence à un article de la base de données. On pose qu'une variable de référence qui ne désigne aucun article est égale à la <référence_nulle> et qu'elle en est différente si elle en désigne un.

L' <opérateur_de_comparaison> se limite à "<>" ou "=" si les <opérande> sont de type booléen, des variables de référence ou la <référence_nulle>.

3.2.4. Structure d'un algorithme

1. STRUCTURE GENERALE D'UN ALGORITHME

<structure_d'algorithme> ::= <entête_d'algorithme>
<corps_d'algorithme>.

<entête_d'algorithme> ::= algorithm <nom>

<corps_d'algorithme> ::= <section_déclaration_interne>;
<section_instruction>

Un algorithme est composé de 3 parties ordonnées :
- l' entête d'algorithme spécifiant le nom de l'algorithme.
  Par exemple : algorithm traitement-client
- la section de déclaration interne qui définit les types de données et les variables utilisées dans l'algorithme.
- la section des instructions décrivant les actions à exécuter.

2. DECLARATION DES DONNEES

Section de déclaration des types et des variables

<section_déclaration_interne> ::= <section_déclaration_type>;
<section_déclaration_variable>

<section_déclaration_type> ::= type <déclarations_types> | <vide>

<déclarations_types> ::= <déclaration_type> <suite_déclarations_types>

<suite_déclarations_types> ::= <vide> | ; <déclarations_types>

<déclaration_type> ::= <nom> = <type_de_variable>

<section_déclaration_variable> ::= var <déclarations_variables> | <vide>

<déclarations_variables> ::= <déclaration_variable> <suite_déclarations_variables>

<suite_déclarations_variables> ::= <vide> | ; <déclarations_variables>

<déclaration_variable> ::= <noms> : <type_de_variable>

<noms> ::= <nom> <suite_noms>

<suite_noms> ::= <vide> | , <noms>

La section de déclaration interne comporte deux sections ordonnées : la section de déclaration des types de données suivie de la section de déclaration des variables. Chacune de ces sections peut être vide.

Si la section de déclaration des types de données est présente, elle spécifie une liste de déclarations de types de données. Chaque élément de cette liste associe un nom identifiant avec un type de données non standard, non offert par ADL (voir point suivant). Ce type de données non standard est décrit en termes de types de données offerts par LDA ou non standard. Les types de données non standard doivent être définis avant leur utilisation dans la déclaration du type qu'ils servent à décrire.

Si la section de déclaration des variables est non vide, elle spécifie une liste de déclarations de variables. Chaque élément de cette liste associe un ou plusieurs noms de variables, identifiant avec un type de données. Ce type de données détermine l'ensemble de valeurs que peuvent prendre les variables déclarées dans l'association.

Types de données

<type_de_variable> ::= <simple> | <structuré> | <nom>

<simple> ::= boolean | real | integer |
numeric(<entier_non_signé>, <entier_non_signé>) |
string(<entier_non_signé>)

<structuré> ::= <groupe> | <tableau> | <items_de> |<ref_de>

<groupe> ::= group <liste_des_champs> end

<liste_des_champs> ::= <déclaration_variable> <suite_liste_des_champs>

<suite_liste_des_champs> ::= <vide> | ; <liste_des_champs>

<tableau> ::= array [<liste_des_indices>] of <type_composant>

<liste_des_indices> ::= <entier_non_signé> <suite2_liste_des_indices>

<suite2_liste_des_indices> ::= <vide> |, <entier_non_signé> <suite3_liste_des_indices>

<suite3_liste_des_indices> ::= <vide> | , <entier_non_signé>

<type_composant> ::= <simple> | <ref_de> | <items_de> | <nom>

<ref_de> ::= ref of <nom>

<items_de> ::= items of <nom>

```
Ex. type  MAT     = array [10] of integer;
          ELEMENT = group
                        MATRICE : MAT;
                        TAILLE  : integer;
                    end;
    var   el1, el2  : ELEMENT;
```

Un type de variables peut être simple. Une variable de ce type ne contient qu'une information.

Une variable de type boolean peut contenir la valeur true ou false.

Une variable de type integer peut contenir une valeur entière positive, négative ou nulle.

Une variable de type real peut contenir une valeur réelle positive, négative ou nulle.

Une variable de type numeric peut contenir une valeur positive ou négative avec un nombre de chiffres spécifié en parties entière et décimale. Une variable de ce type peut également contenir la valeur nulle.

Une variable numeric constitue un cas particulier de variable à valeur réelle.

Une variable de type string peut contenir un nombre spécifié de caractères.

Un second type de variable est le type structuré. Une variable d'un tel type peut contenir plus d'une information.

group

Une variable de type group est composée de différents champs qui peuvent être de différents types. Chaque champ est spécifié par une déclaration de variable associant un nom avec un type de données standard ou préalablement défini.

array

Un type de variables array définit un tableau. Ce type est déclaré en déterminant le nombre de dimensions du tableau, la borne supérieure de chaque dimension et le type des éléments du tableau.

Un tableau peut avoir 1, 2 ou 3 dimensions. Ce nombre est déterminé par le nombre d'indices (<entier_non_signé>) définis dans la liste des indices. Un indice définit le nombre maximum d'éléments dans une dimension du tableau.

Les éléments du tableau ne peuvent être ni de type group, ni de type array.

items_de

Une variable d'item est destinée à contenir les valeurs d'items d'un article qui a appartenu, appartient encore ou appartiendra à la base de données. Le nom spécifié dans la déclaration doit donc être celui d'un type d'articles de la base de données. Ce nom précise le type d'articles dont la variable est autorisée à contenir les valeurs d'items.

Une variable d'item ne contient pas la référence de l'article dont elle contient les valeurs d'items. Une variable d'item permet de conserver des valeurs d'items au-delà de toute modification de la base de données et des accès que l'on y fait.

Une variable d'item se manipule comme une variable hiérarchique. Sa structure est en fait une copie de celle des items du type d'articles tant en termes de noms que de décomposition .

Le programmeur ne peut toutefois déclarer des variables d'items pour des types d'articles ayant au moins un item répétitif. Une variable d'item peut être vue comme un moyen rapide de déclarer une variable de type "group". Dès lors, toute variable d'item doit se plier aux exigences de ce type. En particulier, cela veut dire que pour contenir les valeurs d'un item répétitif, la seule solution serait d'avoir un champ de type tableau. Or, LDA ne reconnaît que les tableaux à longueur fixe. Ceux-ci ne peuvent évidemment contenir les valeurs d'un item à répétitivité variable illimitée.

ref_de

Une variable de type ref of est une variable de référence. Le nom spécifié dans la déclaration doit désigner un type d'article de la base de données. Ce nom précise le type des articles dont la variable peut être référence. Toutes les variables de référence sont déclarées de cette façon; une cependant fait une exception qui mérite d'être signalée. Comme cela a déjà été dit au premier chapitre, il existe dans chaque base de données un article système qui porte le nom de la base de données. Le programmeur d'application doit pouvoir désigner cet article. LDA lui offre donc une variable de référence "spéciale" qu'il n'a pas besoin de déclarer. Cette variable désigne en permanence l'article système et porte le nom de cet article. Le programmeur ne peut évidemment lui assigner aucune autre référence.

type nom

La déclaration d'une variable de type <nom> ne décrit pas en détail le type de la variable. Elle fait référence à un nom de type non standard défini par l'utilisateur et déclaré dans la section de déclaration des types de données.

3. SECTION DES INSTRUCTIONS

<section_instruction> ::= begin <instructions> end

<instructions> ::= <instruction> <suite_instructions>

<suite_instructions> ::= <vide> | ;<instructions>

<instruction> ::= <assignation> |
<instr_for> |
<instr_if> |
<instr_while> |
<instr_next> |
<instr_exit> |
<appel> |
<manipulation_bd> |
<vide>

Ex.
```
begin
      i := 1;
      while i <= n do
        print ('un de plus');
        i := i + 1
       endwhile
end
```

La section des instructions d'un algorithme LDA est une suite d'instructions précédée du mot réservé "begin" et terminée par "end".

EXPRESSIONS DE COLLECTION

<expression_collection> ::= <nom> <suite_expression_collection> |
<entier_non_signé> . . <expr_intervalle>

<suite_expression_collection> ::= . . <expr_intervalle> | <prédicat>

<expr_intervalle> ::= <entier_non_signé> | <nom>

<prédicat> ::= <condition_vide> | <condition_sur_clé> |
<condition_sur_chemin> |
<condition_sur_clé_dans_un_chemin>

<condition_vide> ::= ( )

<condition_sur_clé> ::= <conditions_sélection_items>

<conditions_sélection_items> ::= <cond_sélection_item> |
(<cond_sélection_plus_d'un_item>)

<cond_sélection_item> ::= (: <nom> <suite_cond_sélection_item>)

<suite_cond_sélection_item> ::= <opérateur_de_comparaison> <exp> |
<conditions_sélection_items>

<opérateur_de_comparaison> ::= < | > | = | <= | >=

<exp> ::= <expression_arithmétique> | <string> |
<valeur_logique> | <valeur_null> |
<variable> | <appel>

<cond_sélection_plus_d'un_item> ::= <cond_sélection_item> and
<cond_sélection_item>
<suite_cond_sélection_plus_d'un_item>

<suite_cond_sélection_plus_d'un_item> ::= <vide> | and <cond_sélection_item>
<suite_cond_sélection_plus_d'un_item>

<condition_sur_chemin> ::= (<nom> : <nom>)

<condition_sur_clé_dans_un_chemin> ::= (<condition_sur_chemin> and
<cond_sélection_item>
<suite_cond_clé_chemin>)

<suite_cond_clé_chemin> ::= <suite_cond_sélection_plus_d'un_item>

Ex. 1..5
INF..SUP
client ((: nom = 'DUPONT') and (cc : com))
client (: adresse ((: rue = 'louise') and (: num = ncli)
and (: localite = 'Bruxelles')))

Un intervalle (p. ex. inf..sup) désigne une séquence de valeurs entières consécutives dont la plus petite est donnée par l'expression de gauche, et la plus grande par celle de droite. Ces deux expressions doivent être des entiers ou des variables entières; elles sont appelées bornes (inférieure et supérieure) de l'intervalle. L'intervalle n'est défini que si la borne supérieure est au moins aussi grande que la borne inférieure.

Une expression de collection peut être également un <nom> suivi d'un <prédicat>. Elle désigne alors un ensemble d'articles de la base de données . Le <nom>, qui doit être un nom de type d'articles, désigne l'ensemble de tous les articles de ce type. Le <prédicat> restreint cet ensemble au sous– ensemble des articles vérifiant la condition qu'il énonce; c'est en quelque sorte un filtre.

Le langage permet d'accéder aux articles de l'ensemble ainsi défini. On remarque que les types de conditions auxquelles un <prédicat> peut se ramener correspondent aux primitives d'accès permises par LDA/MAG. La <condition_vide> correspond à l'accès séquentiel, la <condition_sur_ clé> à l'accès par clé, la <condition_sur_chemin> à l'accès par chemin et la <condition_sur_clé_dans_un_chemin> à l'accès par clé dans un chemin.

Les <nom> qui figurent dans une <condition_sur_clé> ne peuvent être que des noms d'items. Rappelons que les accès par clé ne peuvent se faire que sur base d'une seule clé et que si celle–ci est un groupe d'item ou un item décomposable le seul <opérateur_de_comparaison> permis est "=". Dans ce cas, tous les composants élémentaires de la clé doivent apparaître une et une seule

fois dans la <condition_sur_ clé> et doivent en outre être reliés par des "and". De plus, tout nom d'item figurant directement à la gauche d'un <opérateur_de_comparaison> doit être le nom d'un item élémentaire; ceci sera justifié lors de l'exposé des transformations.

Les restrictions sont semblables pour l'accès par clé dans un chemin, cependant l'<opérateur_de_comparaison> est obligatoirement "=" quelle que soit la structure de la clé.

La sémantique d'une <condition_sur_clé> est assez simple. Client(:nom = 'dupont') désigne l'ensemble des articles du type "client" dont la valeur de l'item–clé "nom" est 'dupont'; plus simplement c'est l'ensemble des clients qui s'appelent 'dupont'. Cette sémantique peut être facilement étendue au cas des items répétitifs. Il suffit de dire que l'expression ci–dessus désigne l'ensemble des articles du type "client" dont UNE valeur de l'item–clé "nom" est 'dupont'.

Une <condition_sur_clé> compare une valeur de clé avec <exp>. <variable> et <appel> dans <exp> désignent au contraire des mêmes clauses (<variable> et <appel>) dans <expression_arithmétique>, des variables et des appels de fonctions autres que numériques.

La sémantique d'une <condition_sur_chemin> mérite plus d'attention. A (ch : b) (p. ex.) désigne l'ensemble des articles du type A, cibles du chemin ch dont l'article (désigné par la variable de référence) b est origine (le deuxième <nom> dans la <condition_sur_chemin> est toujours un nom de variable de référence). Le chemin est ici parcouru de b vers les articles de type A. La sémantique de cette forme n'est pas ambigüe parce que b n'est pas de type A.

S'il s'avérait que b fût de type A, il s'agirait d'un type de chemins récursif et la formulation de la <condition_sur_chemin> serait plus délicate.

En effet, considérons l'exemple suivant :

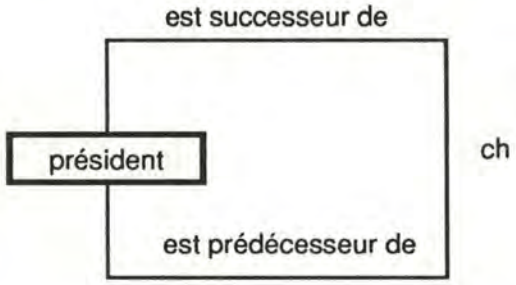


fig 3.2. : exemple de type de chemins récursif

L'expression "président (ch : p)" désigne–t–elle l'ensemble des présidents successeurs de p ou prédécesseurs de p ? (cet ensemble se réduit, le cas présent, à un singleton). La syntaxe ne permet pas de déterminer la sémantique de façon univoque.

On remplacera donc "ch" par un nom de rôle qui à l'avantage d'être non ambigu.

Il reste cependant à déterminer la sémantique de cette nouvelle forme de <condition_sur_chemin>. A priori l'expression "président (est successeur de : p)" désigne l'ensemble des présidents successeurs de p. Cette interprétation est toutefois erronée dans la mesure où une <condition_sur_ chemin> ne se lit pas de gauche à droite mais bien de droite à gauche; en d'autres termes, de l'origine vers la cible. De cette lecture particulière on déduit que c'est p qui joue le rôle de successeur. L'expression désigne donc l'ensemble des présidents dont p est successeur.

Le premier <nom> d'une <condition_sur_chemin> peut donc être un nom de chemin comme un nom de rôle, et dans ce dernier cas, le rôle est joué par l'article origine du chemin.

Si on compare une <condition_sélection_item> à une <condition_sur_chemin>, on remarque une similitude entre "(: <nom>" et "(<nom> : <nom>)". Ces deux expressions expriment les relations qui existent entre les objets de la base de données. La syntaxe veut que le nom de ces relations soit précédé de "(" et suivi de ":". La relation entre un type d'articles et un item étant anonyme, rien n'apparaît entre ces deux symboles dans "(:<nom>". Inversément, un type de chemins ne pouvant être anonyme, c'est son nom qui apparaît à gauche dans "(<nom> : <nom>".

La sémantique d'une <condition_sur_clé_dans_un_chemin> combine celles décrites à propos des clés et des chemins.

Un intervalle de valeurs d'une expression de collection peut être parcouru par l'instruction for décrite ci-après. On peut également cheminer à travers un ensemble d'articles défini par une expression de collection grâce à cette même instruction ou une assignation.

assignation

<assignation> ::= <class_var> := <expression_assignation>

<expression_assignation> ::= <opérande> | <expression_booléenne> | <nom> <prédicat>

Ex. A := 5 + (B * 4)
CLI := CLIENT (:NUM = 10)

Une instruction d'assignation permet d'affecter une valeur à une variable. La variable et la valeur constituent respectivement les parties gauche et droite de l'instruction d'assignation. Ces parties gauche et droite doivent être de même type.

Une instruction d'assignation ne peut affecter une valeur qu'à une variable de type simple (boolean, real, integer, numeric, string), au champ terminal d'une variable <items_de> ou à une variable de type <ref_de>. Dans ce dernier cas, la partie droite de l'instruction ne peut être qu'une référence nulle ou un nom d'un type d'articles de la base de données suivi d'un prédicat. Le nom de ce type d'articles doit être celui qui a servi à déterminer le type de la variable de référence. Le prédicat détermine un ensemble de 0, 1 ou N articles, et la référence d'un (quelconque) de ces articles est affectée à la variable de référence (la variable acquiert une valeur indéterminée dans le cas où aucun article n'est sélectionné). On peut s'étonner de ce que l'on n'exige pas que le prédicat soit identifiant (c'est-à-dire que l'ensemble qu'il détermine contienne au plus un article). Cette singulière prise de position est justifiée au chapitre 4.

Un <appel> dans <opérande> ne peut être un appel de procédure.

instruction for

<instr_for> ::= for <variables_de_parcours> := <séquence> do <instructions> endfor

<variables_de_parcours> ::= <variable> | {<item_répétitif_décomposable>}

<item_répétitif_décomposable> ::= <comp_variable_parcours>,
<comp_variable_parcours>,
<suite_item_répétitif_décomposable>

<suite_item_répétitif_décomposable> ::= <vide> |,<comp_variable_parcours>
<suite_item_répétitif_décomposable>

<comp_variable_parcours> ::= <variable> = <nom>

<séquence> ::= <expression_collection> |<var_refs>

Ex. 1. for x := 1..5
...
endfor;

2. for (num = numero, loc = localité, r = rue) := (cli).adresse
...
endfor;
On obtiendra successivement dans num, loc et r les différentes valeurs de l'item répétitif décomposable adresse de l'article désigné par la variable CLI.

3. for MOT := (OUVR).MOT_CLE
...
endfor;
On obtiendra successivement dans la variable MOT les valeurs de l'item répétitif MOT_CLE de l'article désigné par la variable de référence OUVR.

4. for {c = cours, pt = note} := (élève1).cc où cc est un item répétitif décomposable composé de cours et note.

L'instruction for définit une boucle d'exécution en permettant plusieurs exécutions successives des instructions qui figurent entre les mots réservés "for" et "endfor" ("corps" de la boucle).

A chaque itération, la valeur de l'élément suivant de la séquence est assignée à la variable de parcours. L'exécution de la boucle for se termine s'il n'y a plus d'élément de la séquence à assigner à la variable de parcours. Cette variable acquiert alors une valeur INDETERMINEE, et l'instruction suivant le mot réservé "endfor" est exécutée. Si la séquence ne spécifie aucun élément, la boucle se termine directement; plus précisément, le corps n'est pas exécuté.

La séquence peut spécifier :
- un intervalle de nombres entiers. La variable de parcours est alors une variable integer. L'instruction for prend fin lorsque la variable de parcours a parcouru toutes les valeurs de l'intervalle.
- un <nom> suivi d'un <prédicat>. La variable de parcours est alors une variable de référence du type d'articles dont le nom précède le prédicat. L'instruction for se termine lorsque tous les éléments de la séquence ont été parcourus.
- un champ item répétitif élémentaire d'une variable de référence (<var_refs>; il ne peut donc y avoir qu'un seul champ après la variable de référence de <var_refs>). Dans ce cas, la variable de parcours est de même type que l'item répétitif. L'instruction for a

alors pour but de parcourir les valeurs de l'item répétitif et d'assigner successivement ces valeurs à la variable de parcours. L'instruction se termine lorsque toutes les valeurs de l'item répétitif ont été parcourues.

- un champ item répétitif décomposable d'une variable de référence (<var_refs>; il ne peut donc y avoir qu'un seul champ après la variable de référence de <var_refs>). Dans ce cas, c'est un ensemble de variables de parcours qui permet l'itération. Cet ensemble est constitué d'associations variable, nom de composant de l'item répétitif décomposable (pour tout composant élémentaire de cet item) et est entouré d'accolades ({}), symboles usuels de la notion d'ensemble. L'instruction for a alors pour but de parcourir les valeurs décomposables de l'item répétitif décomposable. A chaque itération, chaque composant de la valeur décomposable est affecté à la variable de parcours qui lui correspond dans l'association variable, nom de composant de l'item répétitif décomposable. Le nom de composant est le <nom> qui apparaît dans <comp_variable_parcours>. L'instruction for prend fin lorsque toutes les valeurs décomposables ont été parcourues.

Si un item répétitif (décomposable ou non) est pseudo-facultatif, l'accès à ses valeurs peut donner une et une seule fois NULL. Or, cela a déjà été dit au premier chapitre, NULL ne doit pas être manipulé librement dans un algorithme (par ex. dans une expression arithmétique). Dès lors, le programmeur doit être prudent en vérifiant si la valeur d'item obtenue est NULL ou pas, et en réagissant en conséquence.

Afin de conserver une logique à une séquence, la séquence de l'instruction for doit désigner la même suite d'éléments durant toute l'exécution de l'instruction.

instruction conditionnelle

<instr_if> ::= if <expression_booléenne> then <instructions> <suite_instr_if>

<suite_instr_if> ::= endif | else <instructions> endif

Ex. if A <> () then PRINT (MESSAGE)
else ENREGISTRER (A)
endif

Une instruction if définit une exécution conditionnelle d'instructions en fonction de l'évaluation de l'expression booléenne. Si l'expression booléenne est évaluée à true, les instructions suivant le mot réservé "then" sont exécutées. Si l'expression booléenne est évaluée à false, les instructions suivant le mot réservé "then" ne sont pas exécutées; dans ce cas, ce sont celles suivant le mot réservé "else" qui sont exécutées, si ce mot réservé apparaît dans l'instruction.

La clause endif se réfère toujours à la dernière instruction if rencontrée.

instruction while

<instr_while> ::= while <expression_booléenne> do <instructions> endwhile

Ex. while (A <> B) and (PRESENT = false) do
CONSULTER (A,PRESENT)
endwhile

Comme l'instruction for, l'instruction while permet l'exécution itérative des instructions qui figurent entre les mots réservés "while" et "endwhile". Cependant, l'instruction while n'effectue aucune assignation à une variable de parcours de boucle.

A chaque entrée dans la boucle, l'expression booléenne est évaluée. Si elle est évaluée à true, les instructions du corps de boucle sont exécutées; dans le cas contraire, l'instruction while prend fin.

instruction next

<instr_next> ::= next <suite_instr_next>

<suite_instr_next> ::= <vide> | <nom> | {<item_répétitif_décomposable>}

Ex. next CLI
next {num = numero, loc = localité, r = rue}

L'instruction next force la terminaison de l'itération courante d'une boucle définie par une instruction for.

Si un nom ou un ensemble d'associations variable, composant d'item répétitif décomposable entouré d'accolades est spécifié, cela doit correspondre aux variables de parcours de boucle. Dans ce cas, la terminaison de l'itération courante concerne la boucle ayant ces variables comme variables de parcours.

La terminaison de l'itération courante d'une boucle implique la fin de l'exécution de toutes les boucles qu'elle contient.

Si next est non accompagné, la terminaison de l'itération courante concerne la boucle la plus imbriquée.

La terminaison de l'itération courante est suivie immédiatement d'une assignation à la ou aux variable(s) de parcours de boucle.

instruction exit

<instr_exit> ::= exit <suite_instr_exit>

<suite_instr_exit> ::= <vide> | <nom> | {<item_répétitif_décomposable>}

Ex. exit PROD
exit {num = numero, loc = localité, r = rue}

L'instruction exit force la fin de l'exécution d'une instruction for. C'est donc une façon supplémentaire de terminer une boucle for. La variable de parcours conserve la valeur qu'elle a au moment où le exit est exécuté.

Si un nom ou un ensemble d'associations variable, composant d'item répétitif décomposable entouré d'accolades est spécifié, cela doit correspondre aux variables de parcours de l'instruction for. C'est alors l'instruction for qui a ces variables de parcours qui est contrainte à prendre fin. Les instructions for imbriquées dans celle-ci sont évidemment contraintes elles aussi à la terminaison.

Si exit est non accompagné, la terminaison forcée concerne l'instruction for la plus imbriquée.

appel de procédure ou de fonction

<appel> ::= <nom> <suite_appel>

<suite_appel> ::= <vide> | (<liste_paramètres_actuels>)

<liste_paramètres_actuels> ::= <opérande> <suite_liste_paramètres_actuels>

<suite_liste_paramètres_actuels> ::= <vide> | ,<liste_paramètres_actuels>

Ex. ENREGISTRER (A);
X := SIN (Y);

Un algorithme LDA peut appeler des procédures et des fonctions.

Aucune déclaration de procédure ou fonction n'apparaît dans un algorithme dans la mesure où celles-ci sont écrites et compilées ailleurs.

Une procédure ou fonction est appelée en spécifiant son nom suivi éventuellement de la liste des paramètres actuels qu'elle exige.

instructions de manipulation d'objets de la base de données

<manipulation_bd> ::= <instr_create> |
<instr_modify> |
<instr_delete>

instruction de création

<instr_create> ::= create <nom> := <nom> <conditions_creation>

<conditions_création> ::= <condition_création> |(<conditions>) |
<condition_vide>

<conditions> ::= <condition_création> and
<condition_création><suite_conditions>

<suite_conditions> ::= <vide> | and <condition_création> <suite_conditions>

<condition_création> ::= <condition_création_chemin> |
<condition_création_item>

<condition_création_chemin> ::= (<nom> : <nom>)

<condition_création_item> ::= (:<nom> <suite_condition_création_item>)

<suite_condition_création_item> ::= = <expression> | <condition_création_item> |
(<cond_création_plus_d'un_item>)

<expression> ::= <exp> | {<liste_exp>}

<liste_exp> ::= <liste_exp_simple> | <liste_exp_décomposable>

<liste_exp_simple> ::= <exp> <suite_liste_exp_simple>

<suite_liste_exp_simple> ::= <vide> | ,<liste_exp_simple>

<liste_exp_décomposable> ::= {<élément>} <suite_liste_exp_décompable>

<suite_liste_exp_décomposable> ::= <vide> | ,<liste_exp_décomposable>

<élément> ::= <vide> | <assign> <suite_élément>

<suite_élément> ::= <vide> | ,<assign> <suite_élément>

<assign> ::= <nom> = <exp>

<cond_création_plus_d'un_item> ::= <condition_création_item> and
<condition_création_item>
<suite_cond_création_plus_d'un_item>

<suite_cond_création_plus_d'un_item> ::= <vide> | and <condition_création_item>
<suite_cond_création_plus_ d'un_item>

Ex. create V_A := A (:ITAREP = {X+Y,5,6})
create V_B := B (:ITBREP = { {F1 = 1 , F2 = 2} , {F1 = 3 , F2 = 4} })
create V_C := C ((:IT = X) and (CH : D))

Un article dont le type a pour nom celui déclaré dans la partie droite de l'assignation est créé. La référence de cet article est assignée à la variable de référence dont le nom figure dans la partie gauche de l'assignation. L'article créé vérifie les conditions de création. Ainsi, il a pour valeurs d'items celles définies dans les <condition_création_item>, et est rattaché par des chemins aux articles spécifiés dans les <condition_création_chemin>.

Seul un nom d'item pseudo-facultatif de l'article créé peut être absent des <condition_ création_item>. Dans ce cas, la valeur d'item correspondante est NULL. La valeur d'item d'un item obligatoire ne peut être NULL.

Pour un item répétitif, on associe à l'article un ensemble de valeurs d'items (entouré d'accolades dans l'expression). Si de plus, l'item répétitif est décomposable, chaque valeur

décomposable est associée à l'article en précisant l'ensemble des associations composant d'item répétitif décomposable, composant de la valeur décomposable. Cet ensemble est entouré lui aussi d'accolades.

Si <élément> est <vide> (ne contient aucune <assignation>), cela veut dire que tous les composants de l'item répétitif sont pseudo-facultatifs et que la valeur d'item correspondante associée à l'article est la valeur décomposable NULL.

Un <appel> dans <exp> ne peut désigner un appel de procédure.

Si l'article créé est soumis à une contrainte d'existence, une <condition_création_chemin> spécifiant le chemin intervenant dans la contrainte, doit accompagner l'instruction create. L'article créé est attaché à un article désigné par la variable de référence dont le nom figure à la droite du signe ":". Cette liaison s'opère par le chemin dont le nom apparaît à la gauche de ce signe.

Tout item apparaît au plus une fois dans les conditions de création. Toute <condition_création_chemin> est identifiée par un nom de type de chemins et un nom de variable de référence. Un même nom de type de chemins peut apparaître plusieurs fois avec des variables de référence différentes (si la classe fonctionnelle du type de chemins le permet) et une même variable de référence peut apparaître plusieurs fois avec des noms différents de type de chemins.

instruction de modification

<instr_modify> ::= modify <nom> <conditions_modification>

<conditions_modification> ::= <cond_mod_item> | <cond_mod_chemin>

<cond_mod_item> ::= <condition_mod_item> | (<conditions_mod_items>)

<conditions_mod_items> ::= <condition_mod_item> and <condition_mod_item> <suite_conditions_mod _items>

<suite_conditions_mod_items> ::= <vide> | and <condition_mod_item> <suite_conditions_mod_items>

<condition_mod_item> ::= (:<nom> <suite_condition_mod_item>)

<suite_condition_mod_item> ::= <suite_condition_création_item> | <addition> {<elem>}

<elem> ::= <exp> | <élément>

<cond_mod_chemin> ::= (<nom> : <suite_cond_mod_chemin>)

<suite_cond_mod_chemin> ::= <nom> | 0 <nom>

Ex. modify CLI ((:NOM = 'DUPONT') and (:NUMERO = 12))
modify COM (CC : 0 CLI)
modify COM (CC : CLI)
modify OUVR (:MOT_CLE = {'ORDINATEUR','SOFT','HARD'})
modify OUVR (:MOT_CLE + {'OS'})
modify OUVR (:MOT_CLE - {'BOTANIQUE'})

Le nom dans l'instruction modify doit désigner une variable de référence. L'article que cette variable désigne est modifié.

L'instruction modify peut être accompagnée de conditions de modification d'items. Si le signe "=" apparaît à la droite d'un nom d'item, l'article est associé à la nouvelle valeur d'item apparaissant à la droite du signe. Les valeurs d'un item répétitif peuvent être remplacées par un ensemble de valeurs entouré d'accolades. Si de plus, l'item répétitif est décomposable, chaque valeur décomposable remplaçante est définie en précisant l'ensemble des associations composant d'item répétitif décomposable, composant de la valeur décomposable remplaçante. Cet ensemble est entouré lui aussi d'accolades.

Une clause <addition> ne peut apparaître qu'après le nom d'un item répétitif à répétitivité variable. Selon le cas, un singleton est ajouté ou retiré des valeurs de l'item répétitif, associées à l'article modifié. Lorsqu'on manipule des valeurs décomposables d'items répétitifs (décomposables), les composants non spécifiés sont supposés à NULL. Pour les items simples, tout nom d'item n'apparaissant pas dans les conditions de modification n'a pas d'influence sur les valeurs d'items correspondantes.

L'article ne peut être associé à NULL pour un item obligatoire.

Le domaine de l'item dont une nouvelle valeur est associée à l'article doit être compatible avec le type de la valeur remplaçante.

Un <appel> ne peut désigner un appel de procédure.

L'instruction modify peut être accompagnée d'une condition de modification de chemin. Le nom apparaissant dans <cond_mod_chemin> est celui du chemin dont la cible est l'article désigné par la variable de référence initiale. Ainsi, se référant à la figure 3.2, modify P1 (est successeur de : P2) fait en sorte que P2 soit le successeur de P1.

Selon que le signe "0" apparaît ou non dans la condition de modification de chemin, l'article modifié est attaché ou détaché d'un article. Cet article est désigné par la variable de référence dont le nom apparaît dans <suite_cond_mod_chemin>.

instruction de destruction

<instr_delete> ::= delete <nom>

Ex. delete CLI

Le nom déclaré dans l'instruction delete doit désigner une variable de référence.

L'article référencé par cette variable est détruit de même que le sont tous les articles qui violent quelque contrainte d'intégrité suite à cette première destruction (effet "en cascade"). Les variables de référence qui désignent les articles détruits ne désignent alors plus rien.

3.3. Conclusion

Ce chapitre s'est attaché à décrire de façon générale le langage LDA pour ensuite en spécifier de façon plus précise la syntaxe et la sémantique.

De cette description, on peut notamment retenir le caractère référentiel (en opposition à contextuel) du langage LDA, ce qui le rend agréable à utiliser dans la mesure où, bien entendu, le concept de variable de référence est maîtrisé.

Le langage étant circonscrit, le problème des transformations peut être abordé dans le chapitre suivant.

Chapitre 4 :

Transformation des algorithmes

4.0. Introduction

Les premier et troisième chapitres ont décrit le SGD virtuel (LDA/MAG) qui intervient notamment dans la phase de conception logique d'une application sur base de données. Ce SGD permet la rédaction d'algorithmes effectifs conformes LDA/MAG.

La phase de conception physique qui s'ensuit débute avec la production d'un schéma conforme à un SGD commercial cible, choisi. Ce schéma s'exprime selon les termes de LDA/MAG.

La production d'un schéma conforme consiste à obtenir, à partir d'un schéma des accès nécessaires (LDA/MAG), un autre schéma LDA/MAG qui lui soit équivalent (tant du point de vue sémantique que de celui des accès), et qui ne contienne que des structures de données permises par le SGD cible.

L'une des étapes consécutives à celle décrite ci–dessus consiste à produire, à partir des algorithmes effectifs conformes LDA/MAG des algorithmes conformes au SGD commercial cible, choisi.

Les algorithmes produits à cette étape sont conformes en ce sens qu'ils doivent travailler sur un schéma conforme, n'utiliser que des primitives permises par le SGD cible et selon un enchaînement permis par ce SGD.

Ce chapitre 4 présente le résultat de l'étude qui a été menée afin de découvrir des régles de transformation d'algorithmes effectifs conformes LDA/MAG en algorithmes conformes à un SGD cible. La démarche qui a abouti à ce qui est présenté ici, a consisté d'abord à distinguer les formes syntaxiques sujettes à transformation de celles qui ne le sont pas. Ces formes syntaxiques font l'objet des paragraphes 4.1 et 4.4.

Les transformations d'algorithmes qui sont étudiées ici opèrent sur un algorithme travaillant sur un schéma non conforme au SGD cible pour aboutir, selon la première exigence de conformité d'un algorithme, à un algorithme travaillant sur un schéma conforme. Dans ce contexte, un ensemble de transformations de schéma a été défini.

Chacune des transformations de cet ensemble permet d'éliminer une structure incompatible avec un ou plusieurs SGD cibles. Par exemple, éliminer un type de chemins 1–N qui est incompatible avec SQL et Cobol. Chaque transformation de schéma a été définie de façon à ce que son application n'entraîne pas l'apparition d'une nouvelle structure incompatible avec le SGD cible. Il faut cependant se garder d'affirmer qu'une transformation rend un schéma conforme. C'est par la mise à profit "intelligente" des transformations qui lui sont offertes, que le concepteur obtient un schéma conforme.

La définition de ce "catalogue" de transformations s'est inspirée du travail fait par C. Charlot et I. Muller [Charlot, Muller 85].

Ces transformations de schéma de données sont donc utilisées dans l'étape de production d'un schéma de données conforme à un SGD cible. A la fin de leur utilisation un schéma totalement conforme au SGD cible choisi est obtenu.

Pour chaque transformation de schéma exécutée, chaque forme syntaxique sujette à transformation et susceptible d'apparaître dans un algorithme a été examinée. La façon dont elle se transforme pour travailler sur la structure de données résultat de la transformation de schéma a été étudiée. Un ensemble de règles de transformation afin qu'un algorithme réponde à la première exigence de conformité a ainsi été défini. Ces règles sont décrites au paragraphe 4.2. Elles sont appliquées à un algorithme lors d'une première étape de transformation et ne se préoccupent nullement du SGD cible dans la mesure où il ne s'agit ici que d'imposer à un

algorithme de travailler sur un schéma (transformé lors d'une étape antérieure).

L'étude de transformation d'algorithmes s'est attachée ensuite au second élément constitutif de la définition d'un algorithme conforme. Cette étude a mis en évidence pour un SGD cible considéré, les formes syntaxiques qui doivent être transformées pour répondre à ce second élément cité. La façon dont chaque forme doit être transformée a été définie et est présentée au paragraphe 4.3.

La conformité des algorithmes selon la troisième partie de sa définition (à savoir n'utiliser que des enchaînements de primitives permis par le SGD cible) n'a pas fait l'objet de l'étude à laquelle le mémoire est consacré.

Il est apparu également que les règles de transformation d'algorithmes définies dans ce chapitre présentent un caractère systématique qui peut nuire à la production d'un algorithme conforme efficace. Le paragraphe 4.5 suggère quelques optimisations de transformation de formes syntaxiques, sans toutefois aborder ce sujet de façon approfondie.

Le paragraphe 4.6 aborde en outre (sommairement) la question de la cohérence de la base de données. Dans le contexte des transformations (de schémas et d'algorithmes), la gestion de la cohérence de la base de données pose, en effet, certains problèmes qui méritent d'être mentionnés. Ce paragraphe aborde d'autre part la question des incidents ou erreurs qui peuvent survenir lorsqu'un algorithme transformé est exécuté.

4.1. Les formes syntaxiques concernées par les transformations

4.1.0. Introduction

Comme cela a déjà été dit, le langage LDA fait la synthèse de deux "types de programmation". LDA offre d'une part la possibilité d'une programmation "classique" avec les instructions communes d'itération, les alternatives, les variables structurées... . Il offre d'autre part la possibilité d'une programmation d'applications sur bases de données avec pour cela les notions de variable de référence, de primitives de manipulation de la base de données... .

Suite aux transformations de schéma de données, les expressions de cette seconde partie du langage doivent être transformées également. La partie d'algorithmique "classique" n'est, elle, pas concernée par le problème de la conformité à un SGD commercial. Cette prise de position quelque peu dichotomique sera toutefois affinée lors du paragraphe 5 qui aborde le problème des formes syntaxiques non concernées par la question de la conformité à un SGD commercial cible.

Le but du présent paragraphe est de présenter les différentes expressions du langage LDA susceptibles d'être transformées ou en d'autres termes, les expressions pour lesquelles des règles de transformation syntaxique (présentées au paragraphe 4.2 et 4.3) ont été définies. L'exposé de ces expressions est indispensable pour circonscrire parfaitement le sous-ensemble du langage LDA directement concerné par le problème de la conformité à un SGD cible.

Ces expressions concernent l'accès à la base de données, sa modification et l'extraction de données qu'elle contient.

Afin d'en faciliter la lecture et la compréhension, les expressions sont exposées en texte libre et sous forme exemplative. La structure de données sur laquelle sont basés les exemples est décrite ci-dessous.

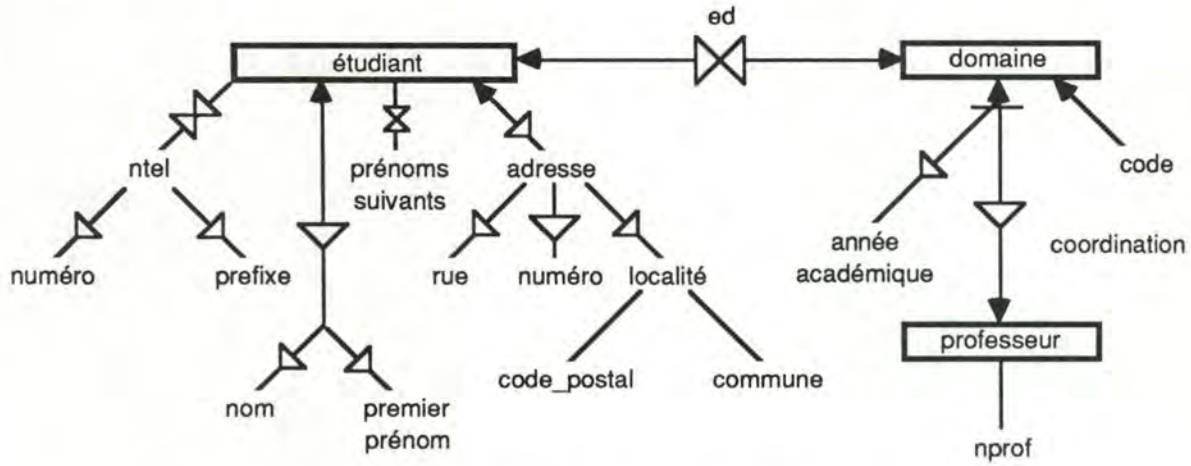


figure 4.0 : exemple de structure de données

4.1.1. Les formes syntaxiques

Les formes syntaxiques susceptibles d'être transformées sont les suivantes :

1. l'accès séquentiel aux cibles d'un chemin dont l'origine est spécifiée.
   for dom := domaine (ed : etud)
   où "etud" est une variable de référence du type d'articles "etudiant".

2. l'assignation d'une variable de référence sur base d'un accès par chemin.
   dom := domaine (coordination : prof1)
   où "prof1" est une variable de référence du type d'articles "professeur".

3. l'association de deux articles par un chemin.
   create etud := etudiant (ed : dom)
   modify etud (ed : dom)
   où "dom" est une variable de référence du type d'articles "domaine".

4. la destruction d'un chemin associant deux articles.
   modify etud (ed : 0 dom)
   où "etud" et "dom" sont des variables de référence des types d'articles "etudiant" et "domaine".

5. l'accès à des articles sur base d'item(s) clé d'accès.
   ```
   for etud := etudiant (: adresse ( (: rue = 'avenue Louise' ) and
                                     (: numero = 100) and
                                     (:localite  ( (: code_postal = 1050) and
                                                   (: commune = 'Bruxelles'))))
   for dom := domaine (: code = '6.1' )
   for etud := etudiant ( (: nom = 'Durant' ) and (: prenom = Jacques ) )
   ```

6. l'assignation d'une variable de référence sur base d'un condition de sélection portant sur un ou plusieurs items.
   ```
   etud := etudiant (: adresse (  (: rue = 'avenue Louise' ) and
                                  (: numero = 100) and
                                  (: localite ( (: code_postal = 1050) and
                                                (: commune = 'Bruxelles'))))
   dom := domaine (: code = '6.1' )
   etud := etudiant ( (: nom = 'Durant' ) and (: prenom = 'Jacques' ) )
   ```

7. l'association entre un article et des valeurs d'items.
   ```
   create prof := professeur (: nprof = maxprof + 1 )
   modify prof (: nprof = 125 )
   ```
   où "prof" est une variable de référence du type d'articles "professeur".
   ```
   modify etud (: adresse (: localite (: commune = 'Bruxelles' ) ) )
   create etud := etudiant(: prenoms_suivants = {'Alexandre', 'Arthur'} )
   modify etud (: ntel = { {prefixe = 071, numero = 123456},
                           {prefixe = 017, numero = 654321} } )
   modify etud (: ntel + {prefixe = 063, numero = 436521} )
   modify etud (: prenoms_suivants + {'Jean'} )
   ```
   où "etud" est une variable de référence du type d'articles "etudiant".

8. la dissociation entre un article et une valeur d'item.
   ```
   modify etud (: ntel - {prefixe = 063, numero = 436521} )
   modify etud (: prenoms_suivants - {'Rene'} )
   ```

où "etud" est une variable de référence du type d'articles "etudiant".

9. l'extraction de valeurs d'items.
   (dom).code
   où "dom" est une variable de référence du type d'articles "domaine".
   (etud).adresse.rue
   for pren := (etud).prenoms_suivants
   for {p = prefixe, n = numero} := (etud).ntel
   où "etud" est une variable de référence du type d'articles "etudiant".

10. l'accès par clé aux cibles d'un chemin dont on spécifie l'origine.
    for dom := domaine ((: coordinateur = prof) and (: année_académique = 8687))
    où prof est une variable de référence du type d'articles "professeur".

11. l'assignation d'une variable de référence sur base d'un accès par clé dans un chemin.
    dom := domaine ((: coordinateur = prof) and (: année_académique =8687))
    où prof est une variable de référence du type d'articles "professeur".

12. la destruction d'un article.
    delete etud ;

13. les instructions de modification de parcours de la boucle "for" portant sur des objets de la base de données (articles, valeurs d'items).
    - next etud
    - exit etud
    - next pren
    - exit pren
      où pren est une variable destinée à recevoir les valeurs d'items "prenoms_suivants".
    - next { p = prefice, n = numero }
    - exit { p = prefice, n = numero }

D'autres expressions peuvent être modifiées dans le cadre du problème de la conformité à un SGD cible. Ces expressions sont des combinaisons de celles vues ci-dessus; il est évidemment impossible de les énumérer. Les règles de transformation syntaxique définies au paragraphe suivant permettent toutefois de transformer toute expression susceptible de devoir être transformée.

4.2. transformation d'un algorithme suite à une transformation de schéma

4.2.0. Introduction

Le but de ce paragraphe est de définir la façon dont sont transformées les formes syntaxiques exposées au paragraphe précédent. Ce paragraphe se présente donc comme une suite de transformations de schéma. Ces transformations sont mises en correspondance avec les règles définissant les transformations d'algorithmes qui en découlent.

Chaque transformation de schéma est une entité indécomposable qui à partir d'une structure de données permet d'en obtenir une autre. Certaines transformations sont cependant exposées de façon "morcelée". En d'autres termes, une transformation est divisée en étapes. Chaque étape travaille sur la structure de données obtenue de l'étape précédente (sauf la première qui travaille sur la structure initiale) et lui fait subir une transformation "simple". La structure finale est donc obtenue de la structure initiale en exécutant toutes les étapes constitutives de la transformation. Ces étapes sont cependant FICTIVES et ne servent qu'à clarifier l'exposé des transformations. Une transformation N'EST PAS une suite de transformations "plus simples", c'est un tout.

Les transformations de schéma sont exposées selon le formalisme graphique connu et sur base d'exemples informels.

Les règles de transformation d'algorithmes sont basées sur des descriptions d'algorithmes en BNF. La règle proprement dite est, quant à elle, en BNF et/ou en texte libre. La grammaire BNF est légèrement modifiée par rapport au chapitre 3 en ce sens qu'elle accepte des expressions optionnelles. Ces expressions apparaissent entre crochets ([]).

Chaque règle est illustrée par un exemple d'utilisation. Les exemples sont à vocation pédagogique et, dès lors, simplifiés à l'extrême. Ils sont un support de l'exposé et ne rendent aucunement compte de l'utilisation qui peut être faite du langage LDA.

Lors de l'analyse de ces transformations, certains problèmes sont apparus. D'aucuns ont pu être résolus, d'autres ont dû être abandonnés, et ont eu, dés lors, pour conséquence de devoir restreindre le champ de l'analyse. Ces restrictions ont été exposées dans différents chapitres (le premier pour les structures de données et les primitives permises par LDA/MAG, le troisième pour la syntaxe LDA). Elles sont analysées, commentées et justifiées dans des réflexions diverses lors de l'exposé des règles. Elles ne pouvaient l'être auparavant dans la mesure où ce n'est qu'à la lumière des transformations d'algorithmes qu'elles peuvent être justifiées.

On peut de plus remarquer que, comme pour les transformations de schéma, une règle de transformation syntaxique ne rend pas un algorithme conforme. Il aurait fallu pour cela définir des règles pour chaque forme qu'est susceptible de prendre un algorithme LDA; ce qui est évidemment impossible. C'est l'utilisation combinée des différentes règles de transformation syntaxique qui permet d'obtenir un algorithme conforme à partir d'un algorithme non conforme.

4.2.1 Transformations

1. APLATISSEMENT TOTAL

Cette transformation consiste à faire disparaître tous les composants non feuilles d'un item décomposable et à rattacher les feuilles directement au type d'articles. Elle a été retenue de façon à pouvoir éliminer un item décomposable non répétitif, identifiant ou pas, qui n'est pas autorisé par SQL.

Exemple

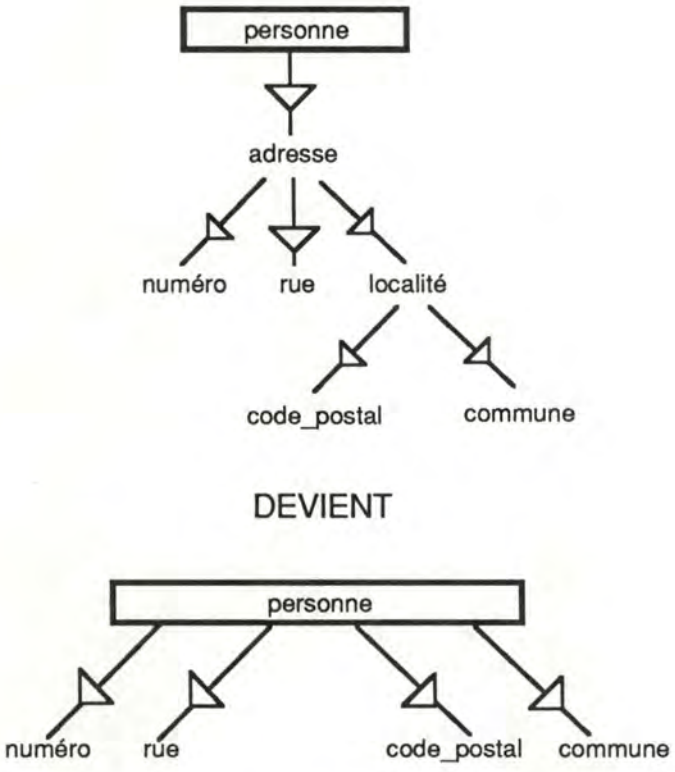


Fig. 4.1 : aplatissement total

Si "adresse" est clé d'accès, le groupe d'items "numéro", "rue", "code_postal", "commune" de la forme transformée est déclaré clé d'accès.

Règle 1

Dans le cadre d'un aplatissement total, (: <nom> <x>) (où la parenthèse fermante est celle qui ferme la condition portant sur l'item <nom>) devient <x> si <nom> est un nom d'item décomposable. Si <x> désigne plusieurs fils de l'item <nom>, les parenthèses l'entourant sont enlevées. Quand tout <nom> d'item décomposable a disparu, si l'expression finale porte sur plusieurs items, elle est entourée de parenthèses.

Exemple

Cet exemple est basé sur la figure 4.1.

```
create p := personne ( : adresse ( ( : numero = 5 ) and ( : rue = 'avenue louise' ) and
                       ( : localite ( ( : code_postal = 1050 )
                       and ( : commune = 'Bruxelles' ) ) ) ) );
```

DEVIENT

```
create p := personne ( ( : numero = 5 ) and ( : rue = 'avenue louise' )
                       and ( : code_postal = 1050 ) and ( : commune = 'Bruxelles' ) );
```

"(:adresse" et "(:localité" sont enlevés, ainsi que leurs parenthèses fermantes correspondantes. Les parenthèses entourant "code_postal" et "commune" sont enlevées. Celles entourant l'ensemble des fils de "adresse" le sont également. L'expression obtenue porte sur plusieurs items (4); elle est donc entourée de parenthèses. L'expression finale est ainsi obtenue.

Règle 2

(<varref>) . <comp> . <feuille>

où <varref> ::= le nom d'une variable de référence
<comp> ::= <item> | <item> . <comp>
<item> ::= le nom d'un composant décomposable de l'item que l'on transforme.
<feuille> ::= le nom d'un composant feuille de l'item décomposable que l'on transforme.

DEVIENT

(<varref>) . <feuille>

L'aplatissement total fait qu'un item initialement feuille d'une décomposition est directement rattaché au type d'articles. Il n'y a donc plus qu'un seul champ dans l'expression après la variable de référence.

Exemple

Cet exemple est basé sur le figure 4.1

```
COM := ( P ) . adresse . localite . commune ;
```

DEVIENT

```
COM := ( P ) . commune ;
```

Seul le champ (commune) à l'extrême droite de l'expression est laissé. C'est celui qui désigne l'item élémentaire.

Réflexions

La manipulation de données structurées

Comme cela a déjà été dit, les procédures et fonctions appelées dans un algorithme LDA sont écrites et compilées séparément. Leurs entêtes ne sont pas envisagés dans les transformations d'algorithmes. Il est dès lors exclu que ces entêtes soient sujets à des transformations syntaxiques suite à des transformations de structures de données.

Le cas présent, ceci amène les conclusions suivantes : aucune expression (<variable de référence>) . <items> ne se terminant pas par un item élémentaire, ne peut apparaître dans un appel de procédure ou de fonction. L'aplatissement de cet item entraînerait, en effet, une remise en cause de la déclaration de la procédure ou de la fonction concernée.

A titre d'exemple, "affiche_adresse((p).adresse)" devrait être transformé en "affiche_adresse ((p).numéro, (p).rue, (p).code_postal, (p).commune)" ce qui n'est pas permis. Par contre, "affiche_adresse ((p).adresse.numéro, (p).adresse.rue, (p).adresse. localité.code_postal, (p).adresse.localité.commune)" peut être transformé sans difficulté en "affiche_adresse ((p).numéro, (p).rue, (p).code_postal, (p).commune)" sans remettre en cause la déclaration de la procédure "affiche_adresse".

Si toutefois les transformations syntaxiques de déclarations de procédures ou de fonctions étaient envisagées, la restriction énoncée ci–dessus à propos des arguments d'un appel serait toujours en vigueur.

Ceci peut être justifié par le fait que le type des paramètres formels doit être déterminé dans l'entête des déclarations de procédures et fonctions. Avant tout développement, il est important de signaler que type a, le cas présent, le sens qui lui a été donné dans le langage LDA (chapitre 3 : signification liée au format de données). Le type n'est donc pas ici une classe regroupant des objets à propriétés communes (type d'articles, de chemins).

Le problème est donc de savoir quel est le type du paramètre formel correspondant à "(a).adresse" dans la déclaration de "affiche_adresse" (p. ex.). Ce problème est insoluble dans la mesure où l'item "adresse" N'A PAS DE TYPE. On peut seulement en dire qu'il est composé de trois items dont deux élémentaires de type connu ("numéro" et "rue") et un décomposable ("localité") composé de deux items élémentaires de type connu ("code_postal" et "commune").

Par contre, en travaillant au niveau des items élémentaires, la déclaration de "affiche_adresse" ne pose aucun problème dans la mesure où le format de tous les items élémentaires est connu.

De façon analogue, "(p).adresse" ne peut être assigné ou comparé à une variable de type "group". Les composants de l'item "adresse", quant à eux, peuvent l'être dans la mesure où leur type est clairement déclaré lors de la définition du schéma de la base de données.

On pourrait toutefois argumenter qu'une expression "if (p).adresse = (p2).adresse" (p.ex.) devrait être permise dans la mesure où même si "adresse" n'a pas de type, les parties gauche et droite de la comparaison répondent à la même définition. Cependant, dans un souci d'homogénéité, on exige en toute généralité que toute manipulation de valeurs d'items soit effectuée au niveau élémentaire.

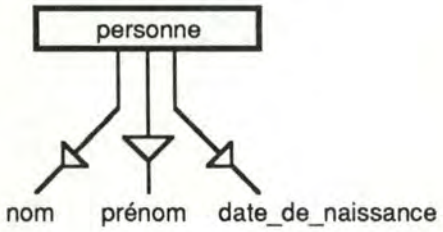
Les restrictions énoncées ci–dessus sont valables également pour les variables d'items.

Il serait néanmoins tout à fait injustifié d'appliquer la même restriction aux variables de type "group". Celles–ci font, en effet, l'objet d'une déclaration conventionnelle.

2. AJOUT D'UN NIVEAU DE DECOMPOSITION

Cette transformation consiste à ajouter un père commun à un groupe d'au moins deux items directement rattachés au type d'articles. Elle permet d'assurer, en Cobol, qu'un identifiant et une clé d'accès ne soient composés que d'un seul item.

Exemple



DEVIENT

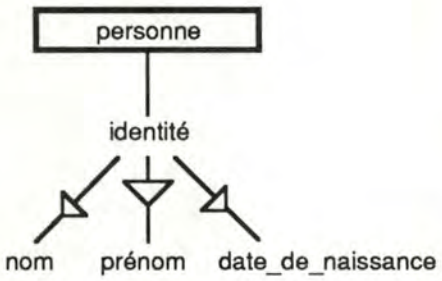


Fig. 4.2. : ajout d'un niveau de décomposition

L'exemple est analogue si le groupe "nom", prénom", "date_de_ naissance" forme une clé non identifiante. "identité" n'est dès lors plus identifiant.

Règle 1

Dans le cadre d'un ajout de niveau de décomposition, soit une expression <x> composée de conditions portant sur des items (un ou plusieurs et à qui on attribue un père commun) lors d'un accès à la base de données, d'une création ou d'une modification d'article. Si <x> porte sur plusieurs items, et qu'elle n'est pas déjà entourée de parenthèses, on l'entoure. Dans les deux cas, elle est transformée en (: <nom> <x>) où <nom> est le nom de l'item père créé.

Exemple

Cet exemple est basé sur la figure 4.2.

```
p := personne ( ( : nom = 'DURANT' ) and ( : prenom = 'Jacques' ) and
     ( : date_de_naissance = date_du_jour ) ) ;
```

DEVIENT

```
p := personne ( : identite ( ( : nom = 'DURANT' ) and ( : prenom = 'Jacques' ) and
```

(: date_de_naissance = date_du_jour))) ;

Les items pour lesquels on crée un père commun ("nom", prenom", "date_de_naissance") sont déjà entourés de parenthèses. Il suffit donc d'ajouter "(:identite" devant et ")" derrière.

Règle 2

(<varref>) . <items>

où <varref> ::= le nom d'une variable de référence
<items> ::= <nom> | <nom> . <items>
<nom> ::= le nom d'un (composant d')item.

DEVIENT

(<varref>) . <nouvel item>. <items>

où <nouvel item> ::= le nom de l'item père créé

Le nouvel item créé doit apparaître dans la décomposition. Il est toujours racine de la décomposition et son nom apparaît dès lors directement à droite de la variable de référence.

Exemple

Cet exemple est basé sur la figure 4.2.

write ((p) . nom) ;

DEVIENT

write ((p) . identite . nom) ;

L'item "nom" a acquis un père "identite" qui doit apparaître dans les champs de l'expression.

Réflexions

Justification de la transformation syntaxique.

Dans le cas où un identifiant ou une clé d'accès est composé de plusieurs items : fils1, ... , filsn, la question est de savoir si l'expression initiale "(:fils1 = x) and ... (:filsn = z)" n'aurait pas pu être transformée (lors de l'ajout d'un niveau de décomposition : père) en "(:père (: fils1 = x)) and ... (: père (: filsn = z))" plutôt qu'en "(:père((:fils1 = x) and... (:filsn = z)))".

La seconde forme a été préférée car elle est plus élégante, et en localisant en un seul endroit de l'expression l'item père créé, elle rend mieux compte de la notion d'item décomposable identifiant ou clé d'accès.

Les composants d'une clé ou d'un identifiant sont directement rattachés au type d'articles.

Le chapitre 1 a exigé que les composants (s'il y en a plusieurs) d'un identifiant ou d'une clé d'accès soient directement rattachés au type d'articles. Ceci s'explique par la définition de la transformation exposée ici qui ne permet de créer un père commun qu'à un groupe d'items directement rattachés au type d'articles.

3. ELIMINATION D'UN TYPE DE CHEMINS PAR DUPLICATION D'ITEM

Cette transformation consiste à éliminer un type de chemins en dupliquant un item identifiant simple, élémentaire et obligatoire d'un des deux membres (on dit aussi que l'on fait une rotation du type de chemins autour du type d'articles membre vers cet item). Elle est utilisée pour éliminer un type de chemins récursif ou non, de classe fonctionnelle 1–N, N–1 ou 1–1. Elle permet d'éliminer cette structure incompatible à SQL et à Cobol ainsi qu'à Codasyl si le type de chemins est récursif de classe fonctionnelle 1–N ou N–1. Dans le cas d'un type de chemins récursif 1–1, une autre transformation expliquée par la suite est appliquée pour assurer la conformité à Codasyl. La transformation présentée ici est également utilisée pour éliminer un type de chemins 1–N supportant une clé d'accès (non identifiante), ce qui n'est pas autorisé par SQL; pour Cobol, une autre transformation est utilisée.

Exemples

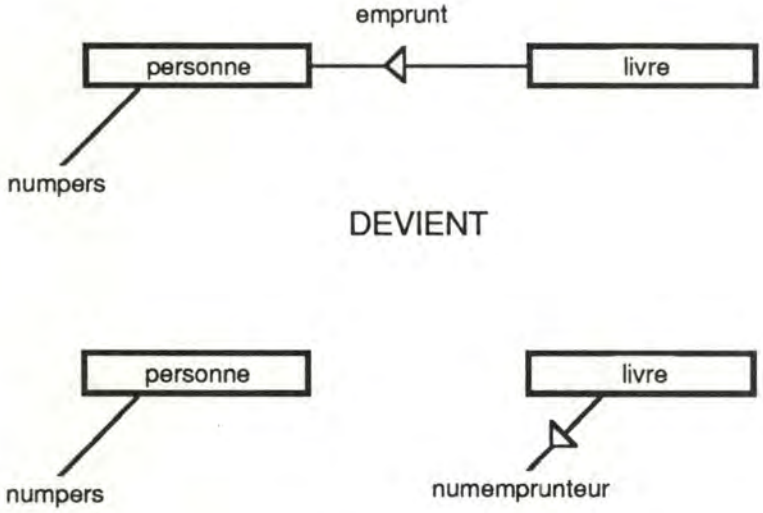


Fig 4.3 : élimination d'un type de chemins 1–N non récursif

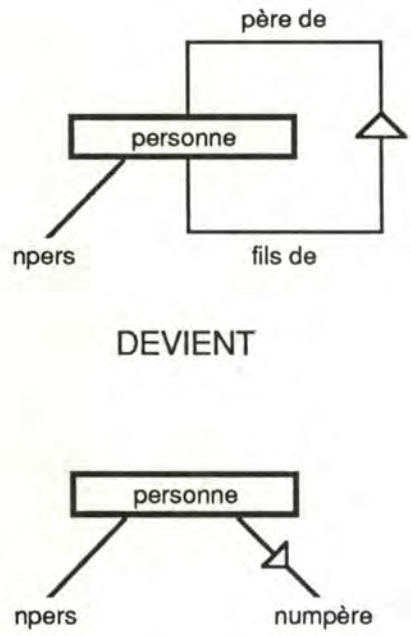


Fig 4.4 : élimination d'un type de chemins 1–N récursif

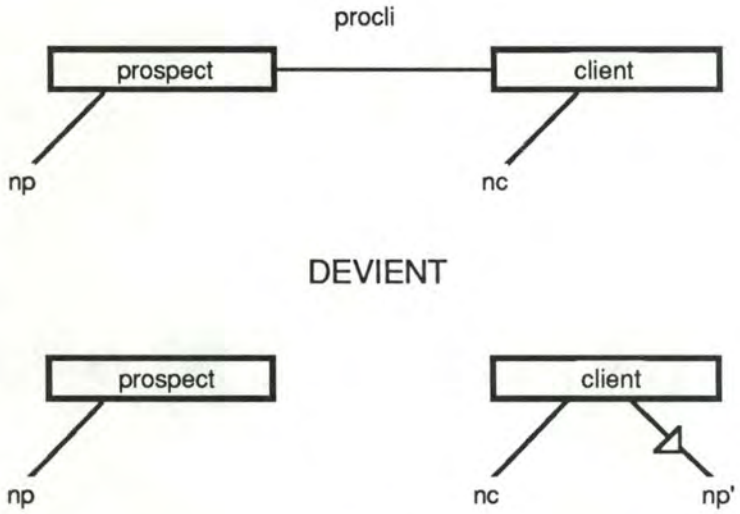


Fig 4.5 : élimination d'un type de chemins 1–1 non récursif

L'élimination, par duplication d'item, d'un type de chemins récursif de classe fonctionnelle 1–1 est analogue.

On peut s'étonner de ce que "np'" n'ait pas conservé le caractère identifiant de "np" dont il est le duplicata. Ceci est expliqué dans les réflexions esposées après les règles.

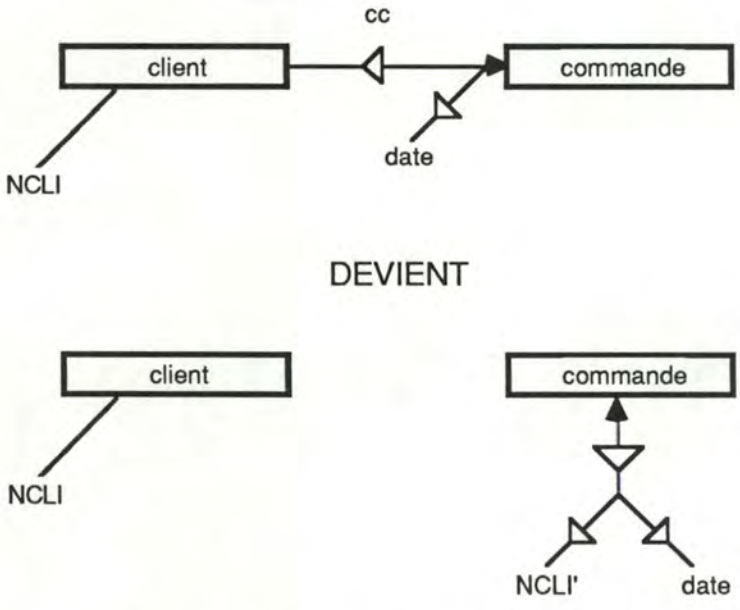


Fig 4.6 : élimination d'un type de chemins supportant une clé d'accès

L'exemple est analogue si le type de chemins est récursif.

On remarque que la duplication d'item se fait de telle sorte que l'item duplicata ne soit pas

répétitif. En d'autres termes, dans le premier exemple (fig. 4.3), on ne duplique pas un éventuel numéro de "livre" pour en faire un item répétitif de "personne".

Règle 1

[for] <varrefc> := <tart> (<lien> : <varrefo>)

où <varrefc> ::= le nom d'une variable de référence
<varrefo> ::= le nom d'une variable de référence
<tart> ::= le nom du type d'articles de <varrefc>

DEVIENT

[for] <varrefc> := <tart> (: <itemc> = (<varrefo>).<itemo>)

1.1. le type de chemins n'est pas récursif

<lien> ::= le nom du chemin entre <varrefc> et <varrefo>

SI la rotation s'est faite autour du type d'articles de <varrefo>
ALORS
<itemc> ::= l'item qui fait l'objet de la duplication
<itemo> ::= l'item duplicata
SINON c'est l'inverse
<itemc> ::= l'item duplicata
<itemo> ::= l'item qui fait l'objet de la duplication

1.2. le type de chemins est récursif

où <lien> ::= nom du rôle joué par <varrefo> dans le chemin le liant à <varrefc>

On distinguera les cas selon que la rotation s'est faite autour d'un rôle ou l'autre (on ne peut distinguer selon le type d'articles vu qu'il n'y en a qu'un).

Que le type de chemins soit récursif ou pas, un accès par chemin devient un accès par clé. La clé utilisée pour cet accès est l'item duplicata ou celui qui a fait l'objet de la duplication.

Exemples

Ces exemples sont basés sur la figure 4.3.

- for l := livre (emprunt : pers)

DEVIENT

for l := livre (: numemprunteur = (pers) . numpers)

- pers := personne (emprunt : l) ;

DEVIENT

pers := personne (: numpers = (l) . numemprunteur) ;

Si on accède à l'article de type autour duquel on a fait la rotation , la clé est l'item duplicata; c'est ici le cas pour l'accès à "livre" sur base de "numemprunteur". Sinon la clé est l'item dupliqué; c'est ici le cas pour l'accès à "personne" sur base de "numpers".

Les exemples avec un type de chemins récursif sont analogues.

Règle 2

[for] <varrefc> := <tart> ((<lien> : <varrefo>) and <accès par clé>)

C'est le cas d'un accès par clé dans un chemin.

où <varrefc> ::= le nom d'une variable de référence
<varrefo> ::= le nom d'une variable de référence
<tart> ::= le nom du type d'articles de <varrefc>
<lien> ::= Si le type de chemins à éliminer et supportant la clé est récursif, <lien> est le nom du rôle que joue <varrefo> dans le chemin le liant à <varrefc>. Sinon <lien> est le nom du chemin liant <varrefo> à <varrefc>.
<accès par clé> ::= Une condition posée sur une clé dans un chemin, telle que définie dans la syntaxe.

DEVIENT

(<lien> : <varrefo>) est transformé en appliquant 1.1 ou 1.2 (voir règle n°1) selon que le type de chemins est récursif ou pas.

Dans le cadre de l'élimination par rotation d'un type de chemins supportant une clé d'accès, ce qui est initialement un accès par clé dans un chemin devient un accès par clé. Cette clé est formée de la clé initiale et de l'item duplicata qui a servi lors de la rotation.

Exemple

Cet exemple est basé sur la figure 4.6.

com := commande ((cc : cli) and (: date = date_du_jour)) ;

DEVIENT

com := commande ((: NCLI' = (cli) . NCLI) and (: date = date_du_jour)) ;

L'accès par clé dans un chemin devient un accès par clé, et la clé est composée des items "NCLI'" et "date".

Les exemples avec un type de chemins récursif sont analogues.

Règle 3

modify <varrefc> (<lien> : <varrefo>)

où <varrefc> ::= le nom d'une variable de référence
<varrefo> ::= le nom d'une variable de référence
<lien> ::= Si le type de chemins à éliminer est récursif, <lien> est le nom du rôle que joue <varrefo> dans le chemin le liant à <varrefc>.

Sinon <lien> est le nom du chemin liant <varrefo> à <varrefc>.

DEVIENT

modify <varrefda> (: <itemda> = (<varrefdé>) . <itemdé>)

où <varrefda> ::= nom de la variable de référence, parmi <varrefc> et <varrefo>, désignant l'article associé à la valeur d'item duplicata.
<itemda> ::= nom de l'item duplicata
<varrefdé> ::= nom de la variable de référence, parmi <varrefc> et <varrefo>, désignant l'article associé à la valeur d'item dupliqué.
<itemdé> ::= nom de l'item dupliqué

Quand un type de chemins est éliminé, l'association entre deux articles par un chemin de ce type est représentée par une valeur de l'item duplicata. Associer deux articles revient donc à associer un article à une valeur d'item duplicata.

Exemple

Cet exemple est basé sur la figure 4.3.

modify l (emprunt : pers) ;

DEVIENT

modify l (: numemprunteur = (l) . numpers) ;

On veut rendre compte du fait que le livre "l" est emprunté par la personne "pers". On associe donc à ce livre une valeur d'item "numemprunteur" égale au numéro de la personne ("pers") qui l'a emprunté.

Règle 4

create <varrefc> := <tart> (<lien> : <varrefo>)

où <varrefc> ::= le nom d'une variable de référence
<varrefo> ::= le nom d'une variable de référence
<tart> ::= le nom du type d'articles de <varrefc>
<lien> ::= Si le type de chemins à éliminer est récursif, <lien> est le nom du rôle que joue <varrefo> dans le chemin le liant à <varrefc>. Sinon <lien> est le nom du chemin liant <varrefo> à <varrefc>.

DEVIENT

create <varrefc> := <tart> ( ) ;
modify <varrefda> (: <itemda> = (<varrefdé>) . <itemdé>) ;

où <varrefda> ::= nom de la variable de référence, parmi <varrefc> et <varrefo>, désignant l'article associé à la valeur d'item duplicata.
<itemda> ::= nom de l'item duplicata
<varrefdé> ::= nom de la variable de référence, parmi <varrefc> et <varrefo>, désignant l'article associé à la valeur d'item dupliqué.
<itemdé> ::= nom de l'item dupliqué

La création d'un article qu'on associe à un autre se scinde après élimination du type de chemins. La première opération se ramène à la création, la seconde à l'association avec l'autre article. Cette association se fait de manière tout à fait semblable à celle exposée lors de la règle précédente (3).

Exemple

Cet exemple est basé sur la figure 4.4.

```
create p1 := personne (fils_de : p2 ) ;
```

DEVIENT

```
create p1 := personne ( ) ;
modify p2 ( : numpere = ( p1 ) . npers ) ;
```

La personne "p2" devient fils de la personne "p1". On lui associe donc une valeur d'item "numpere" (le numéro de son père) égale au numéro de personne "p1".

règle 5

modify <varrefc> (<lien> : 0 <varrefo>)

où <varrefc> ::= le nom d'une variable de référence
<varrefo> ::= le nom d'une variable de référence
<lien> ::= Si le type de chemins à éliminer est récursif, <lien> est le nom du rôle que joue <varrefo> dans le chemin le liant à <varrefc>. Sinon <lien> est le nom du chemin liant <varrefo> à <varrefc>.

DEVIENT

modify <varref> (:<item> = NULL)

où <varref> ::= nom de la variable de référence, parmi <varrefc> et <varrefo>, désignant l'article associé à la valeur d'item duplicata
<item> ::= nom de l'item duplicata

L'association entre deux articles est représentée par une valeur d'item (duplicata). L'absence d'association doit donc être représentée par l'absence de valeur d'item. Cependant LDA/MAG n'accepte pas les items facultatifs. Dès lors, comme cela a déjà été expliqué, c'est NULL qui fait office de valeur absente. Plus concrètement, on met la valeur d'item duplicata à NULL pour indiquer que l'article qui y est associé n'est relié à aucun article via cette valeur.

Exemple

Cet exemple est basé sur la figure 4.4.

```
modify pers1 ( père_de : 0 pers2 ) ;
```

DEVIENT

```
modify pers1 ( : numpere = NULL ) ;
```

La personne "pers2" n'est plus père de la personne "pers1". On met donc le numéro de père ("numpere") de cette dernière à NULL.

Règle 6

modify <varref> (: <item> = <valeur>) ;

où <varref> ::= le nom d'une variable de référence
<item> ::= le nom d'un item simple élémentaire et identifiant faisant l'objet d'une duplication pour l'élimination du type de chemins
<valeur> ::= toute valeur élémentaire pouvant apparaître dans un programme LDA.

DEVIENT

```
for  <varrefc> := <tart> ( : <itemc> = ( <varref> ) . <item> )
     modify <varrefc> ( : <itemc> = <valeur> )
endfor ;
modify <varref> ( :<item> = <valeur> ) ;
```

où <varrefc> ::= nom de la variable de référence, parmi <varrefc> et <varrefo>, désignant l'article associé à la valeur d'item duplicata
<tart> ::= le nom du type d'articles auquel appartient l'item duplicata
<itemc> ::= le nom de l'item duplicata

Cette transformation syntaxique est nécessaire afin d'assurer le respect de la contrainte référentielle apparue suite à la transformation de schéma de données : l'ensemble des valeurs d'item duplicata est inclu dans celui des valeurs d'item dupliqué. Quand on modifie une valeur d'item dupliqué, il faut modifier en conséquence les valeurs d'item duplicata (il peut ne pas y en avoir). Il est nécessaire, pour cela, d'accéder aux articles associés à ces valeurs.

Exemple

Cet exemple est basé sur la figure 4.3.

```
modify pers1 ( : numpers = 250 ) ;
```

DEVIENT

```
for  l := livre ( : numemprunteur = ( pers1 ) . numpers )
     modify l ( : numemprunteur = 250 )
endfor;
modify pers1 ( : numpers = 250 ) ;
```

La personne "pers1" se voit attribuer un nouveau numéro. Tous les livres qu'elle a empruntés doivent donc être associés à ce nouveau numéro. On accède donc à ces livres, et on les associe chacun à une nouvelle valeur d'item "numemprunteur" (250).

Règle 7

delete <varref> ;

où <varref> ::= le nom d'une variable de référence d'un type d'articles membre du type de chemins éliminé par duplication d'item.

DEVIENT

a) SI
- le type de chemins éliminé est de classe fonctionnelle 1–N ou 1–1
- <varref> est une variable de référence du type d'articles origine de ce type de chemins.
- il existe une contrainte d'existence sur les cibles de ce type de chemins

ALORS
Les cibles du chemin dont l'article (désigné par) <varref> est origine doivent être détruites également dans la mesure où leur présence dans la base de données viole la contrainte d'existence.

Cette destruction est gérée par LDA/MAG tant que le type de chemins existe. Dès qu'il est éliminé (il l'est le cas présent) la destruction doit être faite explicitement dans le programme d'application.

La forme syntaxique devient donc :

```
for  <varref1> := <tart> ( : <item1> = ( <varref> ) . <item2> )
     delete <varref1>
endfor ;
delete <varref> ;
```

où <varref1> ::= le nom d'une variable de référence du type d'articles cible obligatoire du type de chemins éliminé.
<tart> ::= le nom du type d'articles de <varref1>, c'est–à–dire, le type d'articles cible obligatoire
si la rotation s'est faite autour de <tart>
<item1> ::= le nom de l'item duplicata
<item2> ::= le nom de l'item dupliqué
sinon
<item1> ::= le nom de l'item dupliqué
<item2> ::= le nom de l'item duplicata

L'article à détruire est associé aux cibles (à détruire également) par une valeur d'item. On accède donc à ces cibles sur base de cette valeur et on les détruit. La destruction initiale peut ensuite être effectuée.

Exemple

Cet exemple est basé sur la figure 4.6.

Si on n'admet de commandes que celles passées par un client, alors quand un client est détruit, il faut également enlever de la base de données toutes les commandes qu'il a passées. Dès lors :

```
delete cli ;
```

DEVIENT

```
for  com := commande ( : NCLI' = ( cli ) . NCLI )
     delete com
endfor ;
delete cli ;
```

On accède par l'item duplicata (NCLI') aux commandes passées par le client "cli" et on les détruit. On détruit ensuite le client.

b) SI

on reprend les mêmes conditions qu'au point "a" mais le type de chemins éliminé est N–1.

ALORS

Le problème est plus délicat dans la mesure où la cible obligatoire du chemin dont l'origine est détruite, n'est détruite que si aucun autre chemin du même type que celui concerné n'y aboutit. Si le type de chemins est éliminé, la gestion de cette destruction par programme d'application s'avère relativement mal aisée; elle sera abordée au point 4.6 qui expose les problèmes de gestion de cohérence de la base de données. Aucune transformation n'est donc exposée ici.

c) Quelle que soit la classe fonctionnelle du type de chemins éliminé (1–N, N–1, 1–1), si les cibles ne sont pas obligatoires (le type de chemins à éliminer peut dans ce cas être récursif) :

c1) delete <varref> ;

Si la rotation s'est faite autour du type d'articles dont <varref> est une variable de référence, dans le cas d'un type de chemins éliminé non récursif. Il n'y a donc pas de transformation de l'instruction.

c2) Sinon : la rotation s'est faite autour de l'autre type d'articles, ou le type de chemins est récursif

```
for  <varref1> := <tart> ( : <itemc> = ( <varref> ) . <itemo> )
     modify <varref1> ( : <itemc> = NULL )
endfor ;
delete <varref> ;
```

où <varref1> ::= le nom de la variable de référence désignant les cibles non obligatoires
<tart> ::= le nom du type d'articles de <varref1>
<itemc> ::= le nom de l'item duplicata
<itemo> ::= le nom de l'item dupliqué

Les articles qui étaient initialement reliés par un chemin, le sont, après rotation, par des valeurs d'items et leurs copies (items duplicatas). Les articles cibles doivent être chacun associés à une valeur d'item duplicata égale à NULL, cela afin d'indiquer qu'ils ne sont plus reliés à l'origine, ce qui est logique dans la mesure où cette origine a été détruite.

Exemple

Cet exemple est basé sur la figure 4.6.

Si on admet que des commandes peuvent exister sans être reliées à un client, la destruction d'un client est transformée de la façon suivante :

```
delete cli ;
```

DEVIENT

```
for  com := commande ( : NCLI' = ( cli ) . NCLI )
     modify com ( : NCLI' = NULL )
```

endfor;
delete cli ;

La valeur d'item "NCLI'" d'une commande est le numéro du client qui a passé cette commande. Si des commandes ne sont plus reliées à aucun client, leur valeur d'item "NCLI'" est NULL. On peut donc voir NULL, le cas présent, comme "le numéro d'aucun client".

Réflexions

Opérateur de comparaison permis dans un accès par clé dans un chemin.

Comme cela a déjà été dit et justifié lors du premier chapitre, dans le cas d'un accès par clé (dans un chemin ou pas) sur base d'un groupe d'items ou d'une clé décomposable, le seul opérateur permis est l'égalité (=).

Il a toutefois été exigé également que cet opérateur soit le seul utilisé dans un accès par clé dans un chemin même si cette clé se résume à un item élémentaire. L'objet de ce point est de justifier cette restriction.

On voit à la règle 2 que l'accès par clé dans un chemin est transformé en accès par clé. La clé déclarée après élimination du type de chemins est formée de la clé initiale et de l'item duplicata. Cette clé est donc composée d'au moins deux items; le seul opérateur permis est donc l'égalité (=).

Elimination d'un type de chemins par duplication d'item

Si on se réfère à la figure 4.5., on remarque que "np'" n'est pas identifiant. Lui laisser son caractère identifiant entraînerait des difficultés. En effet, l'instruction "modify prosp1 (procli : 0 client1)" (p. ex.) (où "prosp1" et "client1" sont des variables de référence des types d'articles "prospect" et "client") devient, après élimination du type de chemins "procli" : "modify client1 (: np' = NULL)". Or si "np'" est identifiant de "client", un seul article "client" peut être associé à une valeur d'item "np'" égale à NULL. Or, une valeur d'item duplicata égale à NULL représente une absence d'association (par chemin). Dès lors, un seul article "client" pourrait ne pas être relié à un article "prospect", ce qui revient presque à déclarer "procli" obligatoire pour "client". La contrainte serait donc du genre : " tout type de chemins 1–1 est obligatoire pour le type d'articles autour duquel se fait la rotation. Un seul article de ce type peut déroger à cette obligation", ce qui n'a pas de sens.

"np'" ne peut donc rester identifiant.

Condition de sélection non identifiante dans une assignation.

L'assignation peut être utilisée pour faire désigner un article par une variable de référence. Par exemple : "client1 := client (procli : prosp1)". Assez paradoxalement, il a été dit au troisième chapitre que la condition portant sur le type d'articles n'a pas à être identifiante.

L'utilité d'une condition non identifiante apparaît cependant pleinement lorsqu'on envisage la transformation exposée à la figure 4.5. Les types d'articles "prospect" et "client" y sont reliés par le type de chemins 1–1 "procli".

Une fois ce type de chemins éliminé, l'assignation exposée ci–dessus devient : "client1 := client(:np' = (prosp1).np)" où la condition n'est pas identifiante si on considère "np'" (cfr. réflexion précédente). Elle l'est cependant de facto dans la mesure où le type de chemins

"procli" est de classe fonctionnelle 1–1.

De plus, on peut signaler qu'une condition non identifiante peut encore être utile dans d'autres cas. En effet, si le SGD cible est Codasyl, le type de chemins "procli" n' a pas à être éliminé. On doit cependant transformer sa classe fonctionnelle en la définissant 1–N de "prospect" vers "client" (p. ex.). Dès lors, l'assignation initiale "client1 := client (procli : prosp1)" n'a pas à être modifiée, et la condition a perdu son caractère identifiant.

Par généralisation, on autorise une assignation sur base d'un accès non identifiant par clé dans un chemin.

L'assignation d'une variable de référence sur base d'une condition de sélection non identifiante répond à une question du type : "y–a–t–il au moins un article vérifiant cette condition ?".

4. INSERTION D'UN TYPE D'ARTICLES

Cette transformation consiste à insérer un nouveau type d'articles dans un type d'associations entre deux types d'objets de la base de données.

Plus concrètement, un type d'articles est inséré dans un type de chemins entre deux types d'articles ou dans la relation qui associe un type d'articles à un item.

On utilise cette transformation pour éliminer des structures incompatibles en Codasyl, supportant des clés d'accès éventuelles, telles que les types de chemins N-N récursifs ou pas, les types de chemins 1-1 récursifs et les items répétitifs clés d'accès.

Exemples

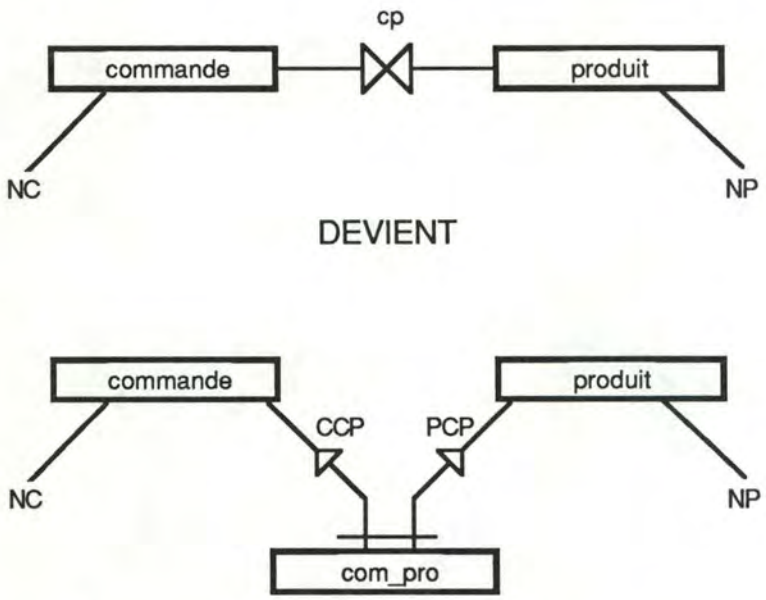
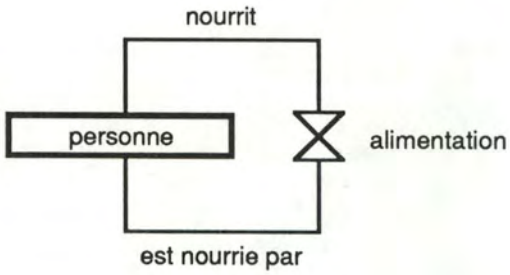


fig. 4.7 : élimination d'un type de chemins N-N non récursif



DEVIENT

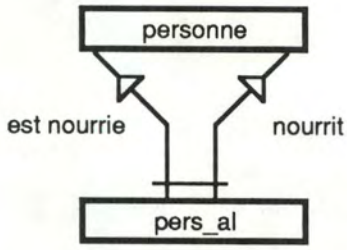
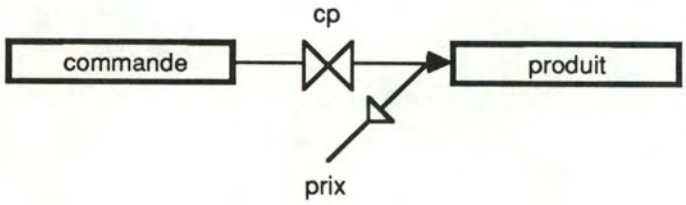


fig. 4.8 : élimination d'un type de chemins N-N récursif



DEVIENT

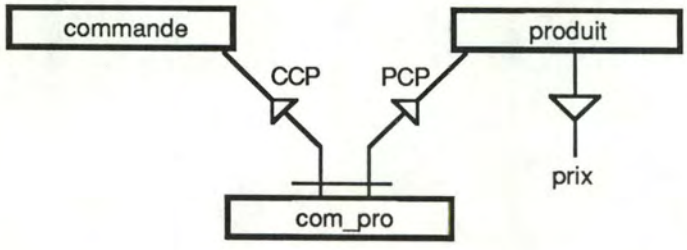
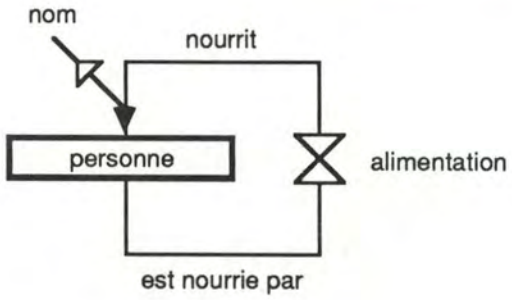


fig. 4.9 : élimination d'un type de chemins N-N non récursif supportant une clé



DEVIENT

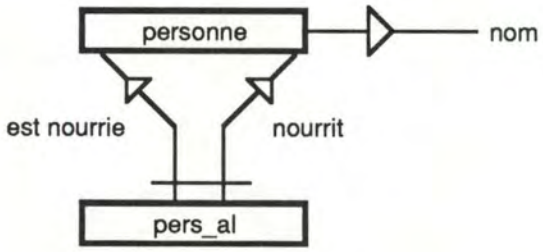
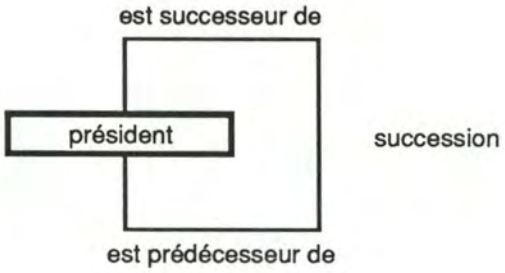


fig. 4.10 : élimination d'un type de chemins N-N récursif supportant une clé



DEVIENT

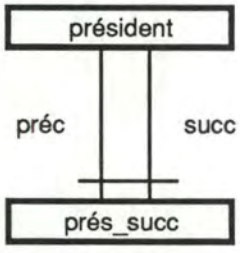
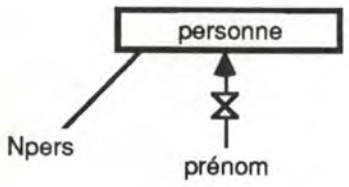


fig. 4.11 : élimination d'un type de chemins récursif 1-1



DEVIENT

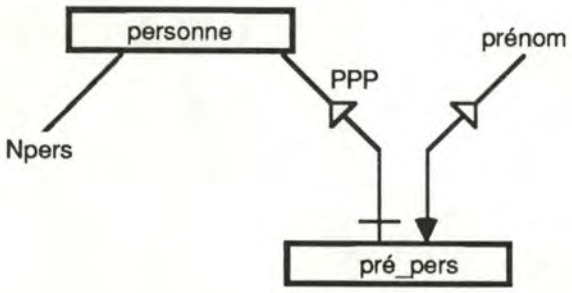


fig. 4.12 : élimination d'une clé répétitive

Remarque : la transformation serait semblable si "prenom" était décomposable.

Règle 1

[for] <varrefc> := <tart> <condition>

où <varrefc> ::= le nom d'une variable de référence.
<tart> ::= le nom du type d'articles de cette variable.
<condition> ::= SOIT (<chemin> : <varrefo>)
où - <chemin> est le nom du chemin reliant <varrefo> à <varrefc>.
- <varrefo> est le nom d'une variable de référence.
s'il s'agit d'une condition sur chemin non récursif.

SOIT (<rôle> : <varrefo>)
où - <rôle> est le nom du rôle que joue <varrefo> dans le chemin le liant à <varrefc>.
- <varrefo> est le nom d'une variable de référence.
s'il s'agit d'une condition sur chemin récursif.

SOIT <c>, une condition portant sur un item s'il s'agit d'un accès par clé répétitive.

DEVIENT

for <varrefint> := <tartint> <condition transformée>
<varrefc> := <tart> (<chemin1> : <varrefint>) ;
...
endfor ;

où <varrefint> ::= le nom d'une variable de référence du type d'articles inséré.
<tartint> ::= le nom du type d'articles inséré.
<condition transformée> ::= SOIT (<chemin2> : <varrefo>)
où <chemin2> est le nom du chemin reliant <varrefo> à

<varrefint> si <condition> portait sur un chemin qu'il soit récursif ou non. Si le type de chemins éliminé était récursif,<chemin2> est le nom du chemin qui représente le rôle que jouait <varrefo> dans la forme initiale (le cas présent, c'est <rôle>).

SOIT <c> dans le cas d'un accès par clé répétitive c'est-à-dire si <condition> portait sur un item.

<chemin1> ::= le nom du chemin reliant <varrefint> à <varrefc>. Si le type de chemins éliminé était récursif, <chemin1> est le nom du chemin représentant le rôle joué par <varrefc> dans la forme initiale.

Dans le cas du "for", tout "next <varrefc>" (ou exit <varrefc>) apparaissant dans le corps de la boucle est transformé en "next <varrefint>" (ou exit <varrefint>).

Dans le cas d'une assignation, l'utilisateur veut obtenir au plus un article. Dès lors, il suffit d'ajouter "exit <varrefint>" juste après l'assignation de <varrefc>. Le corps de la boucle est ainsi exécuté au plus une fois.

Dans le cas d'un accès par chemin ou par clé, on accède d'abord à l'article intermédiaire à partir de l'origine initiale (par clé ou par chemin). Cet article sera l'origine du second accès vers la cible initiale.

Dans le cas du "for", les instructions "next" et"exit" qui apparaissent dans le corps de la boucle, et spécifient la variable de parcours initiale, doivent spécifier, après transformation, la nouvelle variable de parcours (celle des articles intermédiaires).

Exemple

Cet exemple est basé sur la figure 4.7.

1. for p := produit (cp : com)

DEVIENT

```
for cde_prod := com_pro ( CCP : com )
    p := produit ( PCP : cde_prod );
```

L'accès de la commande "com" vers les produits qui lui sont associés se ramène à un accès aux articles intermédiaires "com_pro" à partir de "com"; et de ces articles obtenus, l'accès peut se faire vers les produits.

L'accès par un chemin récursif est similaire.

2. Cet exemple est basé sur la figure 4.12.

```
for pers := personne (: prenom = 'Jean' )
```

DEVIENT

```
for pr_pe := pre_pers (: prenom = 'Jean' )
    pers := personne ( PPP : pr_pe );
```

L'explication est analogue à celle donnée ci-dessus pour l'exemple 1.

Règle 2

create <varrefc> := <tart> <condition>;

où <varrefc> ::= le nom d'une variable de référence.
<tart> ::= le nom du type d'articles de cette variable.
<condition> ::=

A) une condition sur chemin (<chemin> : <origine>)
où - <origine> est le nom d'une variable de référence.
- <chemin> est le nom du chemin reliant <origine> à <varrefc> s'il s'agit d'un chemin non récursif, ou le nom du rôle que joue <origine> dans le chemin le liant à <varrefc> s'il s'agit d'un chemin récursif.

B) une condition sur item répétitif
(:<item> = {<liste_expr>})
où <item> est l'item en question.

1 <liste_expr> ::= <valeur> | <valeur>,<liste_expr>
s'il s'agit d'un item élémentaire

2 <liste_expr> ::= {<élément>} |
{<élément>}, <liste_expr>

où <élément> ::= 0 à N <assignation> selon le nombre de composants feuilles auxquels on donne une valeur.

<assignation> ::= <itemfeuille> = <valeur>
s'il s'agit d'un item décomposable.

<itemfeuille> est le nom d'un composant feuille de l'item répétitif décomposable.

<valeur> est toute valeur élémentaire pouvant apparaître dans un algorithme LDA.

DEVIENT

create <varrefc> := <tart> () ;

suivi de

A) si la condition de création portait sur un chemin

create <varrefint> := <tartint> ((<chemin1> :<varrefc>) and(<chemin2> :<origine>));

où
<varrefint> ::= le nom d'une variable de référence du type d'articles inséré dans le type de chemins.
<tartint> ::= le nom du type d'articles inséré.

<chemin1> ::= le nom du chemin reliant <varrefc> à <varrefint>. Si le type de chemins éliminé était récursif, ce chemin représente le rôle joué par <varrefc> dans la forme initiale.

<chemin2> ::= le nom du chemin reliant <origine> à <varrefint>. Si le type de chemins éliminé était récursif, ce chemin représente le rôle joué par <origine> dans la forme initiale.

La création d'un chemin entre un article créé et un article existant déjà, devient la création d'un article intermédiaire et son rattachement à l'article créé initialement et à celui existant.

B1) si la condition de création porte sur un item répétitif élémentaire

create <varrefint> := <tartint> ((<chemin1> : <varrefc>) and (:<item> = <valeur>));
cette instruction est à répéter autant de fois qu'il y a de <valeur> dans <liste_expr>.
Si <item> n'apparaît pas dans la condition de création, cette instruction apparaît quand même une fois pour créer un article intermédiaire associé à une valeur d'item <item> égale à NULL. La valeur par défaut d'<item> est en effet une seule fois NULL.

B2) si la condition de création porte sur un item répétitif décomposable

create <varrefint> := <tartint> ((<chemin1> : <varrefc>) and (:<item>
(<élément transformé>)))

où <élément transformé> rend compte de la décomposition de <item>. Les feuilles qui reçoivent une valeur ont la forme suivante (: <itemfeuille> = <valeur>).

Cette instruction est à répéter autant de fois qu'il y a de <élément> dans <liste_expr>.
Si <élément> dans <liste_expr> est vide, l'instruction create apparaît une seule fois et ne contient pas "and (: <item>(<élément transformé>))" ni les première et dernière parenthèses.
Si <item> n'apparaît pas dans la condition de création, cette instruction apparaît quand même une fois pour créer un article intermédiaire associé à une valeur décomposable d'item <item> égale à NULL.

Si un item répétitif est éliminé par insertion d'un type d'articles intermédiaire, chaque valeur associée initialement à cet item correspond à un article intermédiaire. Il faut donc créer autant d'articles que de valeurs d'item associées initialement à l'article. Il faut de plus attacher tous ces articles à l'article auquel étaient associées initialement les valeurs d'items.

Exemple

Cet exemple est basé sur la figure 4.12.

```
create pers := personne (: prenom = {'René', 'Alexandre', 'Arthur'} );
```

DEVIENT

```
create pers := personne ();
create pr_pr := pre_pers ( ( PPP : pers ) and (: prenom = 'René' ) );
create pr_pr := pre_pers ( ( PPP : pers ) and (: prenom = 'Alexandre' ) );
create pr_pr := pre_pers ( ( PPP : pers ) and (: prenom = 'Arthur' ) );
```

On crée un article initial (pers) ainsi que 3 articles intermédiaires (pr_pr) qu'on lui

associe. Chaque article intermédiaire représente un prénom de "pers".

Règle 3

modify <varrefc> (<chemin> : <varrefo>) ;

où <varrefc> ::= le nom d'une variable de référence.
<varrefo> ::= le nom d'une variable de référence.
<chemin> ::= le nom du chemin reliant <varrefo> à <varrefc> si le chemin est non récursif; le nom du rôle que joue <varrefo> dans le chemin le liant à <varrefc> s'il s'agit d'un chemin récursif.

DEVIENT

create <varref-int> := <tart-int>((<chemin1> : <varrefc>) and (<chemin2> : <varrefo>));

où <varref-int> ::= le nom d'une variable de référence du type d'articles inséré.
<tart-int> ::= le nom du type d'articles inséré.
<chemin1> ::= le nom du chemin liant <varrefc> à <varref-int>. Si le type de chemins éliminé était récursif, ce chemin représente le rôle que jouait <varrefc> dans la forme initiale.
<chemin2> ::= le nom du chemin liant <varrefo> à <varref-int>. Si le type de chemins éliminé était récursif, ce chemin représente le rôle que jouait <varrefo> dans la forme initiale.

Attacher deux articles par un chemin revient, le cas présent, à créer l'article intermédiaire et à le rattacher à ces deux articles.

Exemple

Cet exemple est basé sur la figure 4.7.

modify com (cp : pro);

DEVIENT

create cde_prod := com_pro ((CCP : com) and (PCP : pro));

Pour réaliser l'association entre l'article "com" et l'article "pro", il faut créer "cde_prod" et le rattacher à "com" et "pro" via les chemins "CCP" et "PCP".

Règle 4

modify <varrefc1> (<chemin> : 0 <varrefo>);

où <varrefc1> ::= le nom d'une variable de référence.
<varrefo> ::= le nom d'une variable de référence.
<chemin> ::= le nom du chemin liant <varrefo> à <varrefc1> ou le nom du rôle que joue <varrefo> dans le chemin récursif N-N le liant à <varrefc1>.

DEVIENT

```
for <varref-int> := <tart-int> (<chemin1> : <varrefo>)
    <varrefc2> := <tartc> (<chemin2> : <varref-int>);
    if <varrefc2> = <varrefc1> then exit <varref-int>
    endif
endfor;
delete <varref-int>;
```

où <varref-int> ::= le nom d'une variable de référence du type d'articles inséré.
<tart-int> ::= le nom du type d'articles inséré.
<chemin1> ::= le nom du chemin liant <varrefo> à <varref-int>. Dans le cas d'un type de chemins récursif, c'est le chemin qui représente le rôle de <varrefo> dans la forme initiale.
<varrefc2> ::= le nom d'une variable de référence de même type que <varrefc1>.
<tartc> ::= le nom du type d'articles dont <varrefc1> est une référence.
<chemin2> ::= le nom du chemin liant <varref-int> à <varrefc2>. Dans le cas d'un type de chemins récursif, c'est le chemin qui représente le rôle de <varrefc1> dans la forme initiale.

Détacher deux articles revient à rechercher l'article intermédiaire les reliant, et à le détruire. Pour ce faire, on recherche parmi les articles intermédiaires reliés à un des articles, celui qui est relié à l'autre des deux articles.

Exemple

Cet exemple est basé sur la figure 4.7.

```
modify com ( cp : 0 pro );
```

DEVIENT

```
for cde_prod := com_pro ( PCP : pro )
    c := commande ( CCP : cde_prod );
    if c = com then exit cde_prod
    endif
endfor;
delete cde_prod;
```

Pour détruire un chemin quand un article intermédiaire (ici, il est désigné par "cde_prod") y est inséré, on recherche cet article et on le détruit.

Règle 5

```
modify <varrefc> (<rôle> : 0 <varrefo>);
```

où <varrefc> ::= le nom d'une variable de référence.
<varrefo> ::= le nom d'une variable de référence.
<rôle> ::= le nom du rôle que joue <varrefo> dans le chemin récursif 1-1 le liant à <varrefc>.

DEVIENT

<varref-int> := <tart-int> (<chemin> : <varrefo>);
delete <varref-int>;

où <varref-int> ::= le nom d'une variable de référence du type d'articles inséré.
<tart-int> ::= le nom du type d'articles inséré.
<chemin> ::= le nom du chemin liant <varrefo> à <varref-int>. Ce chemin représente le rôle (<rôle>) joué par <varrefo> dans la forme initiale.

Détacher deux articles revient à rechercher l'article intermédiaire les reliant, et à le détruire. La recherche est simplifiée par rapport à la règle précédente dans la mesure où un article initial est relié à au plus un article intermédiaire.

Exemple

Cet exemple est basé sur la figure 4.11.

modify pres1 (est_successeur_de : 0 pres2);

DEVIENT

ps := pres_succ (succ : pres2);
delete ps;

On veut que "pres2" ne soit plus successeur de "pres1".

On accède donc à l'article relié par "succ" à "pres2" : "ps" ("ps" est lui-même relié à "pres1" prédécesseur de "pres2"). On détruit ensuite "ps" supprimant ainsi l'association.

Règle 6

modify <varref> <cond-mod>;

où
<varref> ::= le nom d'une variable de référence

a) <cond-mod> ::= (:<item> <opmod> <expr>)

où <item> ::= le nom d'un item répétitif
<expr> ::= {<valeur>} | {<expr-déc>}
<expr-déc> ::= <itemfeuille> = <valeur> |
<itemfeuille> = <valeur>,<expr-déc>|
<vide>
<valeur> ::= toute valeur élémentaire pouvant apparaître dans un algorithme LDA.
<itemfeuille> ::= le nom d'un composant feuille de l'item répétitif décomposable.

a1) <opmod> ::= +
a2) <opmod> ::= -

b) <cond-mod> ::= (:<item> = {<liste-expr>})

où <item> ::= le nom d'un item répétitif.

1 <liste_expr> ::= <valeur> | <valeur>,<liste_expr>
s'il s'agit d'un item élémentaire

2 <liste_expr> ::= {<élément>} | {<élément>}, <liste_expr>

où <élément> ::= 0 à N <assignation> selon le nombre de composants feuilles auxquels on donne une valeur.

<assignation> ::= <itemfeuille> = <valeur>
s'il s'agit d'un item décomposable.

<itemfeuille> est le nom d'un composant feuille de l'item répétitif décomposable.

<valeur> est toute valeur élémentaire pouvant apparaître dans un algorithme LDA.

DEVIENT

a1) create <varref-int> := <tart-int> ((<chemin> : <varref>) and <condition sur item>);

où <varref-int> ::= le nom d'une variable de référence du type d'articles inséré.
<tart-int> ::= le nom du type d'articles inséré.
<chemin> ::= le nom du chemin liant <varref> à <varref-int>.
<condition sur item> ::= une condition portant sur un item, telle que définie dans la grammaire des conditions de modification d'un article. La condition ne porte que sur un seul item.

SI <expr> ::= <valeur>
ALORS <condition sur item> ::= (:<item> = <valeur>)
SINON <condition sur item> rend compte de la décomposition de l'item répétitif décomposable et chaque feuille à laquelle est assignée une valeur se retrouve dans la condition sous la forme (: <itemfeuille> = <valeur>). Si <expr_déc> est <vide> dans la forme initiale, "and <condition sur item>" de même que les première et dernière parenthèses n'apparaissent pas dans l'instruction create.

Pour associer une valeur supplémentaire d'item répétitif à un article, il faut, dans le cas de l'insertion d'un type d'articles, créer un article intermédiaire associé à cette valeur pour valeur d'item, et le rattacher à l'article initial.

Exemple

Cet exemple est basé sur la figure 4.12.

modify pers (: prenom + {'Jean'});

DEVIENT

create pr_pr := pre_pers ((PPP : pers) and (: prenom = 'Jean'));

La personne "pers" se voit attribuer un prénom supplémentaire. On lui rattache donc un article intermédiaire (pr_pr) auquel on associe une valeur d'item égale au prénom.

a2) for <varref-int> := <tart-int> (<chemin> : <varref>)
if <cond> then exit <varref-int>
endif
endfor;
delete <varref-int>;

où <varref-int> ::= le nom d'une variable de référence du type d'articles inséré.
<tart-int> ::= le nom du type d'articles inséré.
<chemin> ::= le nom du chemin liant <varref> à <varref-int>.

si l'item répétitif <item> est élémentaire,
<cond> ::= (<varref_int>) . <item> = <valeur>, où <valeur> a la même signification que précédemment et est la valeur qu'on veut retirer.

si l'item répétitif <item> est décomposable,
<cond> ::= (<cond_elem>) and (<cond_elem>) |
(<cond_elem>) and <cond>
<cond_elem> ::= (<varref_int>) . <item>. <comp>.
<itemfeuille> = <valeur>
<item> ::= même signification que précédemment.
<varref_int> ::= même signification que précédemment.
<comp> ::= rend compte de la décomposition entre l'item décomposable <item> et le composant élémentaire <itemfeuille>.
<itemfeuille> =<valeur> correspond à l'<expr_déc> (<itemfeuille> = <valeur>) définie en 6a et est composant de la valeur (décomposable) que l'on retire de l'item répétitif décomposable.

Il y a autant de <cond_elem> que de composants élémentaires dans <item>. Quand l'utilisateur veut enlever une valeur décomposable de l'item répétitif décomposable, il peut ne pas spécifier les composants facultatifs. Ces derniers sont donc considérés comme NULL et apparaîtront dans <cond> comparés à cette valeur.

De manière générale, si un item répétitif a été éliminé par insertion d'un type d'articles intermédiaire, retirer une valeur de l'ensemble de celles prises par l'item répétitif revient à chercher l'article intermédiaire réprésentant cette valeur (d'où le test de sa valeur d'item) et à le détruire.

Exemple

Cet exemple est basé sur la figure 4.12.

modify pers (: prenom - { 'Jean' }) ;

DEVIENT

```
for  pr_pr := pre_pers ( PPP : pers )
     if ( pr_pr ) . prenom = 'Jean'
     then exit pr_pr
     endif
endfor;
delete pr_pr;
```

On veut enlever le prénom "Jean" de ceux pris par la personne "pers". On recherche alors l'article intermédiaire associé à "pers" et représentant ce prénom, et on le détruit.

b) for <varref-int> := <tart-int> (<chemin> : <varref>)
delete <varref-int>;
endfor;
create <varref-int> := <tart-int> ((<chemin> : <varref>) and <condition sur item>);
Il y a autant de create qu'il y a de <élément> ou de <valeur> (selon que l'item est décomposable ou pas) dans <liste-expr> (cfr forme initiale 6b). Si <élément> dans <liste_expr> est vide, l'instruction create apparait une seule fois et ne contient pas "and <condition sur item>" ni les première et dernière parenthèses.

où

<varref-int>	::=	le nom d'une variable de référence du type d'articles inséré.
<tart-int>	::=	le nom du type d'articles inséré.
<chemin>	::=	le nom du chemin liant <varref> à <varref-int>.
<condition sur item>	::=	une condition de modification d'un item simple telle que définie dans la syntaxe.

SI <item> était élémentaire dans la forme initiale (<liste_expr> est alors une liste de <valeur>)
ALORS <condition sur item> ::= (:<item> = <valeur>)
SINON (<liste_expr> est une liste d'<élément>)
<condition sur item> rend compte de la façon dont <item> se décompose et les correspondances entre <itemfeuille> et <valeur> dans <élément> se retrouvent dans <condition sur item>.

Par ce modify, on remplace un ensemble de valeurs par un autre. S'il y a eu insertion d'un type d'articles, il faut détruire tous les articles correspondant à l'ancien ensemble de valeurs pour après, créer ceux correspondant au nouvel ensemble.

Exemple

Cet exemple est basé sur la figure 4.12.

```
modify pers (: prenom = {'Jean'} );
```

DEVIENT

```
forpr_pr := pre_pers ( PPP : pers )
   delete pr_pr
endfor;
create pr_pr := pre_pers ( ( PPP : pers ) and (: prenom = 'Jean' ) );
```

Etant donné que "pers" ne doit plus être associé qu'à une valeur d'item "prenom" égale à "Jean", on détruit tout article intermédiaire représentant les anciennes valeurs de "prenom". Un article intermédiaire représentant le nouveau prénom est ensuite créé et rattaché à "pers".

Règle 7

for <varrefc> := <tart> ((<chemin> : <varrefo>) and <condition sur item>)
<trt>
endfor ;

(accès par clé dans un chemin)

où <varrefc> ::= le nom d'une variable de référence.
<varrefo> ::= le nom d'une variable de référence.
<tart> ::= le nom du type d'articles dont <varrefc> est une variable de référence.
<chemin> ::= le nom du chemin non récursif N-N liant <varrefo> à <varrefc>, ou le nom du rôle que joue <varrefo> dans le chemin récursif N-N le liant à <varrefc>.
<condition sur item> ::= des conditions portant sur un ou plusieurs items selon que la clé est une clé groupe ou pas. Plus précisément, chaque condition est une <cond_sélection_item> (cfr grammaire) où l'opérateur de comparaison est "=".
<trt> ::= un ensemble d'instructions.

DEVIENT

for <varref-int> := <tart-int> (<chemin1> : <varrefo>)
<varrefc> := <tart> (<chemin2> : <varref-int>);
if <cond> then <trt>
endif
endfor;

où <varref-int> ::= le nom d'une variable de référence du type d'articles inséré.
<tart-int> ::= le nom du type d'articles inséré.
<chemin1> ::= le nom du chemin liant <varrefo> à <varref-int>. Si le type de chemins éliminé était récursif, ce chemin représente le rôle de <varrefo> dans la forme initiale.
<chemin2> ::= le nom du chemin liant <varref-int> à <varrefc>. Si le type de chemins éliminé était récursif, ce chemin représente le rôle de <varrefc> dans la forme initiale.
<cond> ::= la définition est similaire à celle donnée en 6.a.2 lors de l'exposé de la forme transformée, mais elle peut porter ici sur plusieurs items. L'accès par clé dans un chemin s'est en effet transformé en un accès séquentiel aux cibles du chemin (en passant par l'article intermédiaire) avec test des valeurs d'items des articles auxquels on a accédé. Le test porte sur les items qui étaient clé dans le chemin; <cond> rend compte de ce test et peut donc porter sur N items.

Les items ont dû perdre leur propriété de clé dans la mesure où l'insertion d'un type d'articles dans le type de chemins aurait fait d'eux une clé dans un chemin N-1. Ceci est contraire à ce qui avait été exigé au chapitre 1.

Exemple

Cet exemple est basé sur la figure 4.9.

```
for  pro := produit ( ( cp : com ) and (: prix = prix_prod1 ) )
     <trt>
endfor;
```

DEVIENT

```
for  co_pr := com_pro ( CCP : com )
     pro := produit (PCP : co_pr );
     if (pro).prix = prix_prod1 then <trt>
     endif
endfor;
```

L'accès séquentiel aux articles "produit" se fait de façon classique avec passage par les articles intermédiaires "com_pro". Parmi les produits obtenus, il ne faut retenir que ceux qui vérifient la condition posée initialement sur la clé. On fait dès lors un test sur la valeur de l'item "prix" qui était initialement clé (dans le type de chemins "cp"). Si le test est positif, <trt> peut être exécuté.

Règle 8

```
for <exp> := (<varref>).<item>
     <trt>
enfor;
```

où <varref> ::= le nom d'une variable de référence.
<item> ::= le nom d'un item répétitif.
<trt> ::= un ensemble d'instructions.

a) <exp> ::= <variable> si <item> est élémentaire.

b) si <item> est décomposable,
<exp> ::= {<item décomposable>}
<item décomposable> ::= <variable> = <composant>,
<variable> = <composant> |
<variable> = <composant>, <item décomposable>
<composant> ::= le nom d'un composant feuille de l'item répétitif décomposable.

DEVIENT

```
for<varref-int> := <tart-int> (<chemin> : <varref>)
   <exp transformé>;
   <trt>
endfor;
```

où
<varref-int> ::= le nom d'une variable de référence du type d'articles inséré.
<tart-int> ::= le nom du type d'articles inséré.

<chemin> ::= le nom du chemin liant <varref> à <varref-int>.
<item> ::= c'est l'item de la forme initiale; il n'est cependant plus répétitif.

a) <exp transformé> ::= <variable> := (<varref-int>).<item>

b) <exp transformé> ::= <item décomposable transformé>
<item décomposable transformé> ::= <variable> := (<varref-int>).<item>.<comp>;
<variable> := (<varref-int>).<item>.<comp> |
<variable> := (<varref-int>).<item>.<comp>;
<item décomposable transformé>
<comp> ::= <composant> | <composant>.<comp>
<composant> ::= le nom d'un composant de l'item répétitif décomposable.

Les composants feuilles de <comp> sont les <composant> de la forme b) initiale.

Les variables associées à un composant dans la forme initiale le sont au même composant feuille dans la forme transformée.

Tout "next <exp>" (ou exit <exp>) apparaissant dans le corps de la boucle (<trt>) est transformé en "next <varref-int>" (ou exit <varref-int>).

L'accès aux valeurs d'un item répétitif devient donc un accès aux articles représentant chacun une valeur de l'item initial. Cet accès est suivi d'une initialisation explicite (<expr_transformé>) des (de la) variables de parcours aux valeurs d'items des articles auxquels on a accédé. De plus, la variable de parcours n'étant plus celle initiale, les modificateurs de boucle ("next" et "exit") qui spécifient la variable de parcours initiale doivent être transformés en conséquence.

Exemple

Cet exemple est basé sur la figure 4.12.

```
forpren := (pers).prenom
   <trt>
endfor;
```

DEVIENT

```
forpr_pr := pre_pers ( PPP : pers )
   pren := (pr_pr).prenom;
   <trt>
endfor;
```

On parcourt tous les prénoms de la personne "pers" en accédant à tous les articles intermédiaires (qui représentent chacun un prénom) et en obtenant les valeurs d'item "prenom" qui leur sont associées.

Règle 9

<varrefc> := <tart> ((<chemin> : <varrefo>) and <condition sur item>)

(assignation sur base d'un accès non identifiant par clé dans un chemin)

où <varrefc> ::= le nom d'une variable de référence.
<varrefo> ::= le nom d'une variable de référence.
<tart> ::= le nom du type d'articles dont <varrefc> est une variable de référence.
<chemin> ::= le nom du chemin non récursif N-N liant <varrefo> à <varrefc>, ou le nom du rôle que joue <varrefo> dans le chemin récursif N-N le liant à <varrefc>.
<condition sur item> ::= des conditions portant sur un ou plusieurs items selon que la clé est une clé groupe ou pas. Plus précisément, chaque condition est une <cond_sélection_item> (v. grammaire) où l'opérateur de comparaison est "=".

DEVIENT

```
for  <varref-int> := <tart-int> (<chemin1> : <varrefo>)
     <varrefc> := <tart> (<chemin2> : <varref-int>);
     if  <cond> then exit <varref–int>
         else <varrefc> := ()
     endif
endfor;
```

où <varref-int> ::= le nom d'une variable de référence du type d'articles inséré.
<tart-int> ::= le nom du type d'articles inséré.
<chemin1> ::= le nom du chemin liant <varrefo> à <varref-int>. Si le type de chemins éliminé était récursif, ce chemin représente le rôle de <varrefo> dans la forme initiale.
<chemin2> ::= le nom du chemin liant <varref-int> à <varrefc>. Si le type de chemins éliminé était récursif, ce chemin représente le rôle de <varrefc> dans la forme initiale.
<cond> ::= la définition est similaire à celle donnée en 6.a.2 lors de l'exposé de la forme transformée, mais elle peut porter ici sur plusieurs items. L'accès par clé dans un chemin s'est en effet transformé en un accès séquentiel aux cibles du chemin (en passant par l'article intermédiaire) avec test des valeurs d'items des articles auxquels on a accédé. Le test porte sur les items qui étaient clé dans le chemin; <cond> rend compte de ce test et peut donc porter sur N items.

La forme syntaxique originale assigne à la variable de référence <varrefc> la référence d'un article parmi ceux qui vérifient la condition de sélection. La forme transformée accède (par une boucle "for" et via un type d'articles intermédiaire) à tous les articles susceptibles de vérifier la condition de sélection. Dès qu'un article vérifie cette condition, toute recherche ultérieure devient inutile et "exit <varref–int>" assure la sortie de la boucle. <varref> désigne alors un article parmi ceux qui vérifient la condition de sélection. L'instruction "else <varrefc> := ()" est requise pour que <varrefc> ne désigne aucun article si aucun article ne vérifie la condition de sélection.

Exemple

Cet exemple est basé sur la figure 4.9.

```
p := produit ( ( cp : com_1 ) and ( : prix = max_prix ) ) ;
```

DEVIENT

```
for  co_pr := com_pro ( CCP : com )
     p := produit ( PCP : co_pr ) ;
     if ( p ) . prix = max_prix
        then exit co_pr
        else p := ()
     endif
endfor;
```

On accède par le "for" à tous les articles "produit" reliés à "com". Dès qu'on en trouve un dont la valeur d'item "prix" est égale à "max_prix", on peut affirmer qu'il vérifie la condition de sélection initale et on sort de la boucle. Si aucun article n'a été retenu, "p" ne désigne rien.

Réflexions

Gestion des articles intermédiaires associés à un article détruit.

Soit "delete <varrefo>" où <varrefo> est le nom d'une variable de référence. L'article que désigne cette variable est relié à des articles intermédiaires.

Si ces articles intermédiaires rendent compte d'un chemin de classe fonctionnelle N–N auquel participait <varrefo>, et que le type de chemins a été éliminé, ces articles doivent être détruits. En effet, tous les chemins dont un article détruit est origine disparaissent eux aussi; et d'autre part, un article détruit est évidemment éjecté des chemins dont il est cible.

Si ce n'est pas un type de chemins qui a été éliminé, mais bien un item répétitif, le raisonnement est analogue. Les valeurs d'items associées à un article ne peuvent évidemment survivre à la disparition de cet article. Le cas présent, les articles intermédiaires représentent chacun une valeur de l'item répétitif initialement associé à l'article <varrefo> qui est détruit. Ces articles doivent donc être détruits également.

La gestion de ces destructions ne pose cependant pas de difficultés. Aucune transformation d'algorithme n'est nécessaire. LDA/MAG gère, en effet, la destruction des cibles obligatoires de chemins 1–N dont l'origine est détruite. Or, si on se réfère aux transformations de schéma, on constate que les articles intermédiaires sont des cibles obligatoires de chemins 1–N.

Accès par clé et par chemin

Si on se réfère à la transformation exposée à la figure 4.12, l'instruction
`for  p := personne (: prenom = 'Jean' )`

DEVIENT

```
for  pp := pre_pers (: prenom = 'Jean' )
     p := personne ( PPP : pp );
     ...
```

On remarque que, contrairement à la forme initiale, la variable de référence "p" n'est

plus initialisée dans la condition de parcours d'un for mais bien dans une simple assignation. Cela ne pose cependant pas de problème dans la mesure où la condition "personne (PPP : pp)" rend systématiquement un et un seul article (cela est dû à la classe fonctionnelle N-1 de "PPP" et à son caractère obligatoire pour "pre_pers"). La présence d'un for devant "p" devient donc inutile. Elle peut même ne pas être souhaitable car une assignation est plus avantageuse en termes de performances qu'un for exécuté une seule fois.

Le raisonnement est tout à fait analogue dans le cas d'un type de chemins N-N éliminé par insertion d'un type d'articles.

Les clés d'accès répétitives

On peut accepter un item répétitif clé d'accès dans la mesure où sa transformation est aisée et donne lieu à un simple accès par chemin. Cependant, comme cela a déjà été exigé au chapitre 1, si la clé est composée de plusieurs items, aucun ne peut être répétitif. Cela se justifie par le fait que si plusieurs composants étaient répétitifs, ils devraient être éliminés par insertion d'un type d'articles intermédiaire (cfr exemple de la figure 4.12 où un type d'articles "pre_pers" est inséré entre l'item "prenom" et le type d'articles "personne"). Dès lors, ce qui était initialement un accès par clé devient un accès sur base de plusieurs chemins et de valeurs d'items, ce qui est beaucoup plus délicat à gérer.

Si la clé est identifiante, le raisonnement est analogue. La restriction imposée au chapitre 1 à propos des composants d'identifiant qui ne peuvent être répétitifs est donc justifiée dans la mesure où Codasyl impose que tout identifiant soit clé d'accès.

Si on a une clé d'accès dans un chemin, et que la classe fonctionnelle du type de chemins est 1-N, le type de chemins ne doit pas être éliminé car c'est une structure qu'accepte Codasyl. Toutefois, si la clé est un item répétitif ou un groupe d'items dont au moins un est répétitif, ces items doivent être transformés par l'insertion d'un type d'articles similairement à ce qui a été décrit ci-dessus. On obtient donc encore un accès sur base de plusieurs chemins, ce qui n'est pas pris en compte dans la transformation. Cela justifie le refus d'items répétitifs comme composants de clés d'accès dans des chemins.

Si le type de chemins supportant la clé d'accès est de classe N-N, l'accès par clé dans le chemin est transformé, comme cela a déjà été vu, en accès séquentiel avec test des valeurs d'items qui étaient la clé initiale. On pourrait donc accepter les items répétitifs. Cependant, dans un souci de simplification, ils sont réfusés également dans ce cas.

La restriction énoncée au chapitre 1 comme quoi aucun item composant d'une clé d'accès dans un chemin ne pouvait être répétitif est donc justifiée. Dans le cas où cette clé est identifiante, ou en d'autres termes, dans le cas d'un identifiant composé d'items et d'un composant chemin, le raisonnement est différent. Il sera exposé, comme cela a déjà été dit au chapitre 1, dans les annexes lors du détail de ce qui a été abandonné suite à l'analyse.

D'un point de vue plus général, d'aucuns pourraient argumenter qu'au moment où on écrit les algorithmes LDA/MAG, on ne connaît pas le SGD cible et que dès lors, certaines restrictions imposées par l'utilisation de tel ou tel SGD cible peuvent paraître excessives. Elles pourraient même paraître tout à fait injustifiées dès l'instant où le SGD cible utilisé n'est pas celui pour lequel les restrictions ont été imposées. La réponse est que c'est justement parce qu'on ne connaît pas le SGD cible durant la phase de conception logique qu'il faut refuser des structures de données qui pourraient poser des problèmes au niveau de conception physique.

Les clés d'accès répétitives décomposables

Comme cela a déjà été dit au chapitre 1, si une clé d'accès est définie avec les composants d'un item répétitif, elle doit les regrouper tous.

Soit l'exemple suivant

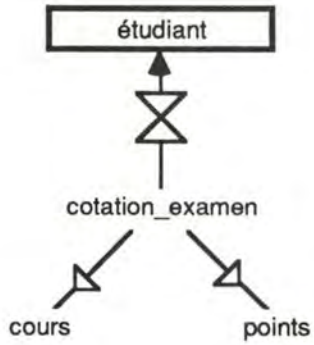


fig. 4.13 : exemple d'item clé répétitif décomposable

La seule clé qui puisse être définie ici doit être composée de "cours" et "points" (ou plus simplement de "cotation"). Si on permettait de définir une clé sur un des composants (par ex. "points"), cela donnerait lieu aux problèmes suivants :

Soit la transformation suivante

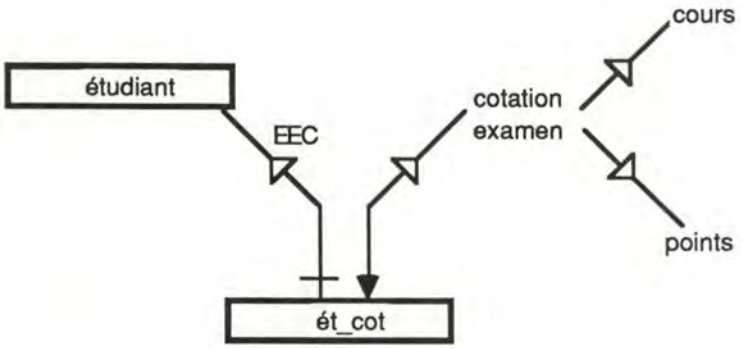


fig 4.14 : transformation d'un item répétitif décomposable

Un accès par clé sur base de "points" dans le schéma initial

```
for  ec := etudiant (: cotation_examen (: points = 12 ) )
```

DEVIENT

```
for  ec := et_cot (: cotation_examen (: points = 12 ) )
     e := etudiant ( EEC : ec );
```

Etant donné le caractère ensembliste des valeurs prises par cotation, il ne peut y avoir deux valeurs décomposables (cours, points) identiques. Certains composants peuvent cependant se répéter plusieurs fois. En particulier, il est logique de penser que certains étudiants peuvent avoir obtenu plusieurs fois 12 comme points d'examens passés. Si la clé est transformée, ces étudiants se retrouveront donc plusieurs fois dans la séquence des étudiants ayant obtenu 12, ce qui n'était pas le cas dans la forme initiale. Cela est tout à fait inacceptable. Si l'accès ne peut se faire que sur base de "cotation", le problème de la survenance de doubles indésirables ne se pose pas.

Accès aux valeurs d'un item répétitif

Soit l'exemple de la figure 4.12 où l'item répétitif "prenom" du type d'articles "personne" est éliminé par insertion d'un type d'articles " pre_pers".

L'accès aux prénoms de la personne "pers1" et leur rangement dans un tableau peut se faire de la façon suivante :

```
i := 1;
for  x := (pers1).prenom
     tab[ i ] := x;
     i := i + 1;
endfor;
```

ET DEVIENT DONC

```
i := 1;
for  pp := pre_pers ( PPP : pers1 )
     x := (pp).prenom;
     tab[ i ] := x;
     i := i + 1;
endfor;
```

Il est évidemment impératif d'exiger que les contenus de "tab" dans l'algorithme initial et dans celui transformé soient identiques. Rien ne le garantit cependant dans la mesure où dans un cas, "tab" est initialisé à partir des valeurs prises par un item répétitif et dans l'autre à partir des valeurs d'items simples d'articles auxquels on a accédé via un chemin.

De façon plus générale, il faut déterminer la façon dont sont obtenues les valeurs d'un item répétitif pour qu'elle s'accorde avec la façon dont ces valeurs sont obtenues dans l'algorithme transformé. On obtient les valeurs d'item dans l'ordre de création c'est-à-dire l'ordre selon lequel elles apparaissent dans la dernière liste de valeurs qui a servi à initialiser l'item (lors d'un create ou d'un modify). Les valeurs ajoutées sont obtenues après celles de la liste et également dans l'ordre de création. Tout retrait d'une valeur provoquera un "tassement" de l'ensemble c'est-à-dire qu'en parcourant ces valeurs, on n'obtiendra jamais de "trou". Plus simplement, on obtient les valeurs d'un item par ordre chronologique.

Ce problème ne concerne pas l'utilisateur qui n'a pas connaissance de la façon dont sont transformés les algorithmes.

La désignation des valeurs d'items répétitifs

Comme cela a déjà été dit au premier chapitre et répété au troisième, il est interdit d'indicer les valeurs d'un item répétitif. En d'autres termes, une expression comme

"(pers).prenom[i]" est illicite.

Cette restriction est imposée dans la mesure où il est anormal de désigner une $i^{ème}$ valeur dans l'ENSEMBLE de celles prises par un item répétitif.

Une raison plus concrète est que la désignation sur base d'indice entraîne des transformations d'algorithmes complexes et peu efficaces. En effet, sur base de la transformation exposée à la figure 4.12, l'expression "(pers).prenom[i]" devient

```
j := 1;
for  pp := pre_pers( PPP : pers )
     if j = i then exit pp
     endif;
     j := j + 1
endfor;
(pp).prenom ...
```

Il faut donc accéder à tous les articles intermédiaires reliés à "pers", les compter et prendre la valeur d'item du $i^{ème}$.

Outre la lourdeur évidente de cet algorithme, on constate qu'il multiplie les accès à la base de données. Il peut donc être condidéré comme coûteux. Il l'est d'autant plus dans les cas où l'utilisateur veut accéder à toutes les valeurs d'un item répétitif.

Par exemple :

```
for  i:= 1..10
     ...
     (pers).prenom[ i ];
     ...
endfor;
```

Dans la forme transformée, la première itération nécessite l'accès à un article intermédiaire, la seconde à 2, la troisième à 3,...

Plus généralement, les accès multiples et inutiles surviennent dans les cas où (pers).prenom[i] apparaît dans une instruction itérative (while, for). En effet, chaque itération nécessite i accès aux articles intermédiaires. Or, il se peut que d'une itération à l'autre, (pers).prenom[i] désigne toujours la même valeur et que, dès lors, les accès soient inutiles. D'autre part, la détection de ce genre de situation relève de l'impossible.

Il est donc plus raisonnable de proscrire les formes du type de "(pers).prenom[i]".

Les types d'articles intermédiaires et leurs variables de référence

Dès l'instant où des types d'articles sont insérés dans des types de chemins ou des associations type d'articles-item, il est fort probable qu'ils devront être manipulés dans des algorithmes transformés. Cette manipulation ne peut se faire évidemment que par le biais de variables de référence (qui apparaissent généralement dans les règles de transformation sous le vocable "<varref_int>").

Le programmeur ne connaît pas la façon dont sont transformés les algorithmes qu'il a écrit. Il lui est évidemment impossible de déclarer ces variables lui-même. Cette déclaration se fera donc lors de la transformation des algorithmes.

La solution la plus simple consiste à déclarer une nouvelle variable (appelons-la intermédiaire) chaque fois qu'il est nécessaire de manipuler un article intermédiaire. Elle

risque cependant d'entraîner un nombre excessif de déclarations par rapport à l'utilisation qui est faite des variables déclarées.

Il est, en effet, souvent possible de réutiliser des variables de référence intermédiaires déjà déclarées. Cela vient du fait qu'elles sont généralement utilisées au plus une fois après assignation. Dès lors après cette utilisation, elles peuvent être réemployées à souhait.

Il y a lieu cependant de nuancer cette affirmation dans le cas des boucles for. Si une variable intermédiaire est la variable de parcours d'une boucle for, elle ne peut pas être considérée comme "libre" (et donc réutilisée à d'autres fins) dans le corps de cette boucle.

Considérons, à titre d'illustration, l'exemple de la figure 4.7 où "commande" est relié à "produit" par le type de chemins "cp" de classe fonctionnelle N-N.

La boucle

```
for  p := produit ( cp : com1 )
...
endfor;
```

DEVIENT

```
for  c_prod := com_pro ( CCP : com1 )
     p := produit ( PCP : c_prod );
     ...
endfor;
```

Si "c_prod" était réutilisée dans le corps de boucle pour désigner un article quelconque, elle perdrait sa signification de variable de parcours qui veut qu'elle désigne successivement tous les articles de la séquence.

La manipulation des données structurées

Il a été exigé lors de la première transformation (aplatissement) qu'une valeur d'item, argument d'un appel, doive être élémentaire. De façon analogue, l'ensemble des valeurs d'un item répétitif ne peut être un argument d'un appel de procédure ou de fonction.

5. INSERTION D'UN TYPE D'ARTICLES ET DUPLICATION D'ITEM

Cette transformation peut être définie comme l'insertion d'un type d'articles suivie de l'élimination des types de chemins obtenus, par rotation autour du type d'articles inséré. Elle est utilisée pour éliminer des items répétitifs élémentaires (identifiants ou pas) et des types de chemins de classe N–N (récursifs ou pas) pour SQL. Elle est employée également pour des items identifiants répétitifs (décomposables ou pas) pour Cobol.

Exemples

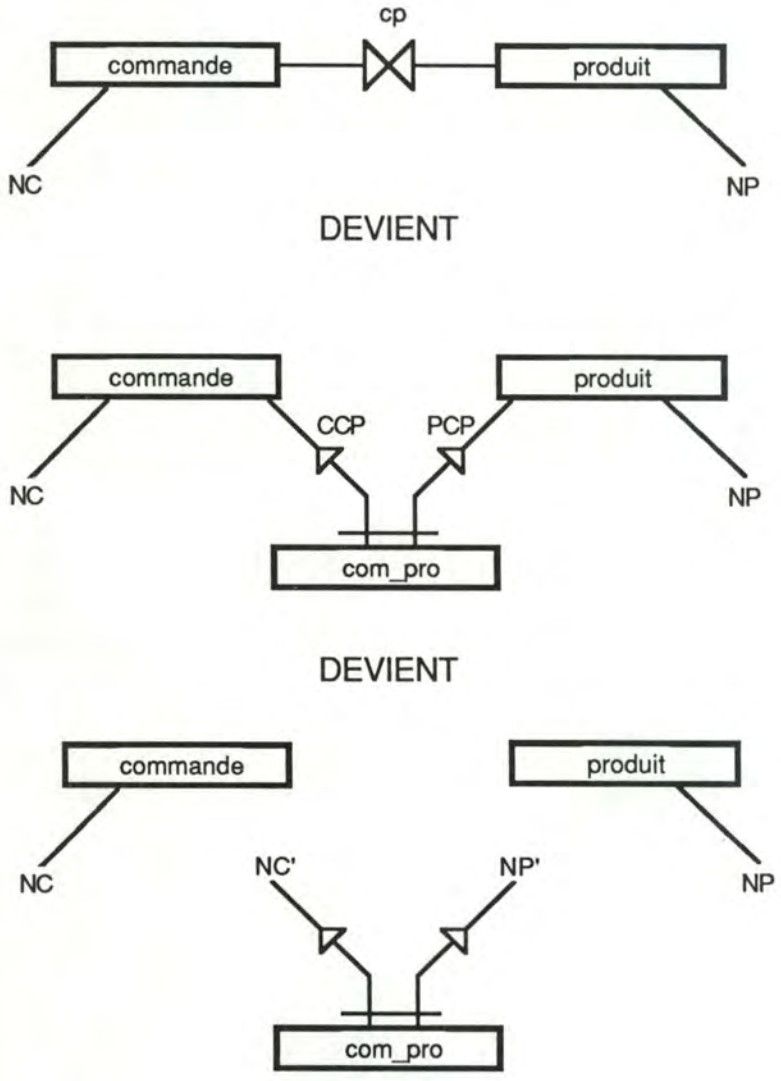
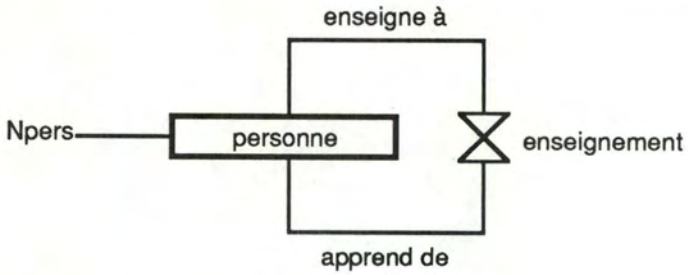
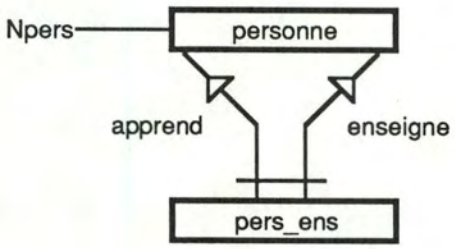


Fig. 4.15 : insertion d'un type d'articles et rotation dans le cas d'un type de chemins non récursif.



DEVIENT



DEVIENT

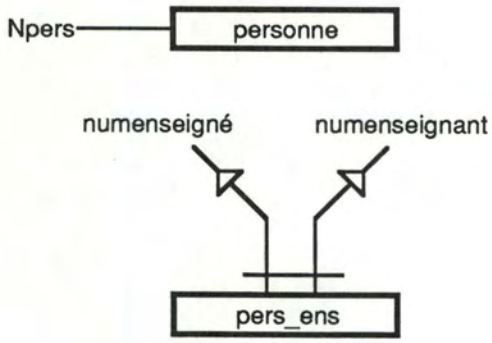


Fig. 4.16 : insertion d'un type d'articles et rotation
dans le cas d'un type de chemins récursif.

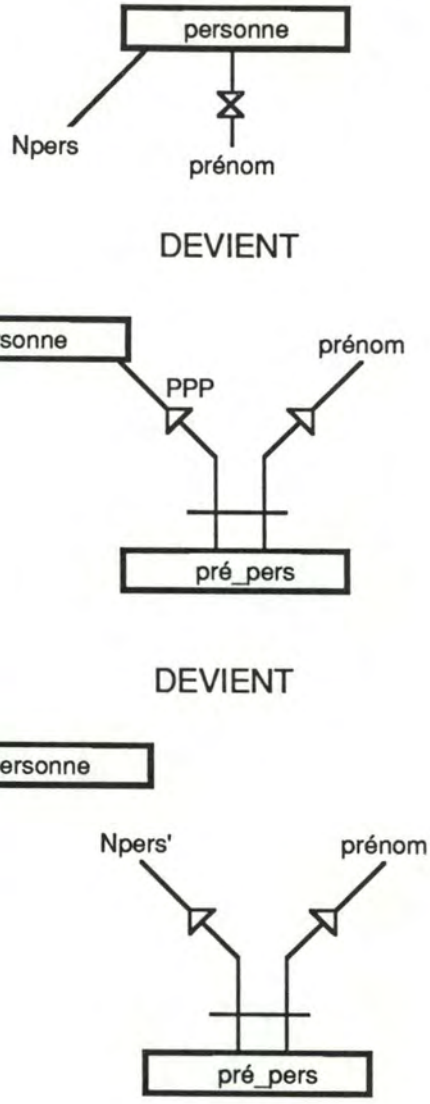


Fig 4.17 : insertion d'un type d'articles et rotation dans le cas d'un item répétitif non identifiant

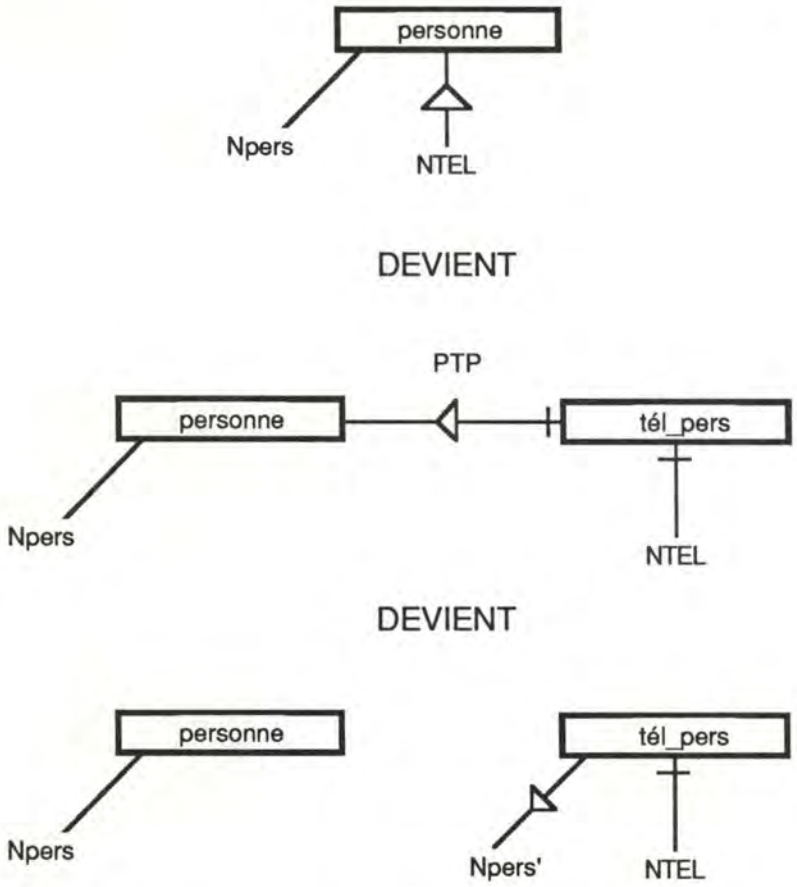


Fig 4.18 : insertion d'un type d'articles et rotation dans le cas d'un item répétitif identifiant

La transformation aurait été similaire si "NTEL" avait été identifiant.

Règle 1

Les règles de transformation syntaxique peuvent être aisément déduites de celles définies à propos de l'insertion d'un type d'articles et de la rotation.

La règle est que selon une forme syntaxique initiale, on applique une des règles définies pour la transformation 4 (insertion d'un type d'articles). A la forme syntaxique obtenue, on applique une des règles de la transformation 3 (élimination d'un type de chemins par duplication d'un identifiant).

Il est important, cependant, de garder à l'esprit que la transformation totale exposée ici forme un tout indécomposable. Seule sa définition a été "morcelée" afin de tirer profit du travail effectué pour d'autres transformations.

Quelques exemples informels de son utilisation sont donnés ci–dessous.

Exemples

Cet exemple est basé sur la figure 4.16.

1. modify pers (: Npers = 250) ;

DEVIENT

```
for  p_e := pers_ens ( : numenseigne = ( pers ) . Npers )
     modify p_e ( : numenseigne = 250 )
endfor ;
for  p_e := pers_ens ( : numenseignant = ( pers ) . Npers )
     modify p_e ( : numenseignant = 250 )
enfor ;
modify pers ( : Npers = 250 ) ;
```

Cette transformation a pour but de faire respecter la contrainte référentielle apparue suite à l'élimination des types de chemins. Les numéros d'enseignant et d'enseigné doivent être des numéros de personne. Dès lors, quand un numéro de personne (Npers) est modifié, il faut modifier en conséquence les numéros d'enseignant (numenseignant) et d'enseigné (numenseigne) correspondants.

Plus concrètement, au vu de la forme initiale, aucune règle de la transformation 4 (insertion) n'est appliquée. Au vu de la forme "transformée", la règle 6 de maintien de la contrainte référentielle de la transformation 3 (rotation) est employée. Cette règle doit être employée deux fois, étant donné que deux rotations sont faites sur base de "Npers".

Rem. Lors de l'exposé de la transformation "insertion d'un type d'articles", il a été question de la réutilisation des variables de référence intermédiaires. Les deux utilisations successives de "p_e" dans l'exemple ci–dessus (dans le premier et le second "for") en sont une parfaite illustration.

Toujours basé sur la même figure (4.16) :

2. modify pers2 (enseigne_a : pers_1) ;

La base de données rend compte maintenant du fait que la personne "pers1" enseigne à la personne "pers2".

DEVIENT

```
create E_1 := pers_ens  ( ( : numenseignant = ( pers1 ) . Npers ) and
                          ( : numenseigne = ( pers2 ) . Npers ) ;
```

Il faut établir une relation d'enseignant à enseigné entre deux articles. On crée donc un article intermédiaire qui rend compte de cette relation. Il est associé pour cela aux valeurs d'items copies des numéros de personne (Npers) des articles reliés. Ces valeurs sont le numéro d'enseignant (numenseignant) pour l'enseignant "pers1" et d'enseigné (numenseigne) pour l'élève "pers2".

La règle générale exposée ci–dessus (règle n°1) admet cependant une exception. Il faut dans ce cas appliquer la règle spécifiquement définie pour cette transformation. Cette règle est exposée ci–dessous.

Règle 2

modify <varrefc> (<chemin>: 0 <varrefo>) ;

où <varrefc> ::= le nom d'une variable de référence
<varrefo> ::= le nom d'une variable de référence

<chemin> ::= le nom du chemin non récursif N-N liant <varrefo> à <varrefc>, ou le nom du rôle que joue <varrefo> dans le chemin récursif N-N le liant à <varrefc>

DEVIENT

<varref-int> := <tart-int> ((: <itemc'> = (<varrefc>).<itemc>) and
(: <itemo'> = (<varrefo>).<itemo>)) ;
delete <varref-int> ;

où <varref-int> ::= le nom d'une variable de référence du type d'articles inséré
<tart-int> ::= le nom du type d'articles inséré
<itemc> ::= le nom de l'item identifiant dupliqué de <varrefc>
<itemo> ::= le nom de l'item identifiant dupliqué de <varrefo>
<itemc'> ::= le nom de l'item duplicata de <itemc>. Dans le cas d'un type de chemins récursif, <itemc'> représente le rôle joué par <varrefc> dans le chemin récursif initial (<chemin>).
<itemo'> ::= le nom de l'item duplicata de <itemo>. Dans le cas d'un type de chemins récursif, <itemo'> représente le rôle joué par <varrefo> dans le chemin récursif initial (<chemin>).

Comme précédemment pour l'insertion d'un type d'articles dans un type de chemins, si on veut détacher deux articles, on détruit l'article intermédiaire qui rend compte de leur association. Lorsque les types de chemins ne sont pas éliminés, (Cfr transformation 4 : insertion d'un type d'articles) la recherche de l'article intermédiaire est assez complexe. Le cas présent, elle se résume à une simple assignation sur base d'un accès par clé, étant donné que le type d'articles intermédiaire est identifié par les deux items duplicatas ou, en d'autres termes, par les identifiants des types d'articles qu'il associe.

exemple

Cet exemple est basé sur la figure 4.15.

modify com_1 (cp : 0 pro_1) ;

DEVIENT

c_p_1 := com_pro((: NC' = (com_1) . NC) and ((: NP' = (pro_1).NP)) ;

Pour détacher "com_1" et "pro_1", on accède à l'article de type "com_pro" qui les relie, et on le détruit. "NP'" et "NC'" forment une clé identifiante qui rend l'opération aisée.

6. INSERTION D'UN TYPE D'ARTICLES, ROTATION ET AJOUT D'UN NIVEAU DE DECOMPOSITION

Cette transformation est définie comme la transformation 5 (insertion d'un type d'articles et duplication d'item) suivie de la transformation 2 (ajout d'un niveau de décomposition). Elle est utilisée pour éliminer des types de chemins N–N (récursifs ou pas) et des items clé répétitifs non identifiants (décomposables ou pas) pour Cobol.

La transformation 5 (insertion et rotation) engendre des structures de données où, dans certains cas, apparaissent des identifiants composés de plusieurs items, ce que Cobol refuse. La transformation présente fait de ces identifiants un seul item et peut donc être utilisée pour Cobol.

Les exemples suivants détaillent les étapes de cette transformation.

Exemples

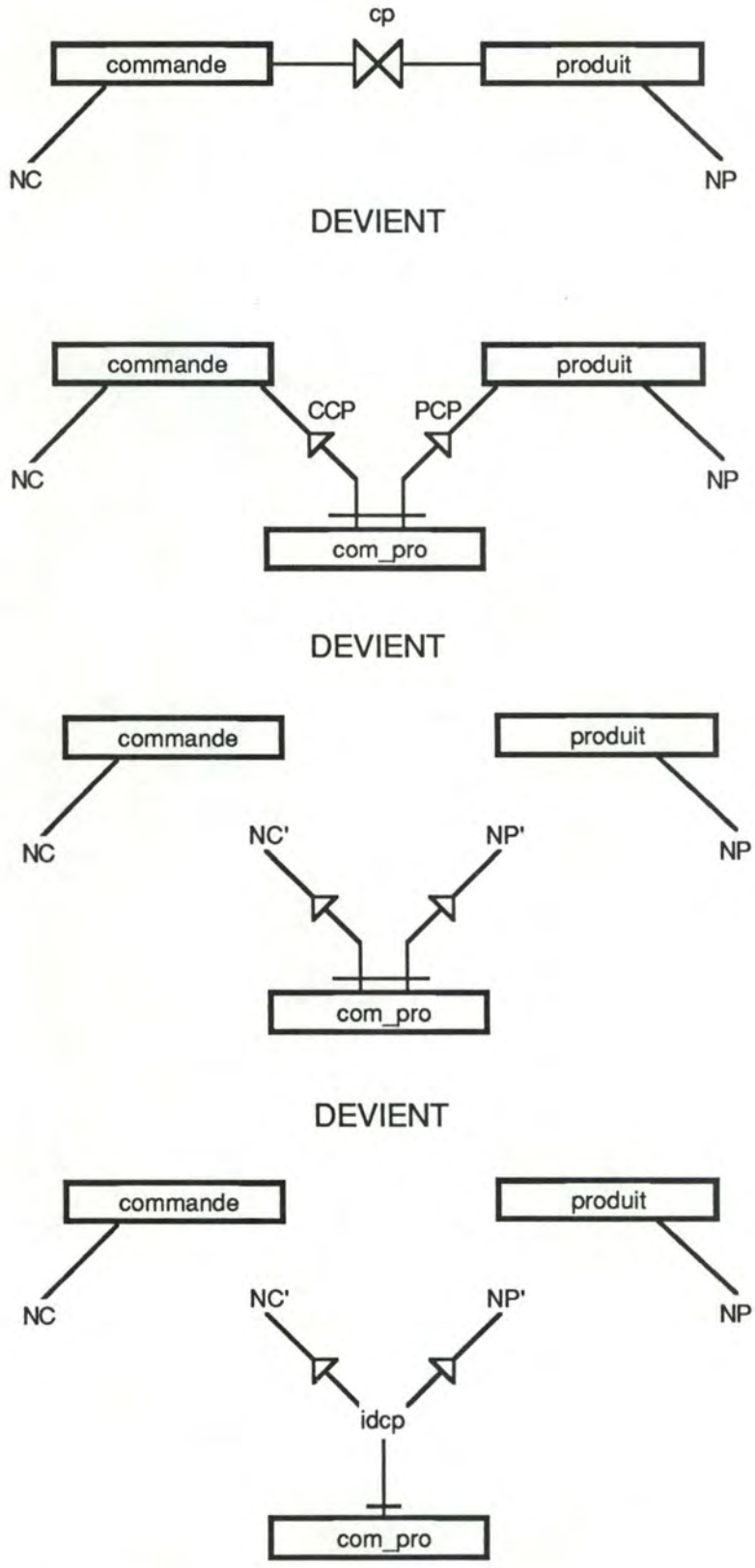
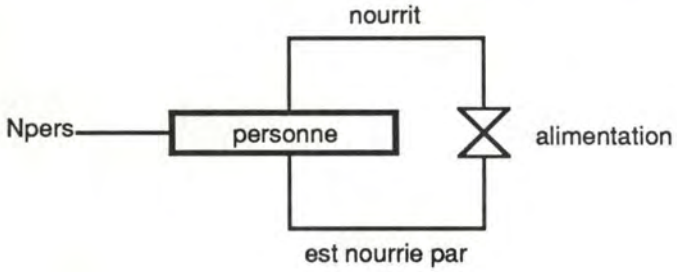
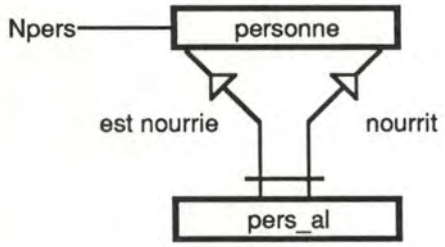


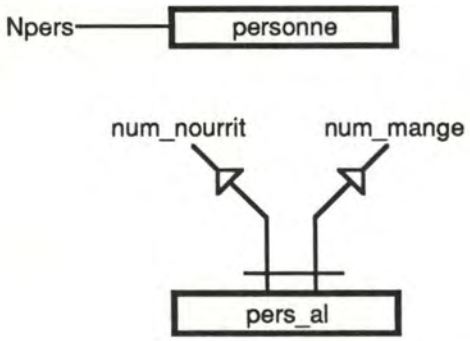
Fig. 4.19 : insertion, rotation et ajout d'un niveau de décomposition dans le cas d'un type de chemins non récursif.



DEVIENT



DEVIENT



DEVIENT

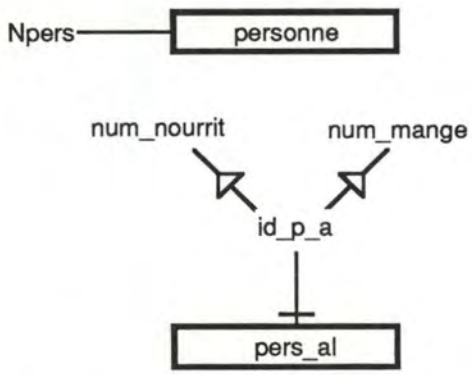
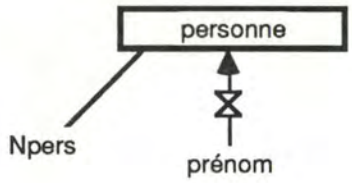
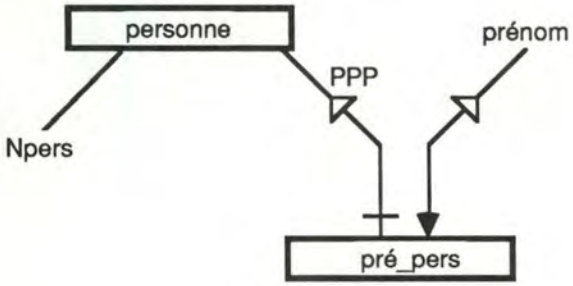


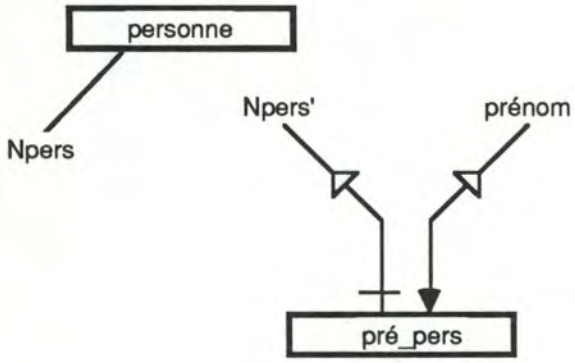
Fig. 4.20 : insertion, rotation et ajout d'un niveau de décomposition dans le cas d'un type de chemins récursif.



DEVIENT



DEVIENT



DEVIENT

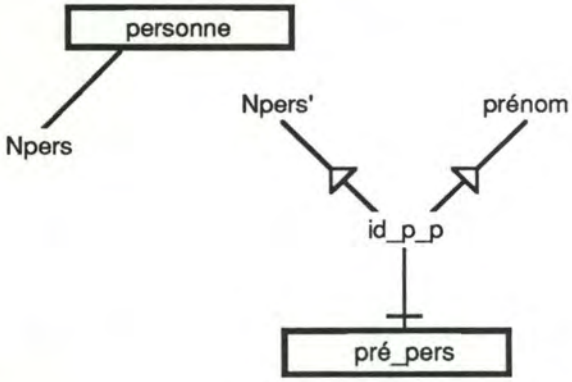


Fig. 4.21 : insertion, rotation et ajout d'un niveau de décomposition dans le cas d'un item clé répétitif.

Si "prénom" avait été un item décomposable, la transformation aurait été similaire.

Règle 1

Le principe de composition des règles exposé à la transformation précédente est toujours valable.

La règle de transformation est que selon une forme syntaxique initiale, on applique une des règles définies pour la transformation 5 (insertion d'un type d'articles et duplication d'item). A la forme syntaxique obtenue, on applique une des règles de la transformation 2 (ajout d'un niveau de décomposition).

Exemples

Cet exemple est basé sur la figure 4.19.

1. for p := produit (cp : com)

DEVIENT

```
for  cp_1 := com_pro ( : idcp ( ( : NC' = ( com ) . NC )
     p := produit ( : NP = ( cp_1 ) . idcp . NP' ) ;
```

L'accès par chemin devient un accès aux articles intermédiaires et de là, aux articles cibles de l'accès initial.

De façon plus précise, cette transformation peut être "découpée" selon les étapes de la transformation de schéma. Il est bon de rappeler une fois de plus que ces étapes sont fictives et ne servent qu'à clarifier l'exposé.

La première étape consiste à appliquer la transformation 5 (insertion et rotation). Or celle–ci est définie à partir de deux autres transformations; dès lors :

On applique une des règles de la transformation 4 (insertion d'un type d'articles) selon la forme syntaxique initiale; on obtient alors :

```
for  cp_1 := com_pro ( CCP : com )
     p := produit ( PCP : cp_1 ) ;
```

De façon classique, l'insertion d'un type d'articles fait qu'un accès initial doit se faire, après insertion, par le biais d'articles intermédiaires.

L'étape suivante consiste à éliminer les types de chemins "CCP" et "PCP". A la forme ci–dessus, on applique donc deux fois une règle de la transformation 3 (rotation) et on obtient :

```
for  cp_1 := com_pro ( NC' = ( com ) . NC )
     p := produit ( : NP = ( cp_1 ) . NP' ) ;
```

où les accès par chemin deviennent des accès par clé quand les types de chemins ont été éliminés.

La transformation 5 est ici terminée.

La dernière étape consiste à ajouter le niveau de décomposition. On applique deux règles (une pour "NP'" et une pour "NC'") de la transformation 2 (ajout d'un niveau de décomposition) et on obtient la forme finale.

L'exemple suivant est basé sur la figure 4.21;

```
2.   for pren := ( pers ) . prenom
     . . .
     endfor ;
```

DEVIENT

```
for  pp := pre_pers ( : id_p_p ( : Npers' = ( pers ) . Npers ) )
     pren := ( pp ) . id_p_p . prenom ;
     . . .
endfor;
```

Pour parcourir les valeurs de l'item répétitif "prenom" d'un article "personne", on accède sur base de la copie du numéro de personne ("Npers'") aux articles intermédiaires. Chacun des ces articles représente un prénom de la personne. On accède ensuite à la valeur de l'item "prenom" de chaque article intermédiaire obtenu.

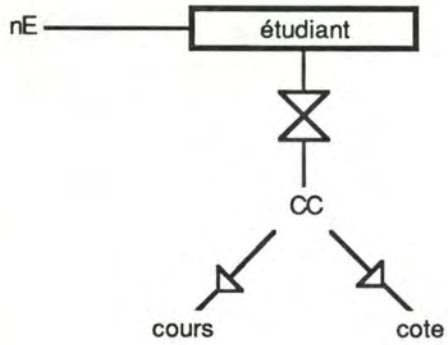
Règle 2

La règle générale exposée ci–dessus (règle n°1) ne couvre cependant pas tous les cas. Il faut alors définir une règle spécifique à la transformation envisagée.

La transformation 5 (insertion et rotation) ne prend pas en compte les items répétitifs non identifiants décomposables. En effet, une insertion d'un type d'articles suivie d'une rotation du type de chemins obtenu crée un identifiant composé de deux items (voir p. ex. fig. 4.17). La transformation ne peut donc être utilisée pour Cobol.

Si l'item répétitif est clé d'accès, il doit être éliminé. Pour ce faire, la règle est de combiner les règles de trois transformations. Selon une forme syntaxique initiale, on applique une des règles de la transformation 4 (insertion d'un type d'articles). A la forme obtenue, on applique une des règles de la transformation 3 (rotation). Pour aboutir à la forme finale, on applique une des règles de la transformation 2 (ajout d'un niveau de décomposition).

Exemple



DEVIENT (sans le détail connu des étapes)

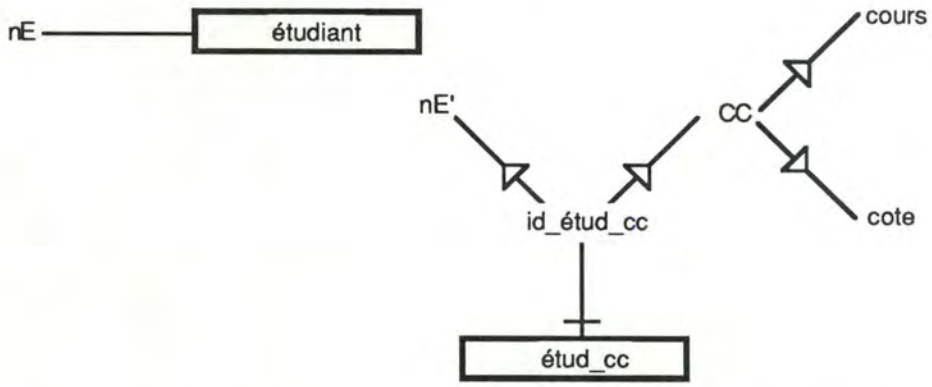


Fig : 4.22 : insertion d'un type d'articles, rotation et ajout d'un niveau de décomposition dans le cas d'un item répétitif décomposable non identifiant

et

```
for  { cte = cote, crs = cours } := ( etud_1 ) . cc
...
endfor ;
```

DEVIENT

```
for  ecc := etud_cc ( : id_etud_cc ( : nE' = ( etud_1 ) . nE )
     cte := ( ecc ) . id_etud_cc . cc . cote ;
     crs := ( ecc ) . id_etud_cc . cc . cours ;
     ...
endfor ;
```

Pour accéder aux valeurs d'un item répétitif décomposable, on accède aux articles intermédiaires qui rendent compte de ces valeurs. On donne ensuite à chaque variable de parcours la valeur de l'item élémentaire correspondant, pour chaque article auquel on a accédé.

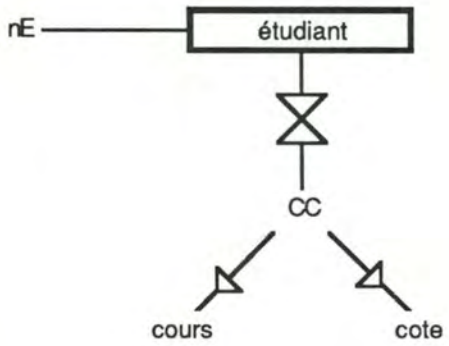
7. INSERTION D'UN TYPE D'ARTICLES, ROTATION ET APLATISSEMENT TOTAL

Cette transformation est définie comme la transformation 4 (insertion d'un type d'articles) suivie de la transformation 3 (rotation) suivie elle-même de la transformation 1 (aplatissement total).

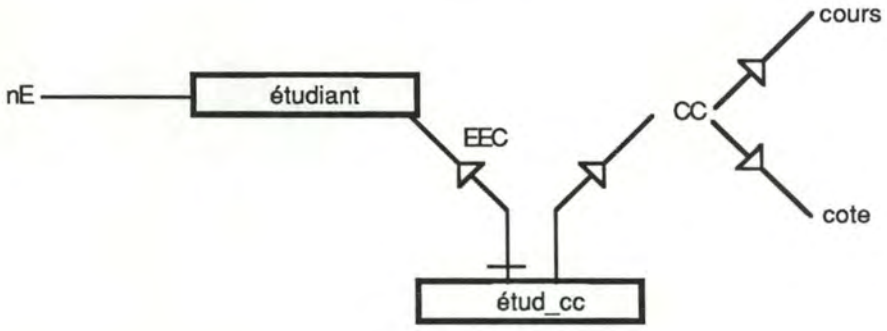
Elle est utilisée pour éliminer un item répétitif décomposable (identifiant ou non) non accepté par SQL.

Exemple

La forme initiale de la figure 4.22 peut être reprise pour détailler les étapes de cette transformation.



DEVIENT



DEVIENT

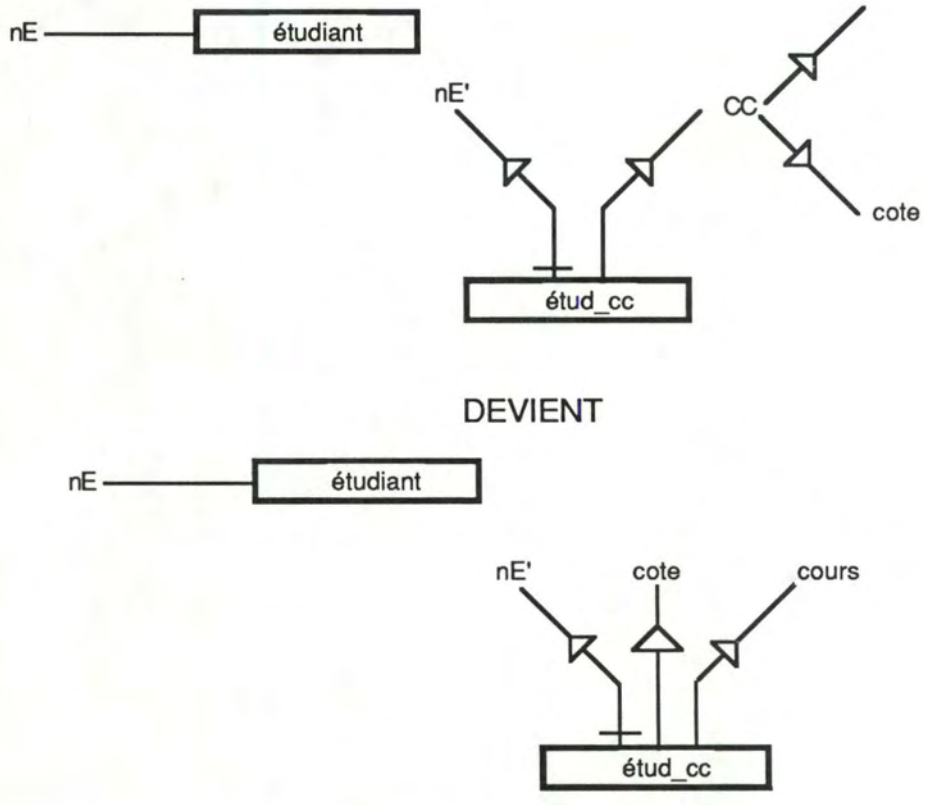
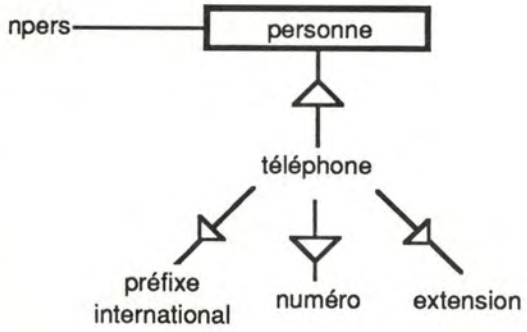
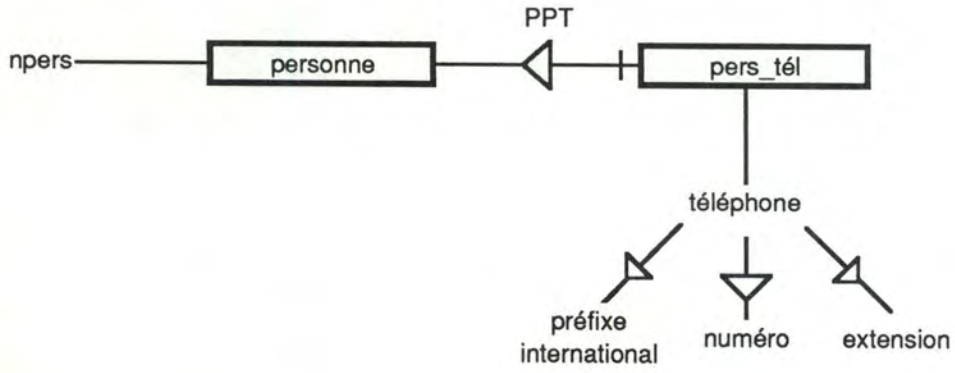


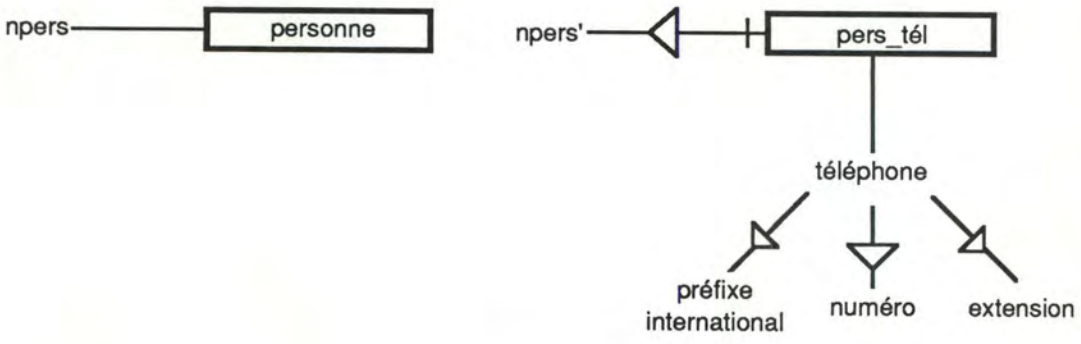
fig. 4.23 : insertion, rotation et aplatissement total
dans le cas d'item non identifiant



DEVIENT



DEVIENT



DEVIENT

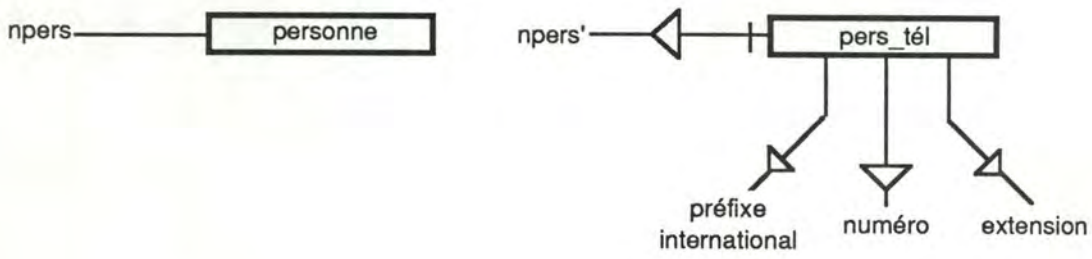


fig. 4.24 : insertion, rotation et aplatissement total
dans le cas d'un item identifiant

Règle 1

Certaines règles définies auparavant sont une fois de plus combinées.

Selon la forme syntaxique initiale, on applique une des règles définies pour la transformation 4 (insertion d'un type d'articles). A la forme ainsi obtenue, on applique une des règles de la transformation 3 (rotation). La forme définitive est obtenue par l'application d'une des règles de la transformation 1 (aplatissement).

Exemples

Ces exemples sont basés sur la figure figure 4.23.

1. L'exemple du point précédent peut être repris.

```
for {cte = cote, crs = cours} := (etud1).cc
...
endfor ;
```

DEVIENT

```
for   ecc := etud_cc (: nE' = (etud1).nE)
      cte := (ecc).cote;
      crs := (ecc).cours;
      ...
endfor;
```

L'explication est similaire à celle donnée lors de l'exemple précédent.

2. Toujours sur le schéma de la figure 4.23,

```
modify etud1 (: CC + {cours = 'conception-bd', cote = 18} );
```

DEVIENT

```
create ec := etud-cc ( (: nE' = (etud1).nE) and (: cours = 'conception-bd')
                       and (: cote = 18) );
```

On veut ajouter une valeur décomposable à l'ensemble de celles prises par un item

répétitif décomposable. Il faut créer un article intermédiaire qui rende compte de cette valeur, et l'associer aux valeurs d'items appropriées.

8. ELIMINATION D'UN COMPOSANT ROLE DANS UN IDENTIFIANT

Cette transformation se ramène à la transformation 3 (élimination d'un type de chemins par duplication d'item). Elle est utilisée pour enlever un composant rôle d'un identifiant, cette structure n'étant pas permise en SQL. Cette structure ne l'est pas non plus en Cobol, mais le cas échéant, elle est éliminée par une autre transformation. Le nouvel identifiant est composé de l'item duplicata utilisé pour l'élimination du type de chemins, et du "reste" de l'identifiant initial.

Exemple

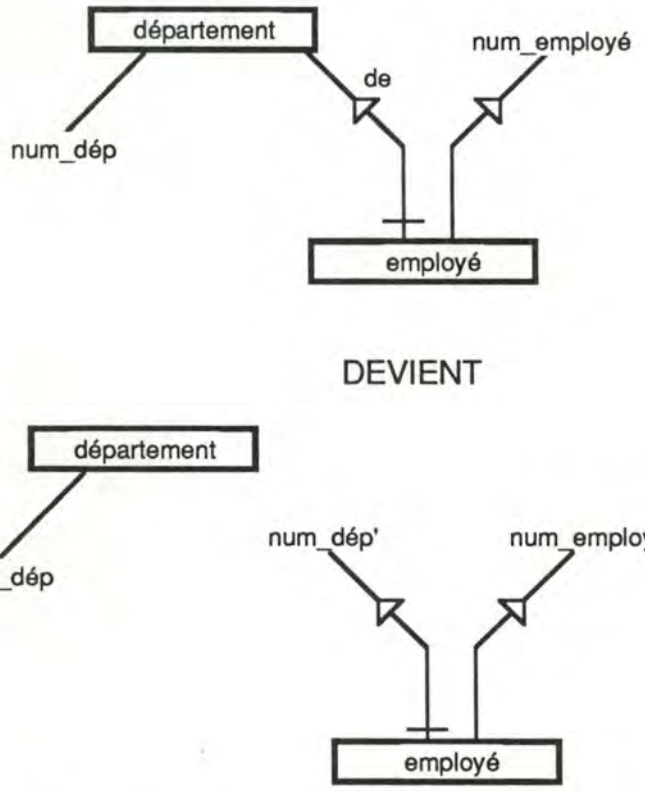


fig. 4.25 : élimination d'un composant rôle dans un identifiant

Règle 1

Les règles de la transformation 3 (rotation) sont appliquées ici selon les mêmes modalités.

Exemples

Ces exemples sont basés sur la figure 4.25.

1. emp := employe ((de : depart1) and (: num_employe = max_num));

DEVIENT

```
emp := employe ((: num_dep' = (depart1).num_dep) and (: num_employe = max_num));
```

L'identifiant de la forme initiale a été défini comme clé dans un chemin. Cet accès par clé dans un chemin est transformé en accès par clé suite à l'élimination du composant rôle ("de"). La nouvelle clé est définie sur l'item duplicata ("num_dep'") et sur le "reste" de l'identifiant initial (un seul item "num_employe").

2. delete depart1;

DEVIENT

```
for  e := employe (: num_dep' = (depart1).num_dep)
     delete e
enfor;
delete depart1;
```

La destruction des cibles obligatoires d'un chemin 1-N dont on détruit l'origine est gérée implicitement par LDA/MAG. Lorsque le type de chemins est éliminé, il faut gérer cette destruction explicitement par programme d'application. Le cas présent, on accède aux employés du département "depart1" (par le biais de leur numéro de département) et on les détruit.

Si on examine les algorithmes transformés des deux exemples ci-dessus, on remarque que le type d'articles "employe" a un item clé d'accès qui est composant d'une autre clé d'accès. "num_dep" est en effet clé d'accès (exemple 2) et composant d'une autre clé d'accès avec "num_employe" (exemple 1).

9. ELIMINATION D'UN TYPE DE CHEMINS PAR DUPLICATION D'ITEM ET AJOUT D'UN NIVEAU DE DECOMPOSITION

Cette transformation est utilisée si le SGD cible est Cobol. Elle sert à éliminer une clé d'accès dans un chemin 1-N. Si la clé est identifiante (toujours dans le type de chemins), on peut formuler la transformation autrement en disant qu'elle sert à éliminer un composant rôle dans un identifiant.

Cette transformation est définie comme la transformation 3 (rotation) suivie de la transformation 2 (ajout d'un niveau de décomposition). La nouvelle clé obtenue est un item composé de la clé initiale et de l'item duplicata qui a servi à éliminer le type de chemins.

Exemples

Dans le cas d'un item clé dans un chemin, la forme initiale de la figure 4.6 exposée lors de la transformation 3 (rotation) peut être reprise.

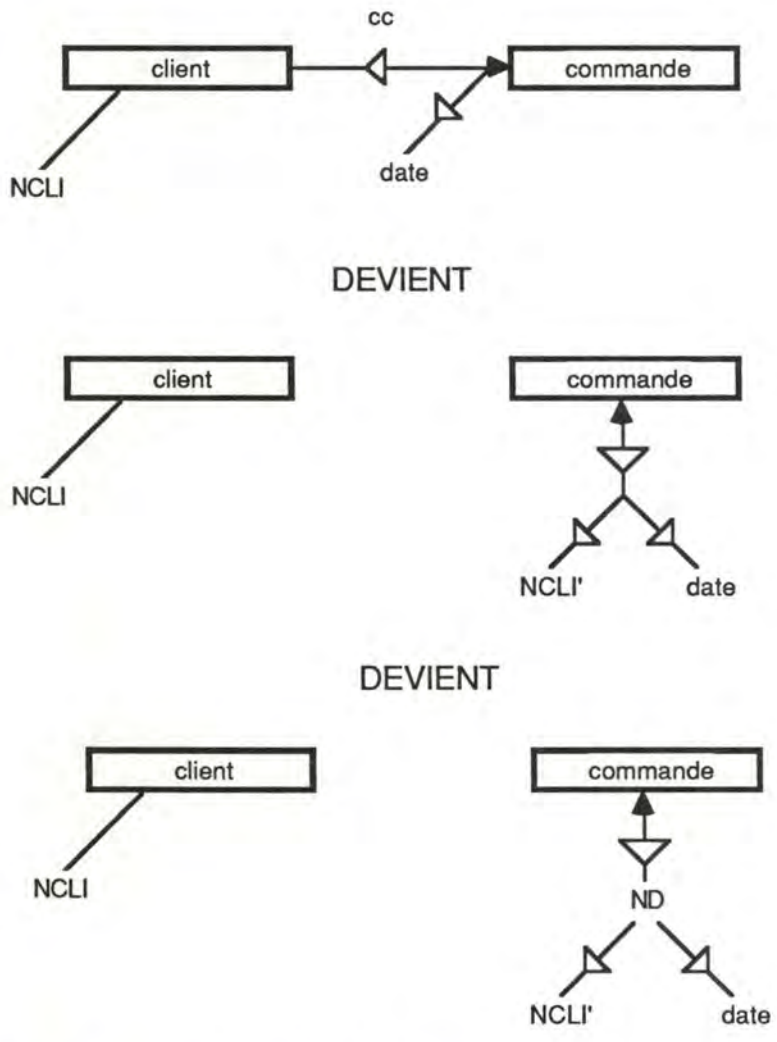
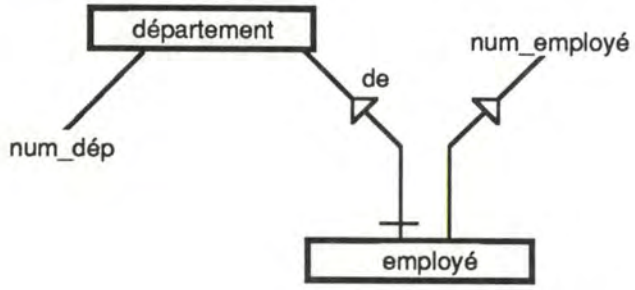


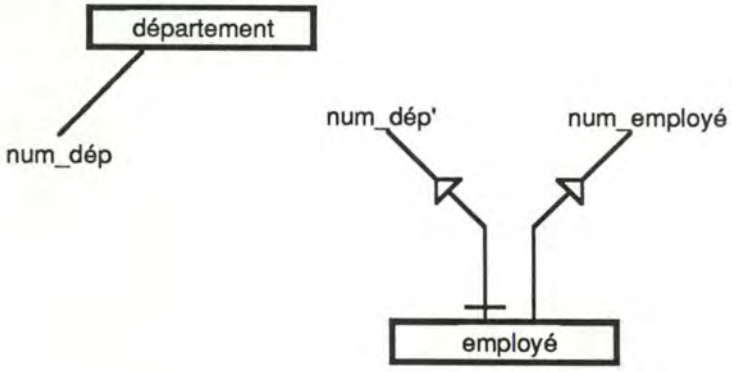
fig. 4.26 : élimination d'une clé non identifiante dans un chemin

L'exemple est analogue si le type de chemins est récursif.

La forme initiale de la figure 4.25 peut être reprise pour un identifiant dont un composant est un chemin.



DEVIENT



DEVIENT

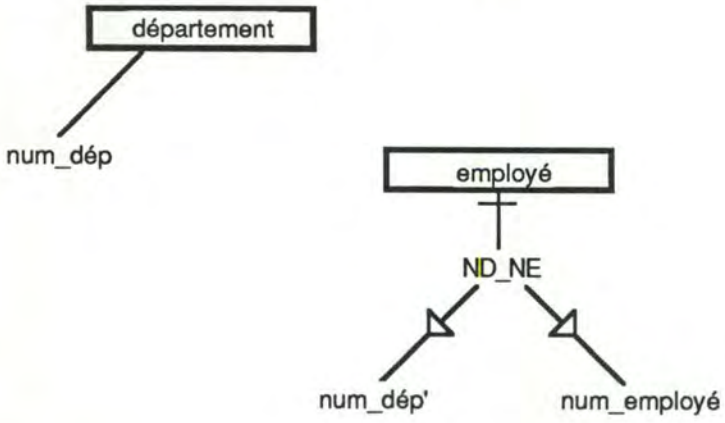


fig. 4.27 : élimination d'un composant rôle dans un identifiant et ajout d'un niveau de décomposition

Règle 1

Selon la forme syntaxique initiale, des règles de la transformation 3 (rotation) ou 8 (élimination d'un composant rôle dans un identifiant) (elles sont de toute façon équivalentes) sont appliquées. Le résultat crée une clé (identifiante ou pas) composée de plusieurs items (deux le cas présent), ce qui est interdit en Cobol. Les règles de la transformation 2 sont donc appliquées pour que l'ajout d'un père commun aux composants de cette clé soit répercuté dans la forme syntaxique.

Exemples

Ces exemples sont basés sur la figure 4.26.

1. for com := commande ((cc : cli) and (: date = dernier_jour))

DEVIENT

for com := commande (: ND ((: NCLI' = (cli).NCLI) and (: date = dernier_jour)))

L'accès par clé dans un chemin devient après rotation du type de chemins, un accès par clé. La clé est un item composé de la clé initiale "date" et de l'item duplicata "NCLI'".

2. for com := commande (cc : cli)

DEVIENT

for com := commande (: ND (: NCLI' = (cli).NCLI))

Un accès par chemin devient un accès par clé sur base de l'item duplicata "NCLI'", après élimination du type de chemins.

Dans le cas d'un identifiant dont un composant est un chemin, les exemples développés à la transformation 8 (élimination d'un composant rôle dans un identifiant) peuvent être repris. Les explications ne le sont pas dans la mesure où elles sont équivalentes; le lecteur peut donc s'y référer.

Les exemples suivants sont basés sur la figure 4.27.

3. emp := employe ((de : depart1) and (: num_employe = max_num));

DEVIENT

```
emp := employe (: ND_NE (  (: num_dep' = (depart1).num_dep ) and
                           (: num_employe = max_num ) ) );
```

4. delete depart1;

DEVIENT

```
for  e := employe (: ND_NE (: num_dep' = (depart1).num_dep ) )
     delete e
endfor;
delete depart1;
```

10. AJOUT D'UN COMPOSANT ROLE A UN IDENTIFIANT

Il ne peut y avoir en Codasyl, par type d'articles, plus d'un identifiant composé uniquement d'items. Cette transformation est utilisée pour garantir cela.

Elle consiste à transformer un identifiant composé d'items simples (un ou plusieurs) en un identifiant composé de ces items et d'un nouveau type de chemins. Le type de chemins est évidemment obligatoire pour le type d'articles identifié. L'autre membre est le type d'articles système. Le classe fonctionnelle de ce type de chemins est 1-N de système vers le type d'articles identifié.

Exemple



DEVIENT

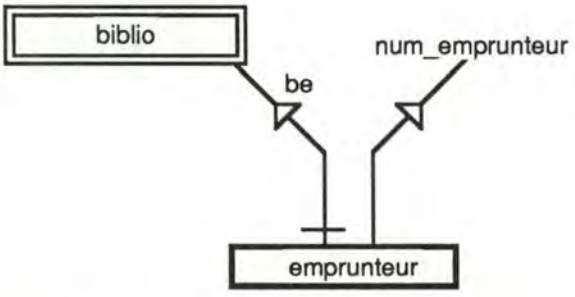


fig. 4.28 : ajout d'un composant rôle à un identifiant

Règle 1

Cette transformation n'a d'influence sur les algorithmes que dans le cas suivant :

create <varref> := <tart> <condition sur item>

où <varref> ::= le nom d'une variable de référence.
<tart> ::= le nom du type d'articles de <varref>.
<condition sur item> ::= une condition portant sur un ou plusieurs items et telle que définie dans la syntaxe des conditions de création. Ces items ou cet item sont (est) l'identifiant à transformer. Cette condition peut aussi être vide.

DEVIENT

create <varref> := <tart> ((<chemin> : <système>) and <condition sur item>)

où <chemin> ::= le nom du chemin reliant l'article système à l'article désigné par <varref> et nouveau composant de l'identifiant.

<système> ::= le nom de la variable de référence de l'article système. C'est donc aussi le nom de la base de données.

Lors d'une création, il faut rattacher l'article identifié à l'article système, et lui associer la (les) valeur d'item qui forme le reste de l'identifiant. Avant transformation, seule cette (ces) valeur est nécessaire.

Exemple

Cet exemple est basé sur la figure 4.28.

create emp := emprunteur (: num_emprunteur = 250);

DEVIENT

create emp := emprunteur ((be : biblio) and (: num_emprunteur = 250));

La création d'un article "emprunteur" s'accompagne de son rattachement à l'article système. La variable de référence de l'article système porte le nom de cet article qui est aussi le nom de la base de données : "biblio".

11. TRANSFORMATION D'UN ACCES PAR CLE EN UN ACCES PAR CHEMIN

Cette transformation consiste à enlever la qualité de clé d'accès à un (groupe d') item(s). Cette qualité est rapportée sur un (groupe d') item(s) duplicata d'un type d'articles intermédiaire créé. Un type de chemins permet l'accès du type d'articles intermédiaire vers le type d'articles cible initial.

Cette transformation est utilisée pour limiter le nombre de clé d'accès par type d'articles en Codasyl.

Exemple

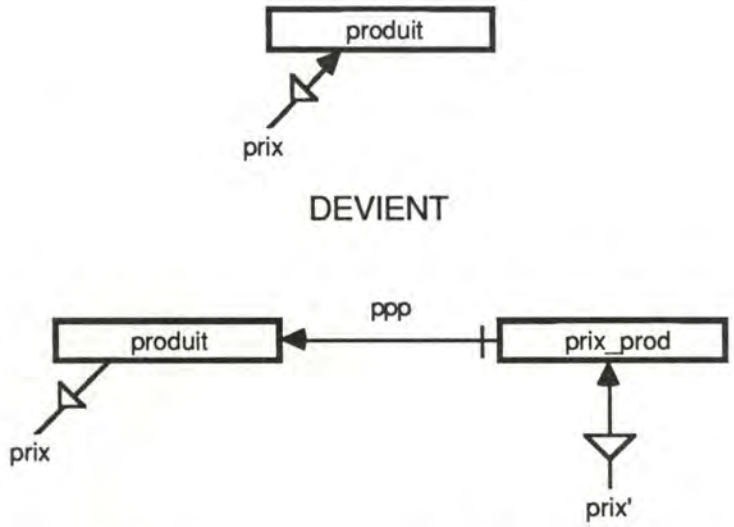


fig. 4.29 : transformation d'un accès par clé en un accès par chemin

"prix'" est un duplicata de "prix". Il est clé d'accès à "prix_prod". "prix_prod" est le type d'articles intermédiaire créé. "ppp" est le type de chemins qui permet l'accès de "prix_prod" à "produit". "prix" est resté item de "produit", mais n'est plus clé d'accès.

Règle 1

[for] <varref> := <tart> <condition sur item>

où <varref> ::= le nom d'une variable de référence
<tart> ::= le nom du type d'articles de <varref>
<condition sur item> ::= une condition portant sur un ou plusieurs items, telle que définie dans la syntaxe des conditions d'accès par clé. Ces items sont la clé à éliminer.

DEVIENT

for <varref-int> := <tart-int> <condition sur item'>
<varref> := <tart> (<chemin> : <varref-int>) ;
...
endfor;

où <condition sur item'> ::= <condition sur item> où les noms des items originaux ont été remplacés par ceux de leur duplicata.
<varref-int> ::= le nom d'une variable de référence du type d'articles intermédiaire.
<tart-int> ::= le nom du type d'articles intermédiaire.
<chemin> ::= le nom du chemin liant <varref-int> à <varref>.

Dans le cas du "for", tout "next <varref>" (ou exit <varref>) apparaissant dans le corps de la boucle est transformé en "next <varref–int>" (ou exit <varref–int>).

Dans le cas d'une assignation, l'utilisateur veut obtenir au plus un article. Dès lors, il suffit d'ajouter "exit <varref–int>" juste après l'assignation de <varref>. Le corps de la boucle est ainsi exécuté au plus une fois.

Un accès par clé devient un accès par clé à l'article intermédiaire. On accède ensuite de l'article intermédiaire à l'article cible de l'accès initial par le chemin reliant ces articles. La transformation pourrait donc être nommée de façon plus précise "transformation d'un accès par clé en un accès par clé suivi d'un accès par chemin".

Il faut de plus que dans le cas du "for", les instructions "next" et "exit" qui apparaissent dans le corps de la boucle, et spécifient la variable de parcours initiale, spécifient, après transformation, la nouvelle variable de parcours (<varref–int>).

Exemple

Cet exemple est basé sur la figure 4.29.

```
for p := produit (: prix > max_prix )
```

DEVIENT

```
for   pr_prod := prix_prod (: prix' > max_prix )
      p := produit ( ppp : pr_prod );
      ...
```

L'accès par clé devient un accès par clé suivi d'un accès par chemin.

Règle 2

create <varref> := <tart> <condition sur item>;

où <varref> ::= le nom d'une variable de référence.
<tart> ::= le nom du type d'articles de <varref>.
<condition sur item> ::= une condition portant sur un ou plusieurs items, telle que définie dans la syntaxe des conditions de création. Ces items sont la clé à éliminer. Cette condition peut aussi être vide.

DEVIENT

create <varref> := <tart> <condition sur item>;
create <varref-int> := <tart-int> ((<chemin> : <varref>) and <condition sur item'>);

où <condition sur item'> ::= <condition sur item> où les noms des items originaux ont été remplacés par ceux de leurs duplicatas.
<varref-int> ::= le nom d'une variable de référence du type d'articles intermédiaire.
<tart-int> ::= le nom du type d'articles intermédiaire.
<chemin> ::= le nom du chemin liant <varref> à <varref-int>.

Après avoir créé l'article initial, on crée l'article intermédiaire. On rattache ce dernier à l'article initial et on l'associe à ses valeurs d'items qui forment la nouvelle clé. Les valeurs d'items dans les deux créations sont identiques pour conserver l'identité qu'il doit y avoir entre les valeurs d'item de l'item (ou du groupe d'items) original et celles du duplicata.

exemple

Cet exemple est basé sur la figure 4.29.

create p2 := produit (: prix = prix_cat1);

DEVIENT

create P2 := produit (: prix = prix_cat1);
create pr_pro := prix_prod ((ppp : p2) and (: prix' = prix_cat1));

La création initiale se double de la création de l'article intermédiaire "pr_pro". Cet article est relié à l'article initial "p2". Les deux articles sont associés à des valeurs d'items (prix et prix') equivalentes "prix_cat1".

Règle 3

modify <varref> <condition sur item>;

où <varref> ::= le nom d'une variable de référence.
<condition sur item> ::= une condition portant sur un ou plusieurs items, telle que définie dans la syntaxe des conditions de modification. Ces items sont la clé à éliminer.

DEVIENT

<varref-int> := <tart-int> (<chemin> : <varref>);
modify <varref-int> <condition sur item'>;
modify <varref> <condition sur item>;

où <condition sur item'> ::= <condition sur item> où les noms des items originaux ont été remplacés par ceux de leurs duplicatas.
<varref-int> ::= le nom d'une variable de référence du type d'articles intermédiaire.
<tart-int> ::= le nom du type d'articles intermédiaire.
<chemin> ::= le nom du chemin liant <varref> à <varref-int>.

Les valeurs d'items de l'item (ou du groupe d'items) original et celles du duplicata doivent toujours être identiques. Dès lors, toute modification apportées aux valeurs originales doit être répercutée sur le duplicata. Pour ce faire, il faut évidemment accéder à l'article intermédiaire relié à l'article dont on modifie les valeurs d'items.

Exemple

Cet exemple est basé sur la figure 4.29.

```
modify p (: prix = 100 );
```

DEVIENT

```
p_pro := prix_prod ( ppp : p );
modify p_pro (: prix' = 100 );
modify p (: prix = 100 );
```

Si on modifie les valeurs d'items originales, il faut modifier aussi les duplicatas afin de conserver l'équivalence.

Réflexions

Si la clé est identifiante

Si la clé à éliminer est identifiante, cette transformation-ci n'est pas utilisée. En effet, la clé initiale perd sa propriété mais le groupe d'items identifiant reste associé au type d'articles. Or, ce groupe est identifiant et le SGD cible est Codasyl (pour lequel tout identifiant doit être clé d'accès). Le groupe d'items doit donc recouvrer sa propriété de clé d'accès et la transformation n'a servi à rien.

Si la clé est constituée d'au moins un item répétitif

Si la clé est constituée d'au moins un item répétitif, cette transformation ne peut être utilisée. En dupliquant la clé originale et en la reportant sur un nouveau type d'articles, la transformation conserve en effet une clé d'accès composée d'un item répétitif. L'inconformité par rapport à Codasyl n'est donc pas résolue.

Le type de chemins est 1-1

Si on examine la schéma de la figure 4.29, on constate que le type de chemins "ppp" est 1-1. Il faudra le transformer en 1-N (avec N du coté "prix_prod" étant donné que c'est à ce type d'articles qu'une contrainte d'existence est imposée).

Ne serait-il pas plus simple d'opter pour la transformation suivante ?

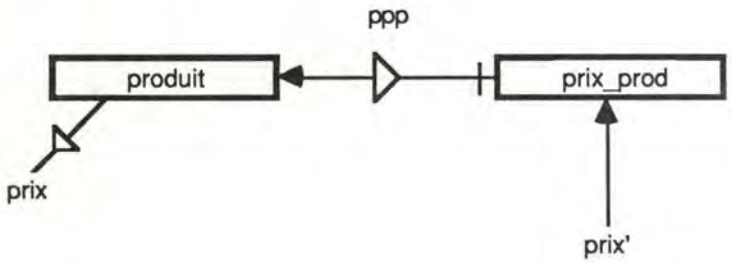


fig. 4.30 : alternative à la transformation 4.29

où "prix" est devenu identifiant et le caractère non identifiant de la clé d'accès initiale (prix) est traduit par la classe fonctionnelle 1-N (de "prix_prod" vers "produit") du type de chemins "ppp". La contrainte d'existence imposée à "prix_prod" en ce qui concerne "ppp" est alors à gérer par programme d'application.

Dans la première solution, "prix_prod" représente les valeurs d'item "prix". Dans cette seconde solution, il représente toutes les valeurs d'items "prix" qui sont différentes. Il faudra donc créer moins d'articles intermédiaires. Cet argument favorise donc la seconde solution.

La première est cependant retenue dans la mesure où elle permet une transformation beaucoup plus aisée de la forme syntaxique exprimant la création d'un article "produit".

En effet, si on se réfère à la règle de transformation syntaxique (règle 2 : création), on remarque que la création d'un article "produit" se double de la création d'un article intermédiaire et de son rattachement à l'article initial. Cette transformation n'est donc pas excessivement compliquée.

Dans le second cas, la création de l'article initial "produit" est similaire. Après cette création, il faut vérifier s'il existe déjà un article intermédiaire "prix_prod" dont la valeur d'item "prix'" est la même que la valeur d'item "prix" de l'article "produit" créé. Si oui, on rattache l'article intermédiaire trouvé et l'article créé. Si non, on crée l'article intermédiaire avec une valeur d'item "prix'" égale à la valeur d'item "prix" de l'article à peine créé, et on rattache les deux articles.

Il est donc préférable de retenir la première solution.

4.3. Des primitives autorisées par le SGD cible : une seconde étape de transformation

4.3.0. Introduction

Le paragraphe précédent a exposé les règles qui permettent d'obtenir des algorithmes travaillant sur un schéma conforme. Suite à l'application de ces règles, les algorithmes respectent donc le premier point de la définition de conformité.

L'étape suivante de la démarche de transformation des algorithmes effectifs conformes à LDA/MAG en algorithmes conformes à un SGD cible fait l'objet du présent paragraphe. Elle consiste à transformer les algorithmes obtenus de la première étape pour que les expressions de désignation et de modification de données qu'ils spécifient, soient totalement évaluables et exécutables par les primitives du SGD cible (Cfr. chapitre 2 pour le détail de ces primitives).

Comme précédemment, seul un sous–ensemble du langage LDA est susceptible d'être transformé. Ce sous–ensemble est orienté vers la manipulation d'objets de la base de données. Une définition en est donnée au paragraphe 4.1.

Le présent paragraphe décrit les transformations à effectuer selon le SGD cible (pour rappel : Cobol, Codasyl, SQL). Les règles de transformation sont exposées selon le formalisme utilisé au paragraphe 4.2.

Avant de développer ce point, il est toutefois important de signaler qu'en termes de désignation et de manipulation de données, aucun des SGD cibles ne reconnaît les notions de variable de référence et de variable d'item. On suppose donc que les algorithmes ne "dialoguent" pas directement avec le SGD, mais bien par l'intermédiaire d'un module chargé de la gestion de ces variables de référence et d'item.

4.3.1. Deuxième étape de la transformation des algorithmes

1. LE SGD CIBLE EST CODASYL

Pour l'accès par clé sur base d'un item clé élémentaire, LDA/MAG reconnaît l'utilisation des opérateurs de comparaison suivants : =, <, >, <=, >=. Codasyl ne permet que l'égalité (=). Les accès par clé sur base des autres opérateurs doivent donc être transformés.

Règle 1

```
for  <varref> := <tart> ( : <item> <opc> <valeur> )
     <trt>
endfor ;
```

où
- <varref> ::= le nom d'une variable de référence
- <tart> ::= le nom du type d'articles de <varref>
- <item> ::= le nom d'un item simple élémentaire, clé d'accès. Il importe peu que cet item soit identifiant ou pas.
- <opc> ::= un opérateur de comparaison parmi <, >, <=, >=
- <valeur> ::= toute valeur élémentaire pouvant apparaître dans un programme LDA
- <trt> ::= le corps de la boucle

DEVIENT

```
for  <varref> := <tart> ( )
     if  ( <varref> ) . <item> <opc> <valeur>
         then <trt>
     endif
endfor ;
```

Codasyl n'admet pas d'autres opérateurs de comparaison que l'égalité. Dès lors, l'accès par clé initial devient un accès séquentiel. Pour chacun des articles obtenus dans cet accès séquentiel, on vérifie s'il est associé à une valeur d'item <item> respectant la condition posée initialement dans l'accès par clé. Si c'est le cas, le corps de la boucle initiale est exécuté. On exécute donc <trt> pour tous les articles <tart> associés à une valeur d'item <item> vérifiant la relation établie par <opc> avec <valeur>; cela correspond bien à l'accès par clé initial.

Exemple

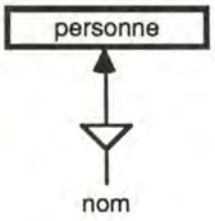


fig. 4.31 : exemple de clé d'accès.

Soit un accès par clé basé sur la figure 4.31 ci–dessus.

```
for  p := personne ( : nom > 'Durant' )
     <trt>
endfor ;
```

Cet accès n'est pas reconnu par Codasyl dans la mesure où l'opérateur de comparaison est >. Il est dès lors transformé en accès séquentiel aux articles "personne" avec un test des valeurs d'item "nom" qui leur sont associées.

```
for p := personne ( )
     if ( p ) . nom > 'Durant'
        then <trt>
     endif
endfor ;
```

Règle 2

<varref> := <tart> (: <item> <opc> <valeur>)

où <varref> ::= le nom d'une variable de référence
<tart> ::= le nom du type d'articles de <varref>
<item> ::= le nom d'un item simple élémentaire, clé d'accès. Il importe peu que cet item soit identifiant ou pas.
<opc> ::= un opérateur de comparaison parmi <, >, <=, >=
<valeur> ::= toute valeur élémentaire pouvant apparaître dans un programme LDA

DEVIENT

```
for  <varref> := <tart> ( )
     if  ( <varref> ) . <item> <opc> <valeur>
         then exit <varref>
     endif
endfor ;
```

L'accès par clé initial devient un accès séquentiel avec test des valeurs d'item. Dans la mesure où il est suffisant de trouver un article vérifiant la condition initiale (<item> <opc> <valeur>), dès qu'on en a trouvé un, le "for" est terminé (exit <varref>).

Exemple

Cet exemple est basé sur la figure 4.31.

```
pers := personne ( : nom < 'Dupont' ) ;
```

DEVIENT

```
for  pers := personne ( )
     if ( pers ) . nom < 'Dupont'
         then exit pers
     endif
endfor ;
```

On accède séquentiellement aux articles "personne". Dès qu'on en trouve un associé à une valeur d'item "nom" inférieure à 'Dupont', on force la terminaison du "for" par l'instruction "exit pers".

LE SGD CIBLE EST SQL

Cette seconde étape de transformation des algorithmes est inutile si le SGD cible est SQL. Les algorithmes obtenus de la première étape ne contiennent que des expressions de désignation et de modification de données totalement évaluables et exécutables par SQL.

LE SGD CIBLE EST COBOL

Cette seconde étape de transformation des algorithmes est inutile si le SGD cible est COBOL. Les algorithmes obtenus de la première étape ne contiennent que des expressions de désignation et de modification de données totalement évaluables et exécutables par COBOL.

4.3.2. Conclusion

Suite à l'application de cette seconde étape, on obtient donc des algorithmes travaillant sur des données conformes, et n'utilisant que des primitives permises par le SGD cible.

Contrairement aux transformations de la première étape (§4.2), celles-ci se préoccupent du SGD cible. Leur principe d'utilisation est que, pour un algorithme et un SGD cible donnés, il faut enlever de l'algorithme toute expression de désignation et de modification de données qui n'est pas évaluable et exécutable par les primitives du SGD cible.

On constate d'autre part que les transformations à effectuer ne concernent que les accès par clé en Codasyl. La plus grande partie du travail a été effectuée lors de la première étape.

4.4 Les formes syntaxiques non concernées par les transformations

4.4.0 Introduction

Certaines formes syntaxiques doivent être transformées afin de rendre un algorithme effectif conforme LDA/MAG, conforme à un SGD cible choisi. Le paragraphe 4.1 les a décrites explicitement. D'autres formes ne sont cependant pas concernées par les restrictions posées par des SGD commerciaux cibles et ne sont donc pas soumises à transformation.

Les SGD cibles n'exprimant d'exigences qu'envers des formes syntaxiques en relation avec une base de données, la partie du langage LDA qui offre la possibilité d'une programmation "classique" ne s'expose à aucune transformation. D'autre part, la partie du langage orientée vers la manipulation d'objets de la base de données, n'est pas non plus totalement sujette à transformation.

Notre propos n'est pas de décrire exhaustivement cette partie du langage LDA non affectée par les transformations mais de procéder à quelques remarques.

4.4.1 Exposé de formes syntaxiques

I. Les variables de type group définies dans un algorithme et en relation (par ex. dans une assignation) avec des données de la base, ne voient pas leur structure transformée alors que celle des données de la base l'a été.

Considérons l'exemple suivant :

```
type  domicile =    group
                        rue : string(10);
                        numero : integer;
                        localite : string(10);
                    end;

      acheteur =    group
                        adr : domicile;
                        numcommande : integer;
                    end;

var x : acheteur;
...
x.adr.rue := (cli).adresse.rue;
x.adr.numero := (cli).adresse.numero;
x.adr.localite := (cli).adresse.localite;
...
```

Si l'item décomposable "adresse" est aplati, l'algorithme n'aura pour seule partie transformée que

```
x.adr.rue := (cli).rue;
x.adr.numero := (cli).numero;
x.adr.localite := (cli).localite;
```

En aucun cas, le type "acheteur" n'est redéfini comme :

```
acheteur =    group
                    rue : string(10);
                    numero : integer;
                    localite : string(10);
                    numcommande : integer;
              end;
```

tandis que le type "domicile" est supprimé.

La variable x n'est donc pas aplatie bien qu'elle soit en relation avec l'item "adresse". Elle a en effet reçu du programmeur une structure particulière pertinente pour ce dernier. Aplatir cette variable alors qu'elle n'est en rien concernée par les exigences imposées par le SGD cible, revient à contester sans raison un choix de structure posé par le programmeur. Il est donc préférable de s'en abstenir.

II. La partie du langage LDA plus orientée vers la manipulation d'objets de la base de données, n'est pas totalement soumise aux restrictions qu'imposent certains SGD commerciaux par rapport à ce qu'autorise le SGD virtuel LDA/MAG. En effet, les variables d'item ne sont pas influencées par les changements de structure des items dont elles peuvent contenir les valeurs.

On peut en effet voir les variables d'item comme un moyen offert au programmeur de déclarer rapidement des variables structurées de même structure que celle des types d'articles. Ces variables sont avant tout destinées à conserver des valeurs d'items indépendamment de la modification de ces valeurs dans la base de données. Cette utilisation des variables d'item n'est cependant pas exhaustive : une variable d'item peut contenir toute valeur qu'il plaît au programmeur de lui affecter. Une variable d'item est donc semblable à une variable group dont la déclaration est simplement plus aisée.

Dès lors, à nouveau, répercuter sur une variable d'item une transformation de structure des items dont la variable peut contenir les valeurs, revient à contester sans raison un choix de structure posé par le programmeur. Les variables d'item ne sont donc pas sujettes à transformation.

4.5. Optimisation

Les paragraphes 4.2 et 4.3 ont exposé des règles de transformation afin d'obtenir à partir d'un algorithme effectif conforme LDA/MAG, un algorithme conforme à un SGD cible choisi.

Ces règles revêtent un caractère systématique en ce sens qu'elles n'ont pour seule préoccupation que de s'appliquer à de nombreuses formes syntaxiques en les transformant "correctement" (la forme finale se doit de réaliser les mêmes spécifications que la forme originale).

Il apparaît cependant que la transformation correcte de formes syntaxiques peut s'enrichir du souci supplémentaire de produire des formes syntaxiques efficaces. L'efficacité permettrait d'obtenir un algorithme final performant en termes d'accès à la base de données.

Ce paragraphe a pour but de présenter quelques perspectives d'optimisation de transformation. Une étude approfondie du problème constitue une extension éventuelle de ce mémoire.

Considérant la transformation de schéma de la figure 4.3,

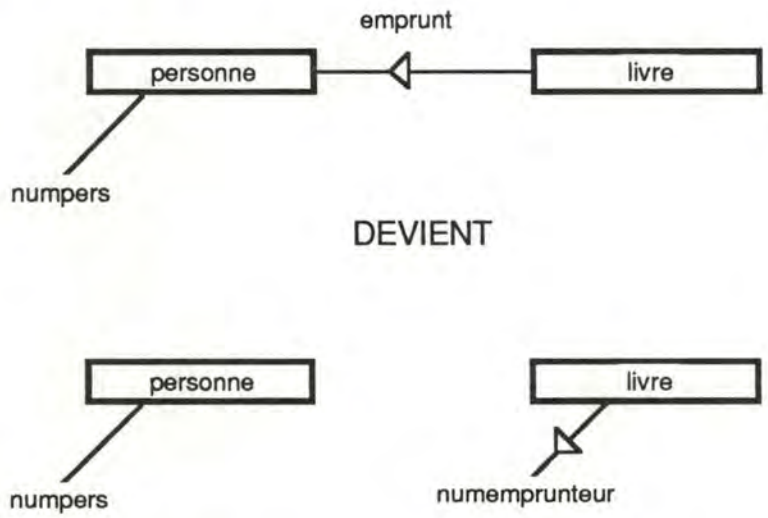


figure 4.32 : élimination d'un type de chemins 1-N
par duplication d'item

l'algorithme initial

```
pers := personne (: numpers = 100);
for  l := livre ( emprunt : pers )
     <traitement>
endfor;
```

qui a pour but de traiter les livres empruntés par la personne de numéro 100, est transformé selon les règles du paragraphe 4.2 (transformation 3 : élimination d'un type de chemins par duplication d'item, règle 1) en

```
pers := personne (: numpers = 100);
for  l := livre (: numemprunteur = (pers).numpers)
     <traitement>
endfor;
```

qui réalise les mêmes spécifications que l'algorithme original. On peut constater cependant que

```
pers := personne (: numpers = 100);
for  l := livre (: numemprunteur = 100)
     <traitement>
endfor;
```

économise l'extraction d'une valeur d'item.

Les instructions create l := livre (emprunt : pers) et create p := personne (emprunt : l) nécessitent un accès à la base de données. Sous leurs formes transformées (paragraphe 4.2, transformation 3 : élimination d'un type de chemins par duplication d'item, règle 4), elles en nécessitent deux :

```
create l := livre ();
modify l (: numemprunteur = (pers).numpers);

create p := personne ();
modify l (: numemprunteur = (p).numpers);
```

Une analyse plus fine remettant en cause le caractère systématique de la transformation, permettrait dans un des cas d'économiser un accès. En effet, si la rotation s'est faite autour du type d'articles dont on crée un article, la création de l'article et la modification de la valeur d'item duplicata peuvent se résumer à une seule opération.

Dès lors, dans le cas où on crée un article "livre", l'expression transformée devient create l := livre (: numemprunteur = (pers).numpers); et ne nécessite qu'un accès à la base de données.

Si l'on considère la transformation d'un item identifiant répétitif clé d'accès,

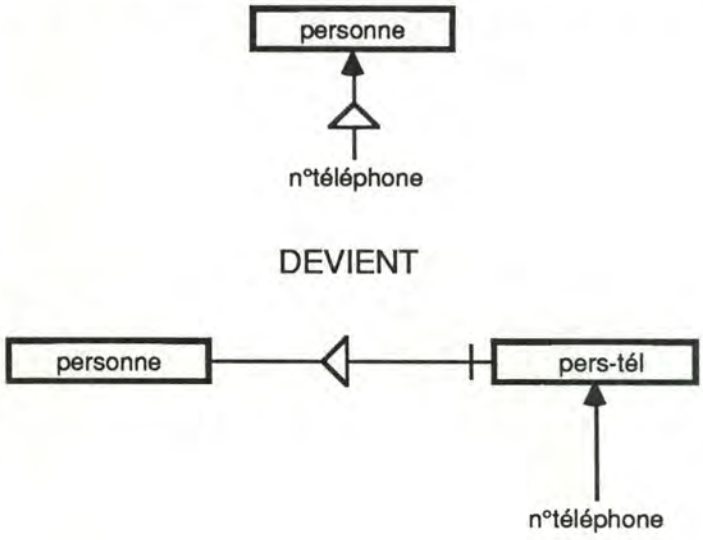


figure 4.33 : transformation d'un item répétitif identifiant clé d'accès en type d'articles

le paragraphe 4.2 (transformation 4 : insertion d'un type d'articles, règle 6a2) a défini la transformation de la forme syntaxique

```
modify pers (: n°telephone - {'02/3332211'});

comme

for  tel := pers-tel ( ch : pers )
     if  (tel).n°telephone = '02/3332211'
         then  delete tel;
               exit tel
     endif
endfor;
```

Les formes originale et finale enlèvent toutes deux '02/3332211' de l'ensemble des numéros de téléphone de la personne désignée par "pers". La forme

```
tel := pers_tel (: n°telephone = '02/3332211');
delete tel;
```

qui peut elle aussi constituer la forme finale, économise cependant bon nombre d'accès à la base de données par rapport à la première forme finale définie. Cette première forme est plus coûteuse dans la mesure où elle ne profite pas du fait que "n°telephone" est clé d'accès.

Si l'optimisation d'une transformation semble donc attrayante, elle n'en est pas pour autant aisée à réaliser. Dans le premier exemple présenté ici, l'optimisation nécessite la détection de la forme pers := personne (: numpers = 100), la mémorisation de la valeur de clé et la reconnaissance dans la suite de l'algorithme de la volonté d'accéder à partir de la personne "pers" aux livres qu'elle a empruntés. La règle du paragraphe 4.2 ne se soucie, elle, que de ce dernier point.

Il ne semble pas raisonnable d'optimiser la transformation d'un algorithme après que celui-ci ait été modifié de façon à le rendre conforme à un SGD commercial cible. Deux raisons étayent cette affirmation.

D'une part, l'optimisation requiert, comme la transformation systématique, une analyse sémantique de l'algorithme initial afin de détecter les formes syntaxiques à transformer. Séparer transformation et optimisation revient à effectuer cette analyse deux fois. Il serait dès lors plus performant de transformer directement de façon efficace.

Ensuite, l'optimisation de la transformation d'une forme syntaxique se révèle parfois être tout à fait différente de la transformation systématique. Il ne s'agit plus alors d'affiner la transformation systématique mais de la remettre en cause totalement : les formes syntaxiques, l'une transformée systématiquement, l'autre de façon efficace, diffèrent de façon majeure. Le troisième exemple d'optimisation illustre parfaitement cela.

4.6. Problèmes liés à la transformation des algorithmes

4.6.0. Introduction

Le but des règles de transformation syntaxique définies au paragraphe 4.2 est d'obtenir des algorithmes conformes à un SGD cible choisi à partir d'algorithmes effectifs conformes à LDA/MAG. Et, si les algorithmes à transformer et ceux transformés "font la même chose", ils ne le font cependant pas de la même manière. Ils sont, et c'est logique, différents. Cette différence peut entraîner des problèmes au niveau de l'exploitation de la base de données, de la gestion de sa cohérence, et de la survenance d'incidents.

Le présent paragraphe a pour objet d'exposer ces problèmes.

Une base de données dans un état cohérent vérifie les contraintes qui ont été définies au niveau conceptuel.

Lors des phases de conceptions logique et physique, les notions de répétitivité d'item, de contrainte d'existence et de classe fonctionnelle participent à la définition de la cohérence de la base.

Lorsque ces notions sont définies sur le schéma des accès nécessaires, elles sont gérées par le SGD LDA/MAG. Cependant, la production d'un schéma conforme à un SGD commercial cible fait en sorte parfois que LDA/MAG ne peut continuer à gérer ces notions. Les algorithmes conformes au SGD cible (c–à–d transformés) doivent alors assurer le respect des notions que LDA/MAG ne peut plus prendre en charge.

Ce paragraphe se propose de présenter les problèmes de gestion de la cohérence de la base de données qui sont à charge des algorithmes transformés.

Ce paragraphe aborde en outre la notion de transaction. Cette notion s'avère utile lors de la survenance d'incidents ou d'erreurs d'exécution dans les algorithmes.

4.6.1. La gestion de la cohérence de la base de données

1. REPETITIVITE

Le chapitre 1 signale que le SGD LDA/MAG reconnaît les notions de répétitivité fixe, de répétitivités variables limitée et illimitée.

Cependant, la transformation d'un item répétitif en type d'articles fait disparaître en même temps que l'item répétitif la possibilité pour LDA/MAG de gérer le caractère fixe ou limité du nombre de valeurs de cet item qui peuvent être associées à un article. LDA/MAG ne peut en effet gérer un nombre fixe ou limité d'articles cibles d'un type de chemins (la répétitivité variable illimitée n'exige aucune gestion particulière). Les algorithmes conformes au SGD cible se chargent dès lors de cette gestion.

L'exemple formel suivant sert de support à l'exposé.

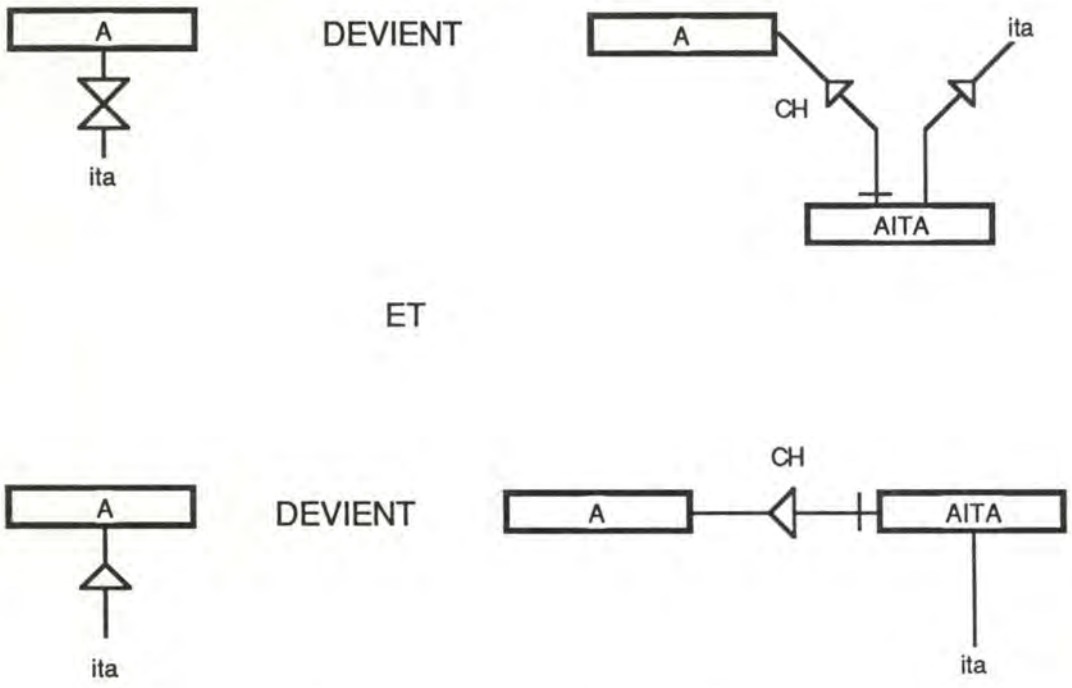


fig. 4.34 : élimination d'un item répétitif
par insertion d'un type d'articles

a) ita est un item à répétitivité fixe

Les instructions susceptibles de mettre en péril le caractère fixe de la répétitivité sont create et modify. L'analyse sémantique des algorithmes à transformer peut garantir le respect du caractère fixe de la répétitivité. En effet, on peut détecter dans une instruction create ou modify le nombre de valeurs d'item (pour l'item répétitif) associées à l'article créé ou modifié (cfr chapitre 3 : nombre de <exp> dans une <liste_exp_simple> si l'item est élémentaire; nombre d'<élément> dans une <liste_exp_décomposable> si l'item est décomposable). Ce nombre doit être égal à celui qui définit la répétitivité fixe de l'item. Si cela est garanti, la forme syntaxique transformée (transformation 4 : insertion d'un type d'articles, règles 2b et 6b) crée autant d'articles intermédiaires représentant chacun une valeur de l'item répétitif, que de valeurs détectées dans la forme originale. Le caractère fixe de la répétitivité est ainsi garanti.

b) ita est un item à répétitivité variable limitée

L'analyse sémantique des algorithmes à transformer peut vérifier dans une instruction create ou modify [create a := A (...(: ita = {v1,...,vn})...) ou modify a (...(: ita = {v1,...,vn})...)] que l'on n'associe pas à l'article une liste de valeurs d'item (cfr chapitre 3 : <exp> dans une <liste_exp_simple> si l'item est élémentaire; <élément> dans une <liste_exp_décomposable> si l'item est décomposable) contenant plus de valeurs que la limite de la répétitivité. Si cela est garanti, la forme syntaxique transformée crée autant d'articles intermédiaires représentant chacun une valeur d'item répétitif, que de valeurs contenues dans la liste. La limite de la répétitivité ne peut donc être dépassée.

L'instruction modify qui retire une valeur d'item répétitif [modify a (: ita - {x})] ne pose pas de problème.

Considérant l'instruction modify qui ajoute une valeur d'item répétitif [modify a (: ita + {x})], il faut vérifier que l'exécution de cette instruction sous sa forme transformée n'enfreint pas la contrainte de répétitivité maximale.

A cette fin, un item à répétitivité variable limitée doit se transformer en type d'articles moyennant l'ajout d'un item compteur au type d'articles qui initialement était associé à l'item répétitif. Cet item compteur (soit cpt) qui rend compte du nombre de valeurs de l'item répétitif transformé, est géré dans les algorithmes transformés.

Create a := A (...(: ita = {v1,...,vn})...) et modify a (...(: ita = {v1,...,vn})...) se transforment (transformation 4 : insertion d'un type d'articles, règles 2b et 6b) en une série de créations d'articles intermédiaires. Il suffit donc de mettre l'item compteur à 0 avant la première création et de l'augmenter de 1 après chaque création.

Ainsi, si l'on désigne par <expression> la façon dont se transforme une instruction modify a (: ita + {x}), l'algorithme transformé

```
if   (a).cpt < <répétitivité maximale>
     then  <expression>;
           <incrémentation de cpt d'une unité>
     else  <déclarer une erreur>
```

assure le respect de la contrainte de répétitivité maximale.

Afin de tenir à jour l'item compteur, toute instruction modify a (: ita - {x}) (sous forme transformée après insertion du type d'articles) doit être suivie de la décrémentation d'une unité de l'item compteur cpt.

2. LES CONTRAINTES D'EXISTENCE

a) les items obligatoires

Une valeur d'item obligatoire est toujours différente de NULL.

Soit un item obligatoire transformé en type d'articles (cfr figure 4.34). Le type de chemins (CH) reliant le type d'articles (A) associé initialement à l'item et le type d'articles intermédiaire créé (AITA) est obligatoire pour les deux types d'articles (A et AITA). D'autre part, l'item simple du type d'articles créé représentant l'item répétitif, est obligatoire.

LDA/MAG ne gère cependant pas deux contraintes d'existence portant sur les types d'articles reliés par un type de chemins. Le respect de la contrainte d'existence portant sur A incombe donc aux algorithmes transformés.

Les algorithmes transformés créent systématiquement des articles intermédiaires représentant les valeurs de l'item répétitif initial, assurant ainsi que l'article est bien associé à au moins une valeur pour cet item. La contrainte d'existence portant sur l'item simple du type d'articles intermédiaire garantit qu'aucun article de ce type ne se voit associer une valeur d'item égale à NULL. Le respect du caractère obligatoire de l'item répétitif initial est ainsi assuré.

D'autre part, il faut veiller à ce que la forme transformée (soit <expression>) de l'instruction "modify a(: ita - {x})" laisse au moins une valeur d'item répétitif initial associée à l'article (l'item est en effet obligatoire). Pour cela, il faut tester avant <expression> s'il y a encore plus d'un article du type d'articles intermédiaire relié à l'article a. Si la condition est respectée, <expression> peut être exécutée; sinon une erreur est signalée :

```
i := 0;
for aita := AITA ( CH : a )
    i := i + 1;
    if i > 1 then exit aita
    endif
endfor;
if i > 1 then <expression>
        else <déclarer une erreur>
endif;
```

Le raisonnement est similaire si les types de chemins sont éliminés.

b) type de chemins N–N obligatoire

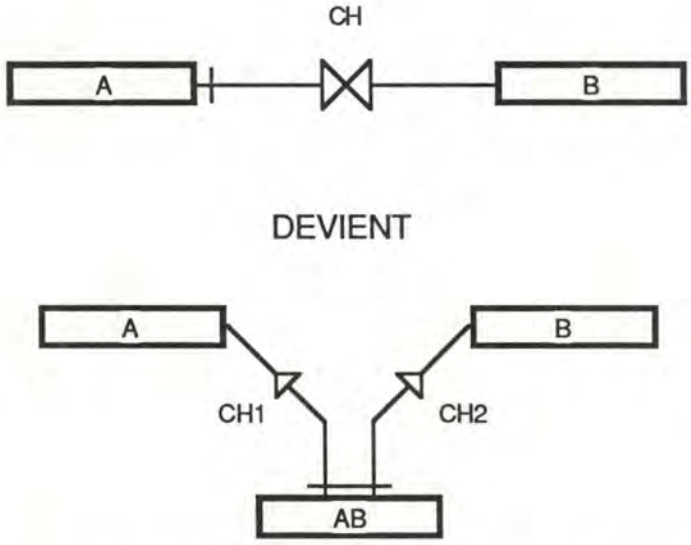


Fig. 4.35 : élimination d'un type de chemins N–N
par insertion d'un type d'articles

La contrainte d'existence initiale se transforme en contrainte d'existence sur A dans le type de chemins CH1. Or, CH1 est aussi obligatoire pour le type d'articles AB. CH1 est donc obligatoire pour les deux membres.

LDA/MAG gère la contrainte d'existence apparaissant dans le schéma initial. Il n'en va pas de même dans le schéma transformé dans la mesure où LDA/MAG ne gère pas deux contraintes d'existence dans un type de chemins.

L'algorithme transformé gère alors la contrainte selon laquelle chaque article A doit être relié à au moins un article AB.

Ainsi, il faut vérifier qu'une instruction "create a ..." spécifie dans les conditions de création au moins une condition du type "(CH : b)". Cela se fait lors de l'analyse sémantique de l'algorithme à transformer. L'algorithme transformé reliera alors a à b en créant un article intermédiaire. La contrainte d'existence est dès lors vérifiée.

Une instruction du type "modify a (CH : b)" ne nécessite aucune vérification. Par contre, "modify a (CH : 0 b)" nécessite un traitement semblable à celui défini pour "modify a (: ita -

{x})" lorsque l'item répétitif est obligatoire : il faut tester avant l'exécution de l'instruction transformée s'il y a encore plus d'un article intermédiaire relié à l'article a.

La destruction des cibles obligatoires du chemin N–N lorsqu'on détruit un article origine pose également un problème. En effet, considérons un article A relié par CH à un seul article B. Si cet article B est détruit, LDA/MAG détruit l'article A qui enfreint la contrainte d'existence selon laquelle tout article A doit être relié à au moins un B. Dans le schéma transformé, LDA/MAG est dans l'impossiblité d'assurer ce traitement : la contrainte d'existence initiale portant maintenant sur A dans CH1 constitue une seconde contrainte d'existence dans ce type de chemins.

La destruction d'un article B doit donc être transformée de façon telle qu'elle est précédée d'instructions vérifiant qu'aucun article A n'est relié à ce seul B. Si ce n'est pas le cas, l'article (ou les articles) A en infraction est (sont) détruit(s) également.

Le raisonnement est similaire si les types de chemins sont éliminés.

c) type de chemins 1–N obligatoire

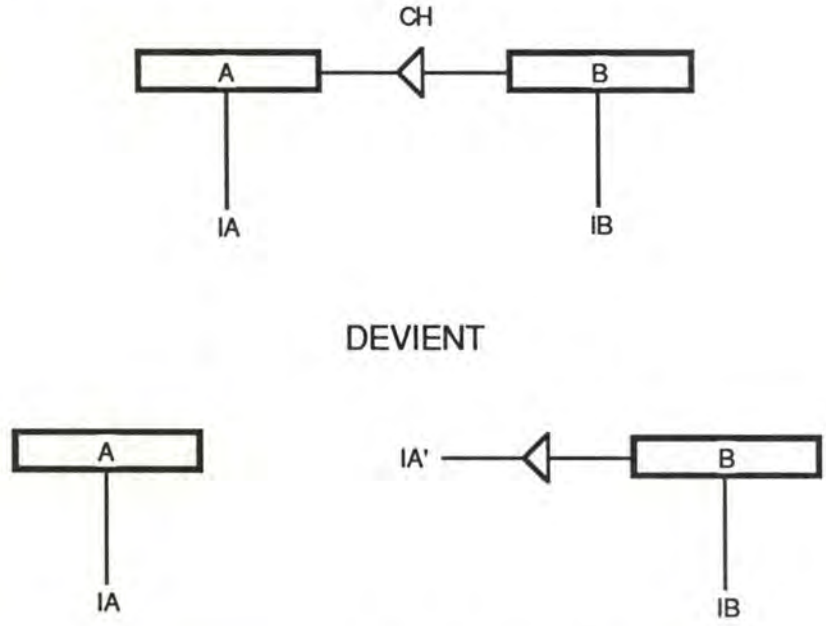


fig. 4.36 : élimination d'un type de chemins 1–N par rotation

Dans le schéma initial, LDA/MAG gère les éventuelles contraintes d'existence portant sur les types d'articles A et B reliés par le type de chemins CH.

De par l'élimination du type de chemins, le respect de ces contraintes ne peut plus être assuré par le SGD. Les algorithmes transformés doivent dès lors faire respecter eux–mêmes ces contraintes.

I. Soit une contrainte d'existence portant sur le type d'articles A. Cette contrainte est gérée de la même façon que dans un type de chemins N–N (Cfr point b).

II. Soit une contrainte d'existence portant sur le type d'articles B.

Dans une instruction de création d'article B apparaissant dans l'algorithme à transformer,

il faut vérifier qu'il existe une et une seule condition du type "(CH : a)". L'algorithme transformé établit alors par des valeurs d'items (ia et ia') une liaison entre l'article désigné par la variable de référence a et l'article créé. Ceci assure le respect de la contrainte d'existence qui veut que tout article B soit relié à au moins un article A. Le respect de la classe fonctionnelle du type de chemins qui veut que tout article B soit relié à au plus un article A, est garanti aussi de cette façon.

LDA/MAG pourrait assurer le respect de la contrainte d'existence lors de l'exécution de l'algorithme transformé. La contrainte d'existence dans le schéma initial se traduit par le caractère obligatoire de l'item IA' dans le schéma final. Dès lors, si un article B n'est pas relié à un article A dans l'algorithme à transformer, cet article B est associé à une valeur d'item égale à NULL lors de l'exécution de l'algorithme transformé. LDA/MAG détecte et refuse cette association. Cette solution offre cependant le désavantage de retarder le moment de détection d'une erreur.

Les instructions du type "modify b (ch : a)" et "modify b (ch : 0 a)" ne peuvent évidemment pas apparaître dans l'algorithme à transformer dans la mesure où chaque article B doit être relié à un et un seul A.

III. Un type de chemins N–1 relie B à A. Lorsqu'on détruit un article origine d'un chemin de ce type, la destruction des cibles obligatoires exige un traitement particulier.

Soit la figure 4.36 où le type de chemins CH serait obligatoire pour le type d'articles A. La destruction d'un article B origine d'un chemin CH entraîne la destruction de l'article A qui lui est associé si cet article A n'est pas relié à d'autres B via CH. Aucun A ne peut en effet exister sans être relié à au moins un B.

LDA/MAG gère le respect de cette contrainte d'existence tant que le type de chemins CH existe. Dès que CH est éliminé, les algorithmes LDA doivent être transformés de sorte à ce qu'ils gèrent cette contrainte eux– mêmes.

La destruction d'un article B (sous sa forme transformée) doit donc être précédée d'instructions vérifiant qu'un article A n'est pas relié au seul B détruit. Si c'est le cas, l'article A en infraction doit être détruit.

d) type de chemins 1–1 obligatoire

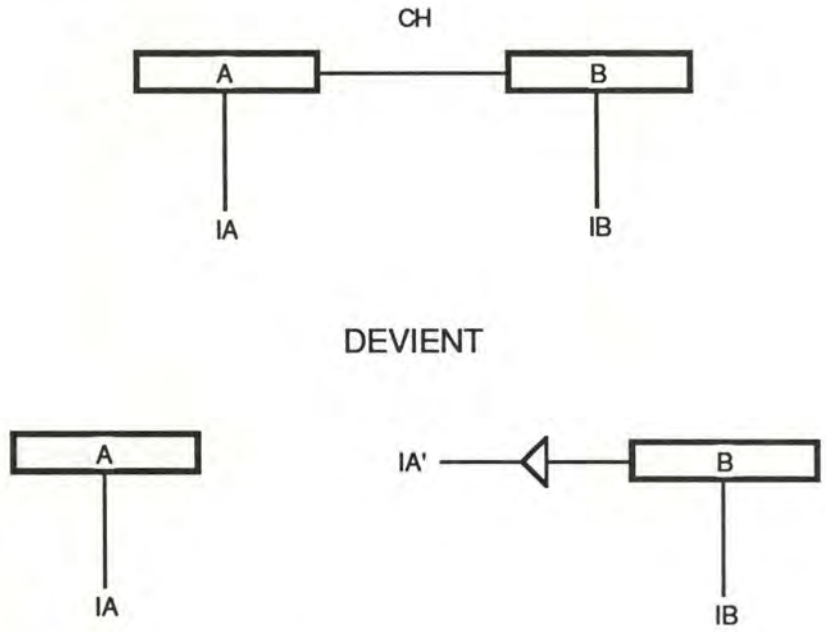


fig. 4.37 : élimination d'un type de chemins 1–1 par rotation

Le raisonnement est identique à celui tenu au point c lorsque la contrainte d'existence porte sur le type d'articles B.

3. LES CLASSES FONCTIONNELLES

a) type de chemins de classe fonctionnelle N–N (Cfr. figure 4.35)

Ces types de chemins ne requièrent aucune gestion particulière.

b) type de chemins de classe fonctionnelle 1–N (Cfr. figure 4.36)

Lors de l'exécution des instructions "create a := A (CH :b)", "modify a (CH :b)" et "modify b (CH :a)", LDA/MAG vérifie que l'article b n'est pas déjà relié à un article A. Si c'est le cas, il y a erreur d'exécution.

Après élimination du type de chemins 1–N par duplication d'item, LDA/ MAG n'est plus en mesure d'effectuer cette vérification. Celle–ci est alors assurée à nouveau par les algorithmes transformés.

Si l'article b n'est pas rattaché à un article A, cela se traduit dans le schéma transformé par le fait qu'il a NULL comme valeur d'item IA'. Les instructions "create a := A (CH :b)", "modify a (CH :b)" et "modify b (CH :a)" peuvent donc être transformées en

```
[ create a := A (); ]
if (b) . IA' = NULL then modify b (: IA' = (a). IA )
                    else <déclarer une erreur>
```

D'autre part, il faut vérifier lors de l'analyse sémantique de l'algorithme à transformer

que dans les instructions du type "create b := B (...)", il existe au plus une condition portant sur le type de chemins CH ((CH:a)).

Les instructions du type "modify a (CH : 0 b)" et "modify b(CH : 0 a)" ne posent aucun problème.

c) type de chemins de classe fonctionnelle 1–1 (Cfr. figure 4.37)

Soit l'instruction "create a := A (CH :b)". Si LDA/MAG peut assurer le respect de la classe fonctionnelle 1–1 de CH dans l'algorithme à transformer, il n'en est plus de même dans l'algorithme transformé après élimination du type de chemins par duplication d'item.

La gestion de la cohérence de la base de données est alors assurée par le même traitement que celui défini au point b. Ce traitement s'ajoute à la vérification, lors de l'analyse sémantique de l'algorithme à transformer, de l'existence d'au plus une condition du type "(CH:b)" dans l'instruction "create a := A (CH:b)". L'explication de ces traitements est identique à celle exposée au point b.

Le point b est d'application ici dans la mesure où la classe fonctionnelle 1–1 n'est qu'un cas particulier de la classe fonctionnelle 1–N.

Une instruction du type "create b := B (...)" exige un traitement particulier. Il faut d'abord vérifier lors de l'analyse sémantique de l'algorithme à transformer qu'au plus une condition du type "(CH:a)" apparaît dans la condition de création de b. Il est de plus nécessaire de vérifier, à l'exécution, que l'article a n'est relié à aucun article B. Pour ce faire, l'algorithme, sous sa forme tranformée, doit vérifier que la valeur d'item IA de cet article est différente des valeurs d'item IA' des articles B. L'instruction de création exposée ci–dessus devient donc :

```
for  b1 := B ( : IA' = ( a ) . IA )
     <déclarer une erreur>
endfor ;
create b := B ();
modify b (: IA' = (a). IA ) ;
```

Si l'on rentre dans la boucle, l'article a est déjà relié à un article B. Aussi faut–il déclarer une erreur et arrêter là l'exécution de l'algorithme. Si l'on ne rentre pas dans la boucle, le respect de la classe fonctionnelle 1–1 se trouve garanti, et l'association de l'article a à l'article b peut avoir lieu.

Les instructions du type "modify a (CH:b)" et "modify b (CH:a)" exigent de vérifier que a n'est relié à aucun B et que b n'est relié à aucun A. les traitements définis pour les instructions "create b (...)" et "create a (...)" sont alors d'actualité ici aussi.

Les instructions du type "modify a (CH: 0 b)" et "modify b (CH: 0 a)" ne posent pas de problème.

4.6.2. La notion de transaction

Disposer de la notion de transaction s'avère utile lors de l'occurence d'incidents ou d'erreurs d'exécution dans les algorithmes.

Abordons d'abord la question des incidents.

Lorsqu'un article origine d'un chemin obligatoire pour les cibles est détruit, les articles cibles enfreignant la contrainte d'existence sont détruits par le SGD LDA/MAG. La transformation du schéma des accès nécessaires en schéma conforme à un SGD cible choisi a cependant requis que les algorithmes transformés assurent ce traitement lorsque LDA/MAG n'en est plus capable.

La solution qui a été exposée dans les pages précédentes n'est cependant pas exempte de problèmes. En effet, soit la figure 4.37 où le type de chemins "CH" est obligatoire pour le type d'articles "A".

L'instruction "delete b" est transformée selon le paragraphe 4.2 (transformation 3 : élimination d'un type de chemins par duplication d'item, règle 7) en :

```
for  a := A (: IA = (b) . IA' )
     delete a
endfor;
delete b;
```

On constate qu'après la destruction de l'article a et avant celle de b, la valeur d'item IA' de l'article b est encore la référence de l'article a avant destruction. La base de données est donc à ce moment dans un état incohérent résolu plus tard par l'instruction "delete b". L'occurence d'un incident avant l'instruction "delete b" aurait pour effet de laisser la base de données dans un état incohérent. Il serait dès lors utile que LDA/MAG dispose de la notion de transaction.

Une transaction est un ensemble d'instructions qui est exécuté totalement ou pas du tout, et qui, exécuté seul à partir d'un état cohérent de la base de données, laisse cette base de données dans un état cohérent.

Le fait de déclarer

```
for  a := A (: IA = (b) . IA' )
     delete a
endfor;
delete b;
```

comme une transaction assurerait la cohérence de la base de données de façon certaine. Un incident avant l'instruction "delete b" ramènerait la base de données dans l'état dans lequel elle se trouvait avant le début de la transaction, évitant ainsi le problème cité auparavant.

De façon plus générale, la notion de transaction permet de reporter le caractère atomique d'une instruction de mise à jour de la base de données au niveau d'un groupe d'instructions. Ceci est particulièrement utile lorsqu'une instruction est transformée en une suite d'instructions, comme c'est le cas ci–dessus.

De plus, cette caractéristique est utile, non seulement dans le contexte restreint des incidents, mais aussi pour le traitement des erreurs qui peuvent survenir lors de l'exécution des algorithmes.

Une erreur d'exécution est ici , hormis les incidents, tout ce qui fait qu'une opération sur la base de données n'a pas pu se dérouler parfaitement. Par exemple, la violation d'une contrainte (d'identifiant, de classe fonctionnelle, d'existence) lors de la création d'un article, une référence non valable (p. ex. dans "for a (CH : b)", b est une référence non valable si b vaut la référence nulle), ...

Une opération de mise à jour provoquant ce genre d'erreur ne peut évidemment être

effectuée. Si elle est transformée en une suite d'instructions, on court le risque qu'elle soit effectuée "en partie". Définir cette suite d'instructions comme une transaction est donc utile.

Un exemple basé sur la figure 4.34 illustre ce problème.

Soit le cas où "ita" est identifiant et où le programmeur veut créer un article A (qui violera cette contrainte) par l'instruction :

```
create a := A ( : ita = { v1, v2, v3 } ) ;
```

où v2 est la valeur litigieuse. Dans ce cas, la création ne se fait pas, et un code d'erreur permet au programmeur prévoyant de réagir en conséquence.

Toutefois, sous sa forme transformée l'instruction ci–dessus devient :

```
create a := A ( ) ;
create aita := AITA ( ( : ita = v1 ) and ( CH : a ) ) ;
create aita := AITA ( ( : ita = v2 ) and ( CH : a ) ) ;
create aita := AITA ( ( : ita = v3 ) and ( CH : a ) ) ;
```

LDA/MAG exécute les deux premières instructions sans problème. Il détecte une violation d'identifiant (ita) lors de la troisième. La création du second article AITA n'est donc pas effectuée, en accord avec le caractère atomique du "create", et un code d'erreur est renvoyé vers le programmeur.

Le résultat est donc que l'article "a" a été créé "sans être associé à toutes ses valeurs d'items", ce qui est inadmissible. Faire de la suite d'instructions ci–dessus une transaction résoudrait ce problème.

Les transactions ne peuvent toutefois résoudre tous les problèmes qui se posent pour gérer les erreurs d'exécution dans le cadre de la transformation d'algorithmes.

En effet, le programmeur est généralement averti d'une erreur par un code de retour (qui, remarquons–le, n'est pas prévu dans LDA mais pourrait l'être très facilement).

Or dans certains cas, ce code dans l'algorithme initial diffère de celui dans l'algorithme transformé.

Un exemple basé sur la figure 4.34 introduit ce problème.

Soit l'utilisateur veut enlever une valeur 'x' des valeurs d'items "ita" de l'article (de type A) désigné par la variable de référence "a". Peu importe que "ita" soit identifiant ou pas. L'instruction est donc :

```
modify a ( : ita - { 'x' } ) ;
```

Si "a" n'est associé à aucune valeur d'item ita égale à 'x', l'opération échoue, et l'utilisateur reçoit un code de retour signifiant quelque chose comme "valeur d'item inexistante".

La forme transformée de cette instruction est :

```
for  aita := AITA ( CH : a )
     if ( aita ) . ita = 'x'
       then exit aita
     endif
```

```
endfor;
delete aita ;
```

Si initialement, "a" n'était associé à aucune valeur d'item ita égale à 'x', le "for" ne rend aucun article AITA, son exécution se termine et aita acquiert une valeur indéterminée. La destruction échoue donc et le programmeur reçoit un code de retour signifiant "référence non valable". Ce code sera évidemment mal interprété.

On voit donc que l'équivalence entre les algorithmes transformés et ceux initiaux est remise en question.

Une solution envisageable pour supprimer cette divergence serait de détecter les cas où elle survient et de modifier en conséquence les règles de transformation d'algorithmes. Il faudrait une réassignation explicite du code de retour à sa valeur "pré–transformation". Une autre solution plus simple serait de rendre ce code binaire : "opération réussie" ou "opération echouée". A l'utilisateur de faire au mieux avec cette pauvre information.

Aucune solution simple et satisfaisante ne semble donc se dégager a priori. Cela montre à quel point l'équivalence des algorithmes est difficile à garantir.

4.6.3. Conclusion

On constate donc que la gestion de la cohérence de la base de données par programme d'application (quand LDA/MAG n'est plus capable de l'assurer) nécessite une étude importante. L'examen de ce problème se limite au présent paragraphe; sa prise en compte peut faire l'objet d'une extension future du mémoire.

D'autre part, la notion de transaction sert l'objectif d'équivalence entre les algorithmes initiaux et ceux transformés en garantissant l'atomicité des opérations de modification de la base de données. Toutefois, les structures de données manipulées dans les algorithmes initiaux diffèrent, par définition, de celles manipulées dans les algorithmes transformés. Dès lors, LDA/MAG peut parfois réagir différemment selon qu'on se trouve dans un cas ou dans l'autre. L'équivalence totale n'est donc pas garantie. L'étude approfondie de ce problème et de la notion de transaction n'entrent pas non plus dans le contexte du présent mémoire.

On peut conclure de cet exposé que certains problèmes survenant lors de la conception d'une application sur base de données méritent plus d'attention qu'on ne leur en accorde généralement. La gestion au niveau des programmes d'application des problèmes qui sont résolus habituellement par le SGD est souvent négligée. On y fait fréquemment référence par des phrases telles que : "il suffit de gérer cette contrainte au niveau du programme d'application", ce qui reflète bien peu les difficultés qu'entraîne cette gestion.

4.7. Conclusion

Ce chapitre a présenté les résultats de l'étude qui a été menée de façon à déterminer des règles de transformation d'un algorithme effectif conforme à LDA/MAG en un algorithme conforme à un SGD cible.

Après avoir circonscrit la partie du langage LDA influencée par les restrictions qu'émettent certains SDG cibles, le paragraphe 4.2. a exposé un premier ensemble de règles de transformation.

Ce paragraphe a montré une série de changements locaux de structure de données. Chaque changement est défini afin d'éliminer un aspect du schéma des accès nécessaires, non conforme à un SGD cible. Chaque changement de structure de données s'accompagne alors d'un ensemble de règles de transformation des formes syntaxiques susceptibles d'être affectées par la modification de schéma. Les règles ainsi définies n'ont pour seul but que d'adapter un algorithme de façon à ce qu'il opère sur un schéma de données transformé. Au terme de ce paragraphe, on dispose des règles qui, suite à la transformation du schéma des accès nécessaires en schéma conforme au SGD cible, sont appliquées à un algorithme de façon à ce qu'il travaille sur le schéma conforme.

Pour être conforme à un SGD cible, un algorithme se doit aussi de n'utiliser que des primitives autorisées par ce SGD. Le paragraphe 4.3. a présenté, pour chaque SGD cible retenu, les règles de transformation des formes syntaxiques qui ne respectaient pas, de ce point de vue, les exigences du SGD considéré.

Sans faire l'objet d'un exposé exhaustif, les formes syntaxiques qui ne s'exposent à aucune restriction de la part des SGD cibles, ont été le lieu de quelques remarques. Les variables structurées et d'item en relation avec des données de la base, dont la structure a été modifiée, ne sont transformées en aucune façon.

Les règles de transformation d'algorithmes définies ont un caractère systématique afin de couvrir le plus de formes syntaxiques possible. Leur généralité nuit parfois aux cas particuliers auxquels elles s'appliquent. La définition d'une transformation particulière pour tel ou tel cas permettrait d'obtenir un algorithme transformé plus efficace que celui obtenu "systématiquement". Des optimisations semblent ainsi nécessaires et requerraient une étude approfondie dont le paragraphe 4.5. n'a posé que les jalons.

La transformation d'un schéma des accès nécessaires en schéma conforme à un SGD cible, fait en sorte parfois que LDA/MAG ne peut continuer à gérer tous les aspects liés à la cohérence de la base de données. Le respect de la répétitivité d'items, des contraintes d'existence, de la classe fonctionnelle de types de chemins doit être alors assuré par les algorithmes transformés. Le paragraphe 4.6 a présenté une étude du problème.

En outre, si un algorithme initial (effectif conforme LDA/MAG) connaît quelqu'incident ou erreur, l'algorithme transformé se doit d'avoir le même comportement en pareil cas. L'enrichissement de LDA/MAG par la notion de transaction se révèle alors être nécessaire. Ce point a constitué un autre propos du paragraphe 4.6.

Ainsi donc, ce chapitre a mis en évidence nombre de problèmes pour transformer un algorithme effectif conforme LDA/MAG en algorithme conforme à un SGD cible. La majorité de ces problèmes ont été etudiés de façon détaillée. L'étude de certains n'a reçu qu'une esquisse, et pourrait faire l'objet d'une extension du mémoire.

Une série de restrictions ont d'autre part été posées sur les arguments des appels de fonctions ou de procédures. Une autre extension du mémoire pourrait les lever. On a, en effet, dit que seuls les "objets ayant un type" pouvaient être les arguments d'un appel. Les valeurs

d'items répétitifs et/ou décomposables ont donc été réfusées. Il suffirait dès lors d'étendre la syntaxe de LDA pour qu'il permette la "déclaration de valeurs d'items" (par le biais d'une déclaration du genre <nom> : <item> of <type d'articles>). Par exemple, si le schéma de la base de données spécifie un type d'articles "personne" associé à un item décomposable "adresse", le programmeur pourrait déclarer : "adr : adresse of personne". Cette déclaration pourrait apparaître dans la déclaration d'un entête de procédure, permettant ainsi qu'un argument valeur d'item (lors d'un appel) ne soit pas élémentaire. Cette extension aurait évidemment pour conséquence de devoir définir des règles de transformation pour les déclarations d'entêtes de procédures ou de fonctions, et les appels correspondants.

On peut conclure que le nombre important de problèmes qui surgissent lors de la transformation d'un algorithme indique à quel point il est utile d'offrir au programmeur d'applications sur bases de données, une aide pour rendre un algorithme conforme à un SGD cible. Les chapitres suivants se consacrent à l'intégration d'un outil d'automatisation de transformation d'algorithmes dans un atelier logiciel de conception d'applications sur bases de données.

Seconde partie :

intégration du mémoire

à

l'atelier logiciel

Chapitre 5 :

L'atelier logiciel de conception d'applications sur bases de données

5.1. Objet de l'atelier

Un atelier logiciel de conception d'applications sur bases de données est actuellement en développement à l'Institut d'Informatique des Facultés Universitaires de Namur.

Cet atelier se donne pour objectif d'offrir une aide au concepteur d'une application sur base de données durant son cheminement dans la démarche décrite en introduction.

Codasyl 71/73, SQL/DS et Cobol ANSI-74 constituent les trois SGD pour lesquels l'atelier peut guider la conception de la base de données. Leur grande diffusion et le fait qu'ils sont souvent considérés comme concurrents ont présidé à leur choix.

Les outils logiciels proposés par l'atelier sont destinés à alléger tant le travail de définition d'un schéma de base de données que celui de rédaction des traitements qui doivent opérer sur cette base. La matière première qui s'offre à la manipulation des outils sont le schéma conceptuel de la base de données à concevoir (schéma entité-association) ainsi que les spécifications des traitements associés à cette base. L'utilisation des outils conduit à la production finale d'un schéma LDD (Langage de Description de Données) et de programmes Cobol. Les outils sont actuellement disponibles sur PC compatibles sous MS-DOS et Lattice-C.

5.2. Schéma de la base des spécifications de l'atelier logiciel

Pour faciliter la mise en œuvre des outils logiciels offerts par l'atelier, une base de spécifications a été développée. Le schéma de cette base permet de décrire en ses termes les composants d'un système d'informations. La base des spécifications de l'atelier organisée selon ce schéma enregistre la description des étapes qui marquent l'évolution du système d'informations.

Le schéma de la base des spécifications de l'atelier est délibérément incomplet sur un certain nombre de plans, soit parce que certains domaines n'ont reçu qu'une description fort générale ou partielle, soit parce que d'autres ont tout simplement été ignorés (suivi de projets,tests, installation, ...). Le schéma est donc sujet à une évolution ultérieure éventuelle.

Les propos qui suivent décrivent les objets du schéma qui ont un lien direct avec :
- la transformation du schéma des accès nécessaires en schéma conforme à un SGD cible.
- la transformation d'un algorithme effectif conforme LDA/MAG en algorithme conforme à un SGD cible.

Les autres objets non directement utiles dans ce cadre sont délibérément passés sous silence. Le lecteur intéressé peut trouver un exposé complet du schéma de la base des spécifications dans [Hainaut 86b].

La description des objets retenus dans le schéma des la base de spécifications est établie dans le formalisme du modèle entité–association. Les attributs des objets sont délibérément omis dans la mesure où l'exposé se veut succinct. Malgré cette restriction, la complexité du schéma conduit à une présentation progressive sous la forme de sous-schémas.

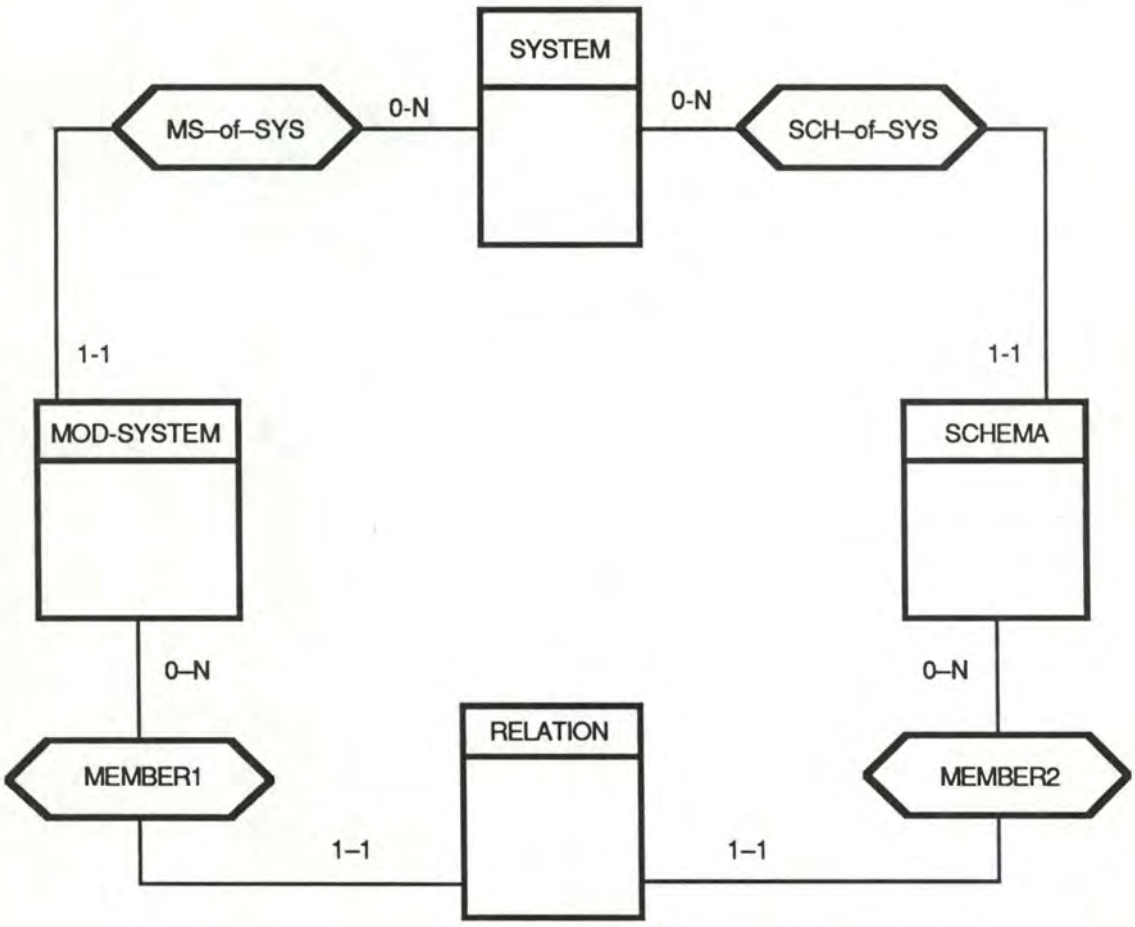


Fig. 5.1. : architecture générale d'un système d'information

Un système d'informations se décompose en deux sous-systèmes. Le premier est constitué de descriptions de traitements qui prennent la forme d'un système de modules (MOD-SYSTEM), et le second de descriptions des données (SCHEMA). Un MOD-SYSTEM d'un SYSTEM travaille sur 0,1 ou plusieurs SCHEMA appartenant à ce SYSTEM, cela est représenté par une RELATION entre le MOD–SYSTEM et le SCHEMA.

L'objet RELATION sert à représenter une association entre deux objets quelconques. Ces deux objets sont attachés à la RELATION par les types d'associations MEMBER1 et MEMBER2. L'association que cette RELATION représente ici entre le MOD–SYSTEM et le SCHEMA, est un exemple de son utilisation.

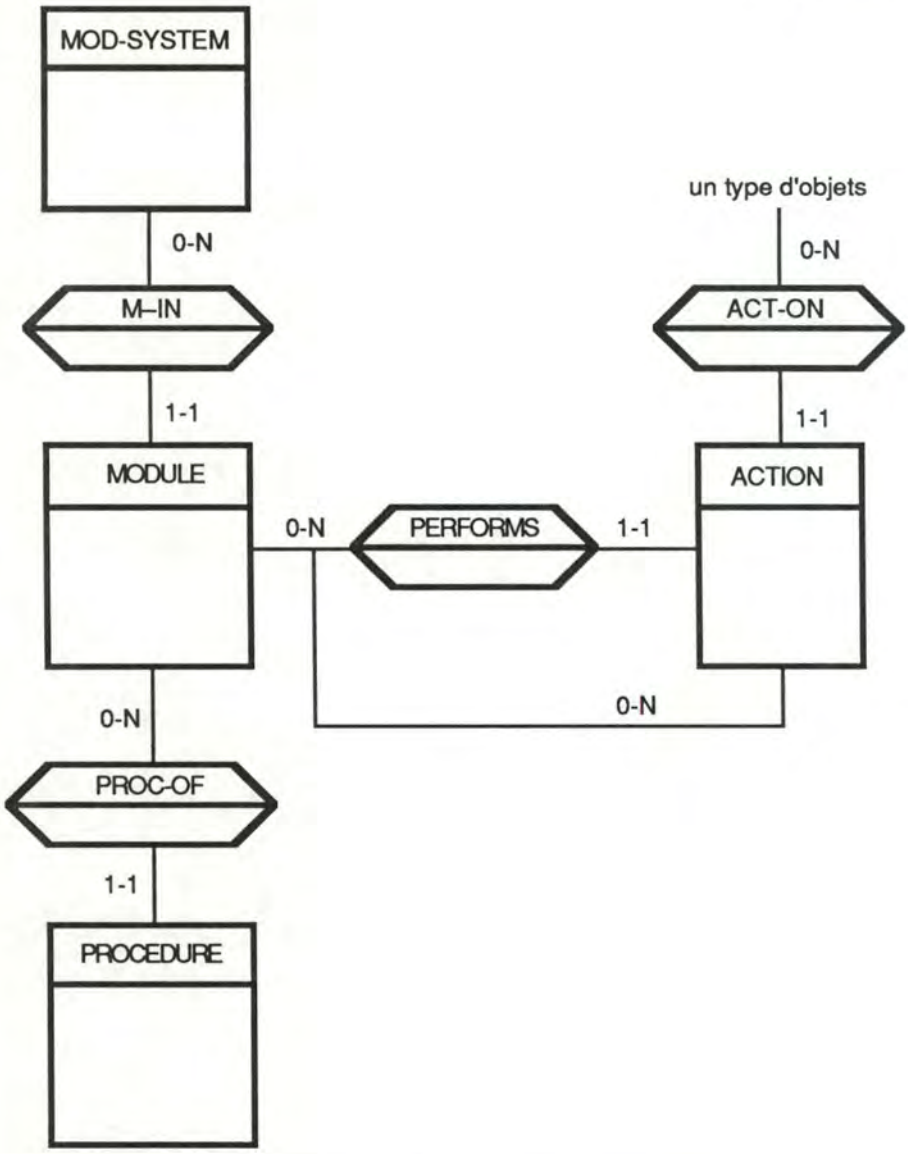


Fig 5.2. : Structure d'un système de modules

L'élément de base d'un système de modules (MOD-SYSTEM) est le MODULE. Tout MODULE appartient à un MOD-SYSTEM.

Le MODULE représente un certain état de perception ou de description d'une unité de traitement.

A un MODULE peuvent être associées des PROCEDURE.

Indépendamment du détail algorithmique éventuel d'un MODULE, on peut spécifier que celui-ci effectue certaines ACTION élémentaires sur des objets. On admettra qu'une ACTION est associée à un MODULE et à un objet. Si une ACTION apparaît comme trop complexe pour être décrite par la structure ci-dessus, on définit une arborescence d'ACTION en admettant qu'une ACTION peut en effectuer d'autres. Si une ACTION agit sur un objet d'un SCHEMA, le MOD-SYSTEM du MODULE de cette ACTION travaille sur ce SCHEMA.

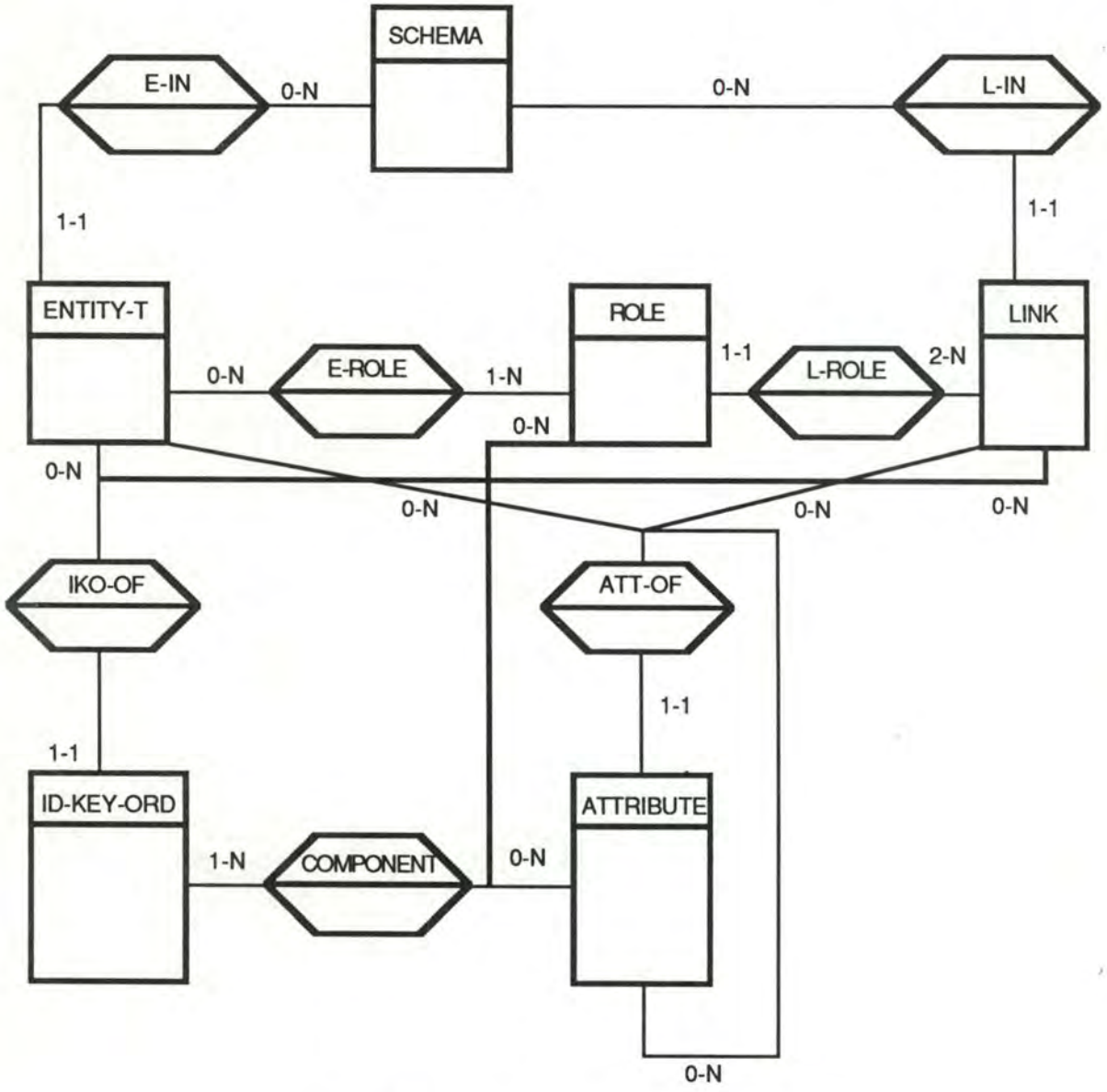


Fig. 5.3. : structure de base d'un schéma

L'élément de base d'un SCHEMA est le type d'entités (ENTITY-T). Tout ENTITY-T appartient à un SCHEMA.

Un type d'associations entre ENTITY-T est représenté par un LINK qui relie au moins deux ENTITY-T (non nécessairement distincts).

Tout ENTITY-T participant à un LINK y joue un ROLE. Un même ROLE d'un LINK peut être joué par plus d'un ENTITY-T. Etant donné un LINK et un ROLE joué par un ou plusieurs ENTITY-T, on indiquera le nombre minimum (MIN-CON) et le nombre maximum (MAX-CON) d'occurences de ce LINK dans lesquels une même occurence de cet ENTITY-T peut apparaître.

Toute propriété caractérisant un ENTITY-T ou un LINK particuliers est représentée par un ATTRIBUTE. Un ATTRIBUTE peut être décomposable en ATTRIBUTE plus élémentaires. On indique le nombre de valeurs d'un ATTRIBUTE que l'on peut trouver associées à un ENTITY-T, LINK ou ATTRIBUTE décomposable. Ce nombre est spécifié par une valeur minimum (MIN-REP) et une valeur maximum (MAX-REP).

Lorsque MIN-REP est nul, l'ATTRIBUTE est dit facultatif. Lorsque MAX-REP est supérieur à 1, L'ATTRIBUTE est dit répétitif. Dans ce cas, La répétitivité peut être fixe (MAX-REP = MIN-REP) ou variable (MAX-REP > MIN-REP), auquel cas il est possible de définir pour cet ATTRIBUTE un compteur qui n'est rien d'autre qu'un ATTRIBUTE voisin. La répétitivité variable peut encore être limitée ou illimitée. Ce dernier cas est représenté par une valeur conventionnelle de MAX-REP.

Un ou plusieurs ATTRIBUTE d'un ENTITY-T peuvent constituer un identifiant pour ce dernier. On représente cet identifiant par un ID-KEY-ORD associé à cet ENTITY-T et à ces ATTRIBUTE. En toute généralité, un identifiant d'un ENTITY-T peut être constitué :
- soit d'un ou plusieurs ATTRIBUTE.
- soit d'au moins deux ROLE.
- soit encore d'un ou plusieurs ATTRIBUTE et d'un ou plusieurs ROLE.

Cette notion d'identifiant peut aussi s'appliquer à un LINK. L'ensemble des ENTITY-T participant à un LINK constitue l'identifiant de ce dernier. Les ENTITY-T sont alors plutôt désignés par le ROLE qu'ils y jouent (indispensable s'ils y jouent plus d'un ROLE).

Au niveau de la description des accès aux données, une notion apparaît: celle de clé d'accès à un ENTITY-T. Une clé est le plus souvent constituée d'un ou plusieurs ATTRIBUTE de l'ENTITY-T. Certaines clés peuvent être contraintes cependant à n'opérer que sur certaines occurences d'un ENTITY-T : celles qui jouent un ROLE dans certaines occurences d'un LINK. On modélise ces structures en considérant qu'une clé d'accès est constituée d'un ou plusieurs ATTRIBUTE et éventuellement d'un ROLE.

Si les notions d'identifiant et de clé d'accès sont conceptuellement strictement indépendantes l'une de l'autre, on observe toutefois que dans la pratique, un identifiant est souvent aussi une clé d'accès. On représente donc un identifiant et une clé d'accès par le même objet ID-KEY-ORD.

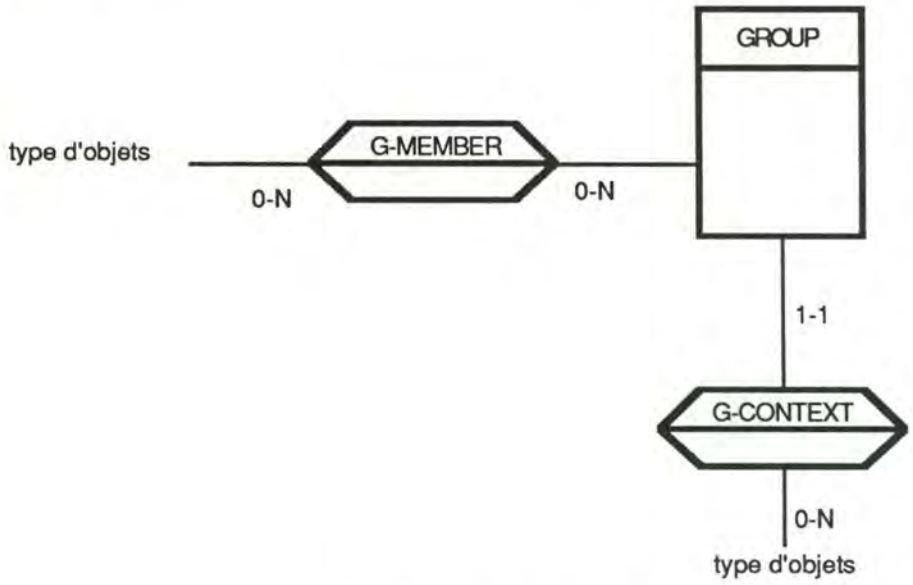


Fig. 5.4. : groupe d'objets

Un groupe (GROUP) est un agrégat d'objets éventuellement hétérogènes. Il est défini dans le contexte d'un objet principal dont il précise le détail (SCHEMA, MODULE, RELATION). L'association du GROUP à cet objet s'effectue via G-CONTEXT. Les membres de l'agrégat constitutif du GROUP lui sont associés via G-MEMBER.

Ce schéma de la base de spécifications permet de représenter facilement tous les objets qui interviennent dans les étapes successives de la démarche de conception d'applications sur bases de données.

On peut aisément entrevoir sans s'y attarder qu'un SCHEMA peut être un schéma conceptuel, un schéma des accès nécessaires, un schéma conforme à un SGD cible,... . Un MODULE peut être un algorithme prédicatif, un algorithme effectif conforme LDA/MAG, un algorithme conforme à un SGD cible,... . Un ENTITY-T peut représenter un type d'articles; un LINK un type de chemins.

La base des spécifications donne donc la possiblité d'enregistrer de nombreux aspects de la démarche pour laquelle l'atelier présente une aide sous la forme d'outils logiciels.

5.3. Les outils logiciels de l'atelier

L'atelier logiciel de conception d'applications sur bases de données offre un ensemble d'outils, les uns destinés à l'utilisateur final, les autres au développeur de l'atelier.

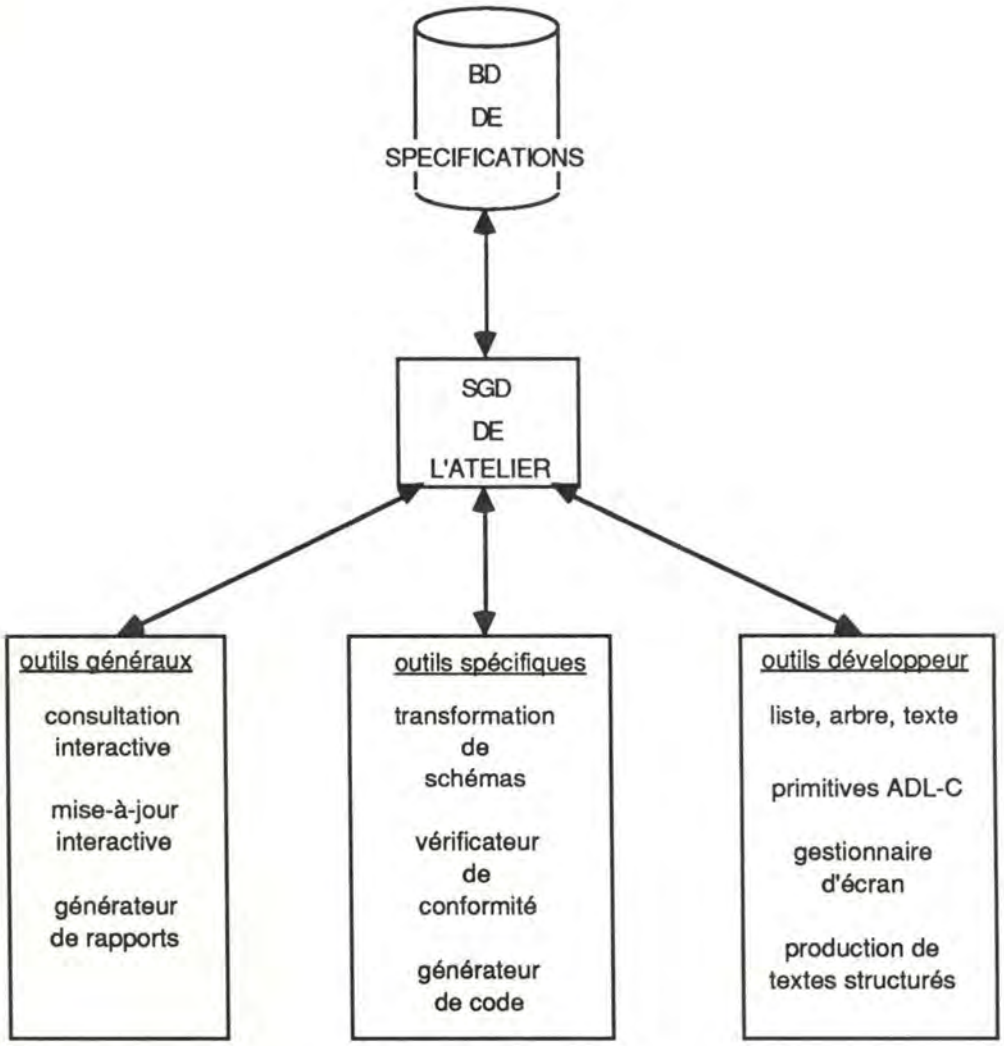


Fig. 5.5. : Les outils de l'atelier

Pour aider l'utilisateur final dans l'application de la démarche de conception d'une application sur base de données, l'atelier lui offre :

- des outils généraux de consultation et de mise à jour du contenu de la base des spécifications; des outils de génération de rapports documentaires présentant certains aspects de l'application développée.
- des outils plus spécifiques qui lui permettent de transformer un schéma des accès nécessaires en schéma conforme à un SGD cible; un outil de vérification de conformité d'un schéma par rapport à un SGD cible; un outil de génération de code COBOL.

Si l'atelier offre déja de nombreux outils logiciels destinés à l'utilisateur final, il n'a pas pour autant négligé le développeur de l'atelier. L'atelier met en effet à la disposition de ce dernier des outils de gestion de données complexes telles que les listes, les arbres et les textes. Des primitives ADL-C lui permettent d'exploiter la base des spécifications. Un gestionnaire d'écran et un outil de production de texte de structure complexe tel qu'un rapport autorisent le

développeur à se concentrer sur les aspects logiques des problèmes à résoudre plutôt que sur des détails techniques. Le chapitre 6 se consacre en partie à la description de l'outil de gestion d'arbre offert par l'atelier. Cet outil trouve en effet dans le cadre de ce mémoire un terrain propice à sa mise en œuvre.

5.4. Contribution du mémoire à l'atelier

L'étude de transformation d'algorithmes effectifs conformes LDA/MAG en algorithmes conformes à un SGD cible qui est menée dans ce mémoire vise à doter l'atelier d'un outil supplémentaire spécifique pour l'utilisateur final.

Ayant transformé un schéma des accès nécessaires en schéma conforme à un SGD cible à l'aide de l'outil adéquat, l'utilisateur pourra alors utiliser l'outil de transformation d'algorithmes. Celui-ci transformera de façon automatique un algorithme effectif conforme LDA/MAG en algorithme conforme au SGD cible choisi. Le travail de l'utilisateur se trouvera dès lors considérablement allégé. Un générateur de code final Cobol pourra alors être utilisé.

Chapitre 6 :

Un type abstrait de données : l'arbre

6.1. Introduction

Le langage LDA dont la syntaxe et la sémantique ont été décrites au chapitre 3 permet la rédaction d'algorithmes qui interviennent dans des applications sur bases de données.

Dans la phase de conception logique d'une application sur base de données, ces algorithmes sont rédigés de façon à être effectifs et conformes au SGD LDA/MAG. Lors de la phase de conception physique, ils sont transformés selon les règles de transformation exposées au chapitre 4 afin d'être rendus conformes à un SGD cible choisi.

L'étude de l'automatisation de la transformation d'algorithmes effectifs conformes LDA/MAG en algorithmes conformes à un SGD cible vise à doter l'atelier logiciel présenté au chapitre 5 d'un outil spécifique supplémentaire destiné à l'utilisateur final.

L'expression d'un algorithme sous la forme d'un arbre syntaxique étudiée dans ce chapitre a pour but de définir la forme sous laquelle un algorithme se soumet au travail du transformateur. Les techniques qui permettent cette expression constituent les éléments de la théorie des compilateurs qui n'est plus un nouveau domaine de recherche.

Dans le cadre de cette théorie, un algorithme LDA est soumis à divers examens et traitements qui forment les phases d'analyses lexicale, syntaxique et sémantique.

La première phase appelée l'analyse lexicale lit l'algorithme source LDA un caractère à la fois et découpe la suite de caractères représentant l'algorithme en unités atomiques appelées "tokens". Tout token est une séquence de caractères dont la signification est liée à la totalité de la séquence plutôt qu'à chaque caractère. Bien que l'ensemble des tokens dépende du langage envisagé, on peut établir certaines règles : sont des tokens les constantes, identificateurs, opérateurs, les mots réservés et symboles de ponctuation. Le langage LDA présente comme tokens : <nom>, <nombre_non_signé>, <mot_réservé>, true, false, null, <string>, <=, >=, <>, (:, .., (), <symbole_spécial> à l'exception de quote (') et underscore(_) dans la mesure où ceux-ci ne sont jamais présents que dans des <string> et des <nom>.

La phase d'analyse syntaxique a deux fonctions. Elle vérifie que les tokens que lui communique l'analyse lexicale apparaissent dans des formes qui sont autorisées par la grammaire du langage LDA (détecte une erreur éventuelle). Le second aspect de l'analyse syntaxique consiste à rendre explicite la structure hiérarchique de la suite des tokens reçus. Cette structure hiérarchique héritée de la grammaire du langage LDA est représentée par un arbre dit syntaxique. Un arbre est un schéma constitué d'un ensemble de nœuds tous reliés par des branches qui ne forment pas de cycle. Chacun des nœuds d'un arbre syntaxique représente un élément d'algorithme. Tout nœud sauf la racine est accessible d'un et un seul nœud. La racine n'est accessible d'aucun nœud

L'analyse lexicale se comporte souvent comme une procédure que l'analyse syntaxique appelle chaque fois qu'elle a besoin d'un nouveau token. L'analyse lexicale communique alors le token qu'elle a lu.

Si l'analyse syntaxique se préoccupe de la forme d'un algorihtme, l'analyse sémantique se concentre sur sa signification. L'analyse sémantique s'attache à la signification des diverses expressions et instructions de l'algorithme LDA.

L'analyse sémantique peut être exécutée durant ou après l'analyse syntaxique. Son but est de vérifier l'exactitude d'une expression reconnue par l'analyse syntaxique. L'analyse sémantique vérifie que les arguments d'une expression ont été définis et détermine leur type. Elle évalue le type des résultats intermédiaires afin d'examiner si l'application des opérateurs

est possible.

Un algorithme LDA effectif et conforme à LDA/MAG est soumis aux examens et traitements de ces trois phases avant d'être transformé en algorithme conforme à un SGD cible. Ce chapitre 6 se préoccupe avant tout de présenter l'algorithme dans une forme utilisée pour la transformation, qui est celle de l'arbre construit lors de l'analyse syntaxique. Les phases d'analyses lexicale et sémantique ne sont dès lors pas décrites davantage.

6.2. L'analyse syntaxique

Un analyseur syntaxique pour la grammaire du langage LDA est un programme qui reçoit une suite de tokens W et produit soit un arbre syntaxique si W constitue une phrase de la grammaire du langage LDA, soit un message d'erreur indiquant que W n'est pas une phrase de cette grammaire.

Diverses techniques d'analyse syntaxique ont été étudiées dans la théorie des compilateurs. Parmi elles, la technique top-down qui construit un arbre syntaxique du haut (top) vers le bas (down), de la racine vers les feuilles a été choisie.

A chaque étape, la technique top-down essaie de trouver une règle de la grammaire du langage LDA telle que la dérivation (un élément de la partie droite d'une régle) la plus à gauche correspond au token reçu de l'analyseur lexical. La technique construit alors l'arbre syntaxique en pré-ordre (parcourir un arbre en pré-ordre est défini récursivement comme : si l'arbre est constitué d'une seule feuille, visiter la feuille; si l'arbre a une racine n et des fils n1,n2, ..., nk, visiter n, visiter en pré-ordre le sous-arbre ayant n1 comme racine, et ainsi de suite jusque nk).

Par exemple, considérons la grammaire <e> ::= a <x>
<x> ::= bc | b
et la suite de tokens W reçue de l'analyseur lexical, abd. L'analyseur syntaxique trouve <e> comme règle dont la dérivation la plus à gauche (a) correspond à abd. Il crée alors un arbre constitué d'un nœud n-e et de 3 fils n-a, n-x, n-d :

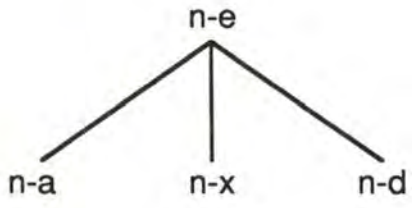


fig 6.1 : arbre 1

Le fils le plus à gauche (n-a) correspondant au premier token de W, l'analyseur syntaxique considère le fils suivant (n-x) et trouve <x> comme règle dont la dérivation la plus à gauche correspond au token suivant (b) de W. L'arbre

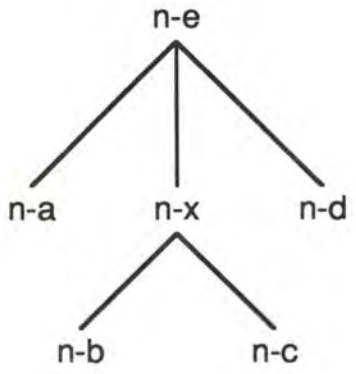


fig 6.2 : arbre 2

est construit. Le fils le plus à gauche (n-b) correspondant au second token de W, l'analyseur syntaxique considère alors le troisième token (d) de W et le fils (n-c) qui ne s'accordent pas. Un retour en arrière au nœud n-x est nécessaire pour voir s'il existe une alternative dans <x> qui pourrait convenir. En retournant en arrière, l'analyseur détruit les fils n-b et n-c de n-x et reconsidère dans W le second token qui permit de construire

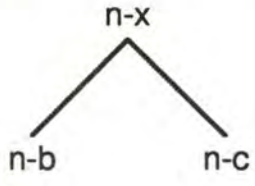


fig 6.3 : sous-arbre 2

Le membre b de l'alternative de <x> correspondant au second token de W, l'arbre

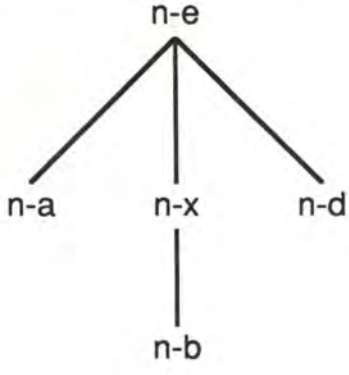


fig 6.4 : arbre 3

est construit et confirmé ensuite par la correspondance entre le fils n-d et le troisième token de W.

Plusieurs difficultés peuvent survenir dans l'emploi d'un analyseur top-down. La première difficulté réside dans une grammaire dite récursive à gauche. Une grammaire est récursive à gauche si elle contient une règle qui comporte une partie gauche et une dérivation la plus à gauche identiques (par ex. <a> ::= <a> b). Ceci entraîne l'analyseur syntaxique dans un bouclage infini. La grammaire du langage LDA exposée au chapitre 3 a été rédigée de façon à éviter ce problème.

Une autre difficulté réside dans l'indécidabilité qui peut exister dans le choix d'un membre d'une alternative dans une règle de la grammaire. Ce problème s'est posé dans l'exemple décrit précédemment où considérant la règle <x> et le token b de W, on ne peut choisir entre les membres bc et b sans encourir le risque d'un retour en arrière lorsque l'on considère le token suivant de W. Afin d'éviter un travail inutile à l'analyseur syntaxique, la grammaire du langage LDA a été définie de manière à éviter aussi ce problème. La solution consiste à factoriser les préfixes communs des membres d'une alternative : <x> ::= bc | b devient <x> ::= b <y>
<y> ::= c | <vide>.

Afin de ne pas alourdir un arbre syntaxique, l'analyseur suntaxique conserve de l'information dans des tables. Cette information peut être la chaîne de caractères représentant un <nom>, un <string> ou un <nombre_non_signé>. Chaque fois qu'un tel objet est rencontré dans l'algorithme LDA, les tables sont parcourues et la chaîne de caractères représentant un <nom>, un <string> ou un <nombre_non_signé> y est ajoutée si elle ne s'y trouve pas déjà. A chaque objet ainsi enregistré dans les tables sont associées, de plus, diverses données telles que par exemple le type d'un <nombre_non_signé> (integer, real, ...). Ces données qui sont utilisées lors de la phase d'analyse sémantique, ne rentrent pas dans la préoccupation immédiate de ce chapitre et dès lors ne sont pas décrites davantage.

Toujours pour ne pas alourdir l'arbre, il convient de ne pas passer systématiquement de la grammaire à la définition des nœuds, dans la mesure où cette dernière est soumise à des considérations d'efficacité alors qu'une grammaire ne l'est pas. Il faut en effet restreindre le nombre de nœuds générés pour un algorithme donné car un arbre volumineux coûte tant en place qu'en termes d'accès.

Il n'existe pas de règles précises pour "compacter" un arbre. Il faut éliminer les nœuds "inutiles" tout en se gardant de ne pas trop en éliminer, ce qui aurait pour conséquence de détruire la structure d'un algorithme. On peut cependant dire que les clauses n'ayant pour but que de rendre la grammaire plus lisible ne sont pas traduites en nœuds. De même, si l'utilisation de la récursivité dans la grammaire se justifie par l'élégance qu'elle y introduit, elle ne se justifie plus au niveau des nœuds car elle entraînerait la génération d'arbres très volumineux.

Plus concrètement, l'algorithme exemplatif présenté au point 6.4.4 requiert un arbre de 47 nœuds pour être représenté. Si la représentation de l'arbre avait été définie à partir d'une "traduction littérale" de la grammaire, plus de 110 nœuds auraient été nécessaires.

6.3. La notion d'arbre dans l'atelier

L'atelier logiciel de conception d'applications sur bases de données décrit au chapitre 5 offre au développeur de l'atelier un outil de manipulation d'arbre qui permet l'implémentation aisée d'un arbre syntaxique. Cet outil est constitué d'un ensemble de primitives opérant sur une structure d'arbre définie dans le cadre de l'atelier.

Un arbre y est un objet qui permet de mémoriser, de gérer et de manipuler une collection d'éléments (en nombre quelconque et variable) structurée en arbre. Chaque élément appelé nœud de l'arbre est composé de deux constituants du type integer dénommés DATA et COMPL. L'interprétation de ces constituants est définie par le développeur de l'atelier et est extérieure au concept d'arbre tel qu'il est connu des primitives.

Un arbre vide ne contient aucun nœud. Un arbre non vide contient un nœud racine et un nombre quelconque (éventuellement nul) de nœuds descendants. Tout nœud, sauf la racine, possède un nœud supérieur que l'on appelle son père. A tout nœud, on peut associer une séquence de nœuds inférieurs que l'on appelle fils. Cette séquence est caractérisée par le fait qu'à un nœud père on peut associer un premier nœud fils, et qu'à chaque nœud fils on peut associer un nœud frère suivant (sauf au dernier) et un nœud frère précédent (sauf au premier). Le niveau d'un nœud dans un arbre est défini comme suit : le niveau de la racine est 1; le niveau d'un nœud non-racine est égal au niveau de son père +1. Le nombre de niveaux d'un arbre est égal à la plus grande valeur de niveau parmi les nœuds de cet arbre, ou à 0 si l'arbre est vide.

Les primitives offertes permettent notamment de :

- se positionner dans un arbre (à la racine, au nœud père d'un nœud, au premier nœud fils d'un nœud, ...).
- accéder aux constituants DATA et COMPL d'un nœud.
- de tester si un nœud a un père, au moins un fils, un nœud frère suivant, ...
- d'insérer un nœud fils d'un nœud spécifié, ...
- de supprimer un nœud.
- de réorganiser la structure d'un arbre (un nœud n1 accompagné de sa descendance est détaché de son père, quitte ses frères, pour être inséré comme unique fils d'un nœud n2. Les anciens fils du nœud n2 deviennent les frères suivants des fils du nœud n1).
- de parcourir un arbre en pré-ordre.

6.4. Représentation d'un algorithme LDA sous la forme d'arbre

La notion d'arbre étant définie, ce paragraphe se propose d'en voir une utilisation concrète à savoir la représentation d'un algorithme LDA utilisée lors de la transformation qui doit le rendre conforme à un SGD cible.

Le travail consiste, comme cela a déjà été dit, à établir des règles de représentation en arbre à partir de la grammaire définissant le langage LDA (cfr chapitre 3, §2). Ces règles énumèrent les types de nœuds que l'on peut trouver dans l'arbre, ainsi que leurs différentes combinaisons possibles par l'énumération des fils qu'un nœud peut avoir.

Dans ce contexte, le présent paragraphe se décompose en trois points. Le premier définit la structure virtuelle d'un nœud indépendamment du type abstrait de données développé dans l'atelier, ainsi que les tables de symboles; le second détaille les règles de représentation toujours en termes virtuels; le troisième décrit la façon dont les nœuds sont concrètement représentés dans les arbres offerts par l'atelier. Une illustration de l'utilisation des règles de représentation d'un algorithme clôture le paragraphe.

6.4.1. Structure d'un nœud et table des symboles

1. Les nœuds

Un arbre peut contenir différents types de nœuds et chaque nœud a un type. Ce type détermine la construction LDA dont le nœud est une représentation. A chaque type est attribué un code identifiant. Ce code est contenu dans le nœud.

Par exemple, un nœud du type "instr_create" est la représentation d'une instruction de création d'article et son code est 30.

Outre son code, un nœud peut avoir un certain nombre de fils et/ou peut faire référence à des entrées de la table des symboles.

Plus précisément, certains types de nœuds ont un nombre fixe de fils (cela peut être 0) et d'autres en ont un nombre variable. Certains types de nœuds ne peuvent avoir qu'un type de fils. Inversément, d'autres peuvent avoir plusieurs types de fils. Les nœuds à nombre et types de fils fixes sont définis à partir de règles ne contenant pas d'alternative.

La figure 6.5 décrit un exemple d'utilisation de nœud "instr_create" dans un formalisme graphique :

soit l'instruction : create a := A (ch : b)

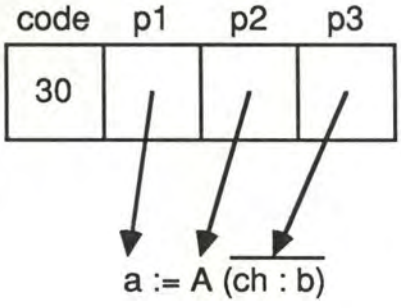


fig 6.5 : exemple de nœud

2. La table des symboles

Certains types de nœuds contiennent des champs faisant référence à une entrée de la table des symboles.

Deux tables sont utilisées :
- la table des constantes qui contient toutes les constantes apparaissant dans l'algorithme LDA. Elle reprend les constantes numériques et les strings.
- la table des symboles contenant les identificateurs non réservés par le langage LDA. On peut y trouver les noms de (types de) variables, fonctions, procédures, types d'articles, types de chemins,....

En l'occurence, la structure de ces tables ne fait pas l'objet d'une description approfondie dans la mesure où celle-ci n'est pas nécessaire à l'objet de ce paragraphe.

6.4.2. Règles de représentation

Ce point présente les différents types de nœuds que peut contenir un arbre ainsi que les fils (nœuds) que ces nœuds peuvent avoir, et les références vers les tables qu'ils peuvent contenir.

Les noms des types sont souvent repris littéralement de la grammaire; par exemple, le nœud de type "instr_create" tire son nom de la règle <instr_create> qui définit la forme d'une instruction de création. Cependant, comme cela a déjà été dit, la structure de l'arbre ne reflète pas exactement celle de la grammaire pour des raisons d'efficacité.

Les nœuds sont présentés par code croissant, et la description particulière d'un nœud se fait selon le formalisme suivant :

<code> <nom du type de nœuds>

<description des fils et/ou des références vers les tables>

La description de chaque fils est d'autre part accompagnée de son code entre parenthèses.

1 structure_d'algorithme

Une référence à la table des symboles pour le nom en entête d'algorithme.

2 fils : section_déclaration_interne (2)
instructions (19)

2 section_déclaration_interne

2 fils : section_déclaration_type (3)
section_déclaration_variable (5)

3 section_déclaration_type

0 à N fils déclaration_type (4), autant qu'il y a de types déclarés.

4 déclaration_type

Une référence à la table des symboles pour le nom du type.

1 fils qui peut être boolean (11) ou real (12) ou integer (13) ou numeric (10) ou string (9) ou groupe (14) ou tableau (15) ou items_de (18) ou ref_de (17) ou nom (8).

5 section_déclaration_variable

0 à N fils déclaration_variable (6) selon le nombre de déclarations que la section contient.

6 déclaration_variable

2 fils : un est noms (7)

L'autre peut être boolean (11) ou real (12) ou integer (13) ou numeric (10) ou string (9) ou groupe (14) ou tableau (15) ou items_de (18) ou ref_de (17) ou nom (8).

7 noms

1 à N fils nom (8) selon le nombre de variables apparaissant dans cette déclaration.

8 nom

Une référence à la table des symboles.

9 string

Une référence à la table des constantes pour la longueur.

10 numeric

2 fils : entier_non_signé (48)
entier_non_signé (48)

11 boolean

pas de fils

12 real

pas de fils

13 integer

pas de fils

14 groupe

1 à N fils déclaration_variable (6) selon le nombre de champs déclarés dans le groupe (group).

15 tableau

2 fils : liste_des_indices (16)
et

boolean (11) ou real (12) ou integer (13) ou numeric (10) ou string (9) ou items_de (18) ou ref_de (17) ou nom (8) selon le type des éléments du tableau.

16 liste_des_indices

1 à 3 fils entier_non_signé (48) selon le nombre de dimensions du tableau.

17 ref_de

Une référence à une entrée de la table des symboles qui désigne le nom du type d'articles dont la variable déclarée est référence.

18 items_de

Une référence à une entrée de la table des symboles contenant le nom du type d'articles dont la variable déclarée est une variable d'item.

19 instructions

0 à N fils selon le nombre d'instructions. Chaque fils peut être de type assignation (20) ou instr_for (22) ou instr_if (25) ou instr_while (26) ou instr_next (27) ou instr_exit (28) ou appel (29) ou instr_create (30) ou instr_modify (39) ou instr_delete (44).

20 assignation

2 fils : class_var (50)
expression_assignation (21)

21 expression_assignation

1 fils qui peut être expression_arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47) ou référence_nulle (81) ou expression_booléenne (76) ou expression_collection (64).

22 instr_for

3 fils : Le premier : class_var (50) ou item_répétitif_décomposable (23) qui dénote la variable de parcours.

Le second : intervalle (63) ou expression_collection (64) ou var_refs (53). C'est la séquence parcourue.

Le troisième : instructions (19). C'est le corps de la boucle for.

23 item_répétitif_décomposable

2 à N fils comp_variable_parcours (24) selon le nombre de composants élémentaires de l'item parcouru.

24 comp_variable_parcours

1 fils class_var (50). C'est la variable à qui sera assignée la valeur du composant.

Une référence à la table des symboles : le nom de l'item composant élémentaire de l'item parcouru.

25 instr_if

2 à 3 fils : expression_booleénne (76)

instructions (19) : instructions à exécuter dans le "then" du if.

instructions (19) : instructions à exécuter dans le "else" du if. Ce fils n'existe pas s'il n'y a pas de "else".

26 instr_while

2 fils : expression_booléenne (76)
instructions (19)

27 instr_next

0 ou 1 fils item_répétitif_décomposable (23) ou nom (8).

28 instr_exit

0 ou 1 fils item_répétitif_décomposable (23) ou nom (8).

29 appel

Une référence à la table des symboles pour le nom de la fonction ou procédure appelée.

0 ou N fils qui sont les paramètres de l'appel. Chaque fils peut être de type expression_arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47) ou référence_nulle (81).

30 instr_create

3 fils : nom (8) : la variable de référence.

nom (8) : le type d'articles.

conditions_création (31) pour les conditions de création.

31 conditions_création

0 à N fils condition_création_chemin (32) et/ou condition_création_ item (33) selon le nombre de conditions élémentaires qui apparaissent dans la condition de création.

32 condition_création_chemin

2 fils : nom (8)
nom (8)

33 condition_création_item

Une référence à la table des symboles pour le nom de l'item.

un fils qui est :
- si l'item est simple et élémentaire, expression_arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47).
- si l'item est répétitif et élémentaire, liste_exp_simple (34).
- si l'item est répétitif et décomposable, liste_exp_décomposable (35).
- si l'item est simple et décomposable et suivi par un de ses fils, condition_création_ item (33).
- si l'item est simple et décomposable et suivi par plusieurs de ses fils, cond_création_ plus_d'un_item (38).

34 liste_exp_simple

1 à N fils expression_arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47), selon le nombre d'éléments de la liste.

35 liste_exp_décomposable

1 à N fils élément (36) selon le nombre de valeurs d'items décomposables que l'on associe à l'article.

36 élément

0 à N fils assign (37) selon le nombre de valeurs d'items (composants de l'item décomposable) spécifiées.

37 assign

Une référence à une entrée de la table des symboles pour le nom de l'item.

1 fils expression_arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47).

38 cond_création_plus_d'un_item

2 à N fils condition_création_item (33).

39 instr_modify

Une référence à la table des symboles pour le nom de la variable de référence qui désigne l'article que l'on modifie.

1 fils qui est
- condition_mod_item (41) si la modification porte sur un item.
- conditions_mod_items (40) si la modification porte sur plusieurs items.
- cond_detach (42) si la modification porte sur un chemin.
- cond_attach (43) si la modification porte sur un chemin.

40 conditions_mod_items

2 à N fils condition_mod_item (41).

41 condition_mod_item

Une référence à une entrée de la table des symboles pour le nom de l'item dont on modifie une valeur.

un fils qui est :
- si l'item est simple et élémentaire, expression_arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47).
- si l'item est répétitif et élémentaire, liste_exp_simple (34); ou plus (59) ou moins (60). Dans ces deux derniers cas, il y a un second fils qui est expression_ arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47).
- si l'item est répétitif et décomposable, liste_exp_décomposable (35) ou élément (36). Il y a, dans ce dernier cas, un autre fils qui est plus (59) ou moins (60).
- si l'item est simple et décomposable et suivi par un de ses fils, condition_création_ item (33).
- si l'item est simple et décomposable et suivi par plusieurs de ses fils, cond_création_ plus_d'un_item (38).

42 cond_detach

2 fils nom (8).

43 cond_attach

2 fils nom (8).

44 instr_delete

Une référence à la table des symboles pour la variable de référence de l'article détruit.

45 true

pas de fils

46 false

pas de fils

47 valeur_null

pas de fils

48 entier_non_signé

Une référence vers la table des constantes.

49 nombre_non_signé

Une référence vers la table des constantes.

50 class_var

1 fils variable_non_hiérar (51).

0 ou 1 fils class_var (50).

51 variable_non_hiérar

Une référence à une entrée de la table des symboles pour le nom de la variable.

0 ou 1 fils indices (52) selon que la variable est indicée.

52 indices

1 à 3 fils expression_arithmétique (54) selon le nombre d'indices.

53 var_refs

Une référence à une entrée de la table des symboles pour le nom de la variable de référence.

1 fils class_var (50)

54 expression_arithmétique

2 fils : soit plus (59) ou moins (60)
et expression_arithmétique.

soit facteur (56)
et suite_expression_arithmétique (55)

55 suite_expression_arithmétique

0 ou 2 fils : plus (59) ou moins (60)
et expression_arithmétique (54)

56 facteur

2 fils : terme (58)
suite_facteur (57)

57 suite_facteur

0 ou 2 fils : fois (61) ou divisé (62)
et
facteur (56)

58 terme

1 ou 2 fils. Chaque fils peut être un nombre_non_signé (49) ou class_var (50) ou var_refs (53) ou appel (29) ou expression_arithmétique (54).

59 plus

pas de fils

60 moins

pas de fils

61 fois

pas de fils

62 divisé

pas de fils

63 intervalle

2 fils qui peuvent être chacun entier_non_signé (48) ou nom (8).

64 expression_collection

Une référence à la table des symboles pour le nom du type d'articles sur lequel on pose des conditions.

1 fils qui est soit condition_vide (65) ou conditions_sélection_items (66) ou condition_sur_chemin (74) ou condition_sur_clé_dans_un_chemin (75).

65 condition_vide

pas de fils

66 conditions_sélection_items

1 à N fils cond_sélection_item (67) selon le nombre d'items sur lesquels porte la condition de sélection.

67 cond_sélection_item

Une référence à la table des symboles pour le nom de l'item sur base duquel la sélection se fait.

1 fils suite_cond_sélection_item (68)

68 suite_cond_sélection_item

1 fils conditions_sélection_items (66)
ou
2 fils : le premier est plus_petit (69) ou plus_grand (70) ou égal (71) ou plus_petit_égal (72) ou plus_grand_égal (73).

Le second est expression_arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47).

69 plus_petit

pas de fils

70 plus_grand

pas de fils

71 égal

pas de fils

72 plus_petit_égal

pas de fils

73 plus_grand_égal

pas de fils

74 condition_sur_chemin

2 fils nom (8)

75 condition_sur_clé_dans_un_chemin

2 fils : condition_sur_chemin (74)
et
conditions_sélection_items (66).

76 expression_booléenne

1 ou 2 fils, facteur_booléen (77) seul ou avec expression_booléenne (76).

77 facteur_booléen

1 fils : not_primaire_booléen (78) ou primaire_booléen (79)
seul ou avec un autre fils : facteur_booléen (77).

78 not_primaire_booléen

1 fils expression_de_test (80) ou expression_booléenne (76).

79 primaire_booléen

1 fils expression_de_test (80) ou expression_booléenne (76).

80 expression_de_test

2 fils qui peuvent être chacun expression_arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47) ou référence_nulle (81).

et 1 fils qui peut être plus_petit (69) ou plus_grand (70) ou égal (71) ou plus_petit_ égal (72) ou plus_grand_égal (73) ou différent (82).

81 référence_nulle

pas de fils.

82 différent

pas de fils.

83 constante_caractère

Une référence à une entrée de la table des constantes contenant la constante caractère.

6.4.3. Représentation de l'arbre selon le type abstrait de données de l'atelier

L'objet de ce point est de voir comment la structure présentée au point 6.4.1. peut être représentée par la notion d'arbre de l'atelier, décrite au paragraphe 6.3.

Le champ DATA des nœuds contient le code du type de nœuds. Les pointeurs vers les fils sont gérés par l'outil lui–même. Le champ COMPL est un indice d'une table qui peut être soit la table des symboles, soit celle des constantes.

6.4.4. Exemple de représentation en arbre

Ce point présente la représentation en arbre d'un algorithme simple selon les règles définies en 6.4.2.

Bien que l'algorithme ne soit qu'un exemple pédagogique, la taille de l'arbre généré exclut toute représentation graphique classique en termes de "boîtes" et de "flèches". Il faut donc adopter les conventions suivantes : les nœuds sont écrits à la suite les uns des autres tels qu'on les obtiendrait si on parcourait l'arbre en pré-ordre. Un nœud est représenté par le nom de son type suivi de son code. Il est de niveau i s'il est précédé de i caractères "Δ" et il est le fils du premier nœud de niveau i-1 rencontré en remontant la liste des nœuds.

Le lecteur perplexe peut se baser sur l'exemple formel suivant :

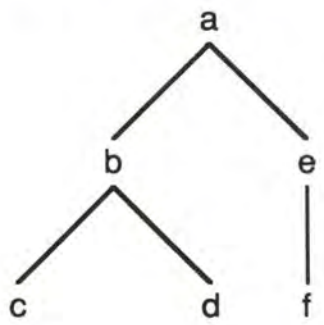


fig 6.6 : représentation graphique d'un arbre

Selon les conventions établies, cet arbre est exprimé de la façon suivante :

Δ a

ΔΔ b

ΔΔΔ c

ΔΔΔ d

ΔΔ e

ΔΔΔ f

L'algorithme est le suivant :

```
algorithm  trt_client

type  tab = array [10] of string(30);

var   tabnomclient : tab;
      c : ref of client;
      i : integer;

begin
      inittableau(tabnomclient);
      for i := 1..10 do
          create c := client (: nom = tabnomclient[i])
      endfor
end.
```

Sa représentation est la suivante :

Δ structure_d'algorithme : 1. Il contient une référence vers une entrée de la table des symboles contenant "trt_client".

ΔΔ section_déclaration_interne : 2

ΔΔΔ section_déclaration_type : 3

ΔΔΔΔ déclaration_type : 4. Il contient une référence vers une entrée de la table des symboles contenant "tab".

ΔΔΔΔΔ tableau : 15

ΔΔΔΔΔΔ liste_des_indices : 16

ΔΔΔΔΔΔΔ entier_non_signé : 48. Il contient une référence vers une entrée de la table des constantes contenant "10".

ΔΔΔΔΔΔ string : 9. Il contient une référence vers une entrée de la table des constantes contenant "30", la longueur du string.

ΔΔΔ section_déclaration_variable : 5

ΔΔΔΔ déclaration_variable : 6

ΔΔΔΔΔ noms : 7

ΔΔΔΔΔΔ nom : 8. Il contient une référence vers une entrée de la table des symboles contenant "tabnomclient".

ΔΔΔΔΔ nom : 8. Il contient une référence vers l'entrée de la table des symboles contenant "tab".

ΔΔΔΔ déclaration_variable : 6

ΔΔΔΔΔ noms : 7

ΔΔΔΔΔΔ nom : 8. Il contient une référence vers une entrée de la table des symboles contenant "c".

ΔΔΔΔΔ ref_de : 17. Il contient une référence vers une entrée de la table des symboles contenant "client".

ΔΔΔΔ déclaration_variable : 6

ΔΔΔΔΔ noms : 7

ΔΔΔΔΔΔ nom : 8. Il contient une référence vers une entrée de la table des symboles contenant "i".

ΔΔΔΔΔ integer : 13. Nœud sans fils.

ΔΔ instructions : 19

ΔΔΔ appel : 29. Il contient une référence vers une entrée de la table des symboles contenant "inittableau".

ΔΔΔΔ class_var : 50

ΔΔΔΔΔ variable_non_hiérar : 51. Il contient une référence vers l'entrée de la table des symboles contenant "tabnomclient".

ΔΔΔ instr_for : 22

ΔΔΔΔ class_var : 50

ΔΔΔΔΔ variable_non_hiérar : 51. Il contient une référence vers l'entrée de la table des symboles contenant "i".

ΔΔΔΔ intervalle : 63

ΔΔΔΔΔ entier_non_signé : 48. Il contient une référence vers une entrée de la table des constantes contenant "1".

ΔΔΔΔΔ entier_non_signé : 48. Il contient une référence vers une entrée de la table des constantes contenant "10".

ΔΔΔΔ instructions : 19

ΔΔΔΔΔ instr_create : 30

ΔΔΔΔΔΔ nom : 8. Il contient une référence vers l'entrée de la table des symboles contenant "c".

ΔΔΔΔΔΔ nom : 8. Il contient une référence vers l'entrée de la table des symboles contenant "client".

ΔΔΔΔΔΔ conditions_création : 31

ΔΔΔΔΔΔΔ condition_création_item : 32. Il contient une référence vers une entrée de la table des symboles contenant "nom".

ΔΔΔΔΔΔΔΔ class_var : 50

ΔΔΔΔΔΔΔΔΔ variable_non_hiérar : 51. Il contient une référence vers l'entrée de la table des symboles contenant "tabnomclient".

ΔΔΔΔΔΔΔΔΔΔ indices : 52

ΔΔΔΔΔΔΔΔΔΔΔ expression_arithmétique : 54

ΔΔΔΔΔΔΔΔΔΔΔΔ facteur : 56

ΔΔΔΔΔΔΔΔΔΔΔΔΔ terme : 58

ΔΔΔΔΔΔΔΔΔΔΔΔΔΔ class_var : 50

ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar : 51. Il contient une référence vers l'entrée de la table des symboles contenant "i".

ΔΔΔΔΔΔΔΔΔΔΔΔΔ suite_facteur : 57. Nœud sans fils.

ΔΔΔΔΔΔΔΔΔΔΔΔ suite_expression_arithmétique : 55. Nœud sans fils.

6.5. Conclusion

Ce chapitre a décrit la représentation d'un algorithme LDA sous la forme d'arbre syntaxique.

L'introduction a montré qu'un algorithme LDA effectif et conforme à LDA/MAG est soumis aux examens et aux traitements des phases d'analyses lexicale, syntaxique et sémantique avant d'être transformé en algorithme conforme à un SGD cible.

La représentation en arbre d'un algorithme LDA est le résultat de l'analyse syntaxique et peut être implémentée par l'outil offert par l'atelier logiciel et décrit au troisième paragraphe.

On peut tirer de ce chapitre les quelques conclusions qui suivent.

La grammaire d'un langage étant l'ensemble des règles qui permettent de déterminer si une suite de symboles est ou non une phrase du langage, on pourrait la considérer indépendante de l'outil utilisé pour analyser ce langage. Ce n'est pas le cas dans la mesure où beaucoup de règles ont été définies de telle sorte qu'elles ne soient ni récursives à gauche ni indécidables, ce qui est exigé par l'utilisation d'un analyseur top-down.

La définition d'une table particulière pour les constantes se justifie par le fait que les entrées de cette table ont une structure beaucoup plus simple (type et valeur) que celle des entrées de la table des symboles qui en plus du type et de la valeur de l'identificateur, peuvent contenir des références vers des nœuds de l'arbre (par ex. vers une déclaration de variable), vers une table de déclaration des objets de la base de données, ... etc... . L'utilisation de cette structure complexe pour une constante aurait entraîné un gaspillage de champs inutilisés.

La correspondance entre la structure virtuelle de nœud définie au point 6.4.1 et sa représentation physique à l'aide de la notion de nœud définie dans l'atelier logiciel, n'a pas posé de difficulté. La simplicité de la structure des nœuds virtuels a garanti ce passage harmonieux.

Le chapitre 7 expose l'application des règles de transformation d'algorithmes définies au chapitre 4 sur un algorithme représenté sous la forme décrite dans ce chapitre.

Chapitre 7 :

Expression des règles de transformation en termes d'arbre syntaxique

7.1. Introduction

Le chapitre 4 a exposé l'ensemble des règles de transformation d'algorithmes développées dans ce mémoire.

Le chapitre 6 a décrit la représentation d'un algorithme sous forme d'arbre.

Le présent chapitre se propose de faire la synthèse des deux précités en exposant la représentation des règles de transformation d'algorithmes sous forme d'arbre.

Rappelons que le but du mémoire est d'étudier la conception d'un outil de transformation d'algorithmes à intégrer dans l'atelier logiciel. Le présent chapitre trouve sa justification dans le fait que l'outil travaille sur l'arbre représentant un algorithme. Indépendamment de toute sémantique, l'effet de l'outil est de transformer des arbres.

De par son objet, ce chapitre fait fréquemment référence aux chapitres 4 et 6 dont la lecture est supposée avoir précédé celle de celui-ci.

Un premier paragraphe décrit en termes d'arbre, les transformations subies par un algorithme suite aux transformations de schéma. Le second paragraphe décrit, toujours en termes d'arbre, les transformations subies par l'algorithme obtenu de la première étape (celle consécutive aux transformations de schéma) pour qu'il n'utilise plus que des primitives permises par le SGD cible choisi.

Chaque règle de transformation syntaxique développée au chapitre 4, est détaillée ici. Sa traduction en termes d'arbre est ensuite définie et est accompagnée d'un exemple si cela s'avère nécessaire à la compréhension de l'exposé.

Les deux paragraphes précités représentent les règles de transformation d'algorithmes selon le formalisme type BNF utilisé au chapitre 4. Les représentations en arbres sont basées sur le formalisme défini dans l'exemple de représentation développé au chapitre 6.

Il est d'autre part souvent fait référence aux nœuds définis au point 6.4.2. Le lecteur est à ce propos invité à consulter ce point, pour connaître la définition exacte de ces nœuds. Leur code est donné entre parenthèses afin de faciliter cette recherche.

Un troisième paragraphe fait en outre un bref exposé de la gestion de la table des symboles, qu'il y a lieu de faire suite à la transformation de l'arbre représentant un algorithme.

7.2. Représentation des règles de transformation de la première étape

Ce point a pour objet d'exposer la représentation en arbre des règles de transformation présentées au paragraphe 4.2.

1. APLATISSEMENT TOTAL

Règle 1

Dans le cadre d'un aplatissement total, (: <nom> <x>) (où la parenthèse fermante est celle qui ferme la condition portant sur l'item <nom>) devient <x> si <nom> est un nom d'item décomposable. Si <x> désigne plusieurs fils de l'item <nom>, les parenthèses l'entourant sont enlevées. Quand tout <nom> d'item décomposable a disparu, si l'expression finale porte sur plusieurs items, elle est entourée de parenthèses.

Transformation en termes d'arbre

Dans le cadre des conditions de création (de modification) d'items, chaque nœud condition_création_item (33) (condition_mod_item (41)) détermine un sous-arbre. On transforme les sous-arbres dont le nœud racine fait référence au nom de l'item aplati, dans la table des symboles. Dans ces sous-arbres, tout nœud (même la racine) condition_création_item (33) (ou condition_mod_item (41)) dont le fils n'est pas expression_arithmétique (54), ni class_var (50), ni var_refs (53), ni constante_caractère (83), ni appel (29), ni true (45), ni false (46), ni valeur_null (47) est détruit. Les fils du nœud détruit sont rattachés au père de celui-ci. Tout nœud cond_création_plus_d'un_item (38) est également détruit.

Exemple

La condition de création (: adresse (: rue = 'boulevard Mélot')) où adresse est aplati, devient (: rue = 'boulevard Mélot').

Cela se représente par :

... Δ condition_création_item (33) fait référence à "adresse" dans la table des symboles.
ΔΔ condition_création_item (33) fait référence à "rue" dans la table des symboles.
ΔΔΔ constante_caractère (83) fait référence à 'boulevard Mélot' dans la table des constantes.

DEVIENT

... Δ condition_création_item (33) fait référence à "rue" dans la table des symboles.
ΔΔ constante_caractère (83) fait référence à 'boulevard Mélot' dans la table des constantes.

L'exemple est analogue dans le cas des conditions de modification.

Soit une expression de collection exprimant un accès sur base d'un item décomposable soumis à un aplatissement. Un nœud cond_sélection_item (67) faisant référence au nom de l'item aplati, détermine un sous-arbre. Tout nœud conditions_sélection_items (66) de ce sous-arbre disparaît en compagnie de ses nœuds père et grand-père suite_cond_sélection_items (68) et cond_sélection_item (67). Les fils du nœud conditions_sélection_items (66) sont alors rattachés au nœud père de cond_sélection_item (67).

Exemple

pers := personne (: n°telephone (: prefixe = 02)) où n°telephone est aplati

DEVIENT

pers := personne (: préfixe = 02)

Cela se représente par :

... Δ conditions_sélection_items (66)
ΔΔ cond_sélection_item (67) fait référence à "n°telephone" dans la table des symboles
ΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔ cond_sélection_item (67) fait référence à "prefixe" dans la table des symboles
ΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔ égal (71)
...

DEVIENT

... Δ conditions_sélection_items (66)
ΔΔ cond_sélection_item (67) fait référence à "prefixe" dans la table des symboles
ΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔ égal (71)
...

Règle 2

(<varref>) . <comp> . <feuille>

DEVIENT

(<varref>) . <feuille>

Transformation en termes d'arbre

Dans le cadre des accès aux valeurs d'items d'un article, la décomposition (<comp>) de l'item est représentée par un sous-arbre de nœuds class_var (50) et variable_non_hiérar (51). Chaque variable_non_hiérar fait référence au nom d'un des items.

Pour l'aplatissement total, il faut déterminer la variable_non_hiérar qui fait référence (dans la table des symboles) au nom de l'item aplati. Ce nœud a un père class_var. Le nœud class_var est détruit ainsi que le sous-arbre dont il est racine. Seuls subsistent dans ce sous-arbre, le nœud class_var sans fils class_var, et son fils variable_non_hiérar (qui fait référence au nom de l'item feuille <feuille>). Ces deux nœuds survivants sont rattachés au nœud père (qui est un nœud var_refs (53) faisant référence le cas présent à <varref>) du class_var racine détruit.

Exemple

(cli).adresse.rue où adresse est aplati

DEVIENT

(cli).rue

Cela se représente par :

... Δ var_refs (53) fait référence à "cli" dans la table des symboles.
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence à "adresse" dans la table des symboles
ΔΔΔ class_var (50)
ΔΔΔΔ variable_non_hiérar (51) fait référence à "rue" dans la table des symboles

DEVIENT

... Δ var_refs (53) fait référence à "cli" dans la table des symboles.
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence à "rue" dans la table des symboles

2. AJOUT D'UN NIVEAU DE DECOMPOSITION

Règle 1

Dans le cadre d'un ajout de niveau de décomposition, soit une expression <x> composée de conditions portant sur des items (un ou plusieurs et à qui on attribue un père commun) lors d'un accès à la base de données, d'une création ou d'une modification d'article. Si <x> porte sur plusieurs items, et qu'elle n'est pas déjà entourée de parenthèses, on l'entoure. Dans les deux cas, elle est transformée en (: <nom> <x>) où <nom> est le nom de l'item père créé.

Transformation en termes d'arbre

Soient N (N > 1) nœuds frères condition_création_item (53) (ou condition_mod_item (41)) qui font référence aux noms des items auxquels on crée un item père commun. On insère un nouveau nœud entre ces nœuds frères et leur ancien nœud père. Ils sont donc tous fils d'un nouveau nœud qui est cond_création_plus_d'un_item (38). On crée un nœud condition_ création_item (ou condition_mod_item) père de ce nœud cond_création_plus_d'un_item. Ce dernier nœud père créé représente le nouvel item obtenu suite à l'ajout d'un niveau de décomposition. Il fait donc référence à une nouvelle entrée de la table des symboles, contenant le nom du nouvel item créé.

Si N (voir paragraphe ci-dessus) = 1, la création de nœud cond_création_plus_d'un_item n'est pas effectuée. Le nœud représentant l'item auquel on ajoute un père devient directement fils du nouveau nœud représentant l'item ajouté.

Exemple

(: nom = 'Dupont') and (premier_prenom = 'Jacques') où nom et premier_prenom sont les items que l'on regroupe sous un père commun (soit identite)

DEVIENT

(: identite ((: nom = 'Dupont') and (premier_prenom = 'Jacques')))

Cela se représente par :

... Δ condition_création_item (33) fait référence à "nom" dans la table des symboles.
ΔΔ constante_caractère (83) fait référence à 'Dupont' dans la table des constantes.
Δ condition_création_item (33) fait référence à "premier_prenom" dans la table des symboles.
ΔΔ constante_caractère (83) fait référence à 'Jacques' dans la table des constantes.

DEVIENT

... Δ condition_création_item (33) fait référence à une nouvelle entrée de la table des symboles, contenant "identite"
ΔΔ cond_création_plus_d'un_item (38)
ΔΔΔ condition_création_item (33) fait référence à "nom" dans la table des symboles.
ΔΔΔΔ constante_caractère (83) fait référence à 'Dupont' dans la table des constantes.
ΔΔΔ condition_création_item (33) fait référence à "premier_prenom" dans la table des symboles.
ΔΔΔΔ constante_caractère (83) fait référence à 'Jacques' dans la table des constantes.

L'exemple est analogue dans le cas des conditions de modification.

Dans le cas des conditions portant sur un ou des items lors d'un accès par clé (dans un chemin ou non), le raisonnement est différent.

Soient N (N >= 1) nœuds frères cond_sélection_item (67) représentant les items sur lesquels porte la condition de sélection, et à qui on ajoute un père commun. On leur crée un nouveau nœud père conditions_sélection_items (66). On crée un père suite_cond_sélection_item (68) au nœud père créé. On crée de plus un nouveau nœud père à ce suite_cond_sélection_item : cond_sélection_item (67). Ce dernier cond_sélection_item représente l'item ajouté; il fait donc référence à une nouvelle entrée de la table des symboles, contenant le nom de cet item. Il est de plus rattaché au père des N nœuds frères cond_sélection_item initiaux.

Exemple

(: nom = 'Dupont') and (premier_prenom = 'Jacques') où nom et premier_prenom sont les items que l'on regroupe sous un père commun (soit identite)

DEVIENT

(: identite ((: nom = 'Dupont') and (premier_prenom = 'Jacques')))

Cela se représente par :

... Δ cond_sélection_item (67) fait référence à "nom" dans la table des symboles
ΔΔ suite_cond_sélection_item (68)
ΔΔΔ égal (71)
ΔΔΔ constante_caractère (83) fait référence à 'Dupont' dans la table des constantes
Δ cond_sélection_item (67) fait référence à "premier_prenom" dans la table des symboles
ΔΔ suite_cond_sélection_item (68)
ΔΔΔ égal (71)
ΔΔΔ constante_caractère (83) fait référence à 'Jacques' dans la table des constantes

DEVIENT

... Δ cond_sélection_item (67) fait référence à une nouvelle entrée dans la table des symboles, contenant "identite"
ΔΔ suite_cond_sélection_item (68)
ΔΔΔ conditions_sélection_items (66)
ΔΔΔΔ cond_sélection_item (67) fait référence à "nom" dans la table des symboles
ΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔ constante_caractère (83) fait référence à 'Dupont' dans la table des constantes
ΔΔΔΔ cond_sélection_item (67) fait référence à "premier_prenom" dans la table des symboles
ΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔ constante_caractère (83) fait référence à 'Jacques' dans la table des constantes

Règle 2

(<varref>) . <items>

DEVIENT

(<varref>) . <nouvel item>. <items>

Transformation en termes d'arbre

Soit un nœud variable_non_hiérar (51) faisant référence, dans la table des symboles, à un nom d'item auquel on ajoute un père (c'est le premier nom d'item apparaissant dans <items>). Ce nœud a un nœud père class_var (50). On crée un père et un frère à class_var. Le nœud père est un nœud class_var. Le nœud frère est un nœud variable_non_hiérar qui fait référence à une nouvelle entrée de la table des symboles, contenant le nom de l'item créé pour l'ajout d'un niveau de décomposition.

Exemple

(pers).nom où on veut ajouter un item père "identite" à l'item "nom"

DEVIENT

(pers).identite.nom

Cela se représente par :

... Δ var_refs (53) fait référence à "pers" dans la table des symboles.
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence à "nom" dans la table des symboles

DEVIENT

... Δ var_refs (53) fait référence à "pers" dans la table des symboles.
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence à "identite" dans la table des symboles
ΔΔΔ class_var (50)
ΔΔΔΔ variable_non_hiérar (51) fait référence à "nom" dans la table des symboles

3. ELIMINATION D'UN TYPE DE CHEMINS PAR DUPLICATION D'ITEM

Règle 1

[for] <varrefc> := <tart> (<lien> : <varrefo>)

DEVIENT

[for] <varrefc> := <tart> (: <itemc> = (<varrefo>) . <itemo>)

Transformation en termes d'arbre

La condition d'accès de la forme initiale est représentée par :

... Δ condition_sur_item (74)
ΔΔ nom (8) fait référence au nom de type de chemins, ou de rôle "<lien>"
ΔΔ nom (8) fait référence au nom de variable de référence "<varrefo>"

DEVIENT

... Δ conditions_sélection_items (66)
ΔΔ cond_sélection_item (67) fait référence au nom d'item "<itemc>"
ΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔ égal (71)
ΔΔΔΔ var_refs (53) fait référence au nom de variable de référence "<varrefo>"
ΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item "<itemo>"

La condition d'accès par chemin "(<lien> : <varrefo>)" est transformée en une condition d'accès par clé : "(:<itemc> = (<varrefo>).<itemo>).

Le sous-arbre ayant comme racine le nœud condition_sur_chemin (74), et un premier nœud nom (8) faisant référence au nom du type de chemins éliminé ou à un nom de rôle dans ce type de chemins, disparaît. Le sous-arbre dont la racine est conditions_sélection_items (66) prend la place du sous-arbre détruit.

Règle 2

[for] <varrefc> := <tart> ((<lien> : <varrefo>) and <accès par clé>)

DEVIENT

[for] <varrefc> := <tart> ((: <itemc> = (<varrefo>) . <itemo>) and <accès par clé>)

Transformation en termes d'arbre

La condition portant sur le chemin dans la condition d'accès par clé dans un chemin, est représentée de la façon suivante;

... Δ condition_sur_clé_dans_un_chemin (75)
ΔΔ condition_sur_chemin (74)
ΔΔΔ nom (8) fait référence au nom de type de chemins ou de rôle "<lien>"
ΔΔΔ nom (8) fait référence au nom de variable de référence "<varrefo>"
ΔΔ conditions_sélection_items (66) racine du sous-arbre représentant l'<accès par clé>

...

DEVIENT

... Δ conditions_sélection_items (66)
ΔΔ cond_sélection_item (67) fait référence au nom d'item "<itemc>"
ΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔ égal (71)
ΔΔΔΔ var_refs (53) fait référence au nom de variable de référence "<varrefo>"
ΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item "<itemo>"
...

La condition d'accès par clé dans un chemin "((<lien> : <varrefo>) and <accès par clé>)" devient une condition d'accès par clé sur base de la clé initiale et de la condition sur item remplaçant l'accès par chemin éliminé : "((: <itemc> = (<varrefo>).<itemo>) and <accès par clé>)".

Dès lors, le sous-arbre ayant condition_sur_clé_dans_un_chemin (75) comme racine, et représentant l'accès par clé dans un chemin, voit sa racine et son sous-arbre représentant l'accès par chemin, éliminés.

Le nœud conditions_sélection_items (66) de ce sous-arbre, qui est racine du sous-arbre représentant l'accès par clé (dans le chemin), se voit attribuer un nouveau premier fils cond_ sélection_item (67).

Ce nœud cond_sélection_item est lui–même racine d'un sous-arbre représentant la condition sur item qui remplaçe l'accès par le chemin éliminé.

Le nœud conditions_sélection_items est rattaché au nœud père de condition_sur_clé_dans_ un_chemin.

Règle 3

modify <varrefc> (<lien> : <varrefo>)

DEVIENT

modify <varrefda> (: <itemda> = (<varrefdé>) . <itemdé>)

Transformation en termes d'arbre

... Δ instr_modify (39) fait référence au nom de variable de référence "<varrefc>"
ΔΔ cond_attach (43)
ΔΔΔ nom (8) fait référence au nom de type de chemins ou de rôle "<lien>"
ΔΔΔ nom (8) fait référence au nom de variable de référence "<varrefo>"

DEVIENT

... Δ instr_modify (39) fait référence au nom de variable de référence "<varrefda>"
ΔΔ condition_mod_item (41) fait référence au nom d'item "<itemda>"
ΔΔΔ var_refs (53) fait référence au nom de variable de référence "<varrefdé>"
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item "<itemdé>"

La condition de modification sur chemin "(<lien>:<varrefo>)" devient une condition de modification d'item "(:<itemda> = (<varrefdé>).<itemdé>)".

Dès lors, le sous-arbre ayant cond_attach (43) comme racine, et un premier nœud nom (8) référençant le nom du type de chemins éliminé ou le nom d'un rôle dans ce type de chemins, disparaît.

Ce sous-arbre est remplacé par celui dont la racine est condition_mod_item (41). Ce dernier sous-arbre représentant la condition de modification d'item est rattaché au nœud instr_ modify (39).

Ce nœud instr_modify référence le nom de variable de référence "<varrefda>".

Règle 4

create <varrefc> := <tart> (<lien> : <varrefo>);

DEVIENT

create <varrefc> := <tart> ( );
modify <varrefda> (: <itemda> = (<varrefdé>) . <itemdé>) ;

Transformation en termes d'arbre

... Δ instr_create (30)
ΔΔ nom (8) fait référence au nom de variable de référence "<varrefc>"
ΔΔ nom (8) fait référence au nom de type d'articles "<tart>"
ΔΔ conditions_création (31), ce nœud est la racine du sous-arbre représentant les conditions de création de l'article considéré. Seul son premier (p. ex.) fils nous intéresse.
ΔΔΔ condition_création_chemin (32)
ΔΔΔΔ nom (8) fait référence au nom de type de chemins ou de rôle "<lien>"
ΔΔΔΔ nom (8) fait référence au nom de variable de référence "<varrefo>"

DEVIENT

... Δ instr_create (30)
ΔΔ nom (8) fait référence au nom de variable de référence "<varrefc>"
ΔΔ nom (8) fait référence au nom de type d'articles "<tart>"
ΔΔ conditions_création (31) est la racine du sous-arbre représentant les conditions de création restant dans la forme transformée (peut–être aucune)
...
Δ instr_modify (39) fait référence au nom de variable de référence "<varrefda>"
ΔΔ condition_mod_item (41) fait référence au nom d'item "<itemda>"
ΔΔΔ var_refs (53) fait référence au nom de variable de référence "<varrefdé>"
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item "<itemdé>"

Le sous-arbre ayant condition_création_chemin (32) comme racine, et un premier nœud nom (8) référençant le nom du type de chemins éliminé, ou un nom de rôle dans ce type de chemins, disparaît. Ce sous-arbre représente la condition de création sur chemin. On a enlevé un fils de conditions_création (31).

Un sous-arbre ayant instr_modify (39) comme racine est ajouté comme frère suivant du sous-arbre de racine instr_create. Ce sous-arbre de racine instr_modify représente l'instruction de modification apparue suite à la transformation.

Règle 5

modify <varrefc> (<lien> : 0 <varrefo>)

DEVIENT

modify <varref> (: <item> = NULL)

Transformation en termes d'arbre

... Δ instr_modify (39) fait référence au nom de variable de référence "<varrefc>"
ΔΔ cond_detach (42)
ΔΔΔ nom (8) fait référence au nom de type de chemins ou de rôle "<lien>"
ΔΔΔ nom (8) fait référence au nom de variable de référence "<varrefo>"

DEVIENT

... Δ instr_modify (39) fait référence au nom de variable de référence "<varref>"
ΔΔ condition_mod_item (41) fait référence au nom d'item "<item>"
ΔΔΔ valeur_null (47)

Le sous-arbre de racine cond_detach (42) ayant un premier nœud nom (8) qui référence le type de chemins éliminé, ou un rôle dans ce type de chemins, disparaît.

Il est remplacé par un sous-arbre de racine condition_mod_item (41), représentant la condition de modification d'item. Ce dernier sous-arbre est rattaché, en toute logique, au nœud instr_modify (39).

Le nœud instr_modify référence le nom de variable de référence "<varref>".

Règle 6

modify <varref> (: <item> = <valeur>) ;

DEVIENT

```
for <varrefc> := <tart> ( : <itemc> = ( <varref> ) . <item> )
    modify <varrefc> ( : <item> = <valeur> )
endfor ;
modify <varref> ( : <item> = <valeur> ) ;
```

Transformation en termes d'arbre

Le nœud instr_modify (39) dont le nœud fils condition_mod_item (41) fait référence au nom de l'item dupliqué, se voit attribuer un frère précédent.

Ce frère précédent est un nœud instr_for (22) racine d'un sous-arbre représentant l'instruction "for" de la forme transformée.

Exemple

Cet exemple est basé sur la figure 4.3.

```
modify pers1 ( : numpers = 250 ) ;

DEVIENT

for l := livre ( : numemprunteur = (pers1).numpers )
    modify l ( : numemprunteur = 250 )
endfor;
modify pers1 ( : numpers = 250 ) ;
```

Cette transformation se traduit en termes d'arbre, par :

```
... Δ instructions (19)
   ...
   ΔΔ instr_modify (39) fait référence au nom de variable de référence "pers1"
   ΔΔΔ condition_mod_item (41) fait référence au nom d'item "numpers"
   ΔΔΔΔ expression_arithmétique (54) racine du sous-arbre représentant 250

                              DEVIENT

... Δ instructions (19)
   ...
   ΔΔ instr_for (22)
   ΔΔΔ class_var (50) racine du sous-arbre représentant "l"
   ...
   ΔΔΔ expression_collection (64) racine du sous-arbre représentant "livre ( : numemp-
       runteur = (pers1).numpers )"
   ...
   ΔΔΔ instructions (19) racine du sous-arbre représentant le corps de la boucle for
   ΔΔΔΔ instr_modify (39) racine du sous-arbre représentant "modify l ( : numemprunteur
       = 250 )"
   ...
   ΔΔ instr_modify (39) racine du sous-arbre représentant l'instruction initiale "modify
      pers1 ( : numemprunteur = 250 )"
```

Règle 7

I. delete <varref> ;

DEVIENT

```
for <varref1> := <tart> ( : <item1> = ( <varref> ) . <item2> )
    delete <varref1>
endfor ;
delete <varref> ;
```

Transformation en termes d'arbre

Soit un nœud instr_delete (44) faisant référence au nom d'une variable de référence du type d'articles origine du type de chemins 1–N ou 1–1 éliminé.

Ce nœud se voit attribuer un nœud frère précédent instr_for (22). Ce nœud instr_for est racine d'un sous-arbre représentant la boucle "for" de la forme transformée.

Exemple

Cet exemple est basé sur la figure 4.6.

```
delete cli ;

DEVIENT

for com := commande ( : NCLI' = ( cli ) . NCLI )
    delete com
endfor ;
delete cli ;
```

Cette transformation se traduit en termes d'arbre, par :

... Δ instructions (19)
...
ΔΔ instr_delete (44) fait référence au nom de variable de référence "cli"

DEVIENT

... Δ instructions (19)
...
ΔΔ instr_for (22)
ΔΔΔ class_var (50) racine du sous-arbre représentant la variable de parcours "com"
...
ΔΔΔ expression_collection (64) racine du sous-arbre représentant "commande (: NCLI' = (cli) . NCLI)"
...
ΔΔΔ instructions (19)
ΔΔΔΔ instr_delete (44) fait référence au nom de variable de référence "com"
ΔΔ instr_delete (44) fait référence au nom de variable de référence "cli"

II. delete <varref> ;

DEVIENT

delete <varref> ;

Transformation en termes d'arbre

Le nœud instr_delete (44) représentant l'instruction "delete <varref>" ne subit aucun traitement.

III. delete <varref> ;

DEVIENT

```
for <varref1> := <tart> ( : <itemc> = ( <varref> ) . <itemo> )
    modify <varref1> ( : <itemc> = NULL )
endfor ;
delete <varref> ;
```

Transformation en termes d'arbre

Soit un nœud instr_delete (44) faisant référence au nom d'une variable de référence du type d'articles associé à l'item dupliqué.

Ce nœud se voit attribuer un nœud frère précédent instr_for (22). Ce nœud instr_for est racine d'un sous-arbre représentant la boucle "for" de la forme transformée.

Exemple

Cet exemple est basé sur la figure 4.6.

```
delete cli ;
```

DEVIENT

```
for com := commande ( : NCLI' = ( cli ) . NCLI )
    modify com ( : NCLI' = NULL )
endfor ;
delete cli ;
```

Cette transformation se traduit en termes d'arbre, par :

```
... Δ  instructions  (19)
   ...
   ΔΔ instr_delete (44) fait référence au nom de variable de référence "cli"
```

DEVIENT

```
... Δ  instructions  (19)
   ...
   ΔΔ  instr_for  (22)
   ΔΔΔ class_var (50) racine du sous-arbre représentant la variable de parcours "com"
   ...
   ΔΔΔ    expression_collection (64) racine du sous-arbre représentant "commande ( : NCLI' =
          ( cli) . NCLI )"
   ...
   ΔΔΔ  instructions  (19)
   ΔΔΔΔ instr_modify (39) racine du sous-arbre représentant "modify com ( : NCLI' =NULL )"
   ...
   ΔΔ instr_delete (44) fait référence au nom de variable de référence "cli"
```

On remarque que la transformation est similaire à celle développée lors du premier cas de cette règle. Seule la destruction est remplacée par la mise à NULL.

4. INSERTION D'UN TYPE D'ARTICLES

Règle 1

I. [for] <varrefc> := <tart> (<chemin> : <varrefo>)
...
[endfor]

DEVIENT

for <varrefint> := <tartint> (<chemin2> : <varrefo>)
<varrefc> := <tart> (<chemin1> : <varrefint>) ;
...
endfor ;

Transformation en termes d'arbre

Soit la forme initiale est une assignation. Elle est représentée par un nœud assignation (20). Ce nœud est le fils d'un nœud instructions (19). On enlève ce fils à instructions et on le remplace par un nœud instr_for (22) représentant la forme transformée.

On suppose la représentation de ce "for" connue. On peut toutefois préciser que le nœud instr_for a un premier fils class_var (50) qui représente <varrefint>, un second fils expression_collection (64) qui représente "<tartint> (<chemin2> : <varrefo>)", et un troisième fils instructions qui a deux fils. Le premier est un nœud assignation qui représente "<varrefc> := <tart> (<chemin1> : <varrefint>)".

Notons que pour ce nœud assignation, on peut récupérer le nœud représentant l'assignation initiale, et le modifier. Il a en effet un fils expression_assignation (21) qui a lui-même un fils expression_collection (60). Il suffit de modifier ce nœud expression_collection pour qu'il représente "<tart> (<chemin1> : <varrefint>)" plutôt que l'expression de collection de la forme initiale "<tart> (<chemin> : <varrefo>)".

Le second fils du nœud instructions est un nœud instr_exit (28) qui représente l'instruction "exit <varrefint>" nécessaire dans le cas d'une assignation.

Exemple

Cet exemple est basé sur la transformation exposée à la figure 4.7 (paragraphe 4.2, règle4).

```
c := commande (cp  : prod);

DEVIENT

for co_pr := com_pro (PCP : prod)
    c := commande (CCP : co_pr);
    exit co_pr
endfor;
```

Cela se représente par :

... Δ instructions (19)
...
ΔΔ assignation (20) est la racine d'un sous-arbre représentant "c := commande (cp : prod)". Le détail de cette représentation n'apporte aucune information à ce niveau; il n'est donc pas donné. Le lecteur intéressé le déduira aisément des règles de représentation décrites au paragraphe 6.3.

DEVIENT

... Δ instructions (19)
...
ΔΔ instr_for (22)
ΔΔΔ class_var (50) est la racine d'un sous-arbre représentant la variable de référence intermédiaire "co_pr"
...
ΔΔΔ expression_collection (64) est la racine d'un sous-arbre représentant "com_pro (PCP : prod)"
...
ΔΔΔ instructions (19)
ΔΔΔΔ assignation (20) est la racine d'un sous-arbre représentant "c := commande (CCP : co_pr)"
...
ΔΔΔΔ instr_exit (28) est la racine d'un sous-arbre représentant "exit co_pr"
...

Dans le cas d'une forme initiale spécifiant une instruction "for", le raisonnement est différent.

Cette instruction est représentée par un nœud instr_for. On modifie le premier fils (class_var (50)) de instr_for pour qu'il représente <varrefint> au lieu de <varrefc>. On modifie le second fils (expression_collection (64)) pour qu'il représente "<tartint> (<chemin2> : <varrefo>)" au lieu de "<tart> (<chemin> : <varrefo>)". Le troisième fils (instructions (19)) de instr_for est modifié de la façon suivante. On lui ajoute un premier fils assignation (20) représentant "<varrefc> := <tart> (<chemin1> : <varrefint>)". De plus, tous les nœuds instr_next (27) ou instr_exit (28) apparaissant dans le sous-arbre dont instructions est racine, sont modifiés également s'ils ont un fils nom (8). Ce nom qui initialement référence <varrefc> doit à présent référencer <varrefint>.

Exemple

Cet exemple est basé sur la transformation exposée à la figure 4.7 (paragraphe 4.2, règle4).

for c := commande (cp : prod)
    <trt>
endfor;
où <trt> est une séquence d'instructions

DEVIENT

```
for co_pr := com_pro (PCP : prod)
    c := commande (CCP : co_pr);
    <trt>
endfor;
```

Cela se représente par :

... Δ instr_for (22)
ΔΔ class_var (50) est racine d'un sous-arbre représentant "c"
...
ΔΔ expression_collection (64) est racine d'un sous-arbre représentant "commande (cp : prod)"
...
ΔΔ instructions (19) est racine d'un sous-arbre représentant <trt>

DEVIENT

... Δ instr_for (22)
ΔΔ class_var (50) est racine d'un sous-arbre représentant "co_pr"
...
ΔΔ expression_collection (64) est racine d'un sous-arbre représentant "com_pro (PCP : prod)"
...
ΔΔ instructions (19)
ΔΔΔ assignation (20) est racine d'un sous-arbre représentant "c := commande (CCP : co_pr)". Les autres fils d'instructions représente <trt>.

II. [for] <varref> := <tart> <condition d'accès portant sur un item>
...
[endfor]

DEVIENT

```
for <varrefint> := <tartint> <condition d'accès portant sur un item>
    <varrefc> := <tart> (<chemin1> : <varrefint> );
    ...
endfor
```

Dans le cas d'un accès par clé, le raisonnement est analogue à celui tenu pour l'accès par chemin. Seule la condition d'accès initial est différente ainsi que la condition portant sur le for dans la forme transformée.

Règle 2

I. create<varrefc>:= <tart> (<chemin> : <origine>);

DEVIENT

```
create <varrefc> := <tart> () ;
create <varrefint> := <tartint> ( (<chemin1> : <varrefc>) and (<chemin2> : <origine>) );
```

Transformation en termes d'arbre

La transformation étant relativement simple, elle peut aisément être expliquée en termes d'arbre.

La forme initiale est représentée par :

... Δ instr_create (30)
ΔΔ nom (8) fait référence à "<varrefc>" dans la table des symboles
ΔΔ nom (8) fait référence à "<tart>" dans la table des symboles
ΔΔ conditions_création (31). Ce nœud a un nombre quelconque de fils représentant chacun une condition élémentaire de création. Le cas présent, seul celui représentant "(<chemin> : <origine>)" nous intéresse.
...
ΔΔΔ condition_création_chemin (32)
ΔΔΔΔ nom (8) fait référence à "<chemin>" dans la table des symboles
ΔΔΔΔ nom (8) fait référence à "<origine>" dans la table des symboles

DEVIENT

... Δ instr_create (30)
ΔΔ nom (8) fait référence à "<varrefc>" dans la table des symboles
ΔΔ nom (8) fait référence à "<tart>" dans la table des symboles
ΔΔ conditions_création (31)
...
Δ instr_create (30). Ce nœud représente l'instruction de création " create <varrefint> := <tartint> ((<chemin1> : <varrefc>) and (<chemin2> : <origine>))".

On constate que le nœud condition_création_chemin représentant "(<chemin> : <origine>)" dans la forme initiale, a été enlevé des fils de conditions_création. La condition sur chemin n'apparaît donc plus dans la condition de création de l'article initial.

On a adjoint un frère au nœud instr_create représentant la création de l'article initial. Ce nœud, instr_create également, représente la création de l'article intermédiaire qui rend compte du chemin N-N initial. On a donc enlevé un petit-fils à instr_create (initial) pour lui ajouter un frère.

Notons que les deux frères instr_create dans la représentation de la forme transformée, doivent être contigus. En d'autres termes, ils sont tous deux fils d'un nœud instructions (19), et si le nœud instr_create initial est le $i^{ème}$ fils d'instructions, le nœud instr_create ajouté après transformation doit être le $i+1^{ème}$.

II. create <varrefc>:= <tart> (:<item> = {<liste_expr>});
Il s'agit ici de la création d'article associé à des valeurs d'item élémentaire répétitif

DEVIENT

create <varrefc> := <tart> () ;
create <varrefint> := <tartint> ((<chemin1> : <varrefc>) and (:<item> = <valeur>));
Cette dernière instruction apparaît autant de fois qu'il y a de valeurs dans <liste_expr>. Elle apparaît une seule fois si <item> n'apparaît pas dans les conditions de création initiales.

Transformation en termes d'arbre

Le parallèle avec le point précédent (création avec condition sur chemin) est évident. La condition de création initiale disparaît. On enlève donc le fils condition_création_item (33) représentant cette condition, du nœud conditions_création. Ce dernier nœud est le troisième fils du nœud instr_create (30) représentant l'instruction de création initiale (création de <varrefc>). Cette condition de création éliminée se traduit en N créations d'articles. On ajoute donc N frères suivants contigus, instr_create, au nœud instr_create représentant la création initiale. Chaque nœud frère ajouté représente une création d'article <tartint>.

Exemple

Cet exemple est basé sur la transformation exposée à la figure 4.12 (paragraphe 4.2, transformation 4).

```
create pers := personne (: prenom = {'René', 'Alexandre', 'Arthur'} );
```

DEVIENT

```
create pers := personne ();
create pr_pr := pre_pers ( (PPP : pers) and (:prenom ='René') );
create pr_pr := pre_pers ( (PPP : pers) and (:prenom ='Alexandre') );
create pr_pr := pre_pers ( (PPP : pers) and (:prenom ='Arthur') );
```

Cela se représente par :

```
... Δ instructions (19)
    ...
    ΔΔ instr_create (30)
    ΔΔΔ nom (8) fait référence à "pers" dans la table des symboles
    ΔΔΔ nom (8) fait référence à "personne" dans la table des symboles
    ΔΔΔ conditions_création (31). Ce nœud est racine du sous-arbre représentant la
        condition de création de "pers"
    ...
    ΔΔΔΔ condition_création_item (33) est la racine du sous-arbre représentant "prenom =
        {'René', 'Alexandre', 'Arthur'}"
```

DEVIENT

```
... Δ instructions (19)
    ...
    ΔΔ instr_create (30)
    ΔΔΔ nom (8) fait référence à "pers" dans la table des symboles
    ΔΔΔ nom (8) fait référence à "personne" dans la table des symboles
    ΔΔΔ conditions_création (31)
    ...
    ΔΔ instr_create (30) est racine du sous-arbre représentant "create pr_pr := pre_pers
        ( (PPP : pers) and (:prenom ='René') );"
    ...
    ΔΔ instr_create (30) est racine du sous-arbre représentant "create pr_pr := pre_pers
        ( (PPP : pers) and (:prenom ='Alexandre') );"
    ...
    ΔΔ instr_create (30) est racine du sous-arbre représentant "create pr_pr := pre_pers
        ( (PPP : pers) and (:prenom ='Arthur') );"
    ...
```

Le raisonnement est identique dans la cas de la création d'un article associé à des valeurs d'item répétitif décomposable.

Règle 3

modify <varrefc> (<chemin> : <varrefo>) ;

DEVIENT

create <varref-int> := <tart-int>((<chemin1> : <varrefc>) and(<chemin2> : <varrefo>));

Transformation en termes d'arbre

L'expression de cette transformation en termes d'arbre est extrêmement aisée. Il suffit en effet de détruire le sous-arbre représentant la forme initiale, pour le remplacer par le sous-arbre représentant la forme transformée. Plus précisément, le nœud instr_modify (39) racine du sous-arbre représentant la forme initiale, est enlevé de l'arbre avec tous ses descendants. Il est remplacé par le nœud instr_create (30) et tous ses descendants, qui forment le nouveau sous-arbre représentant l'instruction de création de la forme transformée.

Règle 4

modify <varrefc1> (<chemin> : 0 <varrefo>);

DEVIENT

for <varref-int> := <tart-int> (<chemin1> : <varrefo>)
    <varrefc2> := <tartc> (<chemin2> : <varref-int>);
    if <varrefc2> = <varrefc1> then exit <varref-int>
    endif
endfor;
delete <varref-int>;

Transformation en termes d'arbre

Le nœud instr_modify (39) et tous ses descendants composent le sous-arbre représentant la modification initiale. Ce sous-arbre est détruit et est remplacé par deux autres sous-arbres. L'un a pour racine un nœud instr_for (22) et représente l'instruction "for" de recherche de l'article intermédiaire. L'autre a pour racine instr_delete (44) et représente l'instruction de destruction de l'article intermédiaire (delete <varref-int>). Les deux racines instr_for et instr_delete sont des nœuds contigus; instr_for est le premier des deux.

Règle 5

modify <varrefc> (<rôle> : 0 <varrefo>);

DEVIENT

<varref-int> := <tart-int> (<chemin> : <varrefo>);
delete <varref-int>;

Transformation en termes d'arbre

Le raisonnement est identique à celui tenu dans la règle 4 étudiée ci-dessus. Un sous-arbre représentant l'instruction initiale est remplacé par deux sous-arbres représentant chacun une des instructions de la forme transformée. Le premier de ces deux sous-arbres diffère de celui exposé à la règle 4 en ce sens que, le cas présent, l'instruction représentée est une simple assignation. D'où l'utilisation d'un nœud assignation (20).

Règle 6

I. modify <varref> (: <item> + {<valeur>});
Il s'agit ici de l'ajout d'une nouvelle valeur d'item répétitif élémentaire à celles déjà associées à un article

DEVIENT

modify <varref> (...);
create <varref-int> := <tart-int> ((<chemin> : <varref>) and <condition sur item>);

Transformation en termes d'arbre

... Δ instr_modify (39) fait référence à "<varref>" dans la table des symboles
ΔΔ conditions_mod_items (40) a N fils représentant les conditions portant sur les items transformés. Un seul nous intéresse.
ΔΔΔ condition_mod_item (41) fait référence à "<item" dans la table des symboles
ΔΔΔΔ plus (59)
ΔΔΔΔ expression _arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47), racine du sous-arbre représentant la nouvelle valeur (<valeur>) d'item associée à l'article

DEVIENT

... Δ instr_modify (39) est racine du sous-arbre représentant "modify <varref> (...);"
...
Δ instr_create (30) est racine du sous-arbre représentant "create <varref-int> := <tart-int> ((<chemin> : <varref>) and <condition sur item>);"
...

On a enlevé le nœud condition_mod_item (41) de la forme initiale. Conditions_mod_items (40) a donc perdu un fils. S'il ne lui en reste qu'un, conditions_mod_items disparaît et le fils restant est directement attaché à instr_modify (39) car l'instruction de modification ne porte plus que sur un seul item. Si déjà initialement, le nœud instr_modify n'avait qu'un fils (condition_mod_item représentant "(: <item> + <valeur>)") et qu'on l'a enlevé, il n'a après transformation, plus de fils. Il n'a dès lors plus de raison d'être et est enlevé de l'arbre. Le cas présent, on suppose qu'il a encore un fils. On a créé un frère contigu au nœud instr_modify. Ce nœud frère instr_create (30) rend compte de l'instruction de création de l'article intermédiaire après transformation. De ces deux nœuds, instr_modify est le premier.

Le raisonnement est identique dans la cas où on ajoute une valeur d'item décomposable. Seules les <condition sur item> dans la forme transformée, diffèrent.

II. modify <varref> (: <item> - {<valeur>});
Il s'agit ici du retrait d'une valeur d'item élémentaire répétitif associée à un article

DEVIENT

```
modify <varref> (...);
for <varref-int> := <tart-int> (<chemin> : <varref>)
    if <cond> then exit <varref-int>
    endif
endfor;
delete <varref-int>;
```

Transformation en termes d'arbre

... Δ instr_modify (39) fait référence à "<varref>" dans la table des symboles
ΔΔ conditions_mod_items (40) a N fils représentant les conditions portant sur les items transformés. Un seul nous intéresse.
ΔΔΔ condition_mod_item (41) fait référence à "<item" dans la table des symboles
ΔΔΔΔ moins(60)
ΔΔΔΔ expression _arithmétique (54) ou class_var (50) ou var_refs (53) ou constante_caractère (83) ou appel (29) ou true (45) ou false (46) ou valeur_null (47), racine du sous-arbre représentant la valeur (<valeur>) d'item que l'on veut dissocier de l'article

DEVIENT

... Δ instr_modify (39) est racine du sous-arbre représentant "modify <varref> (...);"
...
Δ instr_for (22) est racine du sous-arbre représentant l'instruction "for" dans la forme transformée
...
Δ instr_delete (44) est racine du sous-arbre représentant la destruction de l'article intermédiaire : "delete <varref-int>"

La façon dont le nœud instr_modify (39) est modifié ,voire enlevé, de l'arbre, est similaire à celle exposée lors de l'ajout d'une valeur d'item (cas précédent).

On lui ajoute cependant, le cas présent, deux frères contigus instr_for (22) et instr_delete (44). De ces trois nœuds frères, instr_modify est le premier, instr_for le second et instr_delete le troisième.

Le raisonnement est identique dans le cas où on enlève une valeur d'item décomposable. Seul le <cond> dans la forme transformée diffère.

III. modify <varref> (:<item> = {<liste-expr>});
Il s'agit ici d'associer un article à une liste de valeurs d'item répétitif élémentaire

DEVIENT

```
modify <varref> (...);
for <varref-int> := <tart-int> (<chemin> : <varref>)
    delete <varref-int>;
endfor;
create <varref-int> := <tart-int> ((<chemin> : <varref>) and <condition sur item> );
```

Transformation en termes d'arbre

... Δ instr_modify (39) fait référence à "<varref>" dans la table des symboles
ΔΔ conditions_mod_items (40)
ΔΔΔ condition_mod_item (41) est racine du sous-arbre représentant "(: <item> = {<liste_expr>});

DEVIENT

... Δ instr_modify (39) est racine du sous-arbre représentant "modify <varref> (...);"
...
Δ instr_for (22) est racine du sous-arbre représentant l'instruction "for" de la forme transformée
...
Δ instr_create (30) est racine du sous-arbre représentant "create <varref-int> := <tart-int> ((<chemin> : <varref>) and <condition sur item>);"

La façon dont le nœud instr_modify (39) est modifié, est identique à celle exposée lors de l'ajout d'une valeur d'item répétitif (cfr premier cas de cette règle). On lui ajoute un nœud frère contigu instr_for (22). On ajoute à instr_for N nœuds frères contigus instr_create (30). Chacun de ces nœuds représente une instruction de création d'un article intermédiaire. Il y en a autant qu'il y a de valeurs dans la liste <liste_expr> de la forme initiale. Dans ces nœuds frères, instr_modify est le premier, instr_for le second, et les instr_create sont troisième, quatrième,... .

Règle 7

```
for <varrefc> := <tart> ((<chemin> : <varrefo>) and <condition sur item>)
    <trt>
endfor ;
```

DEVIENT

```
for <varref-int> := <tart-int> (<chemin1> : <varrefo>)
    <varrefc> := <tart> (<chemin2> : <varref-int>);
    if <cond> then <trt>
    endif
endfor;
```

Transformation en termes d'arbre

... Δ instr_for (22)
ΔΔ class_var (50) est racine d'un sous-arbre représentant "varrefc"
...
ΔΔ expression_collection (64) est racine d'un sous-arbre représentant "<tart> ((<chemin>: <varrefo>) and <condition sur item>)"
...
ΔΔ instructions (19) est racine du sous-arbre représentant <trt>
...

DEVIENT

... Δ instr_for (22)
ΔΔ class_var (50) est racine du sous-arbre représentant "<varref-int>"

...
ΔΔ expression_collection (64) est racine du sous-arbre représentant "<tart-int> (<chemin1> : <varrefo>)"
...
ΔΔ instructions (19) est racine du sous-arbre représentant le nouveau corps de la boucle "for" dans la forme transformée. Le premier fils de ce nœud est :
ΔΔΔ assignation (20) est racine du sous-arbre représentant "<varrefc> := <tart> (<chemin2> : <varref-int>);"

La transformation consiste à modifier le sous-arbre représentant le "for" initial pour en obtenir un représentant le "for" transformé. La variable de parcours et l'expression de collection sont modifiées.

Le nœud instructions (19) représente le nouveau corps de la boucle. Dans le sous-arbre dont cet instructions est racine, on trouve le nœud initial instructions racine du sous-arbre représentant <trt>.

Règle 8

I. for <variable> := (<varref>).<item>
<trt>
enfor;

DEVIENT

for <varref-int> := <tart-int> (<chemin> : <varref>)
<variable> := (<varref-int>).<item>;
<trt>
endfor;

Transformation en termes d'arbre

... Δ instr_for (22)
ΔΔ class_var (50) est racine du sous-arbre qui représente "<variable>"
...
ΔΔ var_refs (53) est racine du sous-arbre qui représente (<varref>).<item>
...
ΔΔ instructions (19) est racine du sous-arbre représentant <trt>

DEVIENT

Δ instr_for (22)
ΔΔ class_var (50) est racine du sous-arbre représentant "<varref-int>"
...
ΔΔ expression_collection (64) est racine du sous_arbre représentant "<tart-int> (<chemin> : <varref>)"
...
ΔΔ instructions (19) est racine du sous-arbre représentant le nouveau corps de la boucle "for" dans la forme transformée. Le premier fils de ce nœud est :
ΔΔΔ assignation (20) est racine du sous-arbre représentant "<variable> := (<varref-int>).<item>;"

Le nœud class_var (50) représentant la variable de parcours se voit modifié. Le nœud var_refs (53) est remplacé par un nœud expression_collection (64) qui représente la nouvelle

séquence parcourue, cela afin de rendre compte du fait qu'on accède maintenant à des articles. Le nœud instructions (19) représentant initialement <trt>, se voit ajouter un premier nœud fils assignation (20).

II.

Dans le cas d'un accès aux valeurs d'un item répétitif décomposable, le raisonnement est similaire avec toutefois quelques nuances.

```
for {<item décomposable>} := (<varref>).<item>
    <trt>
endfor;
```

DEVIENT

```
for <varref-int> := <tart-int> (<chemin> : <varref>)
    <variable> := (<varref-int>).<item>.<comp>;
    ...
    <trt>
endfor;
```

La variable de parcours initiale est ici représentée par un sous-arbre dont un nœud item_répétitif_décomposable (23) est racine.

Dans le sous-arbre représentant la forme transformée, on ajoute cette fois N nœuds fils contigus assignation (20) au nœud instructions (19) racine du sous-arbre représentant initialement <trt>. Ces nœuds sont les premier, second, troisième,..., $N^{ième}$ fils de instructions.

On peut signaler que si des nœuds instr_next (27) ou instr_exit (28) apparaissent dans le sous-arbre dont instructions est racine, ils doivent être modifiés. Plus précisément, les sous-arbres dont ces nœuds sont racines doivent être modifiés pour qu'ils représentent des next et exit portant sur la nouvelle variable de parcours. Cette remarque est valable pour les items élémentaires et décomposables.

Règle 9

<varrefc> := <tart> ((<chemin> : <varrefo>) and <condition sur item>)

DEVIENT

```
for <varref-int> := <tart-int> (<chemin1> : <varrefo>)
    <varrefc> := <tart> (<chemin2> : <varref-int>);
    if <cond> then exit <varref–int>
              else <varrefc> := ()
    endif
endfor;
```

Transformation en termes d'arbre

La forme initiale est représentée par un sous-arbre dont un nœud assignation (20) est racine. Ce sous-arbre (le nœud et tous ses descendants) est détruit et remplacé par le sous-arbre représentant la forme transformée. Ce second sous-arbre a pour racine instr_for.

Notons encore que certains sous-arbres du sous-arbre initial pourraient être récupérés pour apparaître dans le sous-arbre de la forme transformée (par exemple celui représentant <varrefc>).

5. INSERTION D'UN TYPE D'ARTICLES ET DUPLICATION D'ITEM

Règle 1

Les règles de transformation syntaxique peuvent être aisément déduites de celles définies à propos de l'insertion d'un type d'articles et de la rotation.

La règle est que selon une forme syntaxique initiale, on applique une des règles définies pour la transformation 4 (insertion d'un type d'articles). A la forme syntaxique obtenue, on applique une des règles de la transformation 3 (élimination d'un type de chemins par duplication d'un identifiant).

Il est important, cependant, de garder à l'esprit que la transformation totale exposée ici forme un tout indécomposable. Seule sa définition a été "morcelée" afin de tirer profit du travail effectué pour d'autres transformations.

Transformation en termes d'arbre

Le chapitre 4 définit en termes BNF, certaines règles de transformation de formes syntaxiques, comme la composition d'autres règles.

Comme la définition de ces règles en termes BNF, celles en termes d'arbre possède cette propriété d'être décomposable.

Aussi peut–on affirmer à nouveau que selon une forme syntaxique initiale exprimée cette fois en termes d'arbre, on applique une des règles définies (en termes d'arbre) pour la transformation 4. A l'expression en termes d'arbre ainsi obtenue, on applique une des règles (en termes d'arbre) de la transformation 3.

Le lecteur peut ainsi aisément déduire la transformation en termes d'arbre d'une forme syntaxique exprimée elle aussi de la sorte. Il importe cependant de rappeler que seule la définition d'une règle de transformation est morcelable en règles plus élémentaires. L'application d'une règle de transformation sur l'arbre syntaxique est, quant à elle, atomique.

Exemple

L'exemple suivant donne le détail de la transformation de l'arbre selon la découpe définie par les "étapes" de la transformation syntaxique.

Cet exemple est basé sur la figure 4.15.

```
for com := commande ( cp : pro )

DEVIENT selon la règle 1 de la transformation 4

for c := com_pro (PCP : pro )
    com := commande ( CCP : c )
```

En termes d'arbre, cette transformation se traduit par :

```
... Δ  instr_for  (22)
   ΔΔ  class_var  (50)
   ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "com"
   ΔΔ expression_collection (64) fait référence au nom de type d'articles "commande"
```

ΔΔΔ condition_sur_chemin (74)
ΔΔΔΔ nom (8) fait référence au nom de type de chemins "cp"
ΔΔΔΔ nom (8) fait référence au nom de variable de référence "pro"
ΔΔ instructions (19) racine du sous-arbre représentant le corps de la boucle "for"
...

DEVIENT

... Δ instr_for (22)
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "c"
ΔΔ expression_collection (64) fait référence au nom de type d'articles "com_pro"
ΔΔΔ condition_sur_chemin (74)
ΔΔΔΔ nom (8) fait référence au nom de type de chemins "PCP"
ΔΔΔΔ nom (8) fait référence au nom de variable de référence "pro"
ΔΔ instructions (19) racine du sous-arbre représentant le corps de la boucle "for", un nouveau premier fils lui est ajouté : assignation
ΔΔΔ assignation (20)
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "com"
ΔΔΔΔ expression_assignation (21)
ΔΔΔΔΔ expression_collection (64) fait référence au nom de type d'articles "commande"
ΔΔΔΔΔΔ condition_sur_chemin (74)
ΔΔΔΔΔΔΔ nom (8) fait référence au nom de type de chemins "CCP"
ΔΔΔΔΔΔΔ nom (8) fait référence au nom de variable de référence "c"

Suite à l'élimination des types de chemins CCP et PCP par duplication d'item, on obtient

```
for c := com_pro ( : NP' = ( pro ) . NP )
    com := commande ( : NC = ( c ) . NC' )
```

Selon la règle 1 de la transformation 3, les sous-arbres de racine condition_sur_chemin (74) doivent être remplacés par des sous-arbres de racine conditions_sélection_items (66). On obtient alors : (les triangles (Δ) en traits accentués (Δ) représentent les sous-arbres qui différencient les sous-arbres avant et après rotation des types de chemins CCP et PCP)

... Δ instr_for (22)
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "c"
ΔΔ expression_collection (64) fait référence au nom de type d'articles "com_pro"
ΔΔΔ conditions_sélection_items (66)
ΔΔΔΔ cond_sélection_item (67) fait référence au nom d'item " NP' "
ΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔ var_refs (53) fait référence au nom de variable de référence "pro"
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item "NP"
ΔΔ instructions (19) racine du sous-arbre représentant le corps de la boucle "for"
ΔΔΔ assignation (20)
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "com"
ΔΔΔΔ expression_assignation (21)
ΔΔΔΔΔ expression_collection (64) fait référence au nom de type d'articles "commande"
ΔΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔΔ cond_sélection_item (67) fait référence au nom d'item "NC"

ΔΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔΔ var_refs (53) fait référence au nom de variable de référence "com"
ΔΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item " NC' "

Il est important de rappeler une fois de plus que l'application de la transformation sur l'arbre syntaxique passe directement de la forme initiale à la forme finale. Cette transformation est divisée en étapes afin de clarifier l'exposé.

Règle 2

modify <varrefc> (<chemin> : 0 <varrefo>) ;

DEVIENT

<varref-int> := <tart-int> ((: <itemc'> = (<varrefc>) . <itemc>) and
(: <itemo'> = (<varrefo>) . <itemo>) ;
delete <varref-int> ;

Transformation en termes d'arbre

... Δ instructions (19)
...
ΔΔ instr_modify (39) fait référence au nom de variable de référence "<varrefc>"
ΔΔΔ cond_detach (42)
ΔΔΔΔ nom (8) fait référence au nom de type de chemins "<chemin>"
ΔΔΔΔ nom (8) fait référence au nom de variable de référence "<varrefo>"

DEVIENT

... Δ instructions (19)
...
ΔΔ assignation (20)
ΔΔΔ class_var (50)
ΔΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "<varref-int>"
ΔΔΔ expression_assignation (21)
ΔΔΔΔ expression_collection (64) fait référence au nom de type d'articles "<tart-int>"
ΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔ cond_sélection_item (67) fait référence au nom d'item "<itemc'>"
ΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔ var_refs (53) fait référence au nom de variable de référence "<varrefc>"
ΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item "<itemc>"
ΔΔΔΔΔΔ cond_sélection_item (67) fait référence au nom d'item "<itemo'>"
ΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔ var_refs (53) fait référence au nom de variable de référence "<varrefo>"
ΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item "<itemo>"
ΔΔ instr_delete (44) fait référence au nom de variable de référence <varref-int>

Le sous-arbre de racine instr-modify (39), représentant l'instruction modify initiale, disparaît et est remplacé par deux sous-arbres. Un sous-arbre de racine assignation (20) représente l'instruction d'assignation dans la forme transformée. L'autre sous-arbre est constitué d'un seul nœud instr_delete (44) et représente l'instruction delete de la forme transformée.

Ces deux sous-arbres sont rattachés au père de instr_modify.

6. INSERTION D'UN TYPE D'ARTICLES, ROTATION ET AJOUT D'UN NIVEAU DE DECOMPOSITION

Règle 1

Le principe de composition des règles exposé à la transformation précédente est toujours valable.

La règle de transformation est que selon une forme syntaxique initiale, on applique une des règles définies pour la transformation 5 (insertion d'un type d'articles et duplication d'item). A la forme syntaxique obtenue, on applique une des règles de la transformation 2 (ajout d'un niveau de décomposition).

Transformation en termes d'arbre

Selon la forme syntaxique initiale exprimée en termes d'arbre, on applique une des règles (en termes d'arbre) définies pour la transformation 5. A l'expression en termes d'arbre ainsi obtenue, on applique une des règles (en termes d'arbre) de la transformation 2.

Exemple

Cet exemple est basé sur la figure 4.21.

```
for pren := prenom (pers) . prenom
```

DEVIENT

```
for pp := pre_pers ( : id_p_p ( : Npers' = ( pers ) . Npers ) )
    pren := ( pp ) . id_p_p . prenom ;
```

Cette transformation se traduit en termes d'arbre par :

```
... Δ  instr_for  (22)
   ΔΔ  class_var  (50)
   ΔΔΔ  variable_non_hiérar  (51) fait référence au nom de variable "pren"
   ΔΔ  var_refs  (53) fait référence au nom de variable de référence "pers"
   ΔΔΔ  class_var  (50)
   ΔΔΔΔ  variable_non_hiérar  (51) fait référence au nom d'item "prenom"
   ΔΔ  instructions  (19) racine du sous-arbre représentant le corps de la boucle "for"
```

DEVIENT

```
... Δ  instr_for  (22)
   ΔΔ  class_var  (50) racine du sous-arbre représentant "pp"
   ...
   ΔΔ  expression_collection  (64) racine du sous-arbre représentant "pre_pers ( : id_p_p
       (: Npers' = ( pers ) . Npers ) )"
```

...
ΔΔ instructions (19) racine du sous-arbre représentant le nouveau corps de la boucle "for"
ΔΔΔ assignation (20) racine du sous-arbre représentant "pren := (pp) . id_p_p . prenom"

Le nœud variable_non_hiérar (51) voit sa référence changée. Cette référence désigne, dans l'arbre transformé, le nom de variable de référence "pp", nouvelle variable de parcours.

Le sous-arbre de racine var–refs (53) disparaît et est remplacé par un sous-arbre de racine expression_collection (64), qui représente la nouvelle séquence parcourue.

Enfin, le nœud instructions se voit attribuer un premier fils assignation (20) racine d'un sous-arbre représentant l'assignation dans la forme transformée.

Règle 2

La règle générale exposée ci–dessus (règle n°1) ne couvre cependant pas tous les cas. Il faut alors définir une règle spécifique à la transformation envisagée.

La transformation 5 (insertion et rotation) ne prend pas en compte les items répétitifs non identifiants décomposables. En effet, une insertion d'un type d'articles suivie d'une rotation du type de chemins obtenu crée un identifiant composé de deux items (voir p. ex. fig. 4.17). La transformation ne peut donc être utilisée pour Cobol.

Si l'item répétitif est clé d'accès, il doit être éliminé. Pour ce faire, la règle est de combiner les règles de trois transformations. Selon une forme syntaxique initiale, on applique une des règles de la transformation 4 (insertion d'un type d'articles). A la forme obtenue, on applique une des règles de la transformation 3 (rotation). Pour aboutir à la forme finale, on applique une des règles de la transformation 2 (ajout d'un niveau de décomposition).

Transformation en termes d'arbre

Selon la forme syntaxique exprimée en termes d'arbre, on applique une des règles (en termes d'arbre) définies pour la transformation 4. A l'expression en termes d'arbre ainsi obtenue, on applique une des règles (en termes d'arbre) de la transformation 3, et ensuite de la transformation 2.

Exemple

Cet exemple est basé sur la figure 4.22.

```
for { cte = cote, crs = cours } := ( etud_1 ) . cc
...
endfor ;
```

DEVIENT

```
for ecc := etud_cc ( : id_etud_cc ( : nE' = ( etud_1 ) . nE )
    cte := ( ecc ) . id_etud_cc . cc . cote ;
    crs := ( ecc ) . id_etud_cc . cc . cours ;
    . . .
endfor ;
```

Cette transformation peut se traduire en termes d'arbre, par :

... Δ instr_for (22)
ΔΔ item_répétitif_décomposable (23)
ΔΔΔ comp_variable_parcours (24) fait référence au nom d'item "cote"
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom de variable "cte"
ΔΔΔ comp_variable_parcours (24) fait référence au nom d'item "cours"
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom de variable "crs"
ΔΔ var_refs (53) fait référence au nom de variable de référence "etud_1"
ΔΔΔ class_var (50)
ΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item "cc"
ΔΔ instructions (19) racine du sous-arbre représentant le corps de la boucle "for"

DEVIENT

... Δ instr_for (22)
ΔΔ class_var (50) racine du sous-arbre représentant "ecc"
...
ΔΔ expression_collection (64) racine du sous-arbre représentant "etud_cc (: id_etud_cc (: nE' = (etud_1) . nE))"
...
ΔΔ instructions (19) racine du sous-arbre représentant le nouveau corps de la boucle "for"
ΔΔΔ assignation (20) racine du sous-arbre représentant "cte := (ecc).id_etud_cc.cc.cote"
...
ΔΔΔ assignation (20) racine du sous-arbre représentant "crs := (ecc).id_etud_cc.cc. cours"
...

Les sous-arbres de racines item_répétitif_décomposable (23) et var_refs (50) disparaissent. Ils sont remplacés par des sous-arbres de racine class_var (50) et expression_ collection (64) qui représentent les nouvelles variable de parcours et séquence parcourue.

Le nœud instructions (19) se voit attribuer deux premiers fils assignation (20) racine chacun d'un sous-arbre représentant une instruction d'assignation apparue suite à la transformation.

7. INSERTION D'UN TYPE D'ARTICLES, ROTATION ET APLATISSEMENT TOTAL

Règle

Certaines règles définies auparavant sont une fois de plus combinées.

Selon la forme syntaxique initiale, on applique une des règles définies pour la transformation 4 (insertion d'un type d'articles). A la forme ainsi obtenue, on applique une des règles de la transformation 3 (rotation). La forme définitive est obtenue par l'application d'une des règles de la transformation 1 (aplatissement).

Transformation en termes d'arbre

Selon la forme syntaxique initiale exprimée en termes d'arbre, on applique une des règles (en termes d'arbre) pour la transformation 4. A l'expression ainsi obtenue, on applique une des règles (en termes d'arbre) de la transformation 3 et ensuite de la transformation 1.

Exemple

Cet exemple est basé sur la figure 4.23.

modify etud1 (: cc + { cours = 'conception_bd', cote = x }) ;

DEVIENT

create ec := etud_cc ((: nE' = (etud1) . nE) and (: cours = 'conception_bd') and
(: cote = x) ;

Cette transformation peut être traduite en termes d'arbre, par :

... Δ instr_modify (39) fait référence au nom de variable de référence "etud1"
ΔΔ condition_mod_item (41) fait référence au nom d'item "cc"
ΔΔΔ plus (59)
ΔΔΔ liste_exp_décomposable (35)
ΔΔΔΔ élément (36)
ΔΔΔΔΔ assign (37) fait référence au nom d'item "cours"
ΔΔΔΔΔΔ constante_caractère (83) fait référence à 'conception_bd'
ΔΔΔΔΔ assign (37) fait référence au nom d'item "cote"
ΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom de variable "x"

DEVIENT

... Δ instr_create (30) racine du sous arbre représentant l'instruction de création dans la forme transformée

Le sous-arbre de racine instr_modify (39) disparaît. Il est remplacé par un sous-arbre de racine instr_create (30), représentant l'instruction de création apparue suite à la transformation. Ce dernier nœud racine est rattaché, en toute logique, au nœud père de instr_ modify. La modification est donc remplacée par une création.

8. ELIMINATION D'UN COMPOSANT ROLE DANS UN IDENTIFIANT

Règle

Les règles de la tranformation 3 (rotation) sont appliquées ici selon les mêmes modalités.

Transformation en termes d'arbre

Les règles en termes d'arbre, de la transformation 3 sont appliquées ici selon les mêmes modalités.

Exemple

Cet exemple est basé sur la figure 4.25.

emp := employe ((de : depart1) and (: num_employe = max_num)) ;

DEVIENT

```
emp := employe ( ( : num_dep' = (depart1 ) . num_dep ) and
                 ( : num_employe = max_num ) ) ;
```

Cette transformation peut être traduite en termes d'arbre, par :

```
... Δ assignation (20)
    ΔΔ class_var (50)
    ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "emp"
    ΔΔ expression_assignation (21)
    ΔΔΔ expression_collection (64) fait référence au nom de type d'articles "employe"
    ΔΔΔΔ condition_sur_clé_dans_un_chemin (75)
    ΔΔΔΔΔ condition_sur_chemin (74)
    ΔΔΔΔΔΔ nom (8) fait référence au nom de type de chemins "de"
    ΔΔΔΔΔΔ nom (8) fait référence au nom de variable de référence "depart1"
    ΔΔΔΔΔ conditions_sélection_items (66)
    ΔΔΔΔΔΔ cond_sélection_item (67) racine d'un sous-arbre représentant "(:num_employe =
            max_num )"
```

DEVIENT

```
... Δ assignation (20)
    ΔΔ class_var (50)
    ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "emp"
    ΔΔ expression_assignation (21)
    ΔΔΔ expression_collection (64) fait référence au nom de type d'articles "employe"
    ΔΔΔΔ conditions_sélection_items (66)
    ΔΔΔΔΔ cond_sélection_item (67) fait référence au nom d'item " num_dep' "
    ΔΔΔΔΔΔ suite_cond_sélection_item (68)
    ΔΔΔΔΔΔΔ égal (71)
    ΔΔΔΔΔΔΔ var_refs (53) fait référence au nom de variable de référence "depart1"
    ΔΔΔΔΔΔΔΔ class_var (50)
    ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item "num_dep"
    ΔΔΔΔΔ cond_sélection_item (67) racine d'un sous-arbre représentant "(:num_employe =
            max_num )"
```

Le sous-arbre de racine condition_sur_clé_dans_un_chemin (75) disparaît, sauf son nœud conditions_sélection_items (66) et sa descendance.

Ce nœud conditions_sélection_items de ce sous-arbre, se voit attribuer un fils supplémentaire cond_sélection_item (67). Ce dernier nœud est racine du sous-arbre représentant la condition sur item remplaçant l'accès par le chemin éliminé.

Le nœud conditions_sélections_items est rattaché au nœud père du nœud condition_sur_clé_dans_un_chemin.

9. ELIMINATION D'UN TYPE DE CHEMINS PAR DUPLICATION D'ITEM, ET AJOUT D'UN NIVEAU DE DECOMPOSITION

Règle

Selon la forme syntaxique initiale, des règles de la transformation 3 (rotation) ou 8 (élimination d'un composant rôle dans un identifiant) (elles sont de toute façon équivalentes) sont appliquées. Le résultat crée une clé (identifiante ou pas) composée de plusieurs items (deux le cas présent), ce qui est interdit en Cobol. Les règles de la transformation 2 sont donc

appliquées pour que l'ajout d'un père commun aux composants de cette clé soit répercuté dans la forme syntaxique.

Transformation en termes d'arbre

Selon la forme syntaxique initiale exprimée en termes d'arbre, on applique une des règles (en termes d'arbre) de la transformation 3. A l'expression en termes d'arbre ainsi obtenue, on applique une des règles (en termes d'arbre) de la transformation 2.

Exemple

Cet exemple est basé sur la figure 4.26.

for com := commande (cc : cli)

DEVIENT

for com := commande (: ND (: NCLI' = (cli) . NCLI))

Cette transformation peut être traduite en termes d'arbres, par :

... Δ instr_for (22)
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "com"
ΔΔ expression_collection (64) fait référence au nom de type d'articles "commande"
ΔΔΔ condition_sur_chemin (74)
ΔΔΔΔ nom (8) fait référence au nom de type de chemins "cc"
ΔΔΔΔ nom (8) fait référence au nom de variable de référence "cli"
ΔΔ instructions (19) racine du sous-arbre représentant le corps de la boucle"for"

DEVIENT

... Δ instr_for (22)
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "com"
ΔΔ expression_collection (64) fait référence au nom de type d'articles "commande"
ΔΔΔ conditions_sélection_items (66)
ΔΔΔΔ cond_sélection_item (67) fait référence au nom d'item "ND"
ΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔΔ cond_sélection_item (67) fait référence au nom d'item " NCLI' "
ΔΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔΔ var_refs (53) fait référence au nom de variable de référence "cli"
ΔΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom d'item "NCLI"
ΔΔ instructions (19) racine du sous-arbre représentant le corps de la boucle"for"

Le sous-arbre de racine condition_sur_chemin (74) disparaît. Il est remplacé par un sous-arbre de racine conditions_sélection_items (66) représentant la condition sur item qui remplace la condition sur le chemin éliminé.

10. AJOUT D'UN COMPOSANT ROLE A UN IDENTIFIANT

Règle

create <varref> := <tart> <condition sur item> ;

DEVIENT

create <varref> := <tart> ((<chemin> : <système>) and <condition sur item>) ;

Transformation en termes d'arbre

... Δ instr_create (30)
ΔΔ nom (8) fait référence au nom de variable de référence "<varref>"
ΔΔ nom (8) fait référence au nom de type d'articles "<tart>"
ΔΔ conditions_création (31) un des ses fils est un nœud condition_création_item (33), racine du sous-arbre représentant <condition sur item>
...

DEVIENT

... Δ instr_create (30)
ΔΔ nom (8) fait référence au nom de variable de référence "<varref>"
ΔΔ nom (8) fait référence au nom de type d'articles "<tart>"
ΔΔ conditions_création (31) un des ses fils est un nœud condition_création_item (33), racine du sous-arbre représentant <condition sur item>
ΔΔΔ condition_création_chemin (32)
ΔΔΔΔ nom (8) fait référence au nom de type de chemins "<chemin>"
ΔΔΔΔ nom (8) fait référence au nom de variable de référence "<système>"
...

Le nœud conditions_création (31) se voit attribuer un nœud fils supplémentaire condition_création_chemin (32). Ce nœud fils est racine d'un sous-arbre représentant la condition sur chemin "(<chemin> : <système>)" reliant l'article système à l'article créé par l'instruction create.

11. TRANSFORMATION D'UN ACCES PAR CLE EN UN ACCES PAR CHEMIN

Règle 1

[for] <varref> := <tart> <condition sur item>

DEVIENT

for <varref-int> := <tart-int> <condition sur item'>
<varref> := <tart> (<chemin> : <varref-int>) ;
...
endfor;

Transformation en termes d'arbre

a) la forme syntaxique initiale est une assignation

... Δ assignation (20)
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "<varref>"
ΔΔ expression_assignation (21)
ΔΔΔ expression_collection (64) fait référence au nom de type d'articles "<tart>"
ΔΔΔΔ conditions_sélection_items (66) est racine du sous-arbre qui représente "<condition sur item>"

DEVIENT

... Δ instr_for (22)
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "<varref-int>"
ΔΔ expression_collection (64) fait référence au nom de type d'articles "<tart-int>"
ΔΔΔ conditions_sélection_items (66) est racine du sous-arbre qui représente "<condition sur item'>"
...
ΔΔ instructions (19)
ΔΔΔ assignation (20) est racine du sous-arbre représentant "<varref> := <tart> (<chemin>: <varref-int>) ;"
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "<varref>"
ΔΔΔΔ expression_assignation (21)
ΔΔΔΔΔ expression_collection (64) fait référence au nom de type d'articles "<tart>"
ΔΔΔΔΔΔ condition_sur_chemin (74)
ΔΔΔΔΔΔΔ nom (8) fait référence au nom de type de chemins "<chemin>"
ΔΔΔΔΔΔΔ nom (8) fait référence au nom de variavle de référence "<varref-int>"
ΔΔΔ instr_exit (28)
ΔΔΔΔ nom (8) fait référence au nom de variable de référence "<varref-int>"

Le sous-arbre de racine assignation (20) est remplacé par un sous-arbre de racine instr_for (22). Ce sous-arbre de racine instr_for est rattaché au père de assignation.

Le sous-arbre de racine conditions_sélection_items (66) est enlevé des descendants du nœud assignation.

Ce sous-arbre de racine conditions_sélection_items est remplacé par un sous-arbre de racine condition_sur_chemin (74). C'est la nouvelle condition de sélection.

Le sous-arbre de racine assignation représentant l'assignation de <varref> dans la forme transformée, est rattaché comme premier fils du nœud instructions. Le second fils est le nœud instr_exit (28). Ce nœud instructions est lui-même un fils de instr_for; le sous-arbre dont il est racine représente le corps de la boucle "for".

b) la forme syntaxique initiale est une instruction for

... Δ instr_for (22)
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "<varref>"
ΔΔ expression_collection (64) fait référence au nom de type d'articles "<tart>"
ΔΔΔ conditions_sélection_items (66) est racine du sous-arbre qui représente "<condition sur item>"
...
ΔΔ instructions (19) est racine du sous-arbre qui représente le corps de la boucle
...

DEVIENT

... Δ instr_for (22)
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "<varref-int>"
ΔΔ expression_collection (64) fait référence au nom de type d'articles "<tart-int>"
ΔΔΔ conditions_sélection_items (66) est racine du sous-arbre qui représente "<condition sur item'>"
...
ΔΔ instructions (19)
ΔΔΔ assignation (20) est racine du sous-arbre représentant "<varref> := <tart> (<chemin> : <varref-int>) ;"

Le nœud instr_for (22) devient la racine d'un sous-arbre représentant la forme transformée.

Le sous-arbre de racine class_var représentant la variable de parcours, et le sous-arbre de racine expression_collection (64) sont modifiés pour rendre compte des nouvelles variables de parcours et séquence parcourue.

Le nœud instructions (19) se voit attribuer un nouveau premier fils qui est assignation (20).

Toute instruction "next <varref>" (ou "<exit <varref>") est transformée en "next <varref-int>" (ou "exit <varref-int>"). Cette transformation est représentée par :

... Δ instr_next (27) (ou instr_exit (28))
ΔΔ nom (8) fait référence au nom de variable de référence "<varref>"

DEVIENT

Le nœud nom change de référence et représente alors le nom de la variable de référence "<varref-int>".

Règle 2

create <varref> := <tart> <condition sur item>;

DEVIENT

create <varref> := <tart> <condition sur item>;
create <varref-int> := <tart-int> ((<chemin> : <varref>) and <condition sur item'>));

Transformation en termes d'arbre

... Δ instr_create (30)
ΔΔ nom (8) fait référence au nom de variable de référence "<varref>"
ΔΔ nom (8) fait référence au nom de type d'articles "<tart>"
ΔΔ conditions_création (31) est racine du sous-arbre représentant "<condition sur item>"

DEVIENT

... Δ instr_create (30)
ΔΔ nom (8) fait référence au nom de variable de référence "<varref>"
ΔΔ nom (8) fait référence au nom de type d'articles "<tart>"
ΔΔ conditions_création (31) est racine du sous-arbre représentant "<condition sur item>"
...
Δ instr_create (30)
ΔΔ nom (8) fait référence au nom de variable de référence "<varref-int>"
ΔΔ nom (8) fait référence au nom de type d'articles "<tart-int>"
ΔΔ conditions_création (31)
ΔΔΔ condition_création_chemin (32)
ΔΔΔΔ nom (8) fait référence au nom de type de chemins "<chemin>"
ΔΔΔΔ nom (8) fait référence au nom de variable de référence "<varref>"
ΔΔΔ... qui représente "<condition sur item'>"

Le nœud instr_create (30) se voit attribuer un nœud frère suivant instr_create. Ce nœud frère est racine d'un sous-arbre représentant l'instruction create ajoutée dans la forme transformée.

Règle 3

modify <varref> <condition sur item>;

DEVIENT

<varref-int> := <tart-int> (<chemin> : <varref>);
modify <varref-int> <condition sur item'>;
modify <varref> <condition sur item>;

Transformation en termes d'arbre

... Δ instr_modify (39) fait référence au nom de variable de référence "<varref>"
ΔΔ condition_mod_item (41) ou conditions_mod_items (40) est racine du sous_arbre représentant <condition sur item>

DEVIENT

... Δ assignation (20)
ΔΔ class_var (50)
ΔΔΔ variable_non_hiérar (51) fait référence au nom de variable de référence "<varref-int>"
ΔΔ expression_assignation (21)
ΔΔΔ expression_collection (64) fait référence au nom de type d'articles "<tart-int>"
ΔΔΔΔ condition_sur_chemin (74)
ΔΔΔΔΔ nom (8) fait référence au nom de type de chemins "<chemin>"
ΔΔΔΔΔ nom (8) fait référence au nom de variable de référence "<varref>"
Δ instr_modify (39) fait référence au nom de variable de référence "<varref-int>"
ΔΔ condition_mod_item (41) ou conditions_mod_items (40) est racine du sous_arbre représentant <condition sur item'>
Δ instr_modify (39) fait référence au nom de variable de référence "<varref>"
ΔΔ condition_mod_item (41) ou conditions_mod_items (40) est racine du sous_arbre représentant <condition sur item>

le nœud instr_modify (39) se voit attribuer deux frères précédents assignation (20) et instr_modify (39). Ces nœuds sont racines de sous_arbres représentant respectivement les instructions d'assignation et de modification ajoutées dans la forme transformée.

7.3. Représentation des règles de transformation de la seconde étape

L'objet de ce point est d'exposer la représentation en arbre des règles de transformation décrites au paragraphe 4.3.

La seconde étape de transformation d'algorithmes consiste à transformer les algorithmes obtenus de la première pour qu'ils n'utilisent plus que des primitives permises par le SGD cible. La découpe de ce point est donc basée sur les différents SGD cibles.

1. LE SGD CIBLE EST CODASYL

Règle 1

```
for  <varref> := <tart> ( : <item> <opc> <valeur> )
     <trt>
endfor ;
```

DEVIENT

```
for  <varref> := <tart> ( )
     if (<varref>).<item> <opc> <valeur>
     then <trt>
     endif
endfor ;
```

Transformation en termes d'arbre

La forme initiale est représentée par un sous-arbre dont la racine est un nœud instr_for (22).

```
... Δ  instr_for  (22)
   ΔΔ class_var (50) est racine du sous-arbre représentant "<varref>"
   ...
   ΔΔ expression_collection (64) fait référence à "<tart>" dans la table des symboles
   ΔΔΔ conditions_sélection_items (66) est racine du sous-arbre représentant "(:<item>
        <opc> <valeur> )"
   ...
   ΔΔ instructions (19) est racine du sous-arbre représentant <trt>
```

DEVIENT

```
... Δ  instr_for  (22)
   ΔΔ class_var (50) est racine du sous-arbre représentant "<varref>"
   ...
   ΔΔ expression_collection (64) fait référence à <tart> dans la table des symboles
   ΔΔΔ condition_vide (65)
   ΔΔ instructions (19) est racine du sous-arbre représentant le nouveau corps de la boucle
        "for" dans la forme transformée
   ...
```

Le nœud conditions_sélection_items (64), fils du nœud expression_collection (66) est détruit et remplacé par un nœud condition_vide (65).

Le nœud instructions (19) représente le nouveau corps de boucle. Dans le sous-arbre dont cet instructions est racine, on trouve le nœud initial instructions racine du sous-arbre représentant <trt>.

Règle 2

<varref> := <tart> (: <item> <opc> <valeur>)

DEVIENT

```
for  <varref> := <tart> ( )
     if (<varref>).<item> <opc> <valeur>
     then exit <varref>
     endif
endfor ;
```

Transformation en termes d'arbre

La forme initiale est représentée par un sous-arbre dont une racine est un nœud assignation (20).

Ce sous-arbre est détruit et remplacé par un sous-arbre représentant la forme transformée. Ce second sous-arbre a pour racine un nœud instr_for (22). Il a d'autre part une structure identique au sous-arbre de la forme transformée exposée au cas précédent (règle 1). Seul le sous-arbre représentant <trt> a disparu pour être remplacé par un sous-arbre représentant "exit <varref>".

2. LE SGD CIBLE EST SQL

Aucune transformation n'est nécessaire à ce stade. Aucune représentation n'est dès lors exposée.

3. LE SGD CIBLE EST COBOL

Aucune transformation n'est nécessaire à ce stade. Aucune représentation n'est dès lors exposée.

7.4. GESTION DE LA TABLE DES SYMBOLES

L'arbre représentant l'algorithme initial est donc modifié pour qu'il représente l'algorithme transformé.

Ce second algorithme ne manipule pas les mêmes symboles que le premier. D'aucuns ont disparu (par ex. des noms de types de chemins éliminés par rotation), d'autres sont apparus (par ex. des noms d'items duplicatas ou de types d'articles insérés).

Chaque fois qu'un nœud fait référence à un nouveau symbole, une entrée représentant ce symbole, doit être ajoutée à la table des symboles.

Inversément, après transformation, toute entrée qui n'est plus référencée par aucun nœud est enlevée de la table.

7.5. Conclusion

Le présent chapitre a exposé la représentation en arbre des règles de transformation d'algorithmes exposées au chapitre 4. La façon dont travaille l'outil de transformation d'algorithmes est à ce stade définie. Le chapitre suivant peut donc aborder la question de l'intégration de l'outil à l'atelier logiciel de conception d'applications sur bases de données.

On peut cependant remarquer à propos du présent chapitre que les primitives de gestion d'arbres offertes par l'atelier répondent parfaitement à l'utilisation que le transformateur d'algorithmes peut en faire. Elles permettent par exemple de détruire un nœud et sa descendance, ce qui s'avère parfaitement utile lorsqu'on veut détruire un sous-arbre représentant une instruction qu'on ne veut plus voir apparaître dans l'algorithme transformé.

La structure d'arbre de l'atelier offre en outre l'indispensable caractéristique d'avoir des nœuds ordonnés c'est-à-dire qu'un ordre est établi parmi des nœuds frères. Il existe un premier, un second,..., un dernier. Cet ordre est évidemment indispensable pour représenter une séquence d'instructions. Ces instructions qui constituent les fils d'un nœud "instructions", doivent recevoir une représentation qui situe chacune d'elles dans la séquence à laquelle elle participe. La structure d'arbre offerte par l'atelier permet cette représentation en situant chaque nœud par rapport à son frère (avant ou après).

Chapitre 8 :

Intégration de la notion de transformation dans la base des spécifications de l'atelier

8.0. Introduction

Après avoir défini l'environnement dans lequel doit opérer l'outil de transformation d'algorithmes, et la façon dont ce dernier travaille, il reste à aborder dans cette seconde partie du mémoire le problème de l'intégration des transformations dans le schéma de la base des spécifications de l'atelier logiciel.

Ce chapitre se propose de décrire le contexte nécessaire à une intégration de l'outil de transformation d'algorithmes à l'atelier.

Cet outil qui a besoin d'informations pour travailler (les transformations de schémas et les algorithmes à transformer) peut les trouver dans la base des spécifications de l'atelier.

Ce chapitre décrit donc la représentation des transformations de schémas et la représentation des algorithmes dans les termes des objets de cette base des spécifications. Le lecteur est donc invité à consulter le chapitre 5 pour le détail des objets référencés dans le présent chapitre.

Ce chapitre expose en outre de manière générale la façon dont le transformateur travaille, ainsi que l'état de la base des spécifications avant et après ce travail.

Il faut toutefois remarquer que le but du présent chapitre n'est pas de définir la documentation des transformations qui peut être faite à l'usage d'utilisateurs de l'atelier. Il s'agit de définir les informations dont le transformateur d'algorithmes a besoin.

8.1. Représentation des transformations de schéma

8.1.1. Règles générales de représentation

Dans le contexte du mémoire, le concepteur de base de données applique des transformations de schéma pour obtenir un schéma de base de données conforme à un SGD cible, et ce à partir du schéma des accès nécessaires. Deux schémas sont donc présents : le schéma initial et le schéma transformé. Ces deux schémas sont représentés chacun par une entité SCHEMA (cfr figure 5.3 chapitre 5). La façon dont chaque schéma est représenté (à partir d'une entité SCHEMA) dans la base des spécifications de l'atelier, est supposée connue. Cette question ne sera donc pas abordée ici. Le lecteur intéressé peut se référer à [Hainaut 86b].

A ce stade, deux schémas sont donc représentés dans la base de l'atelier. Ils sont cependant tout à fait indépendants. Rien n'indique que c'est de l'un que l'on a obtenu l'autre.

Il faut donc établir une relation entre ces deux schémas. Le type d'entités RELATION (cfr figure 5.1 chapitre 5) semble, à cette fin, tout à fait approprié. Une entité RELATION relie les entités SCHEMA. On convient qu'elle est associée par l'association MEMBER1 à l'entité SCHEMA représentant le schéma initial, et par l'association MEMBER2 à l'entité SCHEMA représentant le schéma transformé.

A titre d'illustration, si les schémas initial et transformé sont nommés "si" et "st", on a en termes d'occurences :

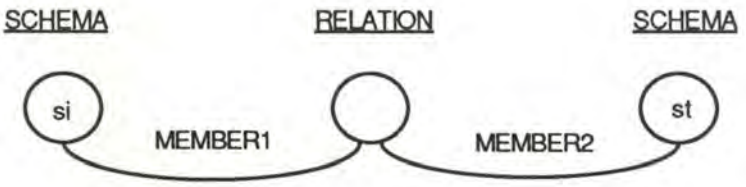


figure 8.1 : relation entre un schéma initial
et un schéma transformé

La base de l'atelier spécifie maintenant que l'on a deux schémas et que le schéma transformé a été obtenu du schéma initial.

On ne sait cependant pas comment ce passage est effectué. Il faut encore représenter les transformations proprement dites.

On détaille pour cela la relation entre deux schémas. En termes de la base des spécifications, on détaille l'entité RELATION qui associe les deux entités SCHEMA. A cette fin, des entités GROUP sont utilisées. N entités GROUP sont donc reliées à l'entité RELATION par des associations G-CONTEXT. Chaque GROUP représente une transformation de schéma qui élimine du schéma initial une structure de données incompatible avec le SGD cible choisi, pour la transformer en une structure de données du schéma transformé.

Le chapitre 4 a présenté les différentes transformations envisagées dans le cadre du mémoire (aplatissement, rotation,...). Un attribut TYPE du type d'entités GROUP spécifie dans ce cas la transformation dont une entité GROUP est une représentation.

Une transformation de schéma travaille sur des objets (type d'articles, type de chemins,...) du schéma initial, les transforme, crée des objets et en détruit. Une structure du

schéma transformé est ainsi obtenue. On peut donc distinguer en deux classes les objets que la transformation concerne : ceux appartenant au schéma initial et ceux appartenant au schéma transformé.

Dans ce contexte, une entité GROUP représentant une transformation de schéma est reliée aux objets que la transformation concerne, par des associations G-MEMBER. Ces objets peuvent être des ENTITY-T, des ATTRIBUTE,... . Le type d'associations G-MEMBER possède un attribut M-ROLE qui rend compte du rôle joué par un objet dans un GROUP. Le cas présent, cet attribut sert donc à représenter le rôle joué par les différents objets dans la transformation. Un objet est susceptible de jouer deux rôles : soit "appartient au schéma initial", soit "appartient au schéma transformé".

En résumé, on a donc qu'une entité GROUP représente une transformation de schéma travaillant sur des objets du SCHEMA initial pour produire des objets du SCHEMA transformé.

Le lecteur peut être contrarié par le caractère flou de cette définition. Ce point (8.1.1) n'est cependant qu'une esquisse de la façon dont les transformations de schéma sont représentées dans la base des spécifications de l'atelier. Le détail de la représentation des transformations exposées au chapitre 4, fait l'objet du point suivant.

8.1.2. Règles de représentation des transformations de schéma dans l'atelier

Ce point décrit la façon dont les transformations de schéma exposées au chapitre 4 sont représentées dans la base des spécifications de l'atelier. Chaque transformation est illustrée par un exemple de son utilisation ainsi que par la représentation de cet exemple dans la base de l'atelier. Il est cependant bon de signaler que seules les informations nécessaires au travail du transformateur d'algorithmes, sont représentées ici.

1. APLATISSEMENT TOTAL

Cette transformation consiste à faire disparaître tous les composants non-feuilles d'un item décomposable et à rattacher les feuilles directement au type d'articles.



figure 8.2 : aplatissement total

remarque : les ovales en trait gras font partie de la représentation du schéma transformé.

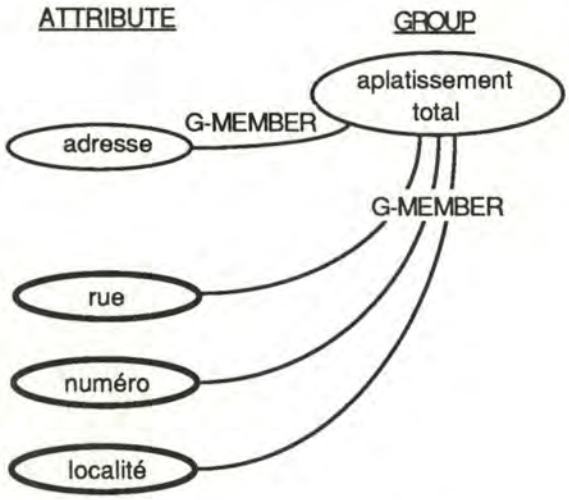


figure 8.3 : représentation de l'aplatissement

Une entité GROUP représentant l'aplatissement d'item décomposable, considéré possède un attribut TYPE ayant pour valeur "aplatissement total".

L'entité GROUP (ici aplatissement total) est associée par une association G-MEMBER à une entité ATTRIBUTE représentant l'item décomposable (ici adresse). Cet ATTRIBUTE joue le M-ROLE "appartient au schéma initial" dans G–MEMBER.

L'entité GROUP est également associée par des associations G-MEMBER aux entités ATTRIBUTE représentant les items résultats de l'aplatissement effectué (ici : rue, numéro, localité sont reliés à "aplatissement total" par trois associations G-MEMBER). Ces ATTRIBUTE jouent le M–ROLE "appartient au schéma transformé" dans ces associations G-MEMBER.

2. AJOUT D'UN NIVEAU DE DECOMPOSITION

Cette transformation consiste à créer un père commun à un groupe d'au moins deux items directement rattachés au type d'articles.

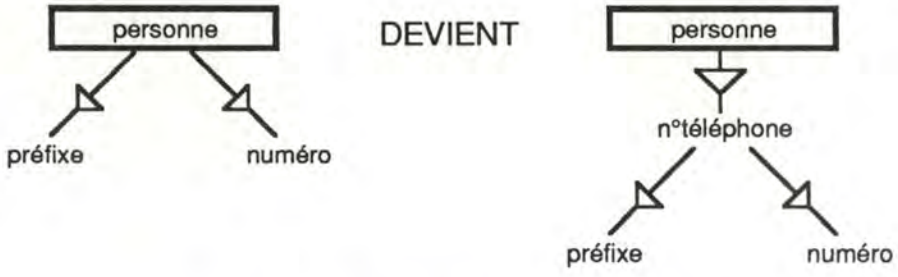


figure 8.4 : ajout d'un niveau de décomposition

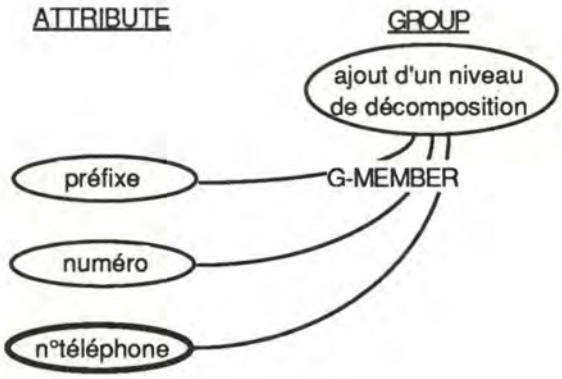


figure 8.5 : représentation de l'ajout d'un niveau de décomposition

Une entité GROUP représentant l'ajout d'un niveau de décomposition, considéré possède un attribut TYPE ayant pour valeur "ajout d'un niveau de décomposition".

L'entité GROUP est associée par des associations G-MEMBER aux entités ATTRIBUTE représentant les items faisant l'objet de la transformation (ici : préfixe et numéro sont reliés à l'entité GROUP "ajout d'un niveau de décomposition" par deux associations G-MEMBER). Ces ATTRIBUTE jouent le M-ROLE "appartient au schéma initial" dans les associations G-MEMBER.

L'entité GROUP est également associée par une association G-MEMBER à l'entité ATTRIBUTE représentant l'item père ajouté (ici : "ajout d'un niveau de décomposition" est relié à n°téléphone par G-MEMBER). Cet ATTRIBUTE joue le M-ROLE "appartient au schéma transformé" dans l'association G-MEMBER.

3. ELIMINATION D'UN TYPE DE CHEMINS PAR DUPLICATION D'ITEM

Cette transformation consiste à éliminer un type de chemins récursif ou non de classe fonctionnelle 1-N, N-1 ou 1-1 en dupliquant un item identifiant simple et élémentaire.

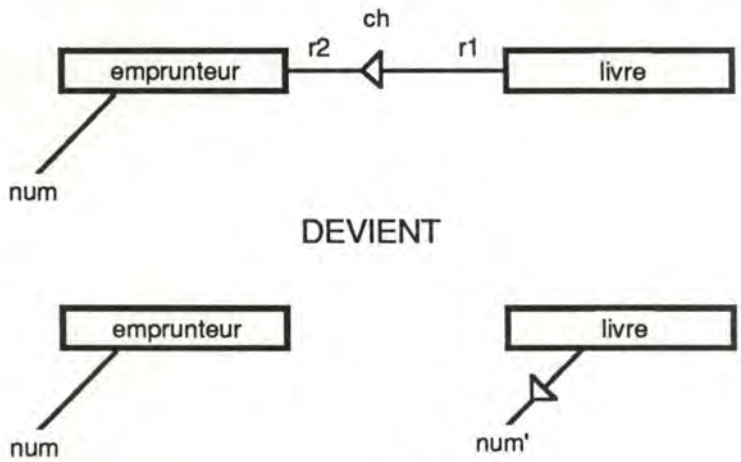


figure 8.6 : rotation

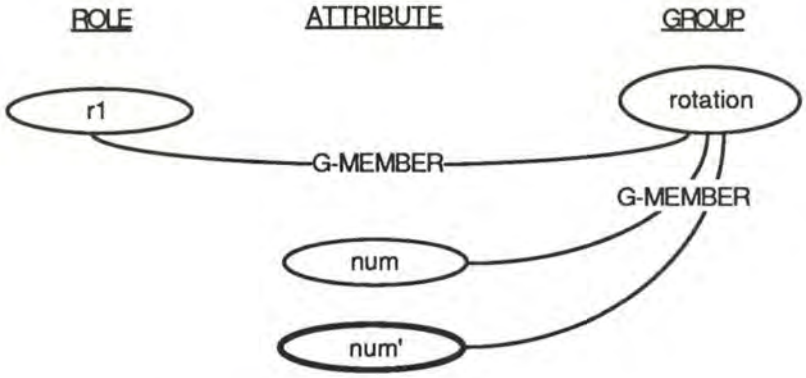


figure 8.7 : représentation de la rotation

Une entité GROUP représentant l'élimination du type de chemins par duplication d'item, considérée possède un attribut TYPE ayant pour valeur "rotation".

Cette entité GROUP est associée par G-MEMBER à une entité ROLE (ici : rotation est liée par G-MEMBER à r qui représente le rôle joué par le type d'articles livre dans le type de chemins ch) représentant le rôle autour duquel s'est faite la rotation. Ce ROLE joue le M-ROLE "appartient au schéma initial" dans cette association G-MEMBER.

L'entité GROUP est également reliée par deux associations G-MEMBER aux deux entités ATTRIBUTE représentant l'item dupliqué et l'item duplicata (ici : rotation est associée à num et num' par G-MEMBER). L'entité item dupliqué joue le M-ROLE "appartient au schéma initial" tandis que celle item duplicata joue le M-ROLE "appartient au schéma transformé", dans les associations G-MEMBER.

On peut de plus signaler que la contrainte référentielle liant num et num' est représentée également dans la base des spécifications de l'atelier.

4. INSERTION D'UN TYPE D'ARTICLES

Cette transformation consiste à insérer un type d'articles dans un type de chemins entre type(s) d'articles ou dans la relation qui associe un type d'articles à un item répétitif.

Une entité GROUP représentant la transformation "insertion d'un type d'articles", considérée possède un attribut TYPE ayant pour valeur "insertion d'un type d'articles".

a) le type d'articles est inséré dans un type de chemins

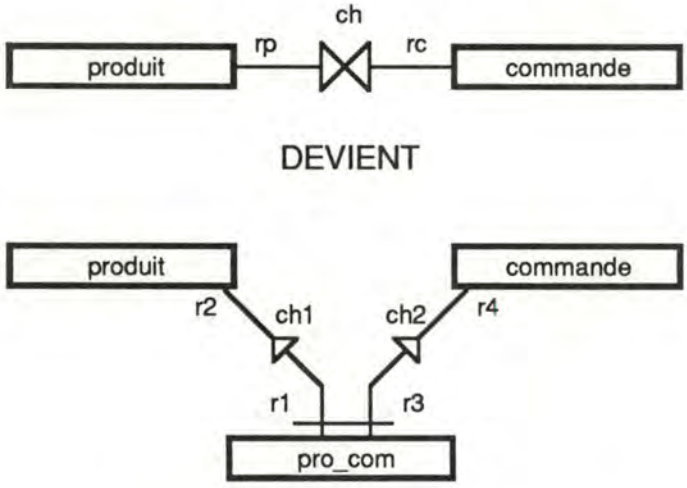


figure 8.8 : insertion d'un type d'articles
pour éliminer un type de chemins

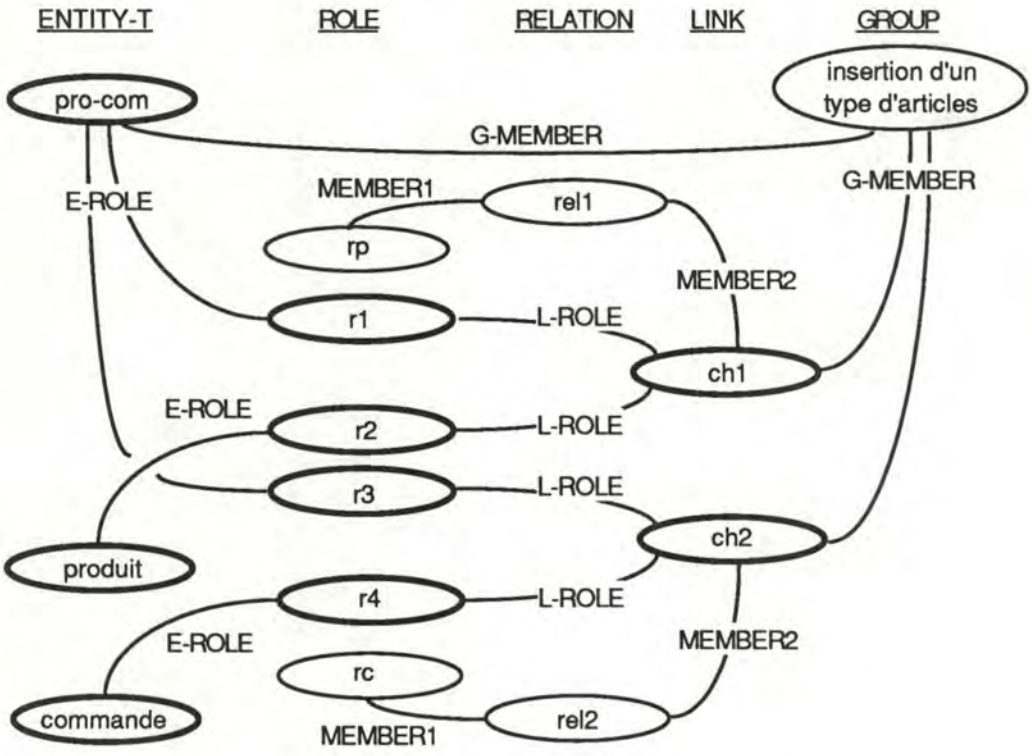


figure 8.9 : représentation de l'insertion d'un type d'articles
pour éliminer un type de chemins

L'entité GROUP est reliée par trois associations G-MEMBER respectivement à une entité ENTITY-T représentant le type d'articles inséré (ici : "insertion d'un type d'articles" est relié à pro-com par G-MEMBER), et deux entités LINK. Ces entités LINK représentent les types de chemins reliant le type d'articles inséré aux types d'articles associés initialement par le type de chemins transformé (ici : "insertion d'un type d'articles" est relié à ch1 et ch2 par deux

associations G-MEMBER). Ces trois entités (deux LINK et une ENTITY-T) associées à l'entité GROUP par G-MEMBER y jouent le M-ROLE "appartient au schéma transformé" (ici : ch1, ch2 et pro-com jouent le rôle "appartient au schéma transformé" dans leurs associations avec "insertion d'un type d'articles").

L'une des deux entités LINK jouant un M-ROLE "appartient au schéma transformé" participe à deux associations L-ROLE la reliant à deux entités ROLE (ici : ch1 est associé à r1 et r2 par deux associations L-ROLE). Ces deux entités ROLE sont associées via E-ROLE aux deux entités ENTITY-T représentant respectivement le type d'articles inséré et un des types d'articles associé initialement au type de chemins transformé (ici : r1 et r2 sont associés respectivement à pro-com et produit par E-ROLE).

La seconde entité LINK jouant un M-ROLE "appartient au schéma transformé" est associée de façon analogue (via ROLE) à deux entités ENTITY-T. Ces deux ENTITY_T représentent respectivement le type d'articles inséré et l'autre type d'articles associé initialement au type de chemins transformé (ici : ch2 est relié par L-ROLE à r3 et r4 eux-mêmes reliés à pro-com et commande par E-ROLE).

D'autre part, les types de chemins apparus suite à l'insertion du type d'articles rendent compte chacun d'un rôle joué par un type d'articles dans le type de chemins initial. On représente cela par une RELATION établie entre les LINK représentant les types de chemins créés après transformation, et les ROLE représentant les rôles dans le type de chemins initial. La RELATION entre un LINK et un ROLE représente le fait que le type de chemins obtenu après transformation rend compte de ce rôle dans le type de chemins initial. On convient que les objets appartenant au schéma initial sont reliés à la RELATION par l'association MEMBER1 et ceux appartenant au schéma transformé, par l'association MEMBER2.

On constate que, le cas présent, le LINK représentant "ch1" est associé via la RELATION "rel1" au ROLE représentant le rôle "rp" du type d'articles "produit" dans le type de chemins "cp", et que le LINK représentant ch2 est associé via "rel2" au ROLE représentant le rôle "rc".

b) le type d'articles est inséré dans la relation qui associe un type d'articles à un item répétitif.

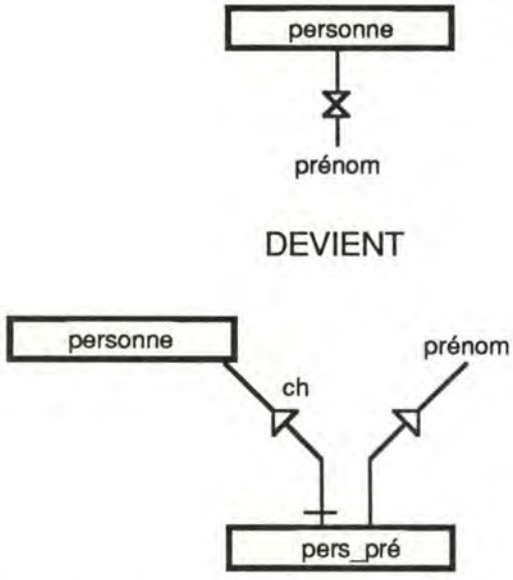


figure 8.10 : insertion d'un type d'articles pour éliminer un item répétitif

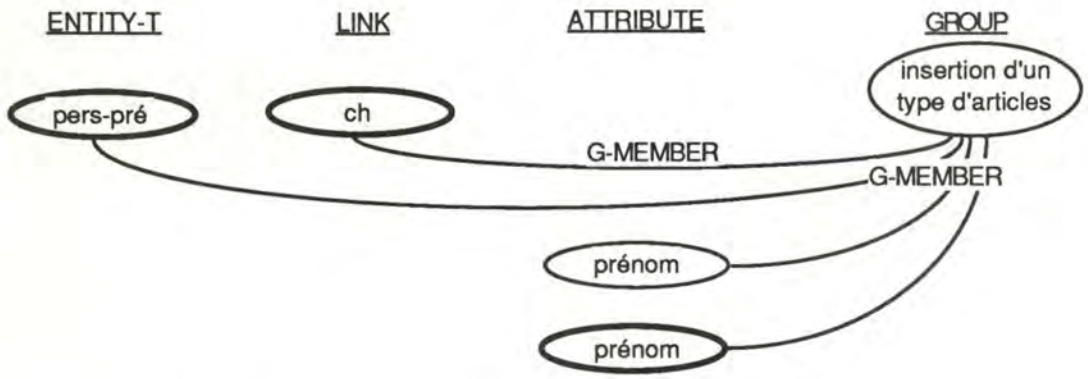


figure 8.11 : représentation de l'insertion d'un type d'articles pour éliminer un item répétitif

L'entité GROUP est associée par une association G-MEMBER à une entité ATTRIBUTE représentant l'item répétitif. Cet ATTRIBUTE joue le M-ROLE "appartient au schéma initial" dans l'association G-MEMBER (ici : "insertion d'un type d'articles" est relié à prénom par une association G-MEMBER).

L'entité GROUP est également associée par trois associations G-MEMBER, successivement à une entité ATTRIBUTE représentant l'item du type d'articles inséré (ici : "insertion d'un type d'articles" est relié à prénom par G-MEMBER), à une entité ENTITY-T représentant le type d'articles inséré (ici : "insertion d'un type d'articles" est relié à pers-pré par G-MEMBER), et à une entité LINK représentant le type de chemins reliant le type d'articles inséré au type d'articles initial (ici : "insertion d'un type d'articles" est relié à ch par G-MEMBER). Ces entités jouent le M-ROLE "appartient au schéma transformé" dans G-MEMBER.

5. INSERTION D'UN TYPE D'ARTICLES ET DUPLICATION D'ITEMS

Cette transformation est définie comme l'insertion d'un type d'articles suivie de l'élimination des types de chemins obtenus par la rotation autour du type d'articles inséré.

A ce stade, le lecteur aura pu se familiariser avec les principes de représentation des transformations. Dès lors, les explications sont données uniquement dans les termes des illustrations.

a) élimination d'un type de chemins

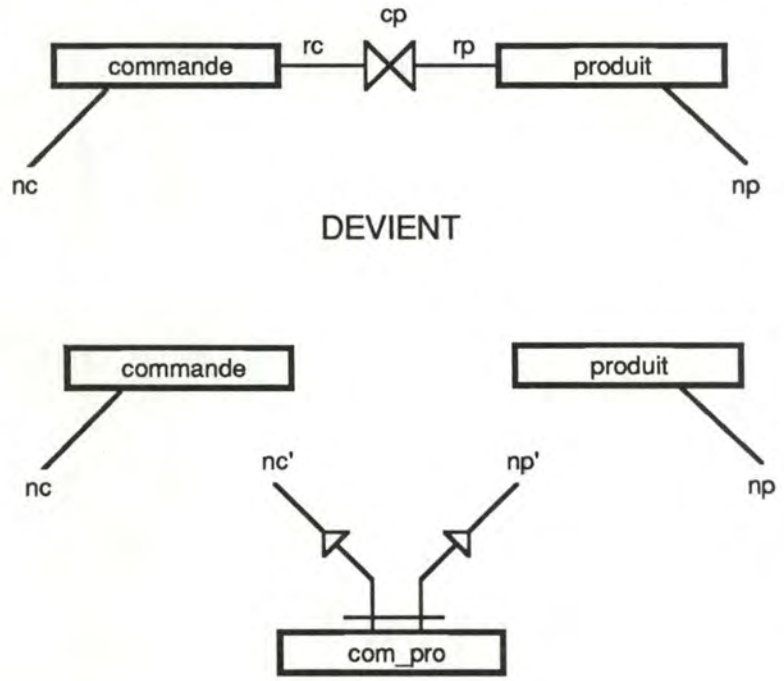


figure 8.12 : insertion et rotation pour éliminer un type de chemins

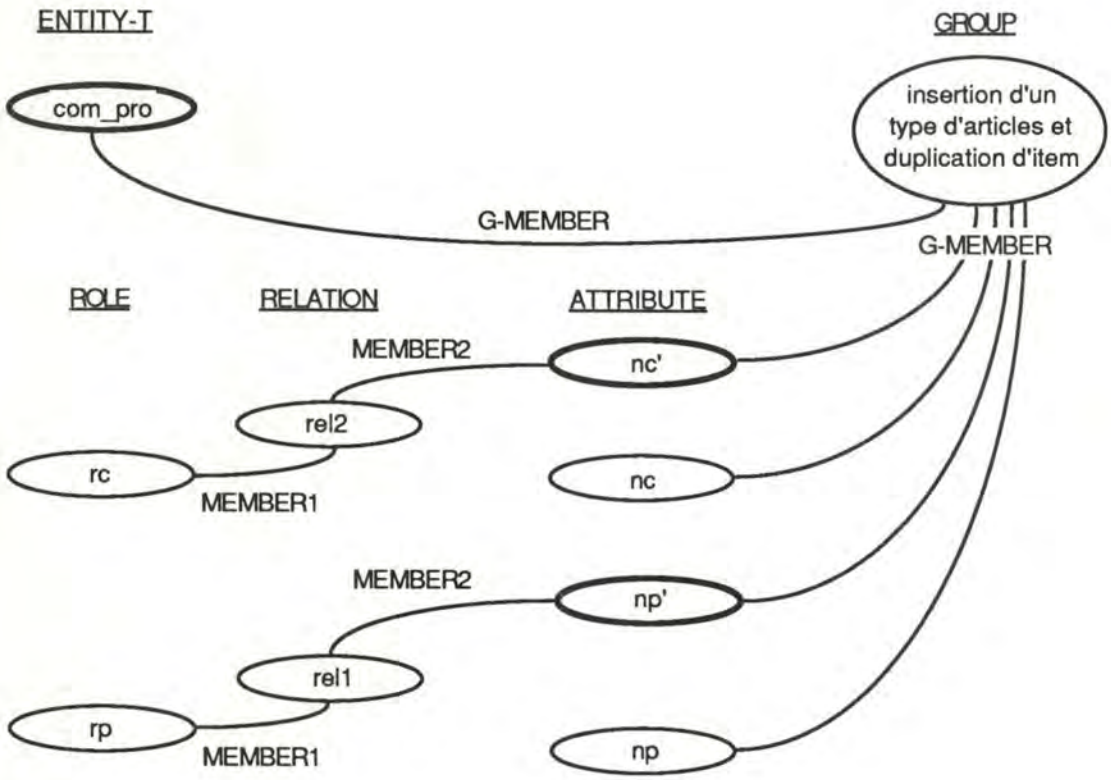


figure 8.13 : représentation de l'insertion et rotation pour éliminer un type de chemins

Une entité GROUP représentant cette transformation a un attribut TYPE qui a pour valeur "insertion d'un type d'articles et duplication d'items".

Cette entité GROUP est associée par G–MEMBER et selon le M-ROLE "appartient au schéma transformé", à l'ENTITY-T représentant le type d'articles inséré com-pro, ainsi qu'à deux ATTRIBUTE représentant les deux items duplicatas nc' et np'.

L'entité GROUP est aussi associée par G-MEMBER aux ATTRIBUTE représentant les items dupliqués nc et np afin de garder la trace des items ayant fait l'objet d'une duplication. Ces ATTRIBUTE jouent le M-ROLE "appartient au schéma initial" dans leurs associations avec l'entité GROUP.

D'autre part, chaque item duplicata rend compte d'un des rôles joués par le type d'articles dans le type de chemins initial. Dès lors, on établit deux RELATION entre les entités ROLE représentant ces rôles, et les ATTRIBUTE représentant les items duplicatas. On convient que les ATTRIBUTE sont reliés à la RELATION par MEMBER2 et les ROLE par MEMBER1. Une RELATION entre un ATTRIBUTE et un ROLE représente donc le fait que l'item représenté par l'ATTRIBUTE rend compte du rôle représenté par l'entité ROLE.

On peut constater qu'ici, l'ATTRIBUTE nc' est relié par la RELATION rel2 au ROLE rc, et que l'ATTRIBUTE np' est relié par la RELATION rel1 au ROLE rp.

b) élimination d'un item répétitif

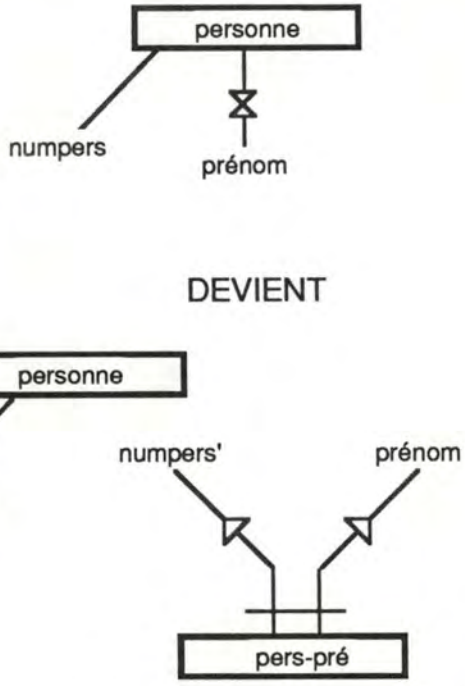


figure 8.14 : insertion d'un type d'articles et rotation pour éliminer un item répétitif

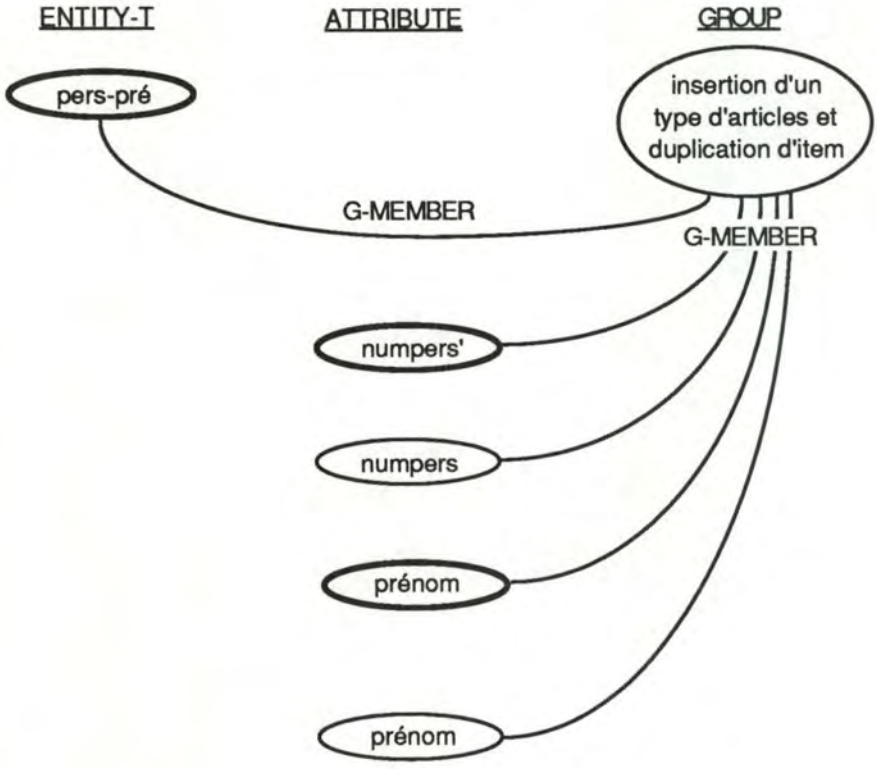


figure 8.15 : représentation de l'insertion d'un type d'articles et rotation pour éliminer un item répétitif

Une entité GROUP représentant cette transformation a un attribut TYPE ayant pour valeur "insertion d'un type d'articles et duplication d'items".

Cette entité GROUP est associée par deux associations G-MEMBER à deux ATTRIBUTE représentant l'item prénom et l'item dupliqué numpers. Ces ATTRIBUTE jouent le M-ROLE "appartient au schéma initial" dans G-MEMBER.

L'entité GROUP est d'autre part associée par trois associations G-MEMBER à l'ENTITY-T représentant le type d'articles inséré pers-pré, et aux deux ATTRIBUTE représentant l'item transformé prénom et l'item duplicata numpers'. Ces deux ATTRIBUTE et l'ENTITY-T jouent le M-ROLE "appartient au schéma transformé" dans les associations G-MEMBER.

6. INSERTION D'UN TYPE D'ARTICLES, ROTATION ET AJOUT D'UN NIVEAU DE DECOMPOSITION.

Cette transformation est définie comme la transformation 5 (insertion d'un type d'articles et duplication d'items) suivie de la transformation 2 (ajout d'un niveau de décomposition).

a) élimination d'un type de chemins

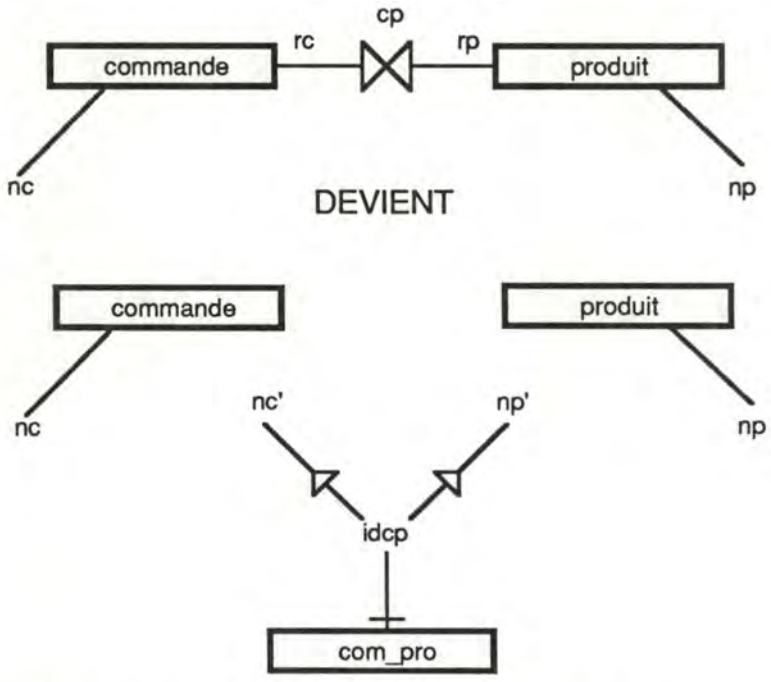


figure 8.16 : insertion, rotation et ajout d'un niveau de décomposition pour éliminer un type de chemins

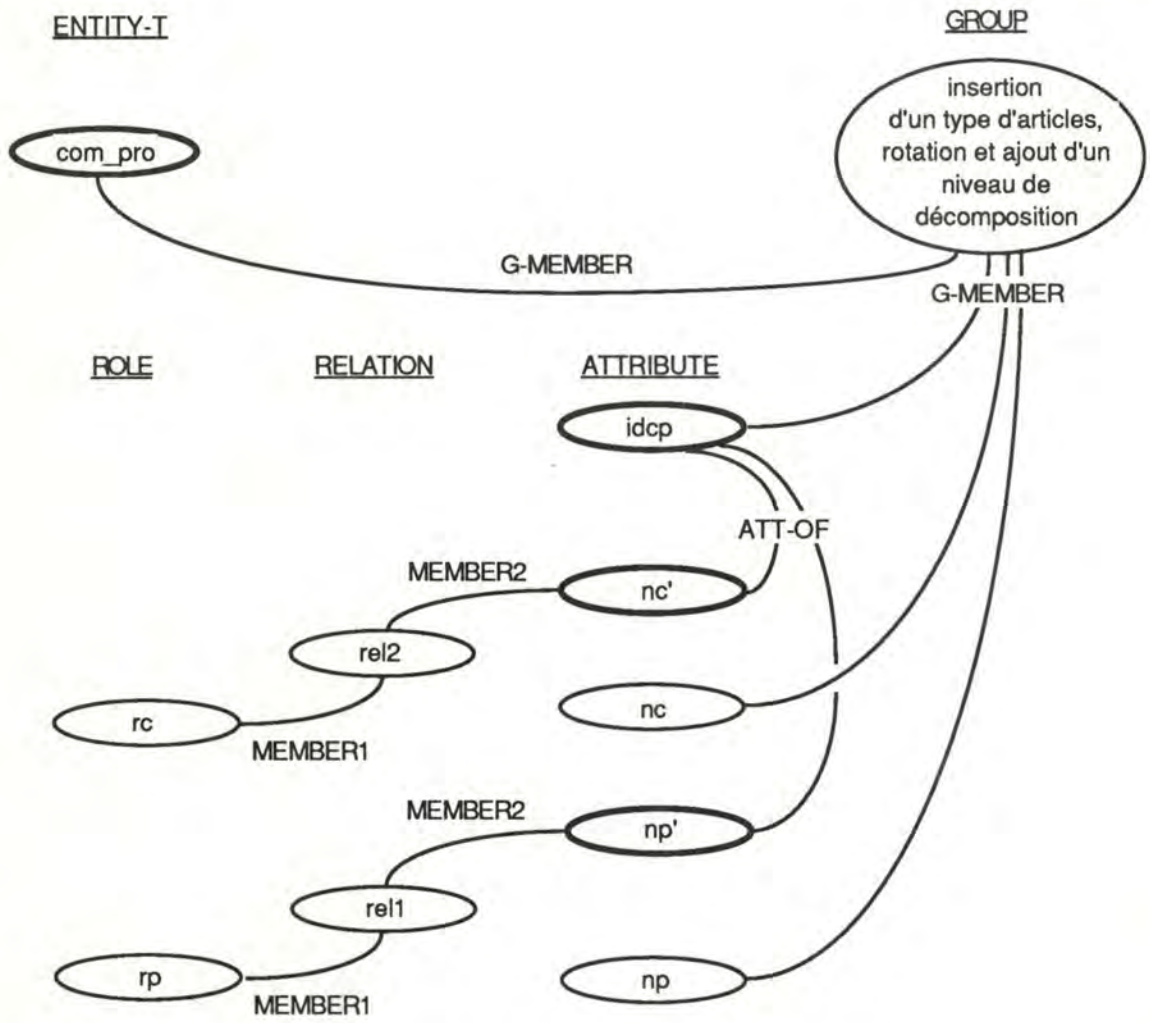


figure 8.17 : représentation de l'insertion d'un type d'articles, rotation et ajout d'un niveau de décomposition pour éliminer un type de chemins

Une entité GROUP représentant cette transformation a un attribut TYPE qui a pour valeur "insertion d'un type d'articles, rotation et ajout d'un niveau de décomposition".

Cette entité GROUP est associée par G-MEMBER à l'ENTITY-T représentant le type d'articles inséré com-pro, ainsi qu'à l'ATTRIBUTE représentant l'item idcp père des deux items dupliqués nc' et np'. L'ATTRIBUTE représentant idcp et l'ENTITY-T représentant com-pro jouent le M-ROLE "appartient au schéma transformé" dans les associations G-MEMBER.

Les items dupliqués nc' et np' sont représentés par des ATTRIBUTE et via des associations ATT-OF comme composants de l'item décomposable idcp.

L'entité GROUP est aussi associée par G-MEMBER aux ATTRIBUTE représentant les items dupliqués nc et np afin de représenter les items ayant fait l'objet d'une duplication.

Comme précédemment, les RELATION (rel1 et rel2) sont utilisées pour indiquer quel item duplicata rend compte de quel rôle du type d'articles dans le type de chemins initial. Ces RELATION associent donc deux ATTRIBUTE à deux ROLE.

Le cas présent, l'ATTRIBUTE nc' est associé au ROLE rc par la RELATION rel2 et l'ATTRIBUTE np' est associé au ROLE rp par la RELATION rel1.

b) élimination d'un item répétitif

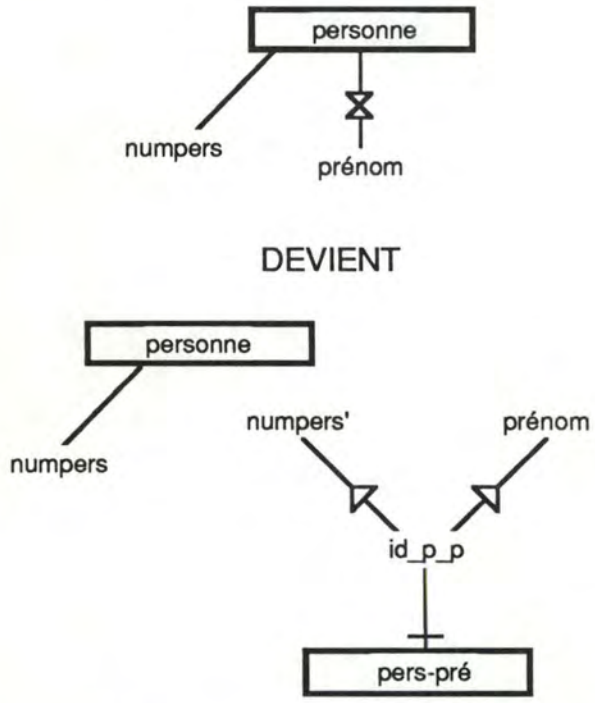


figure 8.18 : insertion d'un type d'articles, rotation
et ajout d'un niveau de décomposition
pour éliminer un item répétitif

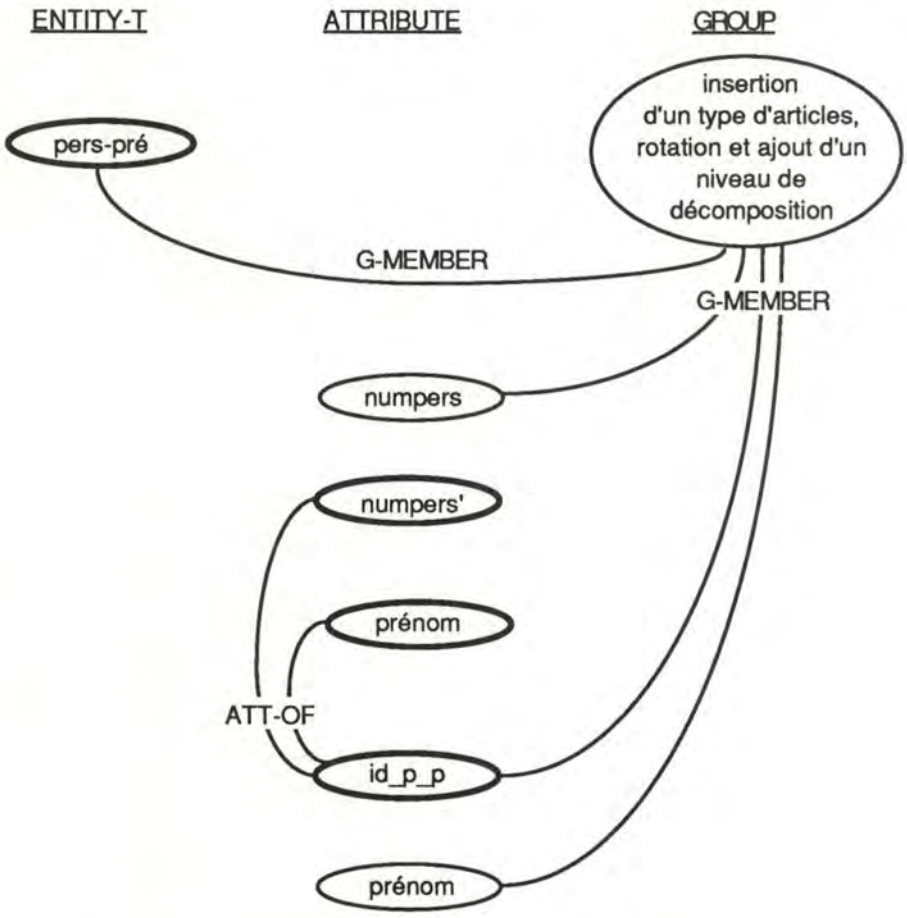


figure 8.19 : représentation de l'insertion d'un type d'articles, rotation et ajout d'un niveau de décomposition pour éliminer un item répétitif

Une entité GROUP représentant cette transformation a un attribut TYPE ayant pour valeur "insertion d'un type d'articles, rotation et ajout d'un niveau de décomposition".

Cette entité GROUP est associée par deux associations G-MEMBER à deux ATTRIBUTE représentant l'item prénom et l'item dupliqué numpers. Ces ATTRIBUTE jouent le M-ROLE "appartient au schéma initial" dans G-MEMBER.

L'entité GROUP est d'autre part associée par deux associations G-MEMBER à l'ENTITY-T représentant le type d'articles inséré pers-pré, et l'ATTRIBUTE représentant l'item id_p_p de ce type d'articles. Cet ATTRIBUTE et cet ENTITY-T jouent le M-ROLE "appartient au schéma transformé" dans les associations G-MEMBER. L'ATTRIBUTE représentant numpers' et prénom sont reliés par ATT-OF à l'ATTRIBUTE représentant leur père, id_p_p.

7. INSERTION D'UN TYPE D'ARTICLES, ROTATION ET APLATISSEMENT TOTAL

Cette transformation est définie comme la transformation 4 (insertion d'un type d'articles) suivie de la transformation 3 (rotation) suivie elle-même de la transformation 1 (aplatissement total).

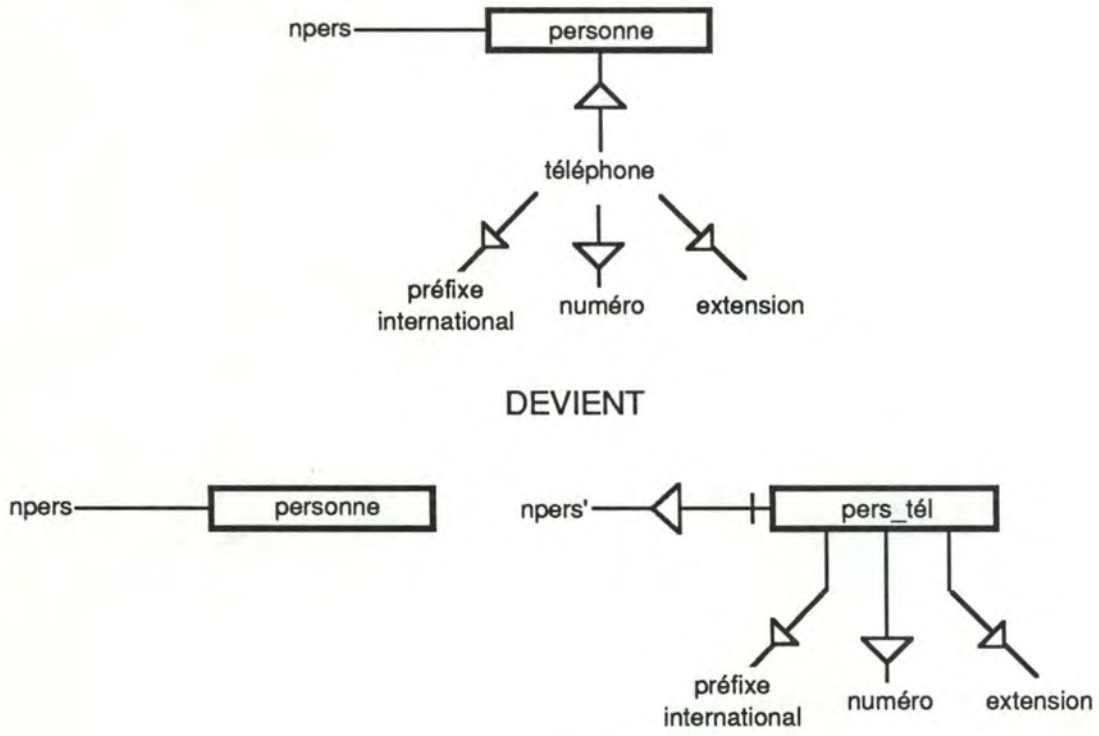


figure 8.20 : insertion d'un type d'articles, rotation et aplatissement total

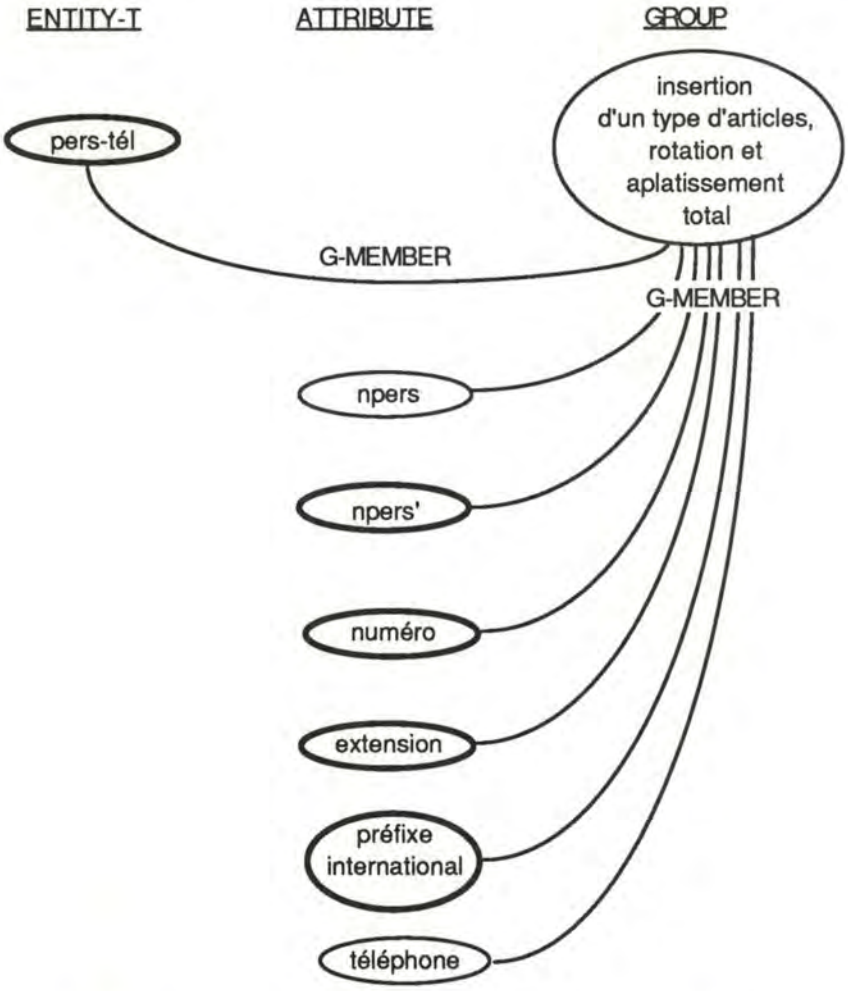


figure 8.21 : représentation de l'insertion d'un type d'articles, rotation et aplatissement total

Une entité GROUP représentant cette transformation a un attribut TYPE ayant pour valeur "insertion d'un type d'articles, rotation et aplatissement total".

Cette entité GROUP est associée par une association G-MEMBER aux entités représentant les objets du schéma initial qui participent à la transformation : l'ATTRIBUTE représentant l'item téléphone transformé et aplati, l'ATTRIBUTE représentant l'item npers qui est dupliqué. Ces ATTRIBUTE jouent le M-ROLE "appartient au schéma initial" dans les associations G-MEMBER.

L'entité GROUP est également associée par G-MEMBER aux entités représentant les objets du schéma transformé : l'ENTITY-T représentant le type d'articles inséré pers-tél, les ATTRIBUTE représentant les items simples et élémentaires résultats de l'élimination de la répétitivité et de la décomposabilité, préfixe international, numéro, extension. L'ATTRIBUTE représentant l'item npers' duplicata est lui aussi associé par G-MEMBER à l'entité GROUP. Cette ENTITY-T et ces ATTRIBUTE jouent le M-ROLE "appartient au schéma transformé" dans les associations G-MEMBER.

8. ELIMINATION D'UN COMPOSANT ROLE DANS UN IDENTIFIANT

Cette transformation se ramène à la transformation 3 (élimination d'un type de chemins par duplication d'item). Le nouvel identifiant est composé de l'item duplicata utilisé pour l'élimination du type de chemins et du reste de l'identifiant initial.

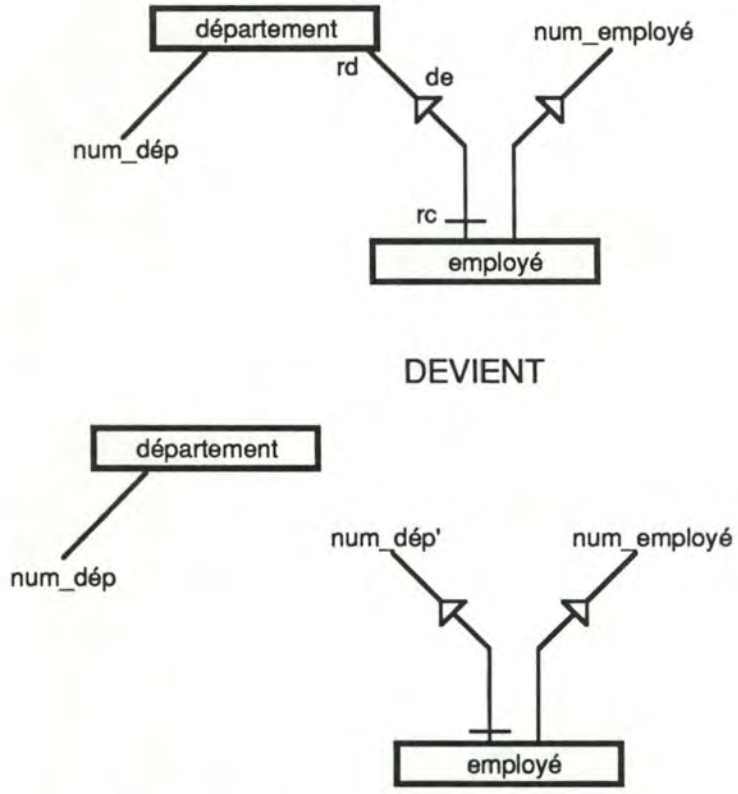


figure 8.22 : élimination d'un composant rôle dans un identifiant

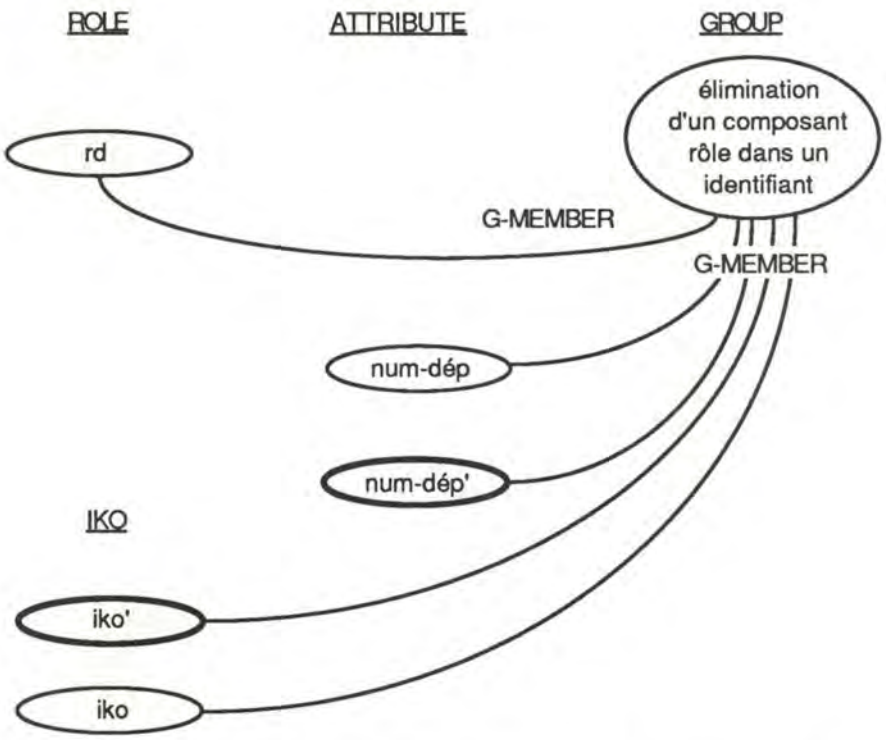


figure 8.23 : représentation de l'élimination d'un composant rôle dans un identifiant

Une entité GROUP représentant cette transformation a un attribut TYPE qui a pour valeur "élimination d'un composant rôle dans un identifiant".

Cette entité GROUP est reliée par une association G-MEMBER à une entité ROLE représentant le rôle participant à l'identifiant, et par une autre association G-MEMBER à l'entité ID-KEY-ORD représentant l'identifiant complet. L'ATTRIBUTE représentant l'item dupliqué (pour éliminer le composant rôle) participe lui aussi à une association G-MEMBER le reliant à l'entité GROUP. Ces entités jouent le M-ROLE "appartient au schéma initial" dans G-MEMBER.

L'entité GROUP est également associée par deux associations G-MEMBER à une entité ATTRIBUTE représentant l'item duplicata, et à une entité IKO représentant le nouvel identifiant. Cet ATTRIBUTE et cet IKO jouent le M-ROLE "appartient au schéma transformé" dans les associations G-MEMBER qui les relient à l'entité GROUP.

L'entité GROUP est associée par des associations G-MEMBER à l'entité ROLE représentant le rôle rd participant à l'identifiant dans le schéma initial, à l'entité ATTRIBUTE représentant l'item dupliqué num-dep pour éliminer le type de chemins, et enfin à l'entité iko représentant l'identifiant. Ces entités jouent le M-ROLE "appartient au schéma initial" dans leurs associations avec l'entité GROUP.

L'entité GROUP est associée aussi à l'entité ATTRIBUTE représentant l'item duplicata num-dep', et à l'entité IKO iko' représentant le nouvel identifiant.

9. ELIMINATION D'UN TYPE DE CHEMINS PAR DUPLICATION D'ITEM ET AJOUT D'UN NIVEAU DE DECOMPOSITION

Cette transformation est définie comme la transformation 3 (rotation) suivie de la transformation 2 (ajout d'un niveau de décomposition). Elle sert à éliminer une clé d'accès dans un type de chemins 1-N.

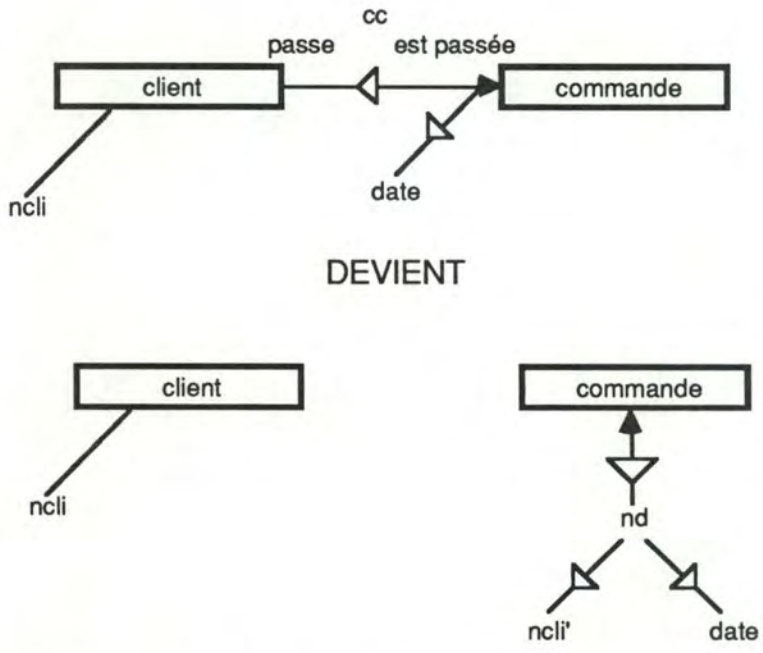


figure 8.24 : élimination d'un type de chemins
par rotation et ajout d'un niveau de décomposition

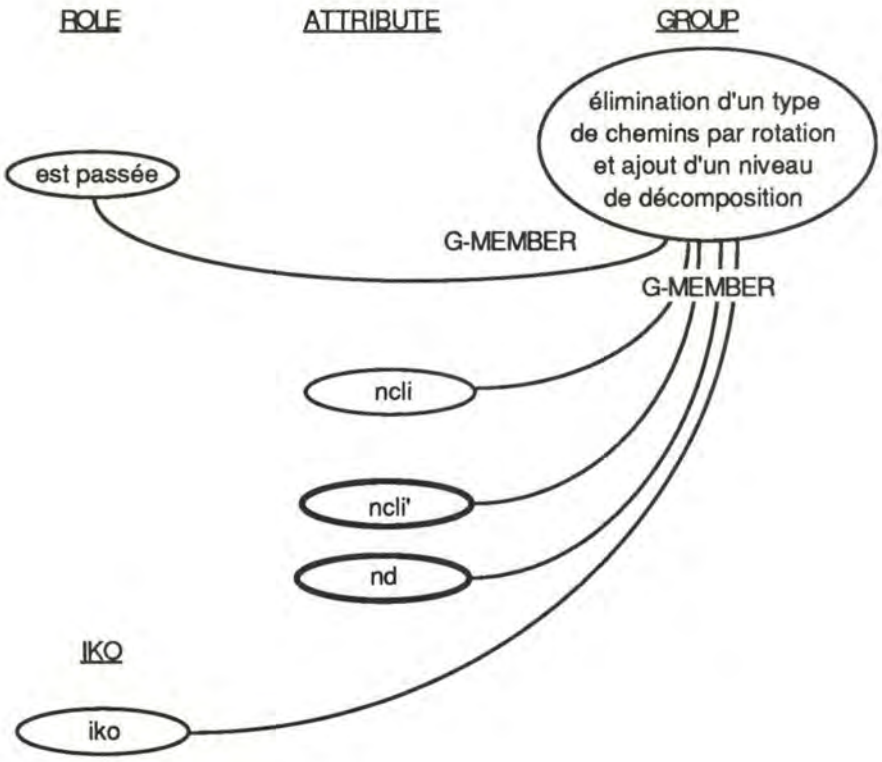


figure 8.25 : représentation de l'élimination d'un type de chemins par rotation et ajout d'un niveau de décomposition

Une entité GROUP représentant la transformation considérée a un attribut TYPE ayant pour valeur "élimination d'un type de chemins par duplication d'item et ajout d'un niveau de décomposition".

Cette entité est associée par une association G-MEMBER à l'entité ROLE représentant le rôle autour duquel s'est faite la rotation. L'entité ATTRIBUTE représentant l'item dupliqué est également reliée par une association G-MEMBER à l'entité GROUP. Une entité IKO est aussi reliée à cette entité GROUP par G-MEMBER. Cette entité IKO représente la clé dans le chemin, de la forme initiale. Toutes ces entités jouent le M-ROLE "appartient au schéma initial" dans leurs associations G-MEMBER avec l'entité GROUP.

L'entité GROUP est associée également par G-MEMBER aux entités ATTRIBUTE représentant l'item père ajouté et l'item duplicata. Ces entités ATTRIBUTE jouent le M-ROLE "appartient au schéma transformé" dans leurs associations avec l'entité GROUP.

L'entité GROUP représentant la transformation est associée par G-MEMBER au ROLE représentant le rôle "est passée" autour duquel s'est faite la rotation, à l'ATTRIBUTE représentant l'item ncli dupliqué pour cette élimination, et à l'IKO représentant la clé iko dans le type de chemins cc. Le ROLE, l'ATTRIBUTE et l'IKO jouent le M-ROLE "appartient au schéma initial" dans ces associations G-MEMBER.

Cette entité GROUP est associée également par G-MEMBER aux ATTRIBUTE représentant l'item duplicata ncli' et l'item père nd résultat de l'ajout d'un niveau de décomposition. Ces deux ATTRIBUTE jouent le M-ROLE "appartient au schéma transformé " dans leurs associations G-MEMBER avec l'entité GROUP.

10. AJOUT D'UN COMPOSANT ROLE A UN IDENTIFIANT

Cette transformation consiste à transformer un identifiant composé d'items (un ou plusieurs) en un identifiant composé de ces items et d'un nouveau type de chemins. Le type de chemins relie le type d'articles identifié à l'article système.



DEVIENT

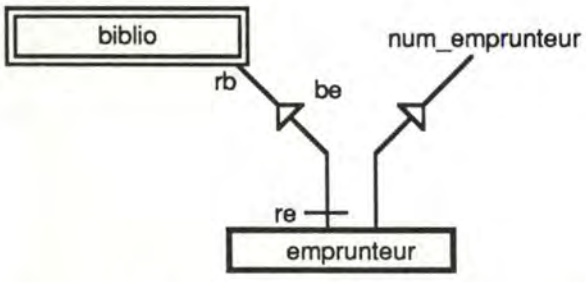


figure 8.26 : ajout d'un composant rôle à un identifiant

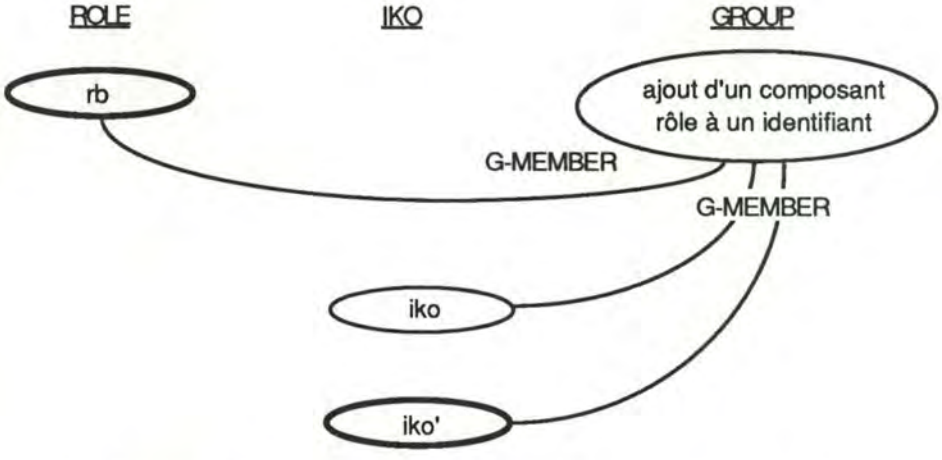


figure 8.27 : représentation de l'ajout d'un composant rôle à un identifiant

Une entité GROUP désignant la transformation considérée a un attribut TYPE ayant pour valeur "ajout d'un composant rôle à un identifiant". Cette entité GROUP est reliée par une association G-MEMBER à une entité IKO représentant l'identifiant auquel on va ajouter un composant rôle. Cette entité IKO joue le M-ROLE "appartient au schéma initial" dans son association G-MEMBER avec l'entité GROUP.

L'entité GROUP est aussi reliée par G-MEMBER selon le M-ROLE "appartient au schéma transformé" à l'entité ROLE représentant le rôle ajouté à l'identifiant. Ce GROUP est également relié par G–MEMBER selon le même M–ROLE à l'entité IKO représentant le nouvel identifiant.

L'entité GROUP est reliée par G-MEMBER à l'entité IKO représentant l'identifiant iko auquel on va ajouter un composant ROLE rb. Ce ROLE rb est représenté par une entité ROLE associée également à l'entité GROUP par G-MEMBER. Le nouvel identifiant est représenté par une entité IKO iko' associée également à l'entité GROUP par G-MEMBER. Dans ces associations, le ROLE et l'entité IKO iko' jouent un M-ROLE "appartient au schéma transformé", et l'entité IKO iko un M-ROLE "appartient au schéma initial".

11. TRANSFORMATION D'UN ACCES PAR CLE EN UN ACCES PAR CHEMIN

Cette transformation consiste à enlever la qualité de clé d'accès à un (groupe d') item. cette qualité est reportée sur un (groupe d') item duplicata d'un type d'articles intermédiaire créé. Un type de chemins permet l'accès du type d'articles intermédiaire vers le type d'articles cible initial.

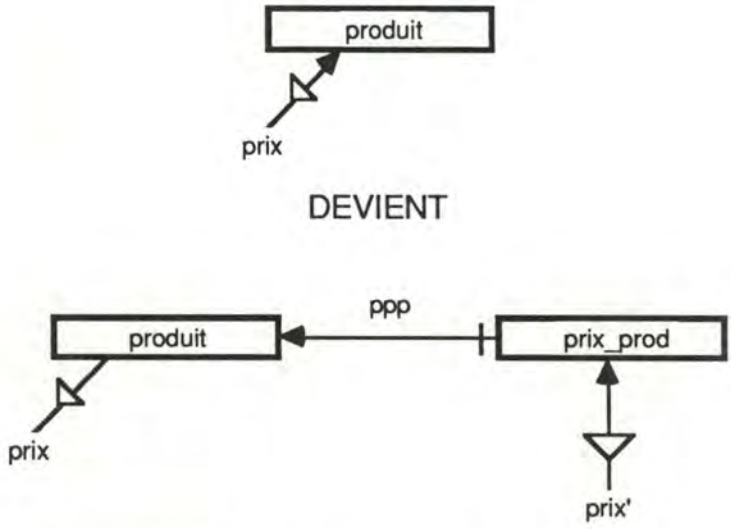


figure 8.28 : transformation d'un accès par clé
en un accès par chemin

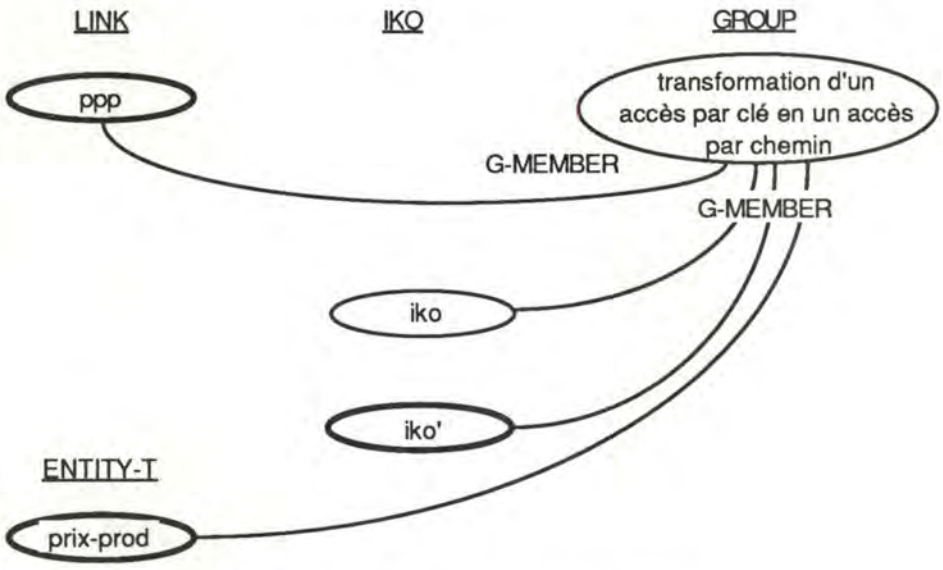


figure 8.29 : représentation de la transformation d'un accès par clé en un accès par chemin

Une entité GROUP représentant la transformation considérée a un attribut TYPE ayant pour valeur "transformation d'un accès par clé en un accès par chemin". Cette entité GROUP est associée par G-MEMBER selon le M-ROLE "appartient au schéma initial" à une entité IKO représentant le (groupe d') item(s) dont on enlève la qualité de clé d'accès.

Cette entité GROUP est associée aussi par des associations G-MEMBER selon le M-ROLE "appartient au schéma transformé" à l'entité ENTITY-T représentant le type d'articles créé, à l'entité LINK représentant le type de chemins reliant le type d'articles créé au type d'articles initial. L'entité GROUP est associée de la même façon à une entité IKO représentant le (groupe d') item(s) duplicata clé d'accès.

L'entité GROUP est reliée par G-MEMBER à l'entité IKO représentant la clé initiale iko (le cas présent, cette clé se résume à l'item prix). iko joue le M-ROLE "appartient au schéma initial" dans son association G-MEMBER avec l'entité GROUP.

L'entité GROUP est d'autre part reliée également par G-MEMBER à l'ENTITY-T représentant le type d'articles créé prix-prod, ainsi qu'au LINK représentant le type de chemins ppp. ppp relie le type d'articles prix-prod au type d'articles initial produit.

L'entité IKO représentant la clé duplicata iko' est reliée toujours par G-MEMBER à l'entité GROUP. Cette entité IKO, le LINK et l'ENTITY-T jouent le M-ROLE "appartient au schéma transformé" dans leurs associations G-MEMBER avec l'entité GROUP.

8.2. Représentation des algorithmes

Le but de ce paragraphe est de définir comment sont représentés dans la base des spécifications de l'atelier les informations concernant les algorithmes initiaux et transformés, et nécessaires au transformateur d'algorithmes.

La base de l'atelier peut contenir des entités MODULE. Un MODULE est une description d'une unité de traitement opérant notamment sur la base de données du système d'information. Ces traitements sont réalisés par l'exécution des programmes qui les décrivent. Ces programmes sont les algorithmes envisagés dans ce mémoire.

Un algorithme fait donc partie de la description d'un traitement et est donc associé à un MODULE.

Plus précisément, le type d'entités PROCEDURE est utilisé pour représenter les algorithmes. Une entité PROCEDURE qui représente un algorithme a un attribut TEXT qui contient le texte de cet algorithme. Elle est d'autre part associée via une association PROC-OF au MODULE dont l'algorithme est une description.

La base des spécifications contient donc un ensemble de MODULE décrits chacun par 0,1 ou N PROCEDURE. La représentation des algorithmes initiaux est ainsi définie.

Les algorithmes transformés sont également représentés de la sorte dans la mesure où la transformation d'algorithmes n'est que la transformation d'un texte de programme en un autre.

La PROCEDURE représentant l'algorithme transformé est reliée par une RELATION à la PROCEDURE représentant l'algorithme initial. On convient que cette dernière est reliée à la RELATION par une association MEMBER1. L'autre PROCEDURE est reliée à cette même RELATION par une association MEMBER2. Ces deux PROCEDURE sont évidemment reliées au même MODULE.

Les PROCEDURE sont chacune reliées à un MODULE. Chaque MODULE est relié à un système de modules MOD-SYSTEM qui est lui-même relié à un SCHEMA par l'intermédiaire d'une RELATION. Un MOD-SYSTEM est associé à une RELATION par MEMBER1, un SCHEMA par MEMBER2. Ce SCHEMA représente le schéma initial.

On peut d'autre part conclure ce point par la remarque suivante.

Le but du présent chapitre est double : d'une part spécifier les informations dont a besoin le programme qui transforme un algorithme effectif conforme LDA/MAG en algorithme conforme à un SGD cible (le transformateur d'algorithmes); d'autre part déterminer là où ces informations sont disponibles et sous quelles formes.

Bien que la base des spécifications de l'atelier logiciel offre, comme cela a été vu au chapitre 5, les objets PERFORMS, ACTION, ACT-ON qui permettent de représenter (indépendamment de tout texte algorithmique) les actions qu'effectue un module sur les éléments d'un schéma de base de données, il n'en est pas fait usage.

Pour rendre un algorithme effectif conforme LDA/MAG en algorithme conforme à un SGD cible, il est nécessaire de connaître la façon dont le schéma des accès nécessaires a été transformé pour le rendre conforme au SGD cible. Les algorithmes peuvent alors être adaptés pour travailler sur ce schéma conforme. Le transformateur d'algorithmes contient en ses procédures les règles de transformation des formes syntaxiques. Ces règles déterminent la forme générique d'une forme syntaxique à transformer, les conditions requises pour que la

forme doive être transformée, et enfin la façon dont la forme syntaxique générique est transformée. La seule information qui fait défaut au transformateur est l'instanciation de ces règles : si une règle du transformateur stipule qu'une forme syntaxique accès par chemin doit être transformée lorsque le type de chemins est éliminé par duplication d'item, en accès par clé sur base de l'item duplicata, il manque au transformateur les informations définissant le type de chemins particulier éliminé, le nom de l'item duplicata. Le transformateur peut trouver ces informations dans la base des spécifications de l'atelier logiciel. A partir de là, il peut reconnaître dans l'algorithme initial la forme syntaxique à transformer et la transformer effectivement. Connaître quelles actions le module qu'il transforme exécute sur tel objet du schéma de la base de données lui est superflu.

On peut toutefois noter que suite à la transformation des algorithmes, la représentation des actions qu'effectuent les modules qu'ils décrivent, doit être modifiée également.

En effet, après transformation, un module peut effectuer de nouvelles actions sur de nouveaux objets (par ex. la création d'un type d'articles intermédiaire).

L'étude de la transformation de la représentation des actions n'entre pas dans le cadre du mémoire.

8.3. Exemple d'une base des spécifications possible

La base des spécifications fournit donc au transformateur d'algorithmes toutes les informations dont il a besoin pour travailler.

On dispose de deux SCHEMA associés par une RELATION qui représente le fait que l'un est obtenu de l'autre. Un ensemble de GROUP précise cette RELATION et décrit comment elle s'est faite par le biais de diverses transformations.

Le transformateur tire profit de ces informations et modifie les TEXT des PROCEDURE représentant les algorithmes initiaux. Ces TEXT ne représentent donc plus les algorithmes initiaux mais bien ceux transformés.Les PROCEDURE avant et après transformation sont en RELATION les unes avec les autres. Elles sont toutes associées (via les associations PROC-OF) à des MODULE eux-mêmes associés (via des associations M-IN) à des systèmes de modules MOD-SYSTEM. Ces MOD-SYSTEM (travaillent sur) sont associés via RELATION au SCHEMA représentant le schéma initial.

En termes d'occurences, on a par exemple :

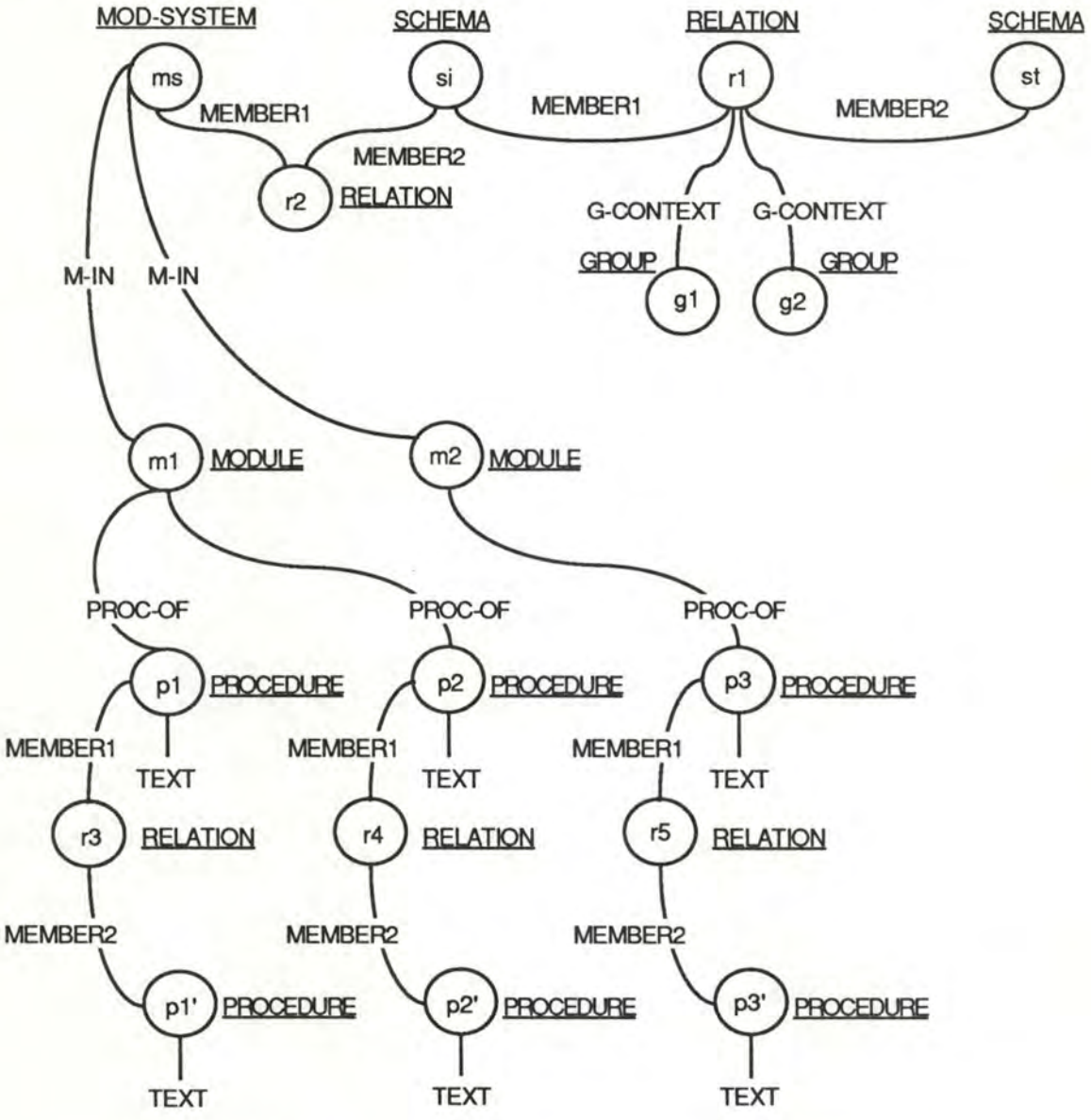


figure 8.30 : Illustration de la représentation de transformation de schéma et d'algorithmes dans la base de l'atelier.

On constate que le système de modules "ms" travaille (par la relation "r2") sur le schéma "si". Ce schéma est transformé en "st" par l'application des transformations "g1" et "g2". "ms" est composé de deux modules "m1" et "m2". Le module "m1" est décrit par les textes des procédures "p1" et "p2"; "m2" est décrit par le texte de "p3".

Les algorithmes initiaux sont représentés par les PROCEDURE "p1", "p2", "p3", les algorithmes transformés par "p1'", "p2'", "p3'". Le passage des uns aux autres est représenté par les RELATION "r3", "r4", "r5".

8.4. Conclusion

Le contexte d'intégration du transformateur d'algorithmes, dans l'atelier est, à ce stade, défini. Certaines remarques méritent, à ce propos, d'être soulignées.

On peut reprocher à la solution adoptée pour la représentation des transformations de schéma, d'être assez consommatrice de place. Les deux schémas sont représentés entièrement dans la base des spécifications, et cela peut parfois donner lieu à des répétitions inutiles d'informations. En effet, considérant un schéma des accès nécessaires volumineux, si seule une structure de données est transformée pour rendre le schéma conforme au SGD cible (p. ex. si on a ajouté un item père commun à deux items), nombre de structures de données du schéma des accès nécessaires se retrouvent comme telles dans le schéma transformé.

Une solution plus économique eût été de ne faire apparaître dans le schéma transformé, associés par G–MEMBER à l'entité GROUP représentant une transformation, que les nouveaux objets que cette transformation introduit. L'inexactitude de cette solution entraîne cependant son rejet. Considérant en effet la transformation qui insère un type d'articles dans un type de chemins N–N, des entités LINK représentant les types de chemins créés entre le type d'articles inséré et les types d'articles initiaux, apparaissent dans le schéma transformé. Dans la mesure où une entité ENTITY–T ne peut appartenir qu'à un seul schéma, il est impossible de relier ces entités LINK aux entités ENTITY–T représentant les types d'articles initiaux sous peine d'affirmer que des chemins relient des objets de schémas différents (ce qui n'a pas de sens). Il est dès lors impossible de spécifier la participation de tel type d'articles du schéma initial à tel type de chemins créé par la transformation.

En outre, documenter les transformations de schémas et d'algorithmes n'a pas constitué le but de ce chapitre. Cela présenterait cependant un attrait certain pour l'utilisateur de l'atelier logiciel et en particulier des outils de transformation de schémas et d'algorithmes. L'horizon sur ce domaine est donc ouvert.

Conclusion

1. Evaluation du travail effectué

Ce mémoire s'est proposé d'étudier le développement d'un outil de transformation automatique d'algorithmes effectifs conformes LDA/MAG en algorithmes conformes à un SGD cible.

La démarche de conception de bases de données dans laquelle s'inscrit la transformation d'algorithmes a pour cela été décrite.

Le SGD virtuel LDA/MAG occupe une place importante dans cette démarche. La description de ce SGD et sa comparaison avec divers SGD cibles envisagés ont occupé les propos des trois premiers chapitres.

Le quatrième chapitre qui est la base du mémoire, a abordé le problème proprement dit de la transformation des algorithmes.

La première partie du mémoire a ainsi été clôturée.

La seconde partie s'est attachée à l'intégration de l'outil étudié, dans l'atelier logiciel de conception d'applications sur bases de données.

Le chapitre 5 a dès lors décrit la base des spécifications et les outils de l'atelier. Un outil particulier est le gestionnaire d'arbre. Le chapitre 6 a décrit cet outil ainsi que l'utilisation qui peut en être faite pour représenter un algorithme LDA.

Les règles de transformation exposées au chapitre 4 ont ensuite été traduites en termes de la représentation exposée au chapitre 6. Ce furent les propos du chapitre 7.

Le huitième et dernier chapitre a enfin défini le contexte nécessaire à une intégration de l'outil de transformation d'algorithmes, à l'atelier.

On peut à ce stade émettre quelques remarques à propos du travail effectué.

L'ampleur que peut prendre ce genre d'étude a entraîné qu'il faille restreindre le cadre de réflexion. En autres choses, l'étude des types de chemins multi-types a été écartée, et le catalogue des transformations de schéma s'est limité aux transformations strictement nécessaires.

Le rejet des types de chemins multi–types a d'autre part entraîné la mise à l'écart de certaines formes syntaxiques permises initialement dans le langage LDA. Offrir au programmeur la possibilité de définir des variables de référence pouvant désigner des articles de type quelconque (x : ref of RECORD) est utile dans le cadre des types de chemins multi–types. Le langage LDA présenté au chapitre 3 ne le signale donc pas.

On peut aussi s'étonner de ce que la transformation de schéma "enlever une contrainte d'existence" n'ait pas été retenue de façon explicite dans le catalogue des transformations. Elle présente en effet une utilité certaine dans la mesure où le SGD Codasyl n'autorise pas de contrainte telle sur l'origine d'un type de chemins 1–N. Le paragraphe 4.6 signale cependant les traitements qu'il y a lieu d'effectuer dans les programmes d'application en cas d'impossibilité pour LDA/MAG de continuer à gérer une contrainte d'existence. La solution présentée est celle que l'on envisagerait si la transformation d'une contrainte d'existence était prise en compte.

Le langage LDA et le modèle MAG ont d'autre part été influencés par le fait qu'ils s'inscrivent dans le contexte de la transformation des algorithmes. Cela s'est illustré particulièrement lors de l'étude de la manipulation des items répétitifs où on a interdit

notamment qu'un item répétitif soit composant d'une clé d'accès, et qu'une valeur d'item répétitif soit désignée par une variable indicée. Cette dernière restriction est consécutive à la transformation d'un item par insertion d'un type d'articles. Suite à cette transformation, la désignation d'une $i^{ème}$ valeur d'item répétitif nécessite, en effet, le comptage du $i^{ème}$ article inséré et l'accès à la valeur d'item qui lui est associée.

Ce mémoire a de plus mis en lumière les difficultés consécutives à la différence entre les algorithmes initiaux et les algorithmes transformés. La gestion de la survenance d'incidents et d'erreurs d'éxécution en est un exemple. On a en effet constaté qu'une instruction de l'algorithme initial pouvait être transformée en plusieurs instructions dans l'algorithme transformé. Dès lors, pour conserver l'atomicité d'une instruction initiale de mise à jour au niveau d'une séquence d'instructions, la notion de transaction s'est avérée indispensable.

De façon analogue, d'autres problèmes sont dus à la disparité entre le schéma des accès nécessaires et le schéma conforme au SGD cible. On peut citer à ce propos la gestion mal aisée par programmes d'application, des diverses contraintes d'intégrité que le SGD ne peut plus prendre en charge. Le SGD qui gère la répétitivité des items ne peut évidemment plus le faire dès l'instant où ces items sont transformés par insertion d'un type d'articles.

Le lecteur a pu d'autre part constater que peu de règles de transformation ont été définies lors de la seconde étape de transformation des algorithmes. Rappelons que cette étape consiste à enlever d'un algorithme toute primitive non reconnue par le SGD cible. D'aucuns pourraient argumenter à ce propos, que le problème de la modification d'identifiant en Cobol a été omis. En effet, la norme spécifie qu'on ne peut modifier la valeur d'une clé primaire (RECORD KEY).

Dès lors, la seconde étape de transformation devrait être complétée par une règle remplaçant la modification de cette clé par une destruction de l'article suivie de la création d'un article identique avec la valeur de clé modifiée.

Cependant, les structures de données, même si elles sont conformes à Cobol, sont toujours à ce stade exprimées dans les termes de LDA/MAG. Or, LDA/MAG ne connaît pas la notion de clé primaire (ou secondaire). Il est dès lors impossible de spécifier l'item qui sera clé primaire dans le LDD et en conséquence, de détecter les modifications de clé primaire pour les transformer.

L'ensemble des règles définies à la seconde étape n'a donc pas à être complété.

Le problème de la transformation des algorithmes présente une complexité importante. Maîtriser celle–ci a exigé d'aborder un aspect du problème à la fois : définir un catalogue de transformations de schéma, déterminer les formes syntaxiques particulières influencées... . Limiter le champ d'investigation lors d'une étape de l'étude a cependant parfois conduit à des réflexions inexactes. Par exemple, la similitude entre les formes "create a := A (ch : b)" et "modify a (ch : b)" avait amené à affirmer que seul "(ch : b)" était à considérer lors de la transformation "élimination d'un type de chemins par duplication d'item". Le chapitre 4 a cependant révélé qu'il ne suffisait pas de transformer cette forme en "(item duplicata = (b). item dupliqué)" mais que d'autres aspects devaient être pris en compte (l'instruction create ou modify précédant la forme syntaxique, le type d'articles autour duquel s'est faite la rotation).

Dès lors, bien que la plupart des transformations d'algorithmes se font suite à des transformations de données, il est erroné d'affirmer que seules les désignations de ces données sont à prendre en considération pour la transformation de ces algorithmes. Les instructions ont une influence que l'on ne peut négliger.

2. Perpectives

Ce mémoire a posé les bases du développement d'un outil de transformation automatique d'algorithmes. Cette étude peut être étendue de diverses façons.

On peut envisager de prendre en compte les aspects qui ont été laissés de côté lors de l'analyse. Par exemple, les types de chemins multi–types et la répétitivité à n'importe quel niveau d'item.

Un décompilateur peut être adjoint à l'ensemble des outils étudiés. Il serait utilisé lors du stockage des algorithmes transformés, dans la base des spécifications de l'atelier, et permettrait ainsi à l'utilisateur de se rendre compte de visu des transformations que ces algorithmes ont subies.

Rappelons d'autre part que le caractère systématique des règles de transformation définies dans ce mémoire peut nuire à la production d'algorithmes efficaces. Ce critère d'efficacité peut être intégré dans la production d'algorithmes conformes.

En conclusion, ce travail et les perspectives qu'il offre s'inscrivent dans le domaine de la conception de logiciels assistée par ordinateur. Il s'agit ici moins d'assister le concepteur que de le libérer totalement d'une tâche complexe et fastidieuse. C'est un des aspects de l'aide qu'on peut lui apporter.

Bibliographie

[Aho, Ullman, 79] : AHO, ULLMAN, "Principles of compiler design", Addison Wesley, 1979.

[Bodart, Pigneur, 83] : BODART, PIGNEUR, "Conception assistée des applications informatiques, 1. Etude d'opportunité et analyse conceptuelle", Masson, 1983.

[Cadelli, Muller, 85] : CADELLI, MULLER, "contribution to a database design workbench ADL–COBOL/GAM compiler", mémoire de fin d'étude, institut d'informatique, 1984–1985.

[Charlot, Muller, 86] : CHARLOT, MULLER, "contribution à l'atelier logiciel de conception de bases de données : étude de transformation de schémas", mémoire de fin d'étude, institut d'informatique, 1985–1986.

[Clarinval, 81] : CLARINVAL, "comprendre, connaître et maîtriser le COBOL, normes ANSI COBOL 1974", Presses Universitaires de Namur, 1981.

[COBOL, 74] : "TOPS–10/TOPS–20 COBOL–74 Language Manual", DEC Documentation.

[Date, 81] : DATE, "An introduction to Data Base Systems", 3rd Edition, Addison Wesley, 1981.

[Date, 83] : DATE, "An introduction to Data Base Systems", Vol 2 , Addison Wesley, 1982.

[Feldman, Gries, 68] : FELDMAN, GRIES, "Translator writing systems", Communications of the ACM, February 1968, vol 11,2.

[Hainaut, 80] : HAINAUT, "Analyse des accès dans les systèmes de gestion de bases de données Codasyl 71", institut d'informatique, juin 1980.

[Hainaut, 84a] : HAINAUT, "Introduction à un système de gestion de bases de données relationnel SQL/DS d'IBM", institut d'informatique, janvier 1984.

[Hainaut, 84b] : HAINAUT, "Le modèle d'accès généralisé", institut d'informatique, janvier 1984.

[Hainaut, 86a] : HAINAUT, "Conception assistée des applications informatiques, 2. Conception de la base de données", Masson, 1986.

[Hainaut, 86b] : HAINAUT, "PROJET Atelier de Base de Données, Spécification – SPEC 86/7–1 Schémas de la base des spécifications", institut d'informatique, juillet 1986.

[Hainaut, 86c] : HAINAUT, "PROJET Atelier de Base de Données, Spécification – SPEC 86/8–1, Introduction aux outils généraux de développement de l'atelier", institut d'informatique, août 1986.

[Lecharlier] : LECHARLIER, "Définition du langage LSD 80", institut d'informatique.

ANNEXES

INTRODUCTION

Ces annexes remplissent deux objectifs : d'une part exposer la raison pour laquelle la notion d'identifiant dans un type de chemins ne reconnaît pas les types de chemins de classe fonctionnelle N–N et les items répétitifs; d'autre part présenter au lecteur un exemple où il peut constater de visu une application concrète de l'étude menée dans le mémoire.

Annexe 1 :
la restriction de la notion d'identifiant dans un type de chemins

Un type de chemins N–N ne peut apparaître comme composant d'un identifiant car quel que soit le SGD cible, sa transformation est problématique.

Considérons en effet l'exemple formel suivant :

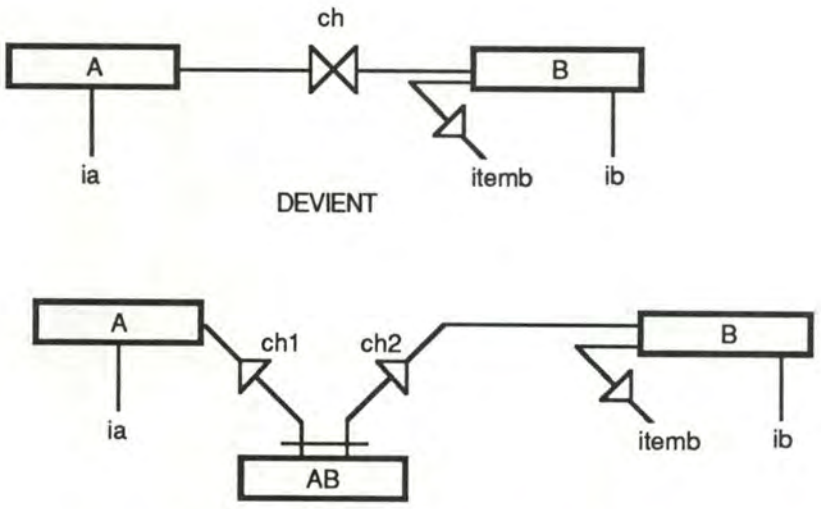


figure A.1 : insertion d'un type d'articles dans un type de chemins composant d'un identifiant, afin de se conformer à Codasyl.

On constate que l'identifiant comprend maintenant un type de chemins (ch2) de classe fonctionnelle N–1, ce qui est interdit.

Si la structure de données est transformée pour se conformer aux SGD SQL ou Cobol (qu'il y ait ajout d'un niveau de décomposition ou non importe peu), d'autres problèmes se posent.

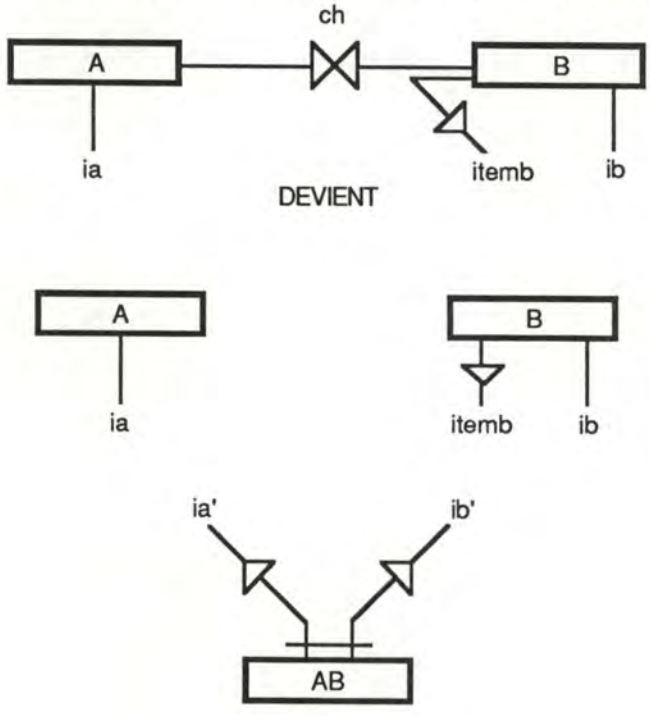


figure A.2 : insertion d'un type d'articles dans un type de chemins N–N composant d'un identifiant, et rotation des types de chemins obtenus.

Cette transformation rejette sur les programmes d'application le soin de gérer l'identification. Celle–ci s'exprime maintenant par un contrainte d'intégrité pour le moins complexe à gérer. La forme initiale spécifie, en effet, que pour tout chemin ch, les cibles B de ce chemin sont identifiées par "itemb". Dès lors, après transformation, il faut vérifier pour chaque article A, que les B qui lui sont associés respectent cette contrainte. Il faut pour cela accéder à partir d'un A à ces B (via AB) et vérifier qu'aucune valeur d'item "itemb" n'apparaît plus d'une fois dans les valeurs d'item "itemb" associées à ces articles (cela signifierait en effet que plusieurs cibles d'un chemin sont associées à une même valeur d'item, ce qui est contraire à la contrainte d'identification). La gestion de cette contrainte est donc laissée de côté.

De la façon similaire dont serait transformé (par insertion d'un type d'articles) un item répétitif composant d'un identifiant, on déduit que les items répétitifs ne peuvent être composants d'identifiant.

Annexe 2 :
développement d'un exemple

Cet exemple a pour but d'illustrer l'articulation du mémoire dans la conception d'une base de données. Pour ce faire sont considérés comme acquis le schéma des accès possibles, les algorithmes prédicatifs, le schéma des accès nécessaires et les algorithmes effectifs.

Soit le schéma des accès nécessaires suivant :

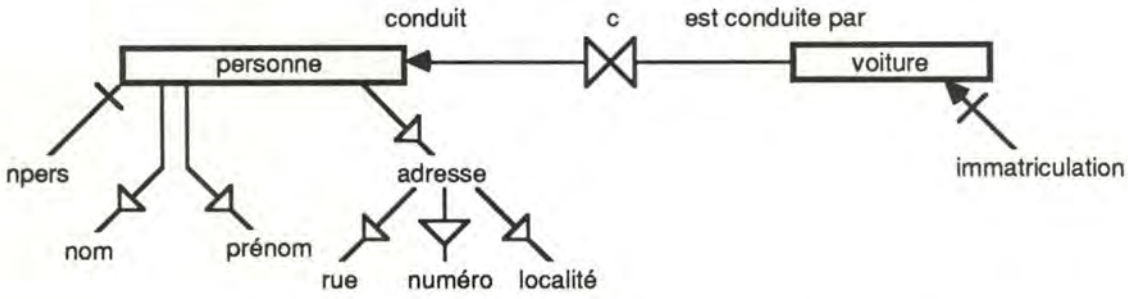


figure A.3 : exemple de schéma des accès nécessaires.

L'algorithme effectif suivant permet de donner la liste des personnes susceptibles d'être impliquées dans un accident occasionné par une voiture dont on connaît l'immatriculation. Cette liste fait mention des nom, prénom et adresse de ces personnes.

```
algorithm accident

var numvoiture :  string  (6);
    v          :  ref of voiture;
    p          :  ref of personne;

begin
    lire  (numvoiture);
    v := voiture (: immatriculation = numvoiture);
    imprimer ('voici la liste des personnes concernees :');
    for p := personne ( c : v )
        imprimer ( (p).nom, (p).prenom, (p).adresse.rue, (p).adresse.numero,
        (p).adresse.localite );
    endfor
end.
```

Cet algorithme est représenté en termes d'arbre par :

Δ structure_d'algorithme (1) fait référence à "accident" dans la table des symboles.
ΔΔ section_déclaration_interne (2)
ΔΔΔ section_déclaration_type (3)
ΔΔΔ section_déclaration_variable (5)
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔΔΔ string (9) fait référence à "6" dans la table des constantes.
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "v" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "voiture" dans la table des symboles.
ΔΔΔΔ déclaration_variable (6)

ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "personne" dans la table des symboles.
ΔΔ instructions (19)
ΔΔΔ appel (29) fait référence à "lire" dans la table des symboles.
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔ assignation (20)
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "v" dans la table des symboles.
ΔΔΔΔ expression_assignation (21)
ΔΔΔΔΔ expression_collection (64) fait référence à "voiture" dans la table des symboles.
ΔΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔΔ cond_sélection_item (67) fait référence à "immatriculation" dans la table des symboles.
ΔΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔ appel (29) fait référence à "imprimer" dans la table des symboles.
ΔΔΔΔ constante_caractère (83) fait référence à 'voici la liste des personnes concernees' dans la table des constantes.
ΔΔΔ instr_for (22)
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "p" dans la table des symboles.
ΔΔΔΔ expression_collection (64) fait référence à "personne" dans la table des symboles.
ΔΔΔΔΔ condition_sur_chemin (74)
ΔΔΔΔΔΔ nom (8) fait référence à "c" dans la table des symboles.
ΔΔΔΔΔΔ nom (8) fait référence à "v" dans la table des symboles.
ΔΔΔΔ instructions (19)
ΔΔΔΔΔ appel (29) fait référence à "imprimer" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "nom" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "prenom" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "adresse" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "rue" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "adresse" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numero" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "adresse" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "localite" dans la table des symboles.

Le SGD cible est Cobol

SCHEMA CONFORME

Le schéma des accès nécessaires devient le schéma conforme suivant :

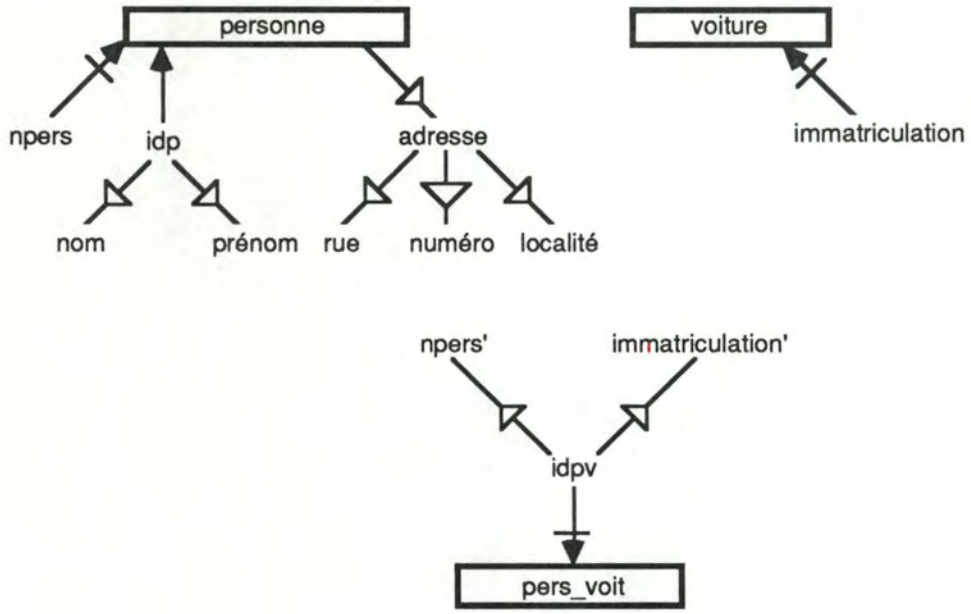


figure A.4 : schéma conforme à Cobol.

REPRESENTATION DE LA TRANSFORMATION DANS LA BASE DE L'ATELIER

On a deux entités GROUP. La première représente l'élimination du type de chemins "c", la seconde l'ajout d'un père commun à "nom" et "prénom".

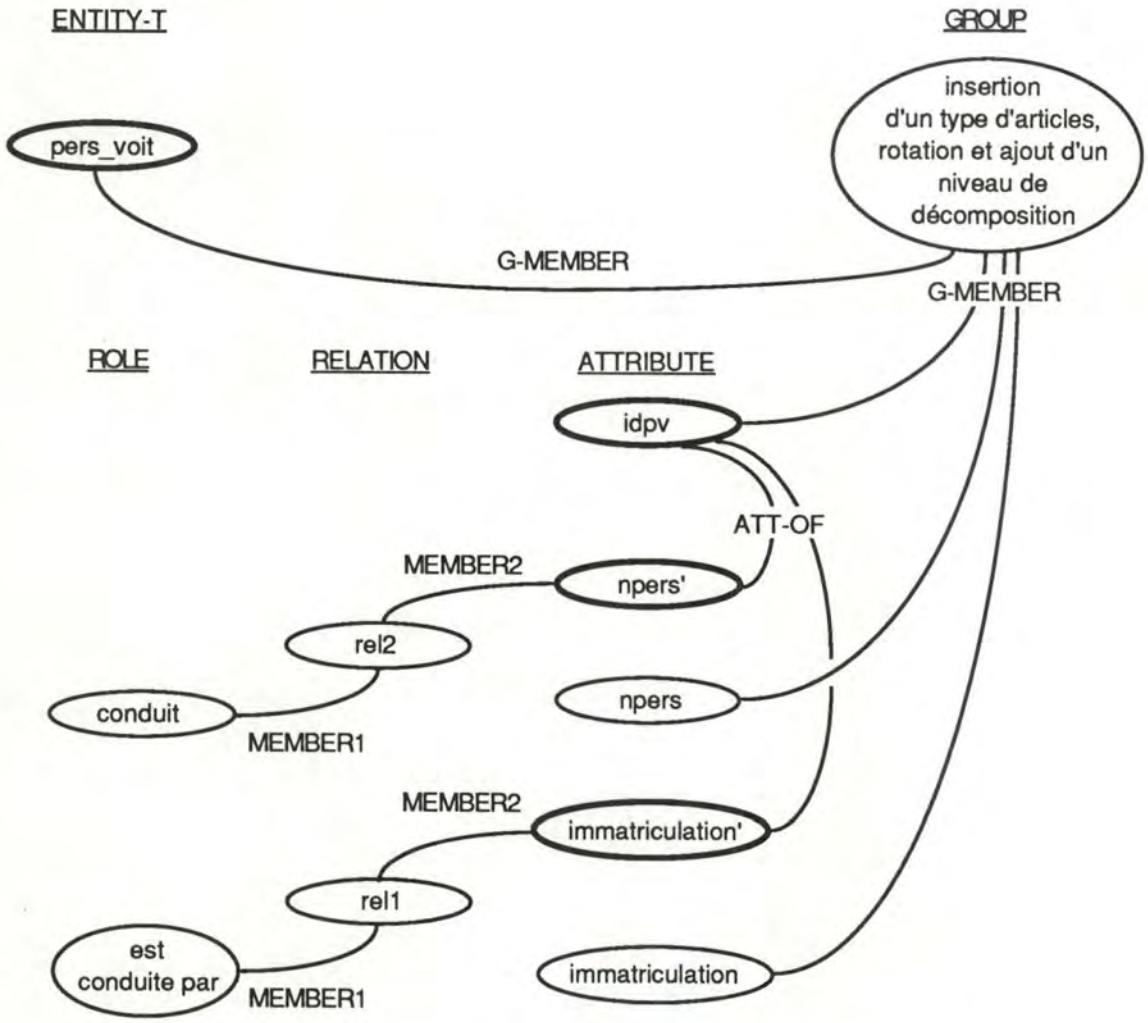


figure A.5 : représentation de l'élimination du type de chemins "c" dans la base des spécifications.

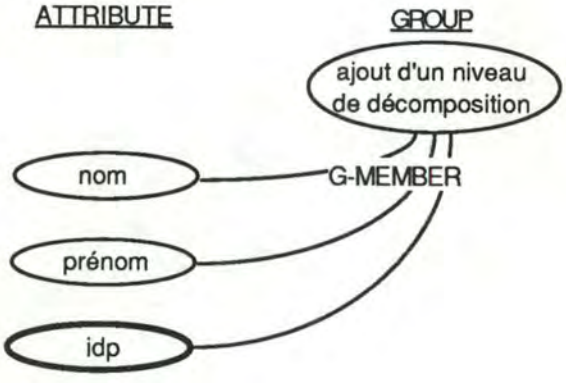


figure A.6 : représentation de l'ajout d'un niveau de décomposition.

ALGORITHME CONFORME

L'algorithme effectif initial devient l'algorithme conforme suivant. Les termes en gras sont nouvellement apparus.

```
algorithm accident

var numvoiture :  string (6);
    v          :  ref of voiture;
    p          :  ref of personne;
    pv         :  ref of pers_voit;

begin
   lire (numvoiture);
   v := voiture (: immatriculation = numvoiture);
   imprimer ('voici la liste des personnes concernees :');
   forpv := pers_voit (: idpv(: immatriculation' = (v). immatriculation))
      p := personne (: npers = (pv).idpv.npers');
      imprimer ( (p).idp.nom, (p).idp.prenom, (p).adresse.rue,
      (p).adresse.numero, (p).adresse.localite );
   endfor
end.
```

REPRESENTATION DE L'ALGORITHME CONFORME

L'arbre représentant l'algorithme effectif initial devient l'arbre suivant. Les "Δ" en trait accentué "Δ" font partie des sous–arbres différenciant les deux arbres.

Δ structure_d'algorithme (1) fait référence à "accident" dans la table des symboles.
ΔΔ section_déclaration_interne (2)
ΔΔΔ section_déclaration_type (3)
ΔΔΔ section_déclaration_variable (5)
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔΔΔ string (9) fait référence à "6" dans la table des constantes.
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "v" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "voiture" dans la table des symboles.
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "personne" dans la table des symboles.
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "pv" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "pers_voit" dans la table des symboles.
ΔΔ instructions (19)
ΔΔΔ appel (29) fait référence à "lire" dans la table des symboles.
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔ assignation (20)
ΔΔΔΔ class_var (50)

ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "v" dans la table des symboles.
ΔΔΔΔ expression_assignation (21)
ΔΔΔΔΔ expression_collection (64) fait référence à "voiture" dans la table des symboles.
ΔΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔΔ cond_sélection_item (67) fait référence à "immatriculation" dans la table des symboles.
ΔΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔ appel (29) fait référence à "imprimer" dans la table des symboles.
ΔΔΔΔ constante_caractère (83) fait référence à 'voici la liste des personnes concernees' dans la table des constantes.
ΔΔΔ instr_for (22)
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "pv" dans la table des symboles.
ΔΔΔΔ expression_collection (64) fait référence à "pers_voit" dans la table des symboles.
ΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔ cond_sélection_item (67) fait référence à "idpv" dans la table des symboles.
ΔΔΔΔΔΔΔ suite_cond_sélection_items (68)
ΔΔΔΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔΔΔΔ cond_sélection_item (67) fait référence à "immatriculation'" dans la table des symboles.
ΔΔΔΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔΔΔΔ var_refs (53) fait référence à "v" dans la table des symboles.
ΔΔΔΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "immatriculation" dans la table des symboles.
ΔΔΔΔ instructions (19)
ΔΔΔΔΔ assignation (20)
ΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔ expression_assignation (21)
ΔΔΔΔΔΔΔ expression_collection (64) fait référence à "personne" dans la table des symboles.
ΔΔΔΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔΔΔΔ cond_sélection_item (67) fait référence à "npers" dans la table des symboles.
ΔΔΔΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔΔΔΔ var_refs (53) fait référence à "pv" dans la table des symboles.
ΔΔΔΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "idpv" dans la table des symboles
ΔΔΔΔΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "npers'" dans la table des symboles.
ΔΔΔΔΔ appel (29) fait référence à "imprimer" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "idp" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "nom" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)

ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "idp" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "prenom" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "adresse" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "rue" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "adresse" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numero" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "adresse" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "localite" dans la table des symboles.

Le SGD cible est Codasyl

SCHEMA CONFORME

Le schéma des accès nécessaires devient le schéma conforme suivant :

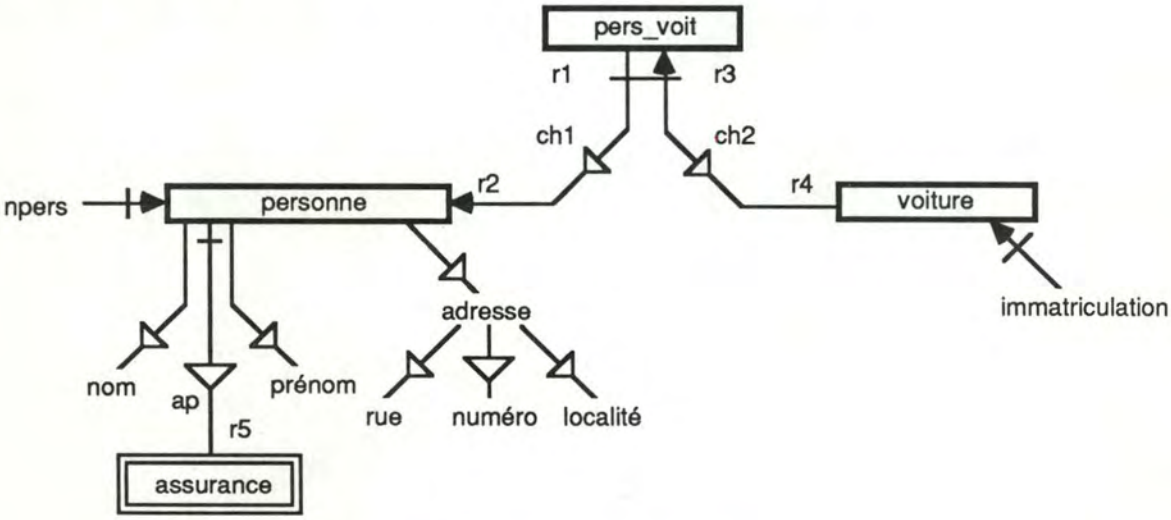


figure A.7 : schéma conforme à Codasyl.

REPRESENTATION DE LA TRANSFORMATION DANS LA BASE DE L'ATELIER

Deux entités GROUP rendent compte des transformations effectuées. L'attribut TYPE de la première est "insertion d'un type d'articles"; l'attribut TYPE de la seconde est "ajout d'un composant rôle à un identifiant.

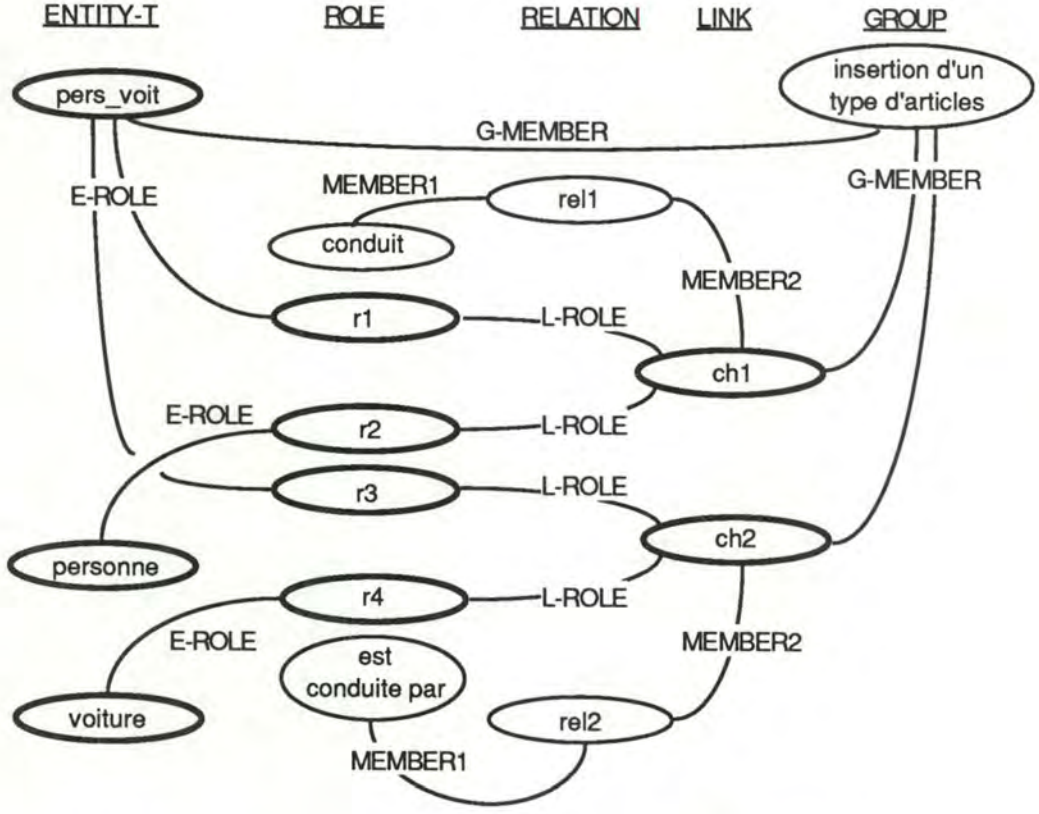


figure A.8 : représentation de l'insertion du type d'articles "pers_voit", dans la base des spécifications.

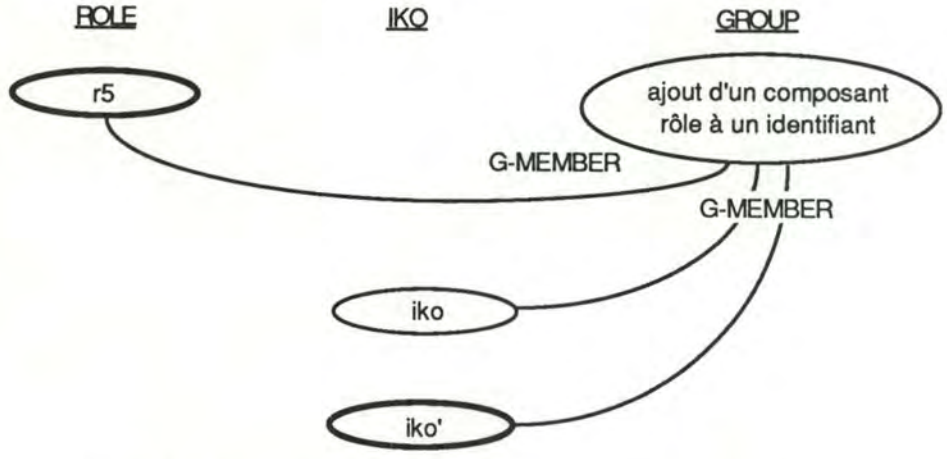


figure A.9 : représentation de l'ajout du composant rôle "r5", dans la base des spécifications.

ALGORITHME CONFORME

L'algorithme effectif initial devient l'algorithme conforme suivant. Les termes en gras sont nouvellement apparus.

```
algorithm accident

var numvoiture :  string (6);
    v          :  ref of voiture;
    p          :  ref of personne;
    pv         :  ref of pers_voit;

begin
    lire (numvoiture);
    v := voiture (: immatriculation = numvoiture);
    imprimer ('voici la liste des personnes concernees :');
    for pv := pers_voit ( ch2 : v )
        p := personne ( ch1 : pv );
        imprimer ( (p).nom, (p).prenom, (p).adresse.rue, (p).adresse.numero,
        (p).adresse.localite );
    endfor
end.
```

REPRESENTATION DE L'ALGORITHME CONFORME

L'arbre représentant l'algorithme effectif initial devient l'arbre suivant. Les "Δ" en trait accentué "Δ" font partie des sous–arbres différenciant les deux arbres.

Δ structure_d'algorithme (1) fait référence à "accident" dans la table des symboles.
ΔΔ section_déclaration_interne (2)
ΔΔΔ section_déclaration_type (3)
ΔΔΔ section_déclaration_variable (5)
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔΔΔ string (9) fait référence à "6" dans la table des constantes.
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "v" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "voiture" dans la table des symboles.
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "personne" dans la table des symboles.
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "pv" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "pers_voit" dans la table des symboles.
ΔΔ instructions (19)
ΔΔΔ appel (29) fait référence à "lire" dans la table des symboles.
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔ assignation (20)
ΔΔΔΔ class_var (50)

ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "v" dans la table des symboles.
ΔΔΔΔ expression_assignation (21)
ΔΔΔΔΔ expression_collection (64) fait référence à "voiture" dans la table des symboles.
ΔΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔΔ cond_sélection_item (67) fait référence à "immatriculation" dans la table des symboles.
ΔΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔ appel (29) fait référence à "imprimer" dans la table des symboles.
ΔΔΔΔ constante_caractère (83) fait référence à 'voici la liste des personnes concernees' dans la table des constantes.
ΔΔΔ instr_for (22)
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "pv" dans la table des symboles.
ΔΔΔΔ expression_collection (64) fait référence à "pers_voit" dans la table des symboles.
ΔΔΔΔΔ condition_sur_chemin (74)
ΔΔΔΔΔΔ nom (8) fait référence à "ch2" dans la table des symboles.
ΔΔΔΔΔΔ nom (8) fait référence à "v" dans la table des symboles.
ΔΔΔΔ instructions (19)
ΔΔΔΔΔ assignation (20)
ΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔ expression_assignation (21)
ΔΔΔΔΔΔΔ expression_collection (64) fait réference à "personne" dans la table des symboles.
ΔΔΔΔΔΔΔΔ condition_sur_chemin (74)
ΔΔΔΔΔΔΔΔΔ nom (8) fait référence à "ch1" dans la table des symboles.
ΔΔΔΔΔΔΔΔΔ nom (8) fait référence à "pv" dans la table des symboles.
ΔΔΔΔΔ appel (29) fait référence à "imprimer" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "nom" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "prenom" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "adresse" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "rue" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "adresse" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numero" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "adresse" dans la table des symboles.
ΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "localite" dans la table des symboles.

Le SGD cible est SQL

SCHEMA CONFORME

Le schéma des accès nécessaires devient le schéma conforme suivant :

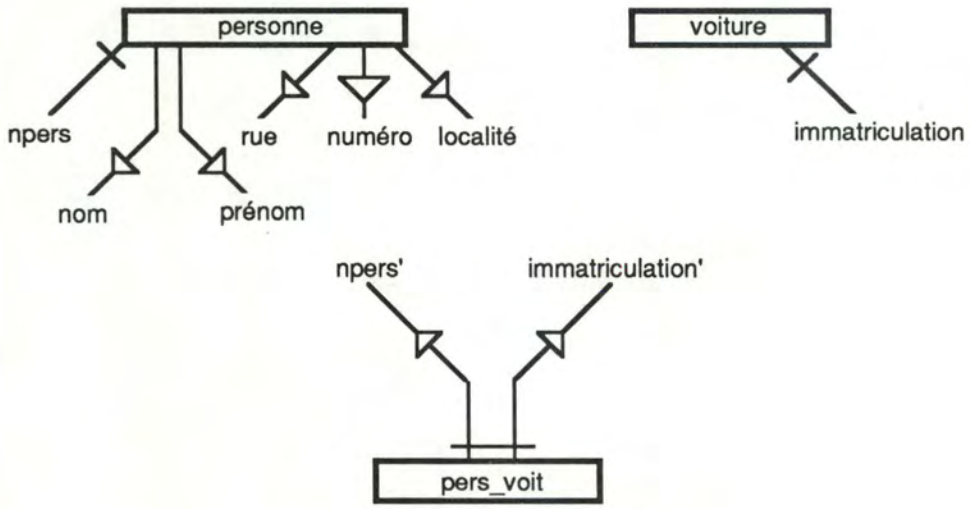


figure A.10 : schéma conforme à SQL.

REPRESENTATION DE LA TRANSFORMATION DANS LA BASE DE L'ATELIER

Deux entités GROUP rendent compte des transformations effectuées. L'attribut TYPE de la première est "insertion d'un type d'articles et duplication d'items"; l'attribut TYPE de la seconde est "aplatissement total".

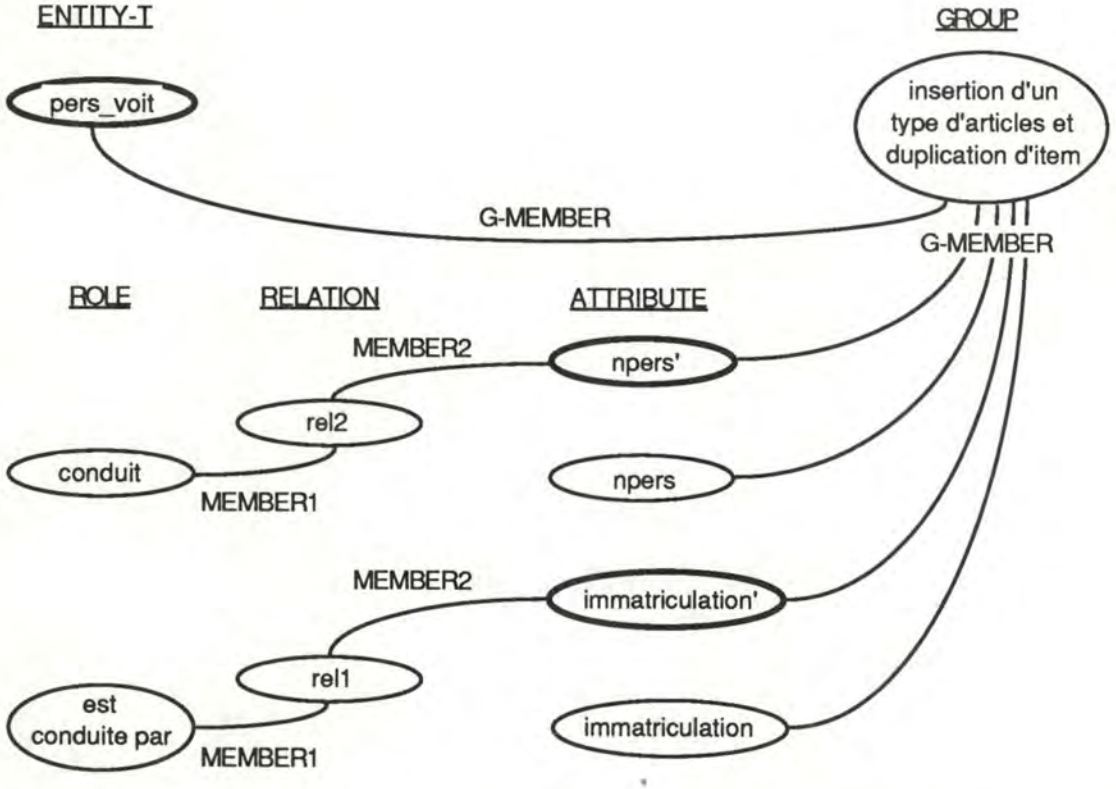


figure A.11 : représentation de l'insertion du type d'articles "pers_voit", et de la duplication d'item, dans la base des spécifications.

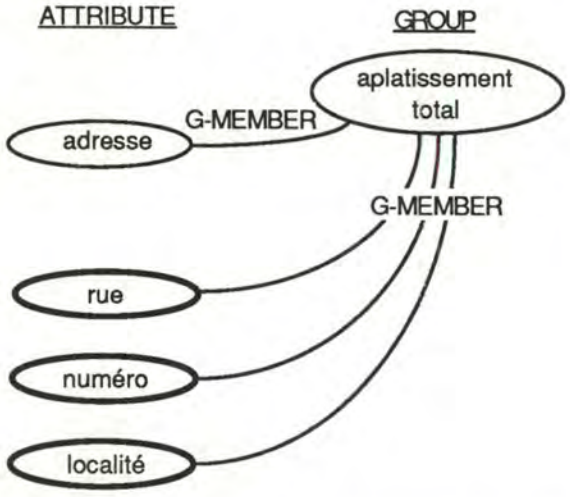


figure A.12 : représentation de l'aplatissement de "adresse", dans la base des spécifications.

ALGORITHME CONFORME

L'algorithme effectif initial devient l'algorithme conforme suivant. Les termes en gras sont nouvellement apparus.

```
algorithm accident

var numvoiture :  string (6);
    v          :  ref of voiture;
    p          :  ref of personne;
    pv         :  ref of pers_voit;

begin
    lire (numvoiture);
    v := voiture (: immatriculation = numvoiture);
    imprimer ('voici la liste des personnes concernees :');
    for pv := pers_voit (: immatriculation' = (v). immatriculation)
        p := personne (: npers = (pv).npers');
        imprimer ( (p).nom, (p).prenom, (p).rue, (p).numero, (p).localite );
    endfor
end.
```

REPRESENTATION DE L'ALGORITHME CONFORME

L'arbre représentant l'algorithme effectif initial devient l'arbre suivant. Les "Δ" en trait accentué "Δ" font partie des sous–arbres différenciant les deux arbres.

Δ structure_d'algorithme (1) fait référence à "accident" dans la table des symboles.
ΔΔ section_déclaration_interne (2)
ΔΔΔ section_déclaration_type (3)
ΔΔΔ section_déclaration_variable (5)
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔΔΔ string (9) fait référence à "6" dans la table des constantes.
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "v" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "voiture" dans la table des symboles.
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "personne" dans la table des symboles.
ΔΔΔΔ déclaration_variable (6)
ΔΔΔΔΔ noms (7)
ΔΔΔΔΔΔ nom (8) fait référence à "pv" dans la table des symboles.
ΔΔΔΔΔ ref_de (17) fait référence à "pers_voit" dans la table des symboles.
ΔΔ instructions (19)
ΔΔΔ appel (29) fait référence à "lire" dans la table des symboles.
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔ assignation (20)
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "v" dans la table des symboles.

ΔΔΔΔ expression_assignation (21)
ΔΔΔΔΔ expression_collection (64) fait référence à "voiture" dans la table des symboles.
ΔΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔΔ cond_sélection_item (67) fait référence à "immatriculation" dans la table des symboles.
ΔΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numvoiture" dans la table des symboles.
ΔΔΔ appel (29) fait référence à "imprimer" dans la table des symboles.
ΔΔΔΔ constante_caractère (83) fait référence à 'voici la liste des personnes concernees' dans la table des constantes.
ΔΔΔ instr_for (22)
ΔΔΔΔ class_var (50)
ΔΔΔΔΔ variable_non_hiérar (51) fait référence à "pv" dans la table des symboles.
ΔΔΔΔ expression_collection (64) fait référence à "pers_voit" dans la table des symboles.
ΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔ cond_sélection_item (67) fait référence à "immatriculation'" dans la table des symboles.
ΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔ var_refs (53) fait référence à "v" dans la table des symboles.
ΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "immatriculation" dans la table des symboles.
ΔΔΔΔ instructions (19)
ΔΔΔΔΔ assignation (20)
ΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔ expression_assignation (21)
ΔΔΔΔΔΔΔ expression_collection (64) fait réference à "personne" dans la table des symboles.
ΔΔΔΔΔΔΔΔ conditions_sélection_items (66)
ΔΔΔΔΔΔΔΔΔ cond_sélection_item (67) fait référence à "npers" dans la table des symboles.
ΔΔΔΔΔΔΔΔΔΔ suite_cond_sélection_item (68)
ΔΔΔΔΔΔΔΔΔΔΔ égal (71)
ΔΔΔΔΔΔΔΔΔΔΔ var_refs (53) fait référence à "pv" dans la table des symboles.
ΔΔΔΔΔΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "npers'" dans la table des symboles.
ΔΔΔΔΔ appel (29) fait référence à "imprimer" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "nom" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "prenom" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "rue" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)
ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "numero" dans la table des symboles.
ΔΔΔΔΔΔ var_refs (53) fait référence à "p" dans la table des symboles.
ΔΔΔΔΔΔΔ class_var (50)

ΔΔΔΔΔΔΔΔ variable_non_hiérar (51) fait référence à "localite" dans la table des symboles.