MASTER EN SCIENCES INFORMATIQUES

Outil d'aide à la conception d'une base de données relationnelle

de Crane d'Heysselaer, Jean; Gonay, André

Award date:
1986

Awarding institution:
Universite de Namur

FACULTÉS
UNIVERSITAIRES
N. D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

Outil d'Aide à la Conception d'une Base de Données Relationelle

Promoteur : F. Bodart

Mémoire présenté par

Jean de Crane d'Heysselaer
André Gonay

en vue de l'obtention du titre de
Licencié et Maître en Informatique

Année Académique 1985-1986

Remerciements

Nous tenons tout d'abord à remercier Monsieur le Professeur François BODART, promoteur de ce mémoire pour les conseils qu'il nous a prodigués et le suivi qu'il a apporté à ce mémoire. Son expérience en la matière nous a été très profitable. Nous remercions Monsieur Béat MICHEL, pour son acceuil à Lausanne ainsi que l'aide, les conseils judicieux et le temps qu'il nous a consacrés. Nous tenons également à remercier Monsieur le Professeur Jean-Luc HAINAUT pour les réponses qu'il a pu donner à nos questions, ainsi que les assistants, Benoît VANHOUTTE, Alain DELCOURT et Mario CADELLI pour l'aide apportée à l'utilisation et le temps consacré à la mise-au-point des programmes qu'ils ont mis à notre disposition et Marcel CLANTIN et Jean-Luc FOUCART pour l'aide technique apportée.

Nous remercions également la société Charles VEILLON et son département informatique, Monsieur Georges GRIMA ainsi que l'équipe de maintenance de la société ADR France de leur acceuil pour notre stage chez eux.

Enfin, il nous tient à coeur de remercier tout particulièrement nos familles, grâçe à qui nous avons pu mener à bien nos cinq années d'études.

TABLE DES MATIERES

Introduction 4

Ie Partie : Cadre méthodologique pour la Conception d'une Base de Données Relationnelle

I.1 Conception d'une base de données partant d'un modèle sémantique Entité-Association 6
I.1.1 Introduction 6
I.1.2 Le Niveau Conceptuel 8
I.1.3 Le Niveau Logique 9
I.1.4 Le Niveau Physique 19

I.2 Démarche choisie dans le cadre du mémoire 20
I.2.1 Objectifs 20
I.2.2 Démarche méthodologique 20
I.2.3 Support logiciel 22
I.2.4 Justification 23
I.2.5 Limitations 24
I.2.6 Comparaison avec les autres démarches 25

IIe Partie : Spécification de l'interface

II.1 Présentation de l'environnement ADR 26
II.1.1 L'utilisateur IDEAL 26
II.1.2 Description du dictionnaire d'ADR 29
II.1.2.1 Description générale 29
II.1.2.2 Analyse des objets du dictionnaire 33
II.1.2.3 Utilisation du dictionnaire 37

II.2 Bref rappel de l'environnement IDA 38
II.2.1 Rappel des outils de l'environnement IDA 38
II.2.2 Le modèle Entité-Association 40
II.2.2.1 Rappel du modèle 40
II.2.2.2 La forme canonique du schéma conceptuel . . . 41

II.3 Un point de vue sur l'environnement "Idéal" . . . 42

II.4 Description de l'architecture fonctionnelle . . . 43
II.4.1 Présentation générale 43
II.4.2 La Base de Spécifications Relationnelles (BSR) . 45
II.4.3 Les automates de transformations de schémas . . 47
II.4.3.1 Le Transformateur IDA/BSR 47
II.4.3.2 Le Transformateur BSR/ADR 51
II.4.4 Les éditeurs de schémas relationnels 56
II.4.4.1 Le Présentateur Relationnel 56
II.4.4.1.1 Transformations de Schémas 56
II.4.4.1.1.1 Transformations sur les Schémas de Relations 57
II.4.4.1.1.2 Transformations sur les Constituants . . . 61
II.4.4.1.1.3 Justifications des opérateurs de transformations 65
II.4.4.1.2 Définition des Accès 66
II.4.4.1.2.1 Gestion des Clés 66

II.4.4.1.2.2 Gestion des Index 68
II.4.4.1.3 Aides à l'Optimalisation 69
II.4.4.1.4 Services 74
II.4.4.1.5 Paramétrages 75
II.4.4.2 L'Editeur des Mises-à-jour 76
II.4.5 Utilitaires 77
II.4.5.1 L'Extracteur Conceptuel 77
II.4.5.2 Le Répercuteur des Correspondances 78
II.4.5.3 Le Transformateur de Textes 80

IIIe Partie : Réalisation de l'interface

III.1 Environnement 81

III.2 Interface Homme/Machine 82
III.2.1 Le dialogue 82
III.2.2 Les procédures de dialogue 83

III.3 Architecture de Programmation 85
III.3.1 Niveaux 4, 5 et 6 87
III.3.1.1 Filière des Dialogues 87
III.3.1.1.1 Introduction 87
III.3.1.1.2 Description des modules 88
III.3.1.2 Filière des Données 140
III.3.2 Niveau 3 172
III.3.3 Niveau 0, 1 et 2 184

Conclusion 200
Bibliographie 201

IVe Partie : Annexe

Mode d'emploi 203

Introduction

Le développement d'un système d'information couvre différentes phases qui sont :

- "l'ETUDE D'OPPORTUNITE qui prépare un avant-projet de solution à partir des besoins exprimés par l'organisation,

- l'ANALYSE CONCEPTUELLE qui, sur base de l'avant-projet, élabore une solution -conceptuelle- détaillée mais indépendante de tout moyen de réalisation,

- la CONCEPTION DE MISE-EN-OEUVRE qui affine et définit la solution retenue par la prise en considération des caractéristiques logiques des moyens de réalisation - humains, techniques et organisationnels-, et

- la REALISATION ET LA MISE-AU-POINT de la solution exécutable en fonction des caractéristiques réelles des matériels, des logiciels et de l'organisation." [BOD-PIG 83]

Des outils logiciels tels que ceux offerts par IDA offrent une aide à l'analyste pour réaliser les deux premières étapes du développement d'un système d'information. Ces outils permettent de vérifier le caractère complet, cohérent, réalisable et conforme aux besoins de la solution conceptuelle développée au cours de ces deux étapes.

D'autres outils logiciels doivent également être mis en oeuvre pour réaliser les deux dernières étapes de développement, et ceci au départ de la solution conceptuelle. Un environnement tel que celui offert par ADR et ses composants DATACOM/DB, DATACOM/DD et IDEAL permet cette réalisation.

Ainsi en est-il des développements réalisés par la société Charles Veillon S.A. (société de distribution de vêtements, textile de maison, bijoux et maroquinerie, opérant principalement par le canal de la vente par correspondance et disposant d'un service informatique occupant 35 employés et équipé d'un IBM 3083). La méthode et l'atelier d'aide à la conception d'applications informatiques IDA y ont été introduits depuis l'automne 1984. Le langage IDEAL est utilisé pour les applications opérationnelles actuelles et futures conjointement avec la base de données DATACOM/DB. En vue de disposer d'un environnement intégré pour le développement d'un système d'information, basé sur une méthode de conception et un langage de quatrième génération, il était nécessaire de disposer d'un interface entre l'environnement IDA et ADR, permettant ainsi de disposer d'une méthodologie et d'un atelier logiciel intégré qui couvre l'ensemble du cycle de vie d'un projet, tout en fournissant des langages de haut niveau.

L'objet de ce mémoire est la réalisation de cet interface dont les objectifs sont :

- Contribuer à la conformité de l'implémentation par rapport aux spécifications. Ceci implique le contrôle des transformations entre les représentations IDA et les représentations IDEAL.

- Assurer la correspondance rigoureuse entre les objets IDA et les objets manipulés dans IDEAL, afin, notamment, de garantir une maintenance efficace.

- Limiter la redondance entre les tâches de conception et d'implémentation, afin d'accroître la productivité des informaticiens, de réduire les obstacles psychologiques et de minimiser les sources d'erreurs.

La méthode de travail pour la réalisation de cette interface se décompose en trois parties. Dans un premier temps, nous avons étudié une démarche d'aide à la conception d'une base de donnée relationelle avec le maître d'oeuvre de chez Charles Veillon. Nous avons ensuite effectué un stage chez ADR France pour l'approfondissement de notre connaissance sur l'environnement IDEAL et l'affinement de nos spécifications conceptuelles. Enfin une stage chez Charles Veillon nous a permis de réaliser des spécifications concrètes de programmation. De retour à Namur, nous avons réalisé l'interface.

Ce mémoire est divisé en quatre parties. La première partie précise la démarche méthodologiques pour la conception d'une base de données relationnelles adoptée par rapport à d'autres démarches de conception. La deuxième partie procure une spécification fonctionnelle détaillée de l'interface ainsi qu'une présentation des environnements en aval (ADR) et en amont (IDA). Nous y décrirons également l'interface qui a été développé pour réaliser le lien entre IDA et ADR. La troisième partie contient une description de la réalisation de cet interface : spécification des modules et des primitives illustrant les programmes qui ont été développés. Un mode d'emploi des différents types de dialogue entre l'utilisateurs et l'interface est décrit en annexe.

Sur recommandation du promoteur de mémoire, nous avons supposé en rédigeant ce document que celui-ci s'adressera à des lecteurs familiers avec les notions du schéma Entité-Association et du schéma relationnel. De même, ce mémoire n'est par un dossier de programmation mais tente de mettre en évidence des choix méthodologiques, fonctionnels et d'implémentations.

PREMIERE PARTIE

CADRE METHODOLOGIQUE POUR LA CONCEPTION D'UNE BASE DE DONNEES RELATIONNELLE

I.1 Conception d'une base de données partant d'un modèle sémantique Entité-Association.

I.1.1 Introduction

Conception à trois niveaux

La conception d'une base de données relationnelle ou autre peut être abordée de différentes manières. Nous avons opté pour une approche décomposée en trois niveaux qui sont les niveaux conceptuel, logique et physique.

Au niveau Conceptuel, on donnera une description du système d'information totalement indépendante de toutes caractéristiques techniques du système informatique sur lequel il sera implémenté. Le but étant de donner une représentation la plus proche possible du système réel. L'on y décrira aussi bien les données, les traitements que les ressources mises en jeu. Pour ce qui est de la structure des données, décrite dans un schéma conceptuel, l'on parlera en terme d'Entités, d'Associations et d'Attributs.

Au niveau Logique, on tentera de donner une description des données, tenant compte des applications s'y rapportant, des quantifications concernant les données et des opérations sur ces dernières, ainsi que des contraintes du Système de Gestion de Base de Données et de tout l'environnement système et logiciel qui sera utilisé. L'on obtiendra ainsi un schéma logique exprimé en terme Record, Item et Relation inter Record.

Au niveau Physique on décrira les caractéristiques physiques et d'implémentations du schéma. Ceci dépendra fortement du système utilisé. On effectuera la définition de Blocs, d'Index, de Pointeurs, de Clusters, de Fichiers, etc.

La frontière entre ces deux derniers niveaux n'est pas toujours aussi claire et dépend fortement des interprétations du concepteur de bases de données.

Le modèle sémantique

Etant donné l'objectif (cfr Introduction) qui nous est assigné, nous nous situons dans une approche où l'on part d'un modèle sémantique exprimé sous le modèle Entité-Association. Une illustration de cette approche sera donnée par les travaux effectués par :

(a) The DATAID-1 Project [CERI 83], [ANT-LEV 84],
(b) l'Atelier de Conception de Base de Données de l'Institut d'Informatique de Namur dans [HAINAUT 85], [CHA-MUL 86],
(c) T.J. Teorey, D. Yang et P. Fry dans [TE-YA-FR-85], [TEO-FRY 82].

Tous partent du Modèle Entité-Association comme modèle pour exprimer le schéma conceptuel de données pour aboutir à la description d'une base de données de type CODASYL ou Relationnel.

Ces trois cas ne sont que des exemples. Nous citerons néanmoins une autre démarche qui consiste à partir d'un modèle Entité-Association normalisé. Le lecteur intéressé pourra consulter [LING 85], [CHUNG 81] pour la définition de cette forme normale. Cette forme normale se base sur les concepts du modèle Entité-Association ainsi que sur la théorie des dépendances fonctionnelles [BERN 76], [FAGIN 77], [DEL-ADI 82]. Ling décrit également dans [LING 85] un algortihme permettant d'obtenir un schéma relationnel, dont les relations sont toutes sous troisième ou cinquième forme normale, à partir d'un schéma Entité-Association normalisé. Il y décrit également une méthode pour normaliser un schéma Entité-Association.

I.1.2 Le Niveau Conceptuel

La description du système d'information est composée d'un ensemble de schémas conceptuels des données et des traitements ainsi que d'un relevé des quantifications de ces données et de ces traitements (Quantifications qui peuvent être obtenues après simulation). Les schémas conceptuels seront consolidés pour former un schéma conceptuel global. Des techniques de consolidation et d'intégration sont décrites dans [BOD-PIG 83] et [TEO-FRY 82].

Comme nous l'avons déjà dit, les trois cas étudiés partent du même schéma conceptuel décrit à l'aide du modèle Entité-Association pouvant également décrire des hiérarchies d'abstractions tels que le sous-typage, la généralisation (également appelée hiérarchie is-a) et la hiérachie complète pour certains tels que dans (a) et (c).

Des divergences existent néanmoins quant au contenu du schéma conceptuel des traitements et de la terminologie utilisée.

Pour (b) ce schéma doit comprendre la description statique et dynamique de chaque fonction de l'application tel que décrite dans [BOD-PIG 83] sur base des modèles de la dynamique et de la statique des traitements.

Pour (c) cette description se fera à l'aide d'un diagramme de flux.

Pour (a) ce schéma est décomposé en deux; le schéma des opérations et le schéma des évènements. Ces deux schémas offrent une spécification des programmes d'accès à la base et devrait servir comme passerelle à des logiciels méthodologiques de conception d'applications informatiques.

Le schéma des opérations est la consolidation d'un ensemble de schémas conceptuels décrivant pour chaque opération ou fonction les données requises par cette fonction, les opérations effectuées sur ces attributs (Accès direct ou séquentiel, argument de sélection, mise-à-jour, suppression, consultation, résultat), ainsi que le chemin suivi pour y accéder. Ce schéma sera décrit à l'aide du modèle Entité-Association.

Le schéma des évènements est la consolidation d'un ensemble de schémas décrivant chaque fonction en termes de conditions et d'opérations. Ces schémas sont décrits à l'aide de Réseaux Petri ([CERI 83] p. 44-45).

Conclusion

Indépendamment des modèles choisis, le niveau conceptuel devrait nous fournir :

- un schéma conceptuel des données,
- un schéma conceptuel des traitements,
- une ensemble d'informations quantitatives, portant aussi bien sur les données, que sur les traitements.

I.1.3 Le Niveau Logique

C'est à ce niveau-ci que les différences se font le plus sentir. Nous présenterons les différents cas séparément.

(a) DATAID-1 (Figure I.1.3.1) :

1. Construction de tables des accès logiques :

Une table est créée pour chaque Entité[m] et chaque Association décrivant les accès logiques de chaque opération opérant sur cette Entité ou Association. Ainsi pour chaque opération, on définit quels sont les Attributs qui sont utilisés et pour quels buts (sélection, suppression, mise-à-jour, etc.). L'on y trouvera également des informations concernant le mode d'accès à ces Entités et Associations ainsi que le nombre moyen d'occurrences d'Entités ou Associations utilisées par unité de temps. Ces tables seront mises-à-jour lors des transformations de schémas et permettront de vérifier si les descriptions sont complètes et si toutes les opérations ont été définies. Ainsi, par exemple, il est nécessaire d'avoir un mode d'accès pour chaque Entité et d'avoir une opération de création et de suppression d'occurrence pour chaque Entité ou Association.

2. Simplification du schéma de données :

Les premières transformations de schémas qui seront effectuées sont celles qui consistent à supprimer des structures du modèle Entité-Association qui n'ont pas de correspondant immédiat dans le modèle CODASYL ou Relationnel. Ainsi l'on devra supprimer :

- les hiérarchies,
- les Associations récursives,
- les Associations d'ordre supérieur à 2.

Une hiérarchie pourra être transformée en introduisant une Association de connectivité 1-1 entre les Entités, ou en fusionnant les deux Entités. Une Association récursive ou d'ordre supérieur à 2 pourra être transformée en une Entité héritant des Attributs de cette Association.

L'on profitera également de l'occasion pour définir un identifiant principal pour chaque Entité qui servira de Clé au Record correspondant. Si une Entité dispose de plusieurs identifiants on peut choisir le meilleur en consultant les tables des accès logiques. On pourrait même envisager de créer un nouvel Attribut identifiant (un code par exemple), plus facile d'utilisation.

---

[m] On parlera : d'Entité au lieu de Type d'Entité
d'Association au lieu de Type d'Association

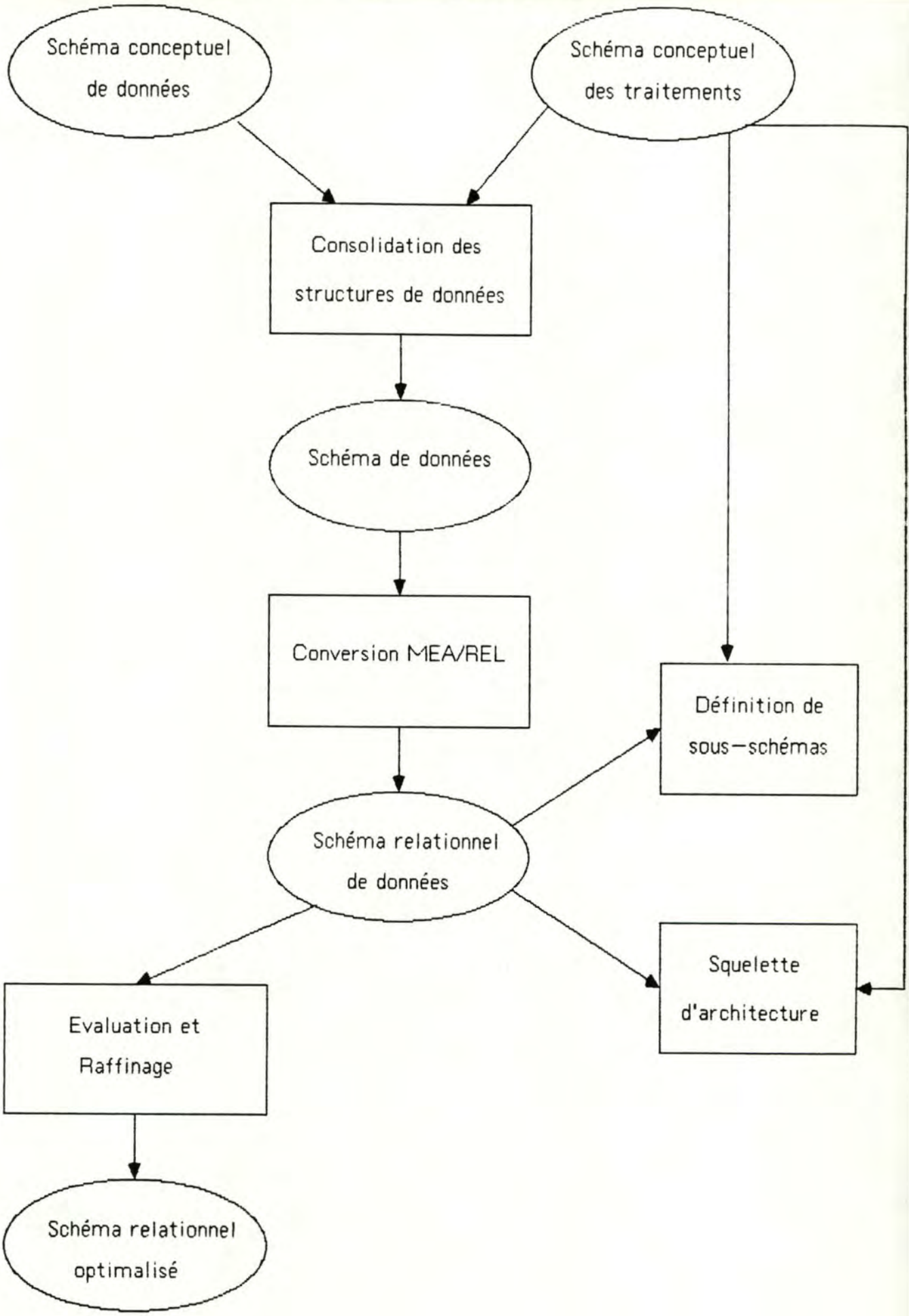


Figure I.1.3.1 : Conception Logique DATAID-1

3. Raffinement du schéma de données :

Le but ici est de transformer les schémas pour optimaliser l'exécution des opérations en tenant compte de deux critères :

- minimiser le nombre d'accès logiques pour une opération,
- minimiser le transfert de données entre mémoire de masse et mémoire centrale.

Trois types de transformations sont proposés :

- la découpe d'une Entité,
- la fusion de deux Entités reliées par une Association de connectivité 1-1,
- la duplication d'Attributs.

Les tables des accès logiques serviront de base à ces transformations et devront également être mises-à-jour après créations ou suppressions de nouvelles Entités, Associations ou Attributs. Il faudra également mettre à jour les opérations s'y rapportant ainsi que définir de nouvelles opérations sur ces nouveaux objets.

La découpe d'Entités :

L'ensemble des Attributs d'une Entité peut être décomposé en plusieurs sous-ensembles héritant chacun des Attributs formant l'identifiant principal. Chacun de ces sous-ensembles fera l'objet d'une nouvelle Entité qui sera reliée à une de ces Entités dite principale via des Associations de connectivité 1-1. Les Associations définies sur l'Entité d'origine devront être transférées vers les Entités reprenant les Attributs les plus souvent accédés par cette Association (cfr. table des accès). Cette transformation permet essentiellement de diminuer les transferts entre mémoire de masse et mémoire centrale.

La fusion d'Entités :

Deux Entités reliées par une Association de connectivité 1-1 peuvent être fusionnées en une seule Entité, entraînant un nombre d'accès logiques moindre si certaines opérations doivent utiliser les deux Entités ainsi que l'Association. La fusion d'Entités reliées par des Associations de connectivité 1-N ou N-M n'est pas envisagée car elle peut entraîner une restructuration complexe aussi bien du schéma de données que des opérations elles-mêmes.

La duplication d'Attributs :

Cette transformation a pour but essentiel de diminuer le nombre d'accès logiques. Cette duplication entraîne bien entendu la création d'opérations supplémentaires devant préserver la cohérence entre données dupliquées.

4. Génération des descriptions logiques.

A partir du dernier schéma de données et du schéma des opérations l'on pourra générer le schéma CODASYL ou Relationnel ainsi que les opérations correspondantes définies sur ces nouveaux objets. L'on abordera ici que la génération de l'aspect Relationnel. Les règles de transformations entre le Modèle Entité-Association et modèle Relationnel sont décrites dans la figure I.1.3.2.

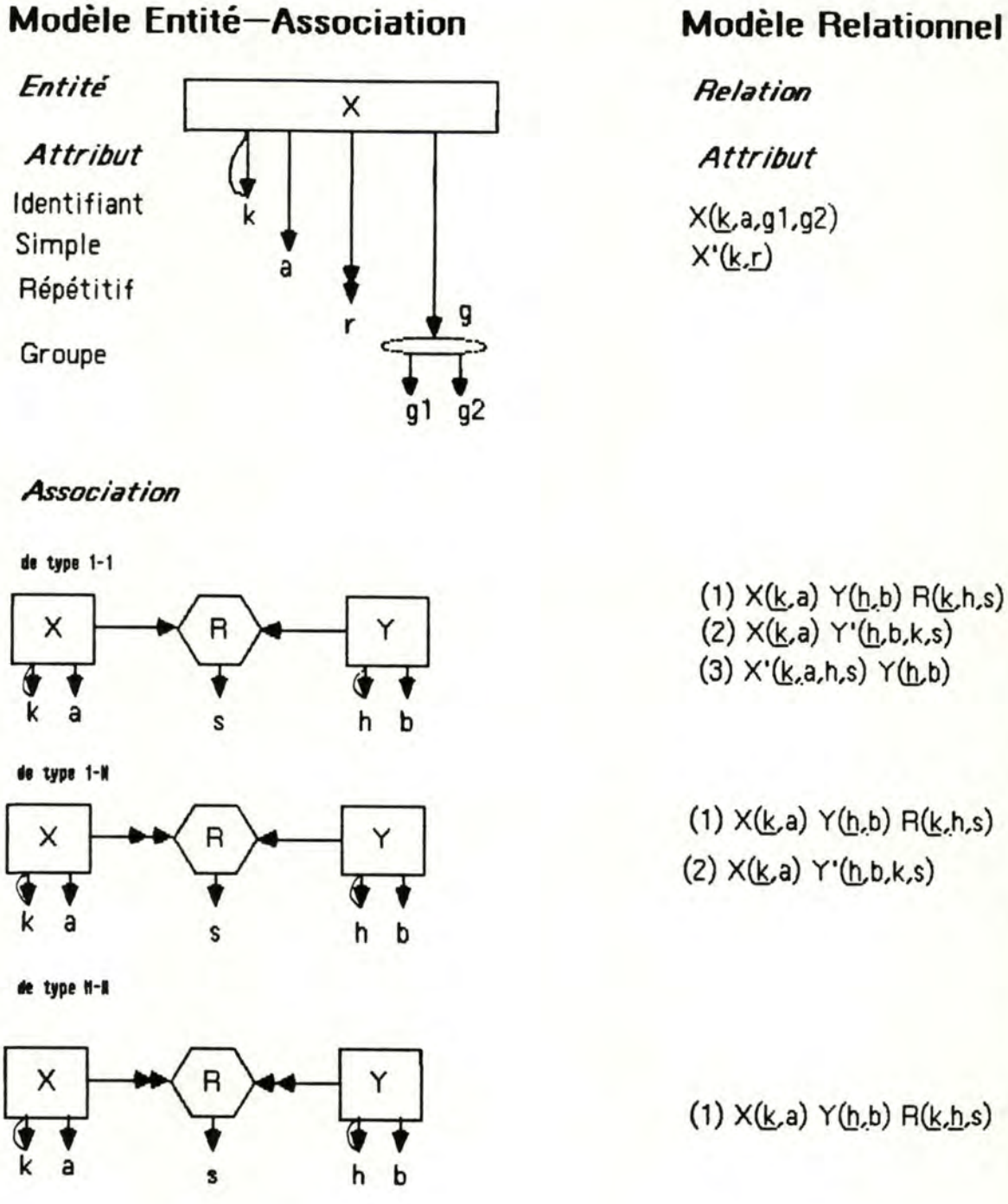


Figure I.1.3.2 : Règles de transformations

(b) l'Atelier de Conception de Base de Données
(Figure I.1.3.3) :

Ici la conception logique de la base de données passe par un modèle intermédiaire appellé Modèle des Accès Généralisé (MAG). "Ce modèle décrit les structures de données non seulement sous l'angle de la sémantique qu'expriment ces données, mais aussi sous celui des accès dont elles peuvent faire l'objet." [HAINAUT 85] Il serait trop long de décrire l'entièreté de ce modèle, ce qui dépasserait de plus le cadre de cette synthèse. Le lecteur intéressé peut consulter la référence précitée.

"Les objets de base de ce modèle sont les articles et types d'articles, les items et valeurs d'items, les chemins d'accès et leurs types, les fichiers, les bases de données, les clés d'accès et les ordres."

La conception logique suit deux filières, la filière des données et celle des traitements. Nous reprenons dans la description qui suit que les processus nécessaires à l'explication de la filière des données.

1. Conversion du modèle Entité-Association en un MAG

   Le schéma conceptuel des données est transformé en un schéma des accès possibles (SAP). Ce dernier est "une représentation correcte aussi directe que possible du schéma conceptuel". Les deux modèles sont assez proches mais certaines transformations doivent néanmoins être réalisées. Ces transformations ainsi que celles utilisées au point 5. ont fait l'objet d'un mémoire [CHA-MUL 86].

2. Construction d'algorithmes

   A partir de la spécification de chaque fonction accédant à la base de données un algorithme prédicatif (décrit sur les conditions de sélection et non sur les spécifications des accès) sera rédigé. Une architecture de modules est également définie.

3. Développement d'algorithmes efficaces

   Sur base du SAP et du relevé des quantifications, on développera pour chaque algorithme prédicatif un algorithme efficace. Le critère envisagé sera celui du nombre d'accès logiques minimum.

4. Dérivation du schéma des accès nécessaires

   Pour chaque algorithme effectif on effectuera le relevé des données et des mécanismes d'accès nécessaires à son bon fonctionnement. La consolidation de tous ces schémas nous fournira le schéma des accès nécessaires (SAN).

Les processus suivants sont classés par l'auteur dans la couche physique dite abstraite. De part ses caractéristiques abstraites, et pour mieux pouvoir comparer avec les autres démarches, nous préférerons les ajouter à la couche logique.

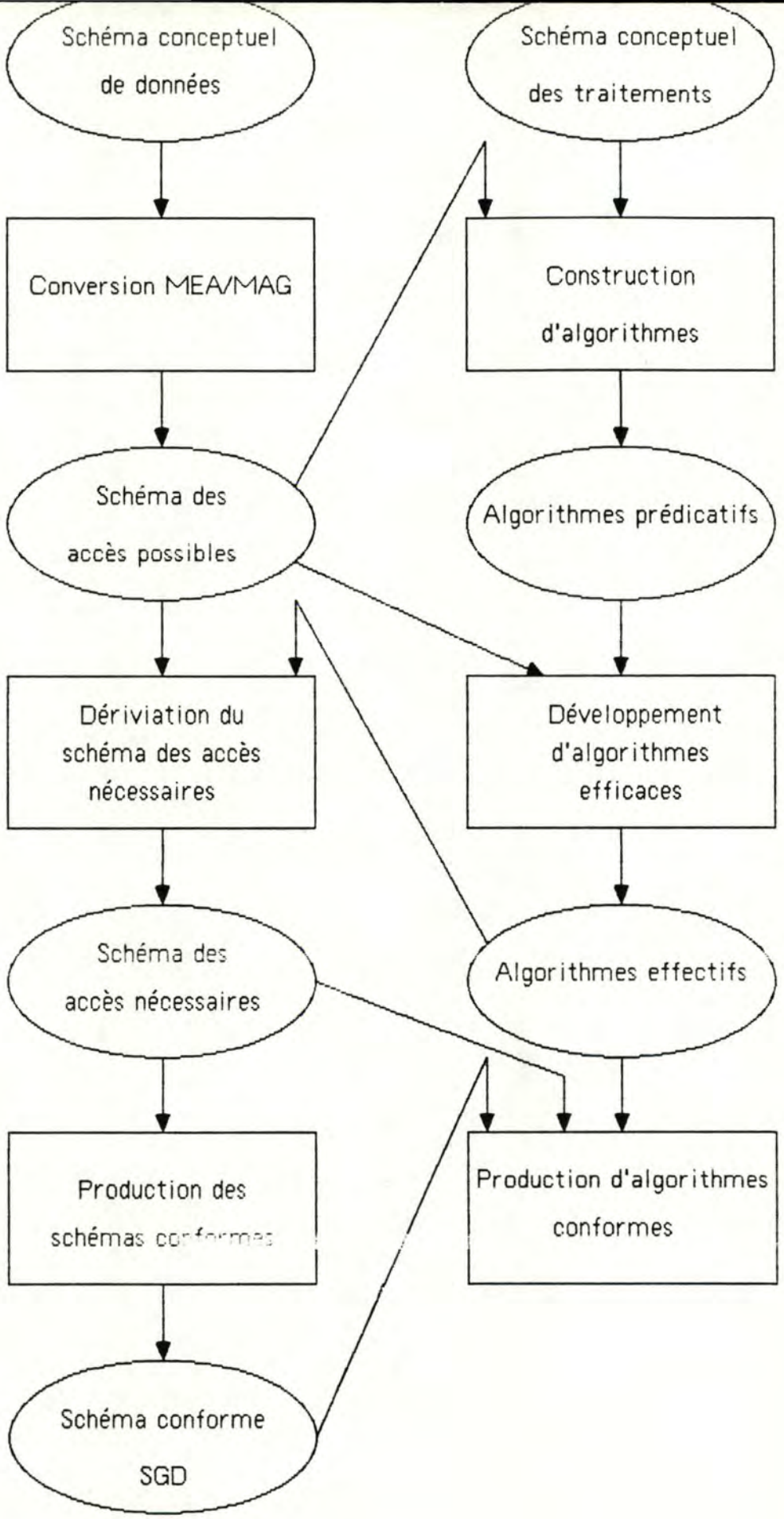


Figure I.1.3.3 : Conception Logique de l'Atelier

5. Production de schémas conformes

A partir du SAN et des contraintes du SGD(*) cible utilisé, on transformera ce schéma en un schéma conforme au SGD. "Un schéma conforme au SGD est donc un schéma MAG qui ne contient que des constructions satisfaisant à la spécification de ce SGD".

6. Production d'algorithmes conformes

Les algorithmes efffectifs doivent eux aussi être transformés pour être conformes au SGD cible, mais également aux constructions admises par le langage de programmation qui sera utilisé. Des règles de transformations de constructions algorithmiques sont décrites dans [HAINAUT 85].

Les transformations proposées par Charlot et Müller dans [CHA-MUL 86] ont pour objectif principal d'assurer l'équivalence sémantique, la réversibilité et de ne pas entraîner de redondances. Hormis les transformations purement théoriques, 7 transformations concrètes ont été proposées ([CHA-MUL 86] p. 54):

- "transformation d'un item en un type d'articles,
- aplatissement d'un item décomposable,
- regroupement d'items en un item décomposable,
- transformation d'un type de chemins en un type d'articles,
- transformation d'un type de chemins par duplication d'items,
- remplacement d'un accès par clé en un accès par type de chemins,
- remplacement d'une connectivité restrictive d'un membre d'un type de chemins par une contrainte d'intégrité à gérer par programme d'application,
- transformation d'un item obligatoire en un item facultatif accompagné d'une contrainte d'intégrité spécifiant la possibilité d'absence de valeur."

(Les transformations permettant l'opération inverse ont également été proposées)

Ainsi pour transformer un SAN en un schéma conforme à un SGD Relationnel, l'on dispose des transformations précitées. Aucune transformation (semi-)automatique ou immédiate de tout le schéma, en un schéma conforme de type Relationnel par exemple, n'est proposée.

(*) SGD Système de Gestion de Données.

(c) Teory, Yang, Fry (figure I.1.3.4) :

1. Définition de structures d'informations locales et consolidation.

   Il s'agit d'analyser chaque processus affectant la base de données pour connaître ces besoins en données. On consolidera ces schémas avec le schéma conceptuel de données existant.

2. Transformations du modèle Entité-Association en un modèle Relationnel.

   En fait la démarche est valable pour d'autres modèles également, mais nous nous limitons au cas Relationnel.

   Le schéma conceptuel est transformé en un schéma analogue, mais décrit sous le modèle relationnel. Les règles de transformations sont décrites dans [TE-YA-FR 85].

   A une Entité correspondra une Relation avec les mêmes informations que celles contenues par l'Entité.

   A une Association correspondra soit une Relation composée des clés des Relations provenant des Entités qu'elle relie, soit une clé d'accès dans une ou dans les deux Relation(s) provenant des Entités qu'elle relie. Ce choix s'effectuera sur base des connectivités de cette Association.

   Des règles précises sont également définies pour transformer des cas particuliers tels que des Associations n-aire (n>2), des hiérarchies de sous-typages ou de généralisation, le cas des aggrégations d'Entités, etc...

   De plus l'analyse des dépendances fonctionnelles devra être faite pour transformer ces relations afin de disposer de relations normalisées (3FN, FNBC, 4FN ou 5FN).

3.1. Définition de sous-schémas.

   Le but de cette étape est de définir l'interface des applications avec le schéma de données. On définira des sous-schémas ou "vues" qui sont des sous-ensembles du schéma de données par lequel chaque programme d'application pourra accéder aux données. Ces "vues" doivent offrir des accès efficaces sans compromettre la sécurité des données.

3.2. Définition de l'architecture de programmation.

   Un squelette de l'architecture sera défini regroupant pour chaque transaction des informations suffisantes pour définir la fonction d'accès aux données.

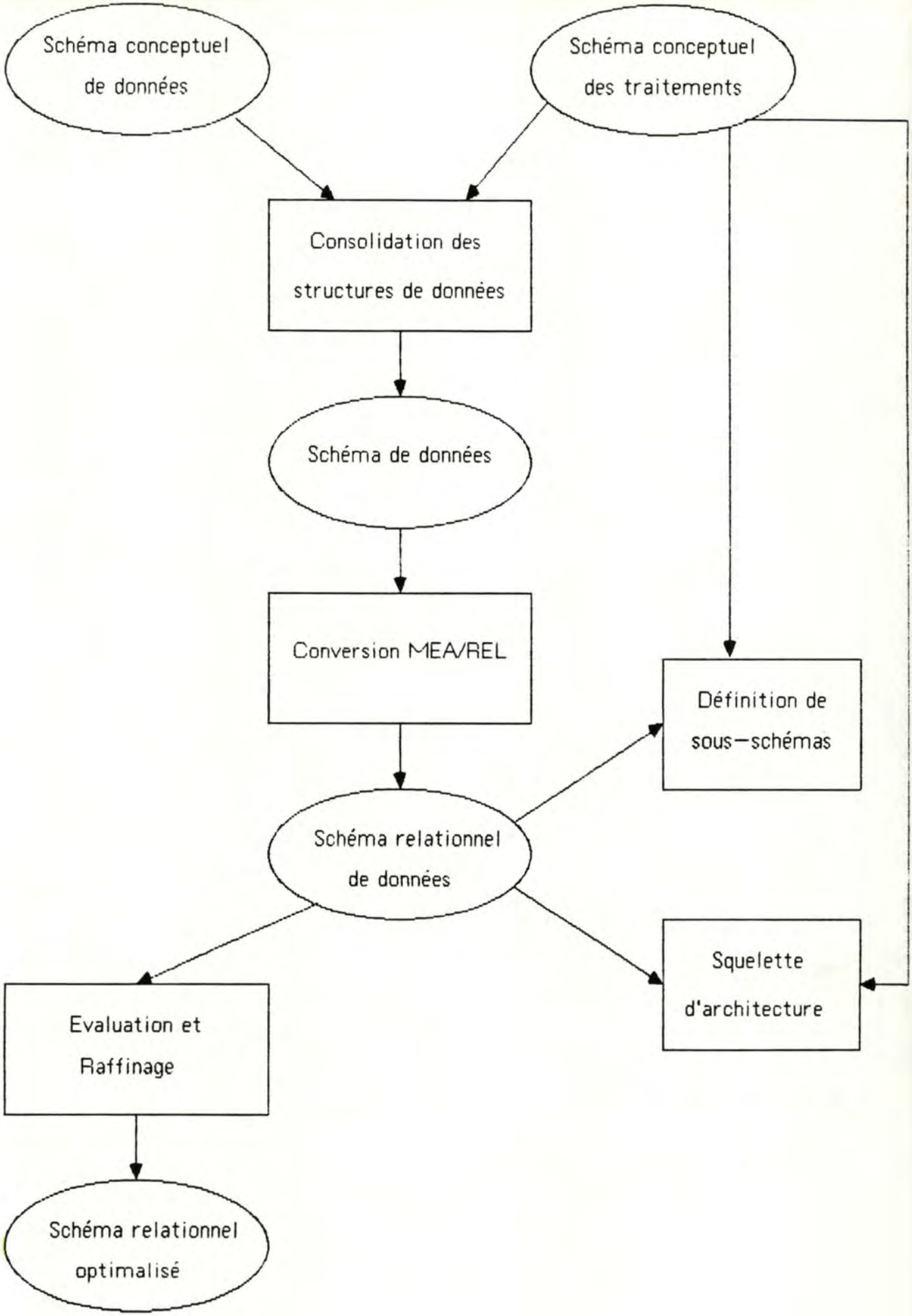


Figure I.1.3.4 : Conception Logique Teory-Yang-Fry

4. Evaluation du schéma et raffinage.

Les auteurs proposent dans [TE-YA-FR 85] de combiner une approche dite "sémantique", qui consiste à décrire le monde réel comme il est, avec une approche dite "d'utilisation" en anticipant les demandes en données des fonctions y accédant. Pour les processus les plus important (en termes de volume accédé, priorité, fréquence d'exécution, temps de réponse exigé), l'on adapte le schéma en ajoutant des attributs ou des relations pour diminuer le nombre de "join" nécessaires. Une fonction de calcul de coût est définie sur base du coût entrées/sorties et sur base du coût de stockage. Cet algorithme, définit dans [TE-YA-FR 85], permet d'évaluer à l'aide de cette fonction, les schémas pour ne retenir que la solution la plus intéressante.

Conclusion

Alors que tous ont pour but de produire un schéma de données de type relationnel (ou autre) qui soit exact, c-à-d décrivant complètement le monde réel, et efficace, c-à-d tenant compte des traitements accompagnant ce schéma, les approches sont différentes.

On passe soit par un modèle intermédiaire (MAG) (b), on continue sur le Modèle Entité-Association (a) ou l'on transforme directement en un Modèle Relationnel (c), qui doit être le modèle final.

Il est à noter que (a) et (b) sont assez semblables quant au modèle utilisé. Le MAG de (b) et le modèle Entité-Association obtenu après simplification du schéma de données sont en réalité des modèles Entité-Association binaires fort proches.

Ceci entraîne que les transformations du schéma de données se fait sur le schéma final (c) ou pas (a) et (b). Peut on assurer dès lors qu'il n'y a pas de perte d'optimalisations lors des transformations si celles-ci ne sont pas réalisées sur le modèle final ? Ceci dépendra fortement des transformations restant à réaliser. Celles-ci devront être effectuées avec grande prudence, tenant compte des critères d'optimalisation (Ex. : Si un mécanisme d'accès à un article A, tel qu'une clé, doit être transformé en un autre article B, ceci aura une influence sur le nombre d'accès logiques à l'article A via ce mécanisme d'accès).

Quant aux critères d'optimimalisation, ils sont généralement semblables :

- minimiser le nombre d'accès logiques par unité de temps,
- minimiser les transferts de données entre mémoire de masse et mémoire centrale,
- minimiser les volumes de données.

Les différentes démarches ne proposent pas d'algorithmes pour optimaliser une structure de données, mais des fonctions d'évaluation ou de calcul de coûts des structures

a posteriori, permettant ainsi d'évaluer les transformations réalisées.

I.1.4 Le Niveau Physique

La conception physique de la base de données dépend fortement de la base de données cible utilisée, en l'occurrence une base relationnelle, et de ses proriétés. Il s'agira de générer la description du schéma et des sous-schémas (ou vue) dans le formalisme du gestionnaire de données, si celle-çi n'a pas déjà été réalisée comme dans (a).

On y effectuera également la définition des Index, l'organisation physique des données en fichiers, "clusters" (agrégats), etc..., et la définition des supports physiques.

I.2 Démarche choisie dans le cadre du mémoire

I.2.1 Objectifs

La démarche méthodologique adoptée dans le cadre du mémoire doit en premier lieu pouvoir répondre aux objectifs généraux définis dans l'introduction. De plus, il ne s'agit pas de se donner une démarche générale comme celles présentées en I.1. Le but étant de se donner une méthode pour construire un schéma physique ADR/DATACOM/DB à partir d'un schéma conceptuel Entité-Association IDA.

I.2.2 Démarche méthodologique

Cette démarche a été proposée par B. Michel dans [MICHEL 85] et est illustrée par la Figure I.2.2.1.

(1) Le Schéma Conceptuel Entité-Association provenant de la base IDA est transformé en un schéma relationnel dit "brut" (Schéma qui fait correspondre à chaque Entité et à chaque Association, un Schéma de Relations).

(2) Le Schéma Relationnel Brut est optimisé par rapport à l'espace disque et au temps d'accès en tenant compte de la spécification des traitements. L'on éffectuera des opérations tels que la réunion ou l'éclatement de Schémas de Relations.

(3) Le Schéma Relationnel Optimalisé ainsi obtenu est complété par la spécification de mécanismes d'accès en spécifiant les constituants qui doivent former les clés d'accès.

(4) Le Schéma Relationnel Logique de l'étape précédente sera transformé en un Schéma Logique ADR/DATACOM/DB. Il sera complété par les paramètres physiques pour former le Schéma Physique DATACOM/DB.

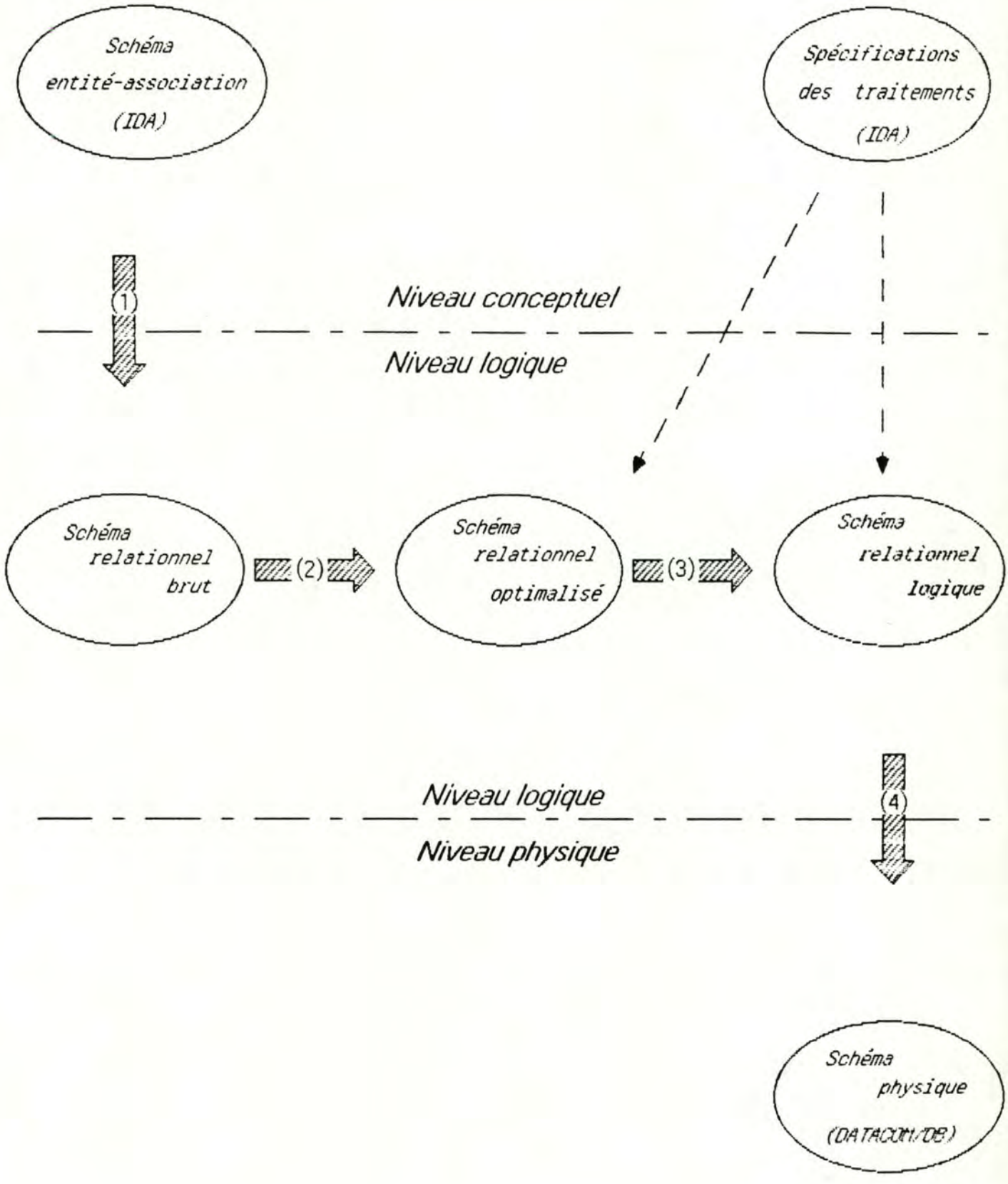


Figure I.2.1.1 : Démarche de conception d'une BD relationnelle

I.2.3 Support logiciel

Pour mener à bien la conception d'une base de données DATACOM/DB à partir d'un schéma conceptuel provenant de la base IDA, nous disposerons des composants logiciels suivants :

(1) Un processeur permettant d'extraire un schéma conceptuel non redondant (II.2.2.2) de la base IDA.

(2) Un générateur automatique du schéma relationnel brut à partir du schéma conceptuel Entité-Association.

(3) Un éditeur de schéma relationnel permettant de transformer un schéma relationnel brut en un schéma relationnel logique. Ces transformations seront contrôlées par l'éditeur.

(4) Un générateur automatique du schéma logique ADR/DATACOM/DB à partir du schéma relationnel logique.

(5) Des facilités "batch" du dictionnaire des données ADR/DATADICTIONARY pour introduire le schéma logique au sein du dictionnaire de données d'ADR.

(6) Des facilités "on-line" ou "batch" du dictionnaire des données ADR/DATADICTIONARY pour donner les paramètres physiques du schéma permettant de définir le schéma physique DATACOM/DB.

(7) D'un processeur permettant de définir les correspondances entre occurrences d'objets DSL et occurrences d'objets ADR/IDEAL.

(8) Des facilités de l'analyseur DSL-SPEC de l'environnement IDA pour répercuter ces correspondances au sein de la base IDA, ainsi que d'extraire des rapports de correspondances.

(9) D'un processeur permettant d'adapter les descriptions faites sur les occurrences d'objets DSL définis dans la base IDA aux occurrences d'objets ADR/IDEAL correspondantes, afin de pouvoir les introduire dans le dictionnaire de données d'ADR.

Tous les composants logiciels autres que ceux d'ADR et d'IDA sont décrits en II.4. L'environnement ADR est décrit en II.1 et l'environnement IDA en II.2 (Consultez également [DD2G-IN] pour ADR et [DSL-SPEC 84], [BOD-PIG 83] pour IDA).

I.2.4 Justification

Le support logiciel décrit en I.2.3 devrait permettre de répondre aux trois objectifs définis dans l'introduction :

"Contribuer à la conformité"

La conformité entre les différents schémas est assurée par :

- Des transformations automatiques et correctes des schémas par le processeur de transformation d'un schéma conceptuel Entité-Association en un schéma relationnel brut (2)[m] et par le processeur de transformation d'un schéma relationnel logique en un schéma logique DATACOM/DB (4).

- Des transformations contrôlées et assistées du schéma relationnel brut en un schéma relationnel logique par l'éditeur relationnel (3).

"Assurer la correspondance"

La gestion de la correspondance entre occurrences d'objets DSL et occurrences d'objets ADR-IDEAL est réalisée lors de chaque transformation de (2), (3) et (4). La répercussion de cette correspondance au sein d'IDA est assurée par (7) et (8).

"Limiter la redondance"

Les transformations automatiques des processeurs (2) et (4) ainsi que l'extraction (1) et les facilités "batch" d'ADR/DATADICTIONARY (5) permettent de rationaliser les tâches de conception et d'implémentation en évitant de saisir manuellement et plusieurs fois les mêmes informations.

---

[m](i) Référence au composant (i) de I.2.3.

I. 2. 5 Limitations

Etant donné que cet interface devait être réalisé dans le cadre d'un mémoire et sur une période d'un an, nous n'avons pu réaliser ni l'entièreté des composants logiciels, ni assurer l'entièreté de la méthodologie.

Limite méthodologique :

Nous ne prenons en compte que le schéma conceptuel des données. La prise en compte du schéma conceptuel des traitements exige une étude approfondie. En effet les occurrences d'objets DSL sur lesquelles sont définis ces traitements sont transformées en occurrences d'objets relationnels, qui eux sont transformées à l'aide de l'éditeur relationnel. Les traitements devraient être également transformés pour tenir compte de ces modifications (Le lecteur intéressé par les transformations d'algorithmes pourra consulter [HAINAUT 85]).

Cette limite implique qu'il n'y aura pas d'assistance sur base des traitements lors de l'élaboration des schémas relationnels optimisé et logique.

Limites sur les composants logiciels :

Les composants :

- Générateur automatique du Schéma Logique DATACOM/DB (4),
- Transformations des descriptions entre occurrences d'objets DSL et ADR/IDEAL (9),

décrits en I. 2. 3 n'ont pas été réalisés, faute de temps. Ils seront néanmoins décrits en II. 4. 3. 2 et II. 4. 5. 3.

I. 2. 6 Comparaison avec les autres démarches

La méthodologie adoptée dans notre démarche pour la conception d'une base de données relationnelle se décompose également en trois niveaux, qui sont conceptuel, logique et physique. Pour ce qui est de la conception logique, nous partons du même principe que celui proposé par Teory, Yang et Fry (I. 1. 3). Nous effectuons d'abord une transformation du schéma conceptuel en un schéma relationnel, permettant ainsi d'éffectuer les transformations sur des objets relationnels. Cette transformation est plus fine pour Teory, Yang et Fry qui tiennent compte des connectivités des Associations. Ainsi par exemple, une Association de connectivité "1-N" ne sera pas transformée en un Schéma de Relations mais en un mécanisme d'accès en définissant une clé dans le Schéma de Relations correspondant à l'Entité du côté "N", permettant l'accès à une relation du Schéma de Relations correspondant à l'Entité du côté "1". Le schéma relationnel de Teory, Yang et Fry, qui est obtenu par des règles de transformation qui comportant des aspects d'optimalisations automatiques, est donc moins "brut" que le nôtre, qui n'a fait l'objet d'aucune tranformation de ce genre.

L'avantage d'effectuer une transformation directe du schéma conceptuel Entité-Association en un schéma relationnel est de pouvoir effectuer des transformations pour optimaliser des structures de données définies sous le modèle final, le modèle du SGBD cible.

Nous avons limité le nombre de modèles utilisés par cette méthodologie à deux, le modèle Entité-Association et le modèle relationnel, tout comme deux des trois démarches proposées dans I. 1. Une généralisation de la démarche, passant par un (troisième) modèle intermédiaire, ne nous paraissait pas nécessaire dans l'optique où nous nous situons (optique partant d'un schéma conceptuel Entité-Association pour aboutir à un schéma logique ADR/DATACOM/DB).

DEUXIEME PARTIE

SPECIFICATION DE L'INTERFACE

II.1 Environnement ADR

II.1.1 L'utilisateur IDEAL [SI2G-01]

IDEAL fournit à l'utilisateur un environnement interactif pour le développement, la maintenance et l'exécution d'applications. Une application IDEAL est constituée de toutes les ressources nécessaires à faire fonctionner une base de données "online" ainsi que pour exécuter les diverses utilisations de cette base de données. Par ressources nécessaires, on entend les types d'entité[m] (T.E.) suivants : program, subprogram, panel, report et dataview.

Une application IDEAL est développée en définissant ces ressources grâce à :

- Des écrans spécialement définis ("fill-in-the-blank") qui permettent à l'utilisateur de concevoir interactivement les T.E. program, dataview, report et panel.
- Un langage structuré de haut niveau (Program Definition Language) pour le design et le développement des diverses procédures d'une application. Ce langage compilé comprend également un éditeur syntaxique et une gestion de bibliothèques.

L'utilisateur a donc la possibilité de développer ses applications en définissant les ressources de celles-ci et ce, au moyen d'outils que lui procure l'environnement IDEAL. Ces outils sont les suivants (fig II.1.1 A):

PANEL DEFINITION FACILITY

Procure à l'utilisateur des facilités pour créer et mettre à jour la définition et le "layout" de ses propres écrans (PANEL). Une fois créés, ceux-ci peuvent être testés, imprimés et édités pour un usage immédiat.

REPORT DEFINITION FACILITY

Procure le moyen de créer, mettre à jour et de tester les définitions des rapports (REPORT) ainsi que le "layout" de ceux-ci (colonnes, champs, entêtes ...).

DATAVIEW DEFINITION

Une dataview est une vue logique des données qui permet de faire des requêtes indépendamment de la structure de stockage. Les définitions des dataview sont stockées dans le dictionnaire. Il n'existe pas de procédure dans IDEAL pour décrire et pour maintenir les définitions des dataview. IDEAL utilise le dictionnaire pour stocker et maintenir les définitions des dataview. L'utilisateur IDEAL ne peut que consulter ces définitions.

---

[m] Appelons les déjà types d'entité puisque ces ressources seront répertoriées dans le dictionnaire de IDEAL.

PROGRAM DEFINITION FACILITY

Permet à l'utilisateur de décrire :

- l'identification d'un programme (PROGRAM).
- les ressources dont le programme à besoin (table des ressources) c.a.d. les dataview qu'il utilise, les panel qui doivent lui être transmis, les report à générer et les program appelés.
- le contenu du programme en PDL (éditeur syntaxique).
- les paramètres et les variables de travail.

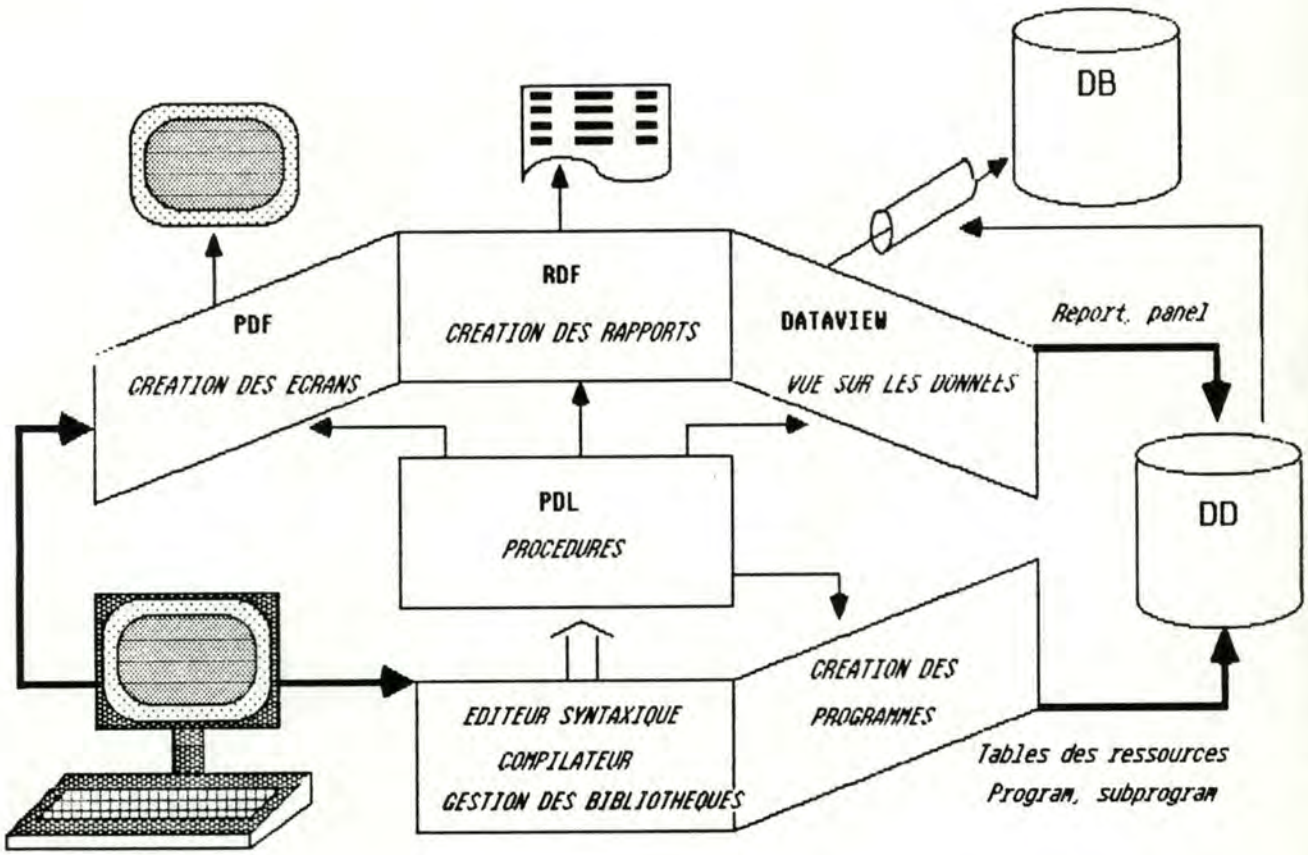


Figure II.1.1.1 : Les outils de développement l'utilisateur IDEAL

L'environnement IDEAL est étroitement lié avec le dictionnaire DATACOM/DD (fig II.1.1 B). En effet, chaque définition d'un objet IDEAL correspond à l'ajout d'une occurrence d'entité dans le dictionnaire pour le T.E. standard correspondant à cet objet. Ainsi, l'utilisateur peut créer les occurrences des T.E. program, panel, report dans le dictionnaire à partir de l'environnement IDEAL. Les occurrences du T.E. dataview, elles, sont seulement créées à partir du dictionnaire. Les types de relation (T.R.) elles aussi, ont leurs occurrences mises à jour automatiquement lors de la définition de la table des ressources du programme.

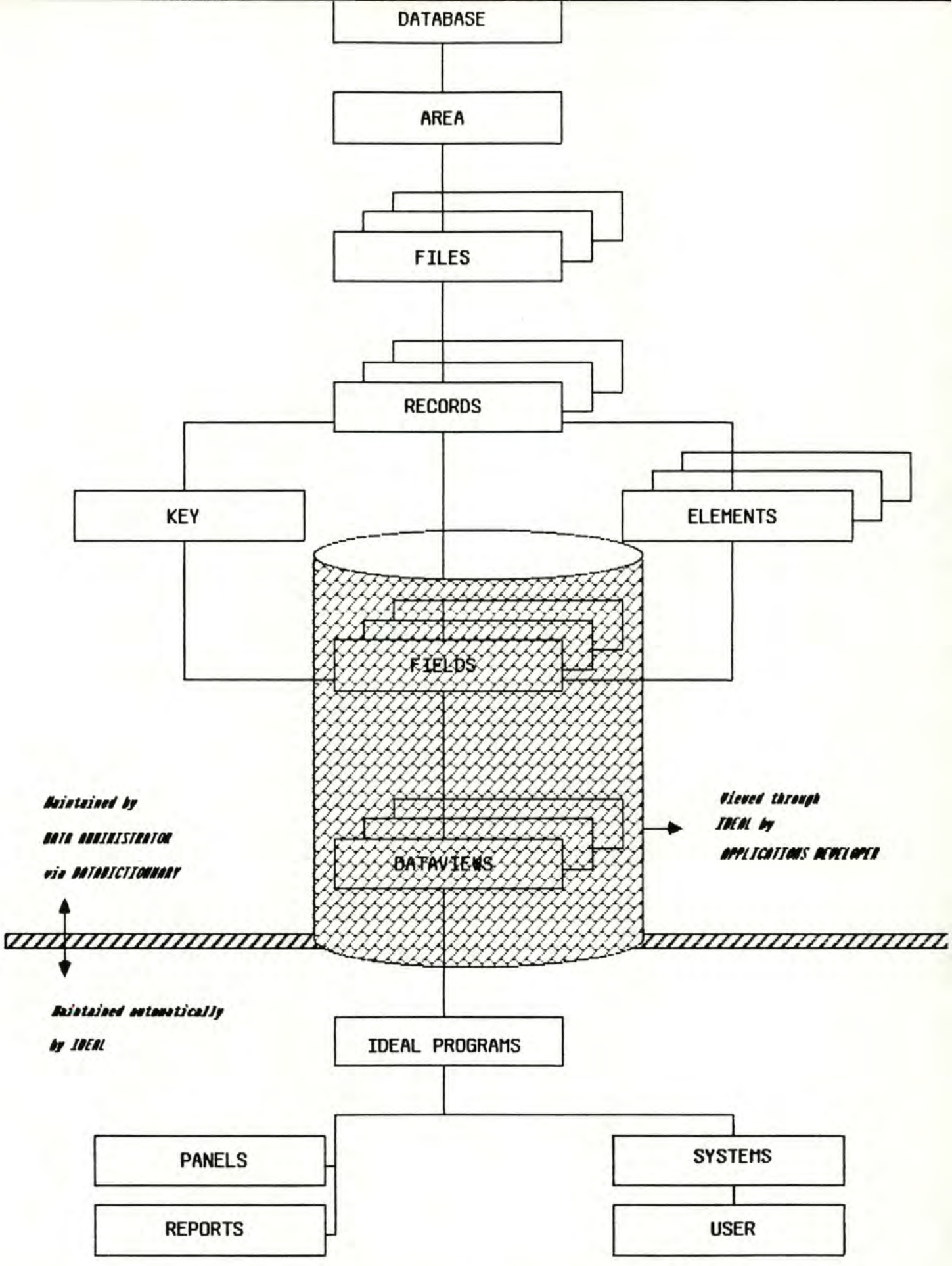


Figure II.1.1.2 : Le dictionnaire vu par l'utilisateur IDEAL

II.1.2 Description du dictionnaire d'ADR [DD2G-IN]

II.1.2.1 Description générale

Le dictionnaire sert de répertoire à toute l'information qui est référencée pour décrire une base de donnée et son environnement. Il peut ainsi décrire toutes les composantes de l'information, de même que les relations entre ces composantes.

Le dictionnaire est structuré en termes de type d'entité. Un type d'entité est une catégorie d'occurrences d'entité qui partagent des caractéristiques communes et qui sont regroupées pour les référencer plus facilement. Une occurrence d'entité est un élément de la classe constituée par un type d'entité. Pour chaque occurrence d'entité, on enregistre certaines caractéristiques. Ces caractéristiques sont appelées des valeurs d'attribut et ces valeurs correspondent à des types d'attribut définis pour un type d'entité. Le dictionnaire contient aussi des informations en ce qui concerne les liens entre :

* un type d'entité et un type d'entité (Type de relation)
* une occurrence d'un type d'entité et une autre occurrence de ce même type d'entité (Occurrence de relation sur une relation réccursive)
* une occurrence d'un type d'entité et une occurrence d'un autre type d'entité (Occurrence de relation).

Chaque donnée entrée dans le dictionnaire est associée à un type d'entité. Le dictionnaire fournit vingt types d'entité standards (standard entity-type) à l'utilisateur pour décrire sa base de données et son environnement. En plus de ces vingt types d'entité standards, l'utilisateur peut définir ses propres types d'entité (user-defined entity-type). Ces types d'entité peuvent être définis après installation du dictionnaire et sont implémentés de la même manière que les types d'entité standards.

Un type d'attribut est une caractéristique type que l'on utilise pour décrire les occurrences d'un type d'entité. La valeur d'attribut est la donnée associée avec le type d'attribut. Il existe trois sortes de type d'attribut: commun, dépendant et utilisateur. Le type d'attribut commun est celui qui est utilisé par tous les types d'entité (standard ou user-defined). Le type d'attribut dépendant est celui qui est associé a un type d'entité particulier. Enfin, le type d'attribut utilisateur est celui qui a été défini par l'utilisateur.

Le dictionnaire ne possède pas uniquement des types d'entité prédéfinis mais il possède aussi des relations entre ces types. Une relation est identifiée par son type de relation dont le nom est une concaténation du type d'entité sujet de la relation, du type d'entité objet et du nom de la relation. Le nom de la relation peut être un des neuf noms fournis par le dictionnaire mais il peut être aussi un nom

d'une relation définie par l'utilisateur. Par exemple la relation entre un programme et un panel est représentée comme ceci: PGM-PNL-USE.

Chaque relation possède quatre types d'attribut:

MAPPING — Qui peut prendre les valeurs 1-1, 1-N, N-1, N-N

REQUIRED — Qui indique si oui ou non chaque occurrence d'entité objet doit toujours être reliée à au moins une occurrence d'entité sujet.

AUTOMATIC — Qui indique si oui ou non chaque occurrence d'entité objet doit être reliée automatiquement à une occurrence d'entité sujet si cette partie sujet existe déjà.

ORDERED — Qui indique si l'on veut retrouver ou non les occurrences d'entité objets dans un certain ordre.

La figure II.1.2.1 nous montre l'organisation standard des objets contenus dans le dictionnaire.

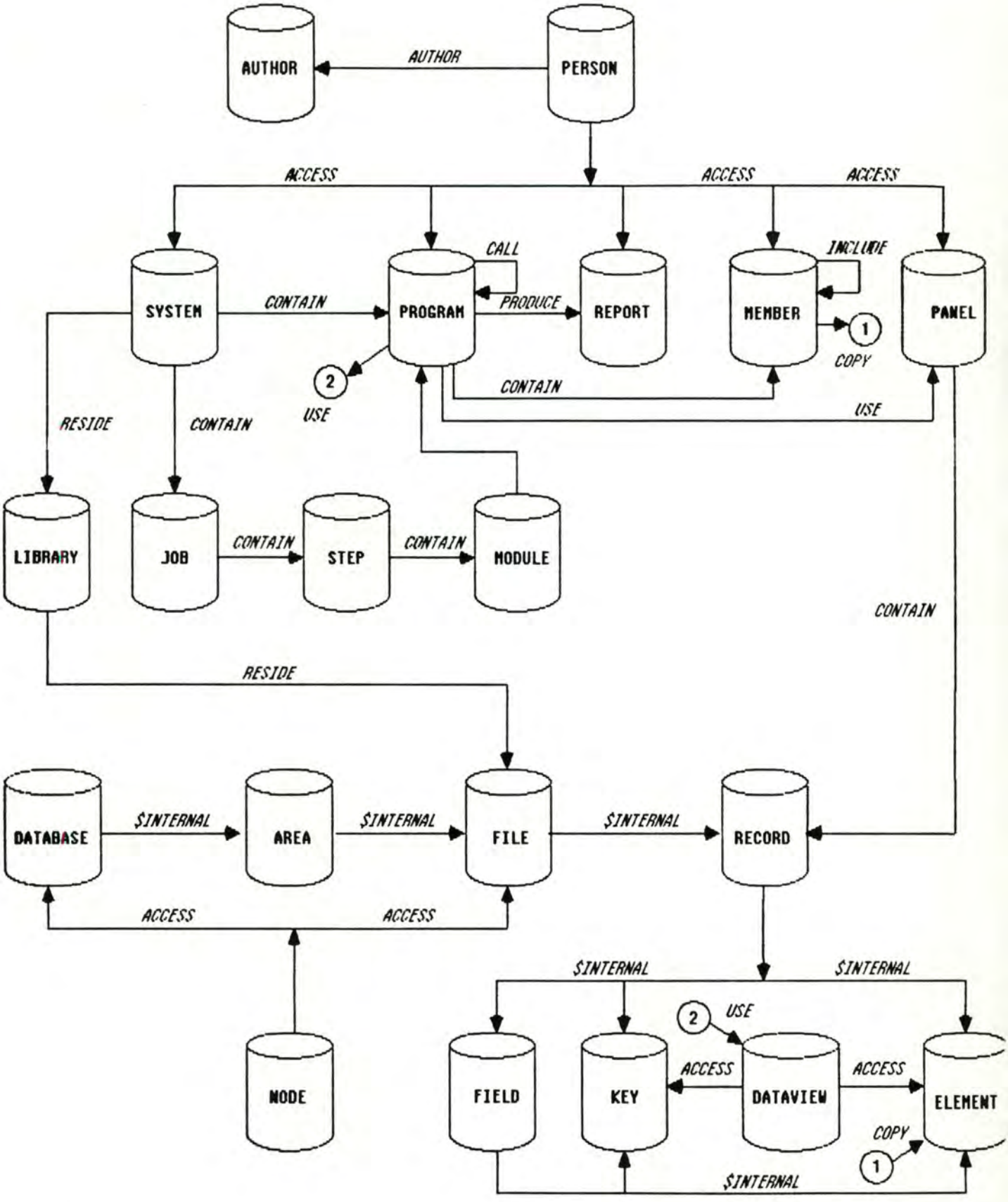


Figure II.1.2.1 : Types de relation et types d'entité standards du dictionnaire

Remarques:

(1) Lorsque l'on définit une occurrence il est nécessaire de respecter les caractéritiques suivantes :

- de 1 à 15 caractères (sauf pour les occurrences définies dans IDEAL et répercutées dans le dictionnaire qui auront au maximum 8 caractères)

- unique dans l'entité

- doit commencer par un caractère alphanumérique

- les caractères suivants peuvent être employés

  A-Z 0-9 $ - #

- mots et caractères réservés

ALL	*	+
END	:	.
REDEFINES	,	?
REMOVE	=	"
START	(	'
UNIVERSAL	)	

(2) Durant le cycle de vie d'une occurrence d'entité, il est parfois nécessaire de garder plusieurs copies d'une même occurrence au même moment. Ceci peut être réalisé grâce au numéro de version. Il est possible d'avoir jusqu'à 999 versions différentes d'une même occurrence d'entité.

Il est possible aussi de définir des étapes dans le cycle de vie d'une occurrence d'entité grâce au type d'attribut status. Le status peut avoir cinq valeurs d'attribut différentes. Les deux valeurs qui nous intéressent particulièrement sont TEST et PROD. TEST représente une occurrence qui est en développement et qui est sujet à des changements. PROD représente une occurrence qui est en production et donc qui, comme telle n'est pas changeable. Les trois autres valeurs sont HIST, INCO, QUAL.

II.1.2.2 Analyse des objets du dictionnaire

Point de vue structure des données (T.R. $INTERNAL)

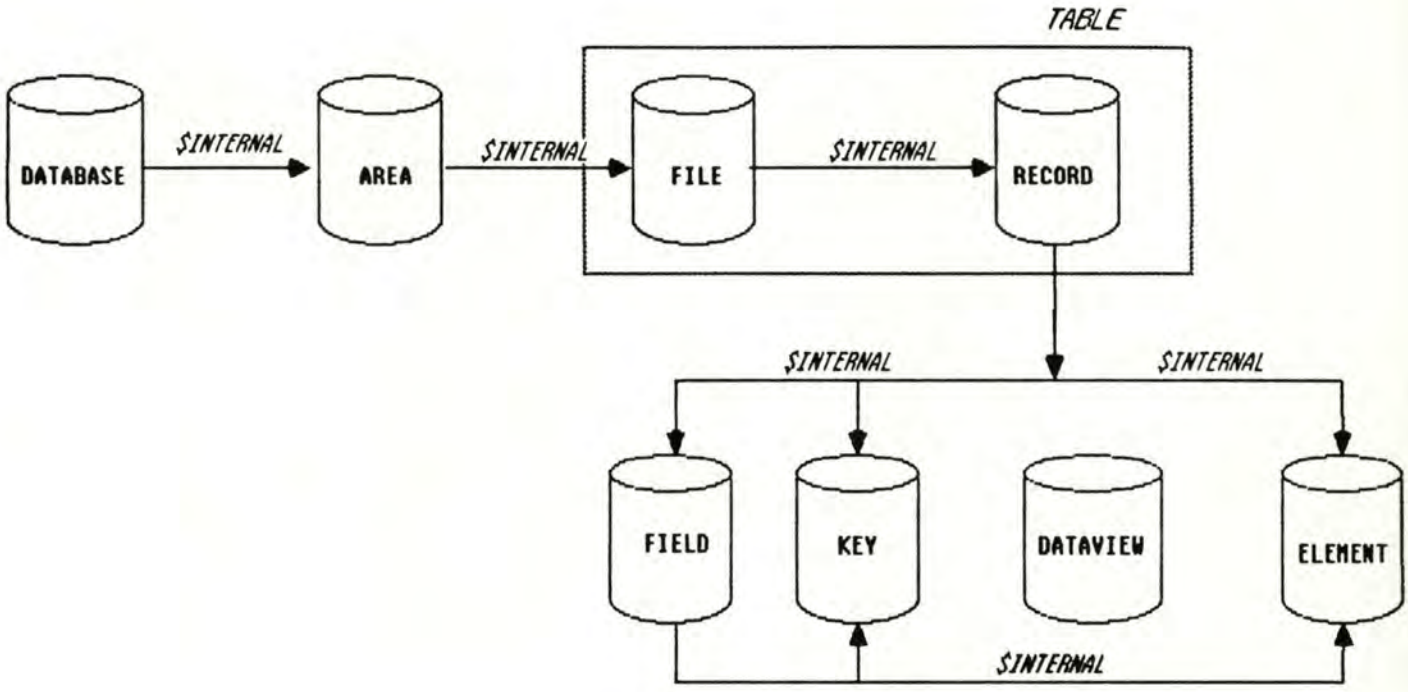


Figure II.1.2.2.1 : Le type de relation standard $INTERNAL

[DB2G-IN]

DATABASE : décrit chaque occurrence du T.E. database de l'environnement du SGBD DATACOM/DB. Dans celui-ci une database est une collection de données reliées ainsi qu'un index sur ces données. Une database est contenue dans une base d'informations qui peut contenir plusieurs database. Chaque database peut avoir plusieurs area.

AREA : décrit chaque occurrence du T.E. area de DATACOM/DB. Dans celui-ci une area désigne le groupement physique de fichiers logiques dans une database. Une area consiste en un ou plusieurs fichiers logiques. Chaque database est constitué d'une INDEX AREA (IXX) et d'une ou de plusieurs DATA AREA. L'INDEX AREA permet entre autres de retrouver le fichier logique auquel appartient un enregistrement et le pointeur logique vers l'enregistrement dans la DATA AREA.

FILE : décrit chaque occurrence du T.E. file qui représente les fichiers de l'environnement de l'utilisateur. Dans l'environnement DATACOM/DB un fichier logique = record = groupement d'enregistrements similaires

(c. a. d. composés des mêmes champs).

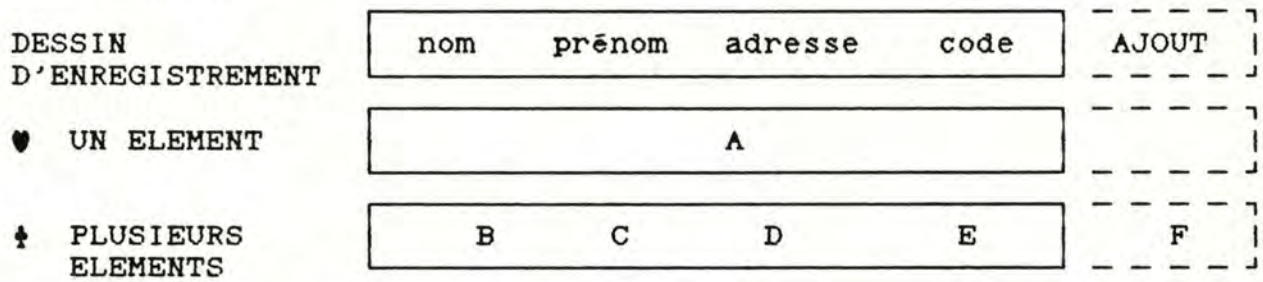
RECORD : décrit chaque occurrence du T. E record c. a. d. un ensemble de field reliés. Dans l'environnement DATACOM/DB il n'existe pas de notion de record car il n'existe pas moins de notion de field.

ELEMENT : décrit chaque occurrence du T. E. element c. a. d un ensemble de field reliés contigus connu comme element. Dans un environnement DATACOM/DB, toutes les données sont définies comme composantes d'un element. L'element est le plus petit groupement logique des données, il s'agit de l'unité logique de transfert.

Remarques:

L'element est la vue externe des données. Il ne fait pas partie du schéma conceptuel et peut être ajouté à la base n'importe quand. La notion d'element permet aux programmes d'être indépendants de la stucture de l'enregistrement.

EXEMPLE :



Les element sont choisis en fonction des besoins de l'utilisateur.

PROGRAMME	♥	♦
MAJ	A	B,C,D,E
LECTURE	A	C
EDITION	A	B,C
EXTRACTION	A	B,E
SUPPRESSION	A	D

Le programme accède aux element dont il a besoin. L'ajout d'une nouvelle donnée n'a pas de répercussion sur les programmes existant.

KEY : décrit chaque occurence du T. E. key. Dans l'environnement DATACOM/DB la clé qui peut être construite sur un ou plusieurs champs non nécessairement contigus (max 4 et de longueur totale de 180 caractères), est utilisée pour qualifier ou déqualifier les données dans une recherche basée sur elle. L'ordre des field dans la clé peut être différent que dans le record. Il existe deux types de clé spéciales : la MASTER KEY et la

NATIVE KEY. Chaque record doit avoir une clé déclarée MASTER KEY. Cette clé est celle pour laquelle il existe toujours une valeur dans l'index et qui peut être définie comme identifiante et non modifiable. De même chaque record doit avoir une clé déclarée NATIVE KEY. Cette clé est utilisée pour ordonner les enregistrements lorsque le fichier est chargé ou déchargé.

DATAVIEW : décrit chaque occurrence du T.E. dataview qui représente une vue logique de l'utilisateur sur les données indépendemment de l'endroit où elles sont stockées. Utilisée par IDEAL une dataview procure une amélioration de l'isolation entre les requêtes des utilisateurs et l'allocation physique des données.

Le dictionnaire détermine la manipulation des données par le SGBD. En effet, DATACOM/DB utilise le CONTROL FILE (CXX) pour stocker les définitions des database, area, file, record, field, key et element d'une base d'information. Ce fichier joue comme un répertoire a grande vitesse lorsque l'on demande un accés à une donnée. La maintenance de ce fichier se fait par l'intermédiaire du dictionnaire DATACOM/DD.

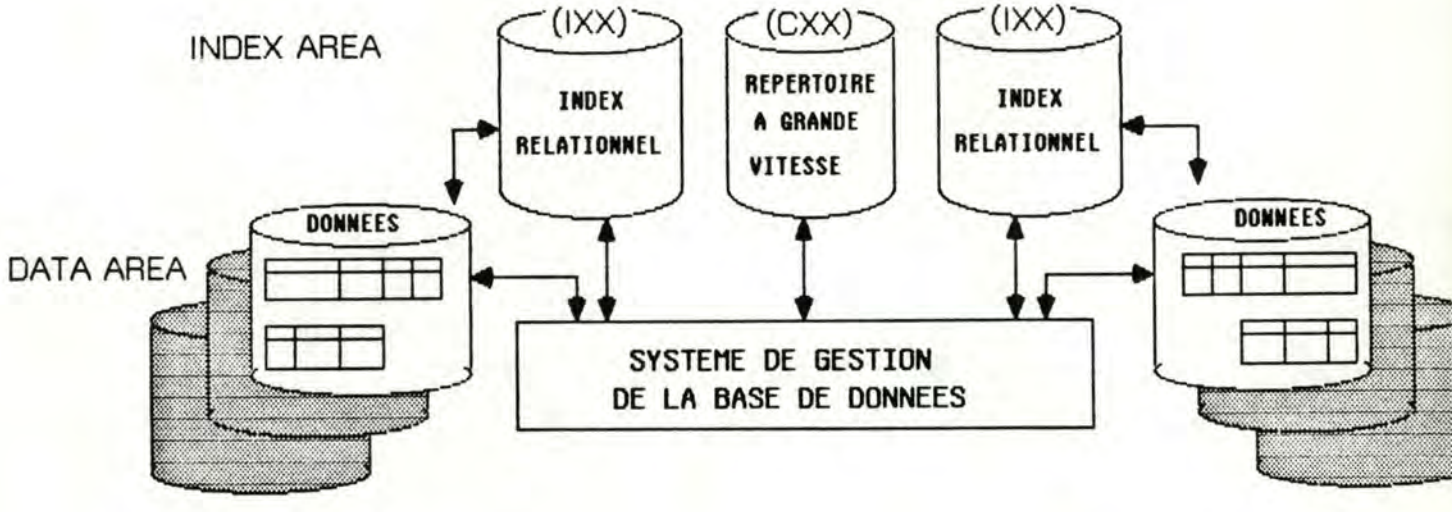


Figure II.1.2.2.2 : Interactions entre le dictionnaire et le SGBD

Point de vue statique des traitements (T. R. USE, CALL, PRODUCE)

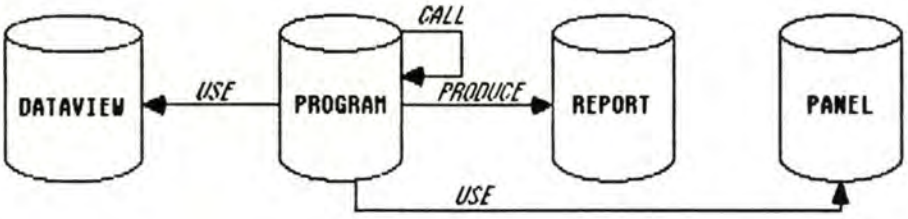


Figure II. 1. 2. 2. 3 : Les types de relation standards USE, CALL, PRODUCE

PROGRAM : décrit chaque occurrence du T. E. program, représente un ensemble d'instructions d'ordinateur compilées ou assemblées comme une unité.

DATAVIEW : cfr supra.

REPORT : décrit chaque occurrence du T. E. report, représente les données qui sont produites sur un papier dans un certain format.

PANEL décrit chaque occurrence du T. E. panel, contient des descriptions à propos des données qui sont produites sur un écran.

Les relations USE, PRODUCE, CALL sont générées automatiquement dans le dictionnaire lorsque la table des ressources d'un programme IDEAL est remplie.

Propriétés communes aux occurrences d'entités du dictionnaire (Types d'attribut communs aux entités)

ALIAS : Synonyme d'une occurrence (1 à 15 caractères, nombre illimité).

DESCRIPTOR : Mot clé qui décrit une occurrence (1 à 15 caractères, 8 max par occurrence, peut être commun à plusieurs occurrences).

TEXT : Description narrative d'une occurrence (nombre quelconque de lignes composées de 1 à 79 caractères).

Autres propriétés communes non développées dans ce chapitre : password, lock, override code, status, version, author, controller.

II.1.2.3 Utilisation du dictionnaire [DD2G-IN]

L'utilisateur en général communique avec le dictionnaire de manière interactive ("online") au moyen d'écrans et de menus spécialement définis pour cette tâche. Cependant, il est possible de communiquer avec le dictionnaire en différé en définissant et en exécutant des commandes "batch" au moyen des utilitaires du système d'exploitation. Ces commandes sont appelées transactions. Le dictionnaire autorise plusieurs types de transactions qui permettent d'ajouter, de mettre à jour ou de supprimer l'information qu'il contient. Ces transactions sont utilisées en groupe. Chaque groupe de transactions est composé d'une transaction entête suivie de paramètres permettant de déterminer le type d'action à effectuer (ADD, DEL, UPD, CPY) et sur quel(s) objet(s) du dictionnaire l'effectuer. Les transactions et leurs paramètres qui compose le reste du groupe servent quand à elles à complèter l'opération introduite par la transaction entête.

EXEMPLE DE GROUPE DE TRANSACTIONS

```
ADD DATABASE,VIN (1,PASS)
1001 'ADR','ADR'
1002 'DATABASE DES VINS'
1101 ADD LIQUIDE
1101 ADD SPIRITUEUX
1200 TEXT ADD
TEXTE DE LA DATABASE DES VINS
1200 TEXT END
3000 300

[ajout d'une occurrence au TE DATABASE]
[définit l'auteur et le controlleur]
[décrit brièvement l'occurrence]
[ajout d'alias]
[ajout d'un texte de description]
[code interne d'identification de la database]
```

Ces transactions permettent aussi de définir des objets non standard dans le dictionnaire.

II.2 Bref rappel de l'environnement IDA

II.2.1 Rappel des outils de l'environnement IDA

IDA (Interactive Design Approach) est un atelier logiciel d'aide à la conception de Systèmes d'Informations. Il est composé (Figure II.2.1.1) :

- d'une Base de Données (BD), coeur du système et contenant toutes les spécifications du Système d'Information,
- d'un Analyseur permettant de mettre-à-jour la BD sur base de spécifications rédigées en DSL (Dynamic Spécification Language), de fournir des rapports documentaires et d'analyses concernant le contenu de la BD, et de contrôler la cohérence des descriptions introduites à l'aide du QS (Query System),
- d'un Outil de Simulation permettant de vérifier le caractère réalisable du système décrit,
- d'un Outil de Prototypage permettant de réaliser une maquette du système décrit pour tester le caractère effectif des spécifications [BOD-PIG 83].

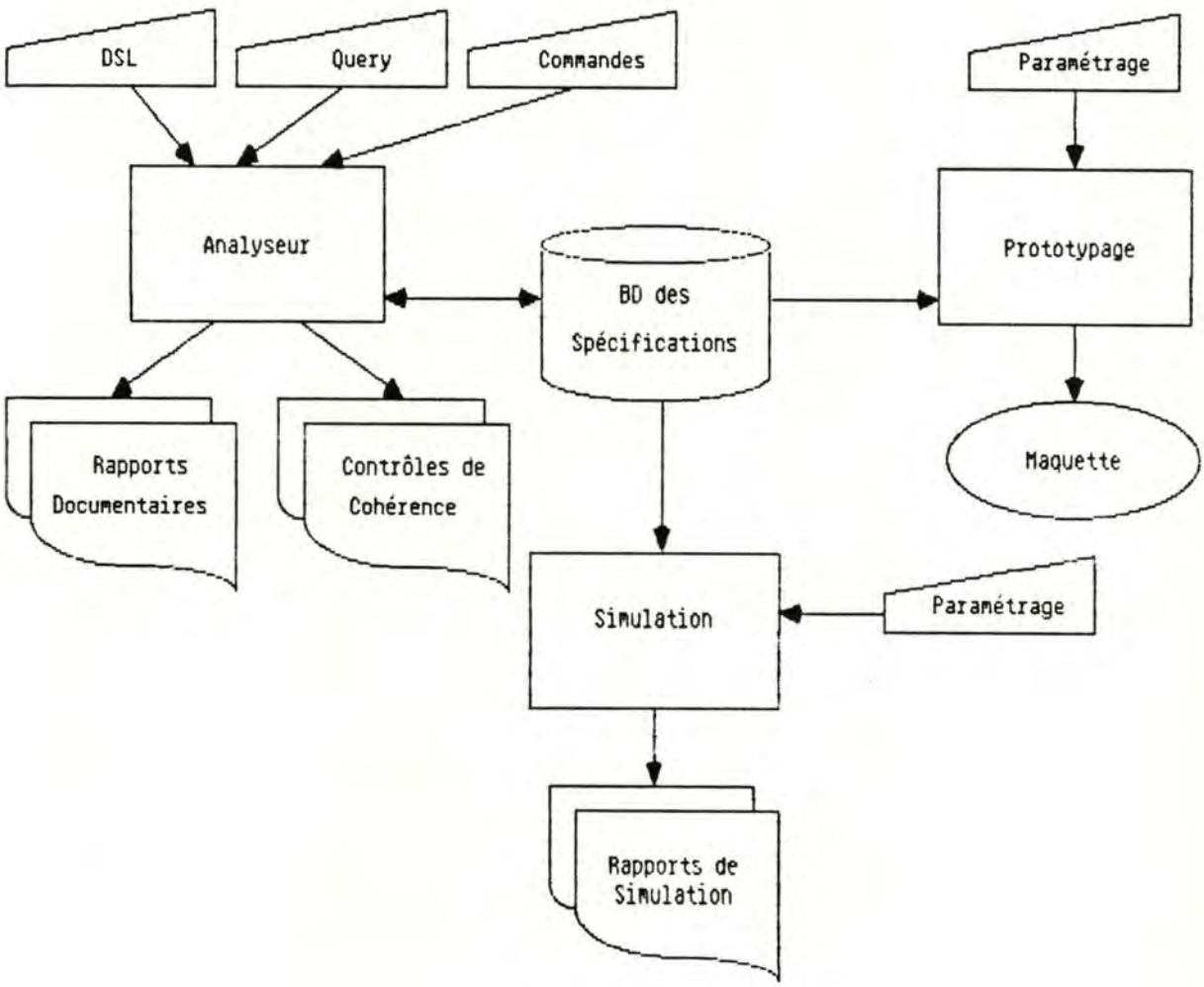


Figure II.2.1.1 : Atelier Logiciel IDA

La spécification du Système d'Information se fait par le biais de différents modèles :

Le modèle de structuration des informations :

Par l'intermédiaire du modèle Entité-Association ce modèle permet de "définir la sémantique des données appartenant à la mémoire ou base des informations du Système d'Information". Il permet également de "définir les messages qui véhiculent les informations". Il permettra de décrire un schéma conceptuel de données.

Le modèle de structuration des traitements :

Ce modèle doit permettre de décrire l'ensemble des traitements définis dans le système. Il permet de décrire ces traitements par "décomposition, par raffinements successifs".

Le modèle de la dynamique des traitements :

Il permet de décrire l'enchaînement de ces mêmes traitements.

Le modèle de la statique des traitements :

Il "a pour but, d'une part, de préciser pour un traitement donné les messages-données et la partie de la mémoire du Système d'Information nécessaires à l'obtention des messages-résultats et d'autre part, de spécifier, sous une forme adéquate, la procédure de traitement qui assure la transformation, à l'aide de la base de données du Système d'Information, des messages-données en messages-résultats."

Le modèle des ressources :

Il sert à "caractériser le comportement des processeurs qui exécutent les procédures de traitement" [BOD-PIG 83].

Dans le cadre de la conception d'une base de données relationnelle, c'est le modèle de la structuration des informations qui nous intéresse le plus. Les autres modèles sont également important mais ne sont pas pris en compte par notre démarche.

II.2.2 Le Modèle Entité-Association

II.2.2.1 Rappel du modèle

Le modèle Entité-Association est un modèle conceptuel de structuration des données. On lui reconnaît "de grandes qualités de communication et une bonne capacité de représentation opérationnelle des informations appartenant au réel perçu" [BOD-PIG 83]. Un exposé complet de ce modèle peut être retrouvé dans [CHEN 76] et [BOD-PIG 83]. Nous ne ferons que rappeler les concepts de base.

Le modèle se base sur quatre concepts :

- l'Entité, objet du réel perçu,

- l'Association, définissant la correspondance entre plusieurs Entités,

- l'Attribut, caractéristique d'une Entité ou Association, pouvant être simple ou répétitif, élémentaire ou décomposable, obligatoire ou facultatif,

- la Contrainte d'Intégrité, "proprieté, non représentée par les concepts de base du modèle, que doivent satisfaire les informations appartenant à la mémoire du Système d'Information". Elles sont de type "Statique" (devant être verifiées à tout moment) ou "Dynamique" (définissant la validité de transition d'états de la base). Ces contraintes peuvent être des contraintes d'existence et d'identification définies sur les Entités ou Associations, des contraintes d'exlusion et d'inclusion, qui avec les contraintes de connectivité caractérisent la participation d'une occurrence d'un type d'Entité dans une occurrence d'un type d'Association, des contraintes de valeur définies sur les Attributs ainsi que des dépendances fonctionnelles.

II.2.2.2 La forme canonique du schéma conceptuel

Une forme canonique ou "idéale" du schéma conceptuel des données décrit sous le modèle Entité-Association à été décrite dans [GAT-BER 86]. Le but est de disposer d'un schéma complet, cohérent et irredondant. Les caractéristiques principales de cette forme canonique sont :

- "Unicité des noms de type d'entité, de type d'association, d'attributs, de nom de rôle.
- Au moins un identifiant par type d'entité et d'association.
- Pas d'attribut ou de groupe répétitif sauf demande explicite de l'utilisateur.
- Pas d'attributs ne nécessitant pas l'entièreté de l'identifiant pour l'identifier.
- Les attributs d'un type d'entité ou d'un type d'association doivent être mutuellement indépendants.
- Présence obligatoire d'une spécification minimale pour tout type d'entité, tout type d'association, et tout attribut."

En fait le schéma conceptuel admis par l'interface peut ne pas répondre à certaines de ces caractéristiques :

- Bien que les trois dernières caractéristiques ne sont pas obligatoires, elles sont, d'un point de vue méthodologique, néanmoins souhaitable.
- L'unicité de nom d'attributs ne doit être pas vérifiée.
- Les groupes et attributs répétitifs sont admis.

De plus les dépendances fonctionnelles ne sont pas traitées par l'interface.

Une contrainte supplémentaire sera ajouté à la forme canonique exigeant que pour chaque Entité ou Association au moins un Identifiant réponde aux caractéristiques d'un Identifiant Principal (II.4.2.1).

II.3 Un point de vue sur l'environnement "idéal"

IDEAL fournit au concepteur d'applications un environnement interactif pour le développement de ses programmes. Grâce à des outils appropriés, le concepteur peut définir les programmes et leurs ressources et bien sûr, il peut ensuite les décrire. En ce qui concerne la partie "structure des données" des ressources d'une application IDEAL (dataview), elle ne peut être définie et décrite qu'à partir du dictionnaire (cfr II.1.1). En effet lors du développement d'une application, l'utilisateur IDEAL ne peut que consulter cette définition.

Notre interface ne peut en tout cas pas communiquer avec un programme interactif, il ne peut qu'accéder avec l'environnement A.D.R. qu'en "batch" et par conséquent qu'avec les éléments de cet environnement qui propose un tel moyen de communication c.a.d. le dictionnaire. Ceci est parfait pour la structure des données, il n'en est pas de même pour une partie des ressources d'une application (statique des traitements : panel, report, program, subprogram). En effet, si on peut toujours ajouter une occurrence au T.E. panel dans le dictionnaire, cet ajout ne crée pas pour autant un objet panel dans l'environnement IDEAL, tandis que si on crée un objet panel au moyen de P.D.F. (cfr II.1.1) dans l'environnement IDEAL, son occurrence viendra s'ajouter automatiquement au T.E. panel du dictionnaire. Il est donc inutile de descendre la statique des traitements au niveau physique, c.a.d. de la répertorier dans le dictionnaire, tant qu'A.D.R. ne permettra pas de créer des objets IDEAL avec d'autres outils que ceux proposés par l'environnement IDEAL. Ceci est peut être dommage dans la mesure où le dictionnaire détermine déjà l'organisation externe des données pour DATACOM/DB. En fait, le dictionnaire a deux fonctions différentes. Il permet de créer le schéma physique des données et permet le rassemblement de l'information concernant les procédures d'exploitation de ce schéma. Une fonction est active, l'autre est passive. Pourquoi ne pas pouvoir créer des objets IDEAL à partir du dictionnaire ? Pourquoi le dictionnaire n'est-il pas complètement un centre actif de l'environnement IDEAL ?

A.D.R. veut sans doute par là distinguer centre de production des données et centre de développement d'applications. Mais l'un n'empêchait pas l'autre !

II.4 Description de l'architecture fonctionnelle

II.4.1 Présentation générale

L'architecture de l'interface décrite à la figure II.4.1 est décomposée en trois niveaux, qui sont conceptuel, logique et physique (I.1). Les processeurs qui y sont décrits ont pour but de concrétiser la methodologie définie en I.2.

Le niveau conceptuel :

C'est à ce niveau que se situe l'environnment IDA permettant la spécification conceptuelle des informations (II.2). Nous y intégrerons un seul processeur qui est l'Extracteur Conceptuel. Il ira extraire de la base de données IDA la spécification concernant la structure des données (schéma conceptuel des données) et la description des propriétés communes.

Le niveau logique :

L'atelier de manipulation organique, sujet principal de notre mémoire, est l'outil principal de la conception logique d'une base de données relationnelle. Après avoir extrait le schéma conceptuel des données à l'aide de l'Extracteur Conceptuel, nous le transformerons en un schéma relationnel brut via le Transformateur IDA/BSR. Ce processeur effectuera la transformation automatique du schéma conceptuel, conformément aux règles définies en II.4.3.1, et le garnissage automatique du schéma relationnel brut dans la Base de Spécifications Relationnelles (BSR). Ce schéma relationnel brut peut être transformé en un schéma relationnel logique optimalisé à l'aide des fonctions offertes par le Présentateur Relationnel et décrites en II.4.4.1. Un Editeur des mises-à-jour devra permettre d'effectuer des mises-à-jour ponctuelles et locales à la BSR provenant de mises-à-jour ponctuelles et locales du schéma conceptuel des données au sein d'IDA. Enfin la génération complète ou partielle des descriptions logiques du schéma ADR peut être réalisée automatiquement par l'Extracteur Logique, aussi appellé Transformateur BSR/ADR, et ceci conformément aux règles définies en II.4.3.2. La correspondance entre occurrences d'objets DSL et occurrences d'objets ADR peut être générée par le Répercuteur des Correspondances. Après avoir effectué l'extraction logique, nous pouvons répercuter les descriptions faites sur les occurrences d'objets DSL, générées par l'Extracteur Conceptuel, aux occurrences d'objets ADR de notre choix à l'aide du Transformateur de Textes.

Le niveau physique :

Nous arrivons dans l'environnement ADR décrit en II.1. Nous pouvons y mettre-à-jour le Dictionnaire de Données à partir des fichiers générés par l'Extracteur Logique et le Transformateur de Textes. A l'aide des outils de cet environnement, nous y compléterons également la définition du schéma de données par des paramètres physiques propres à l'environnement du système et des programmes d'applications.

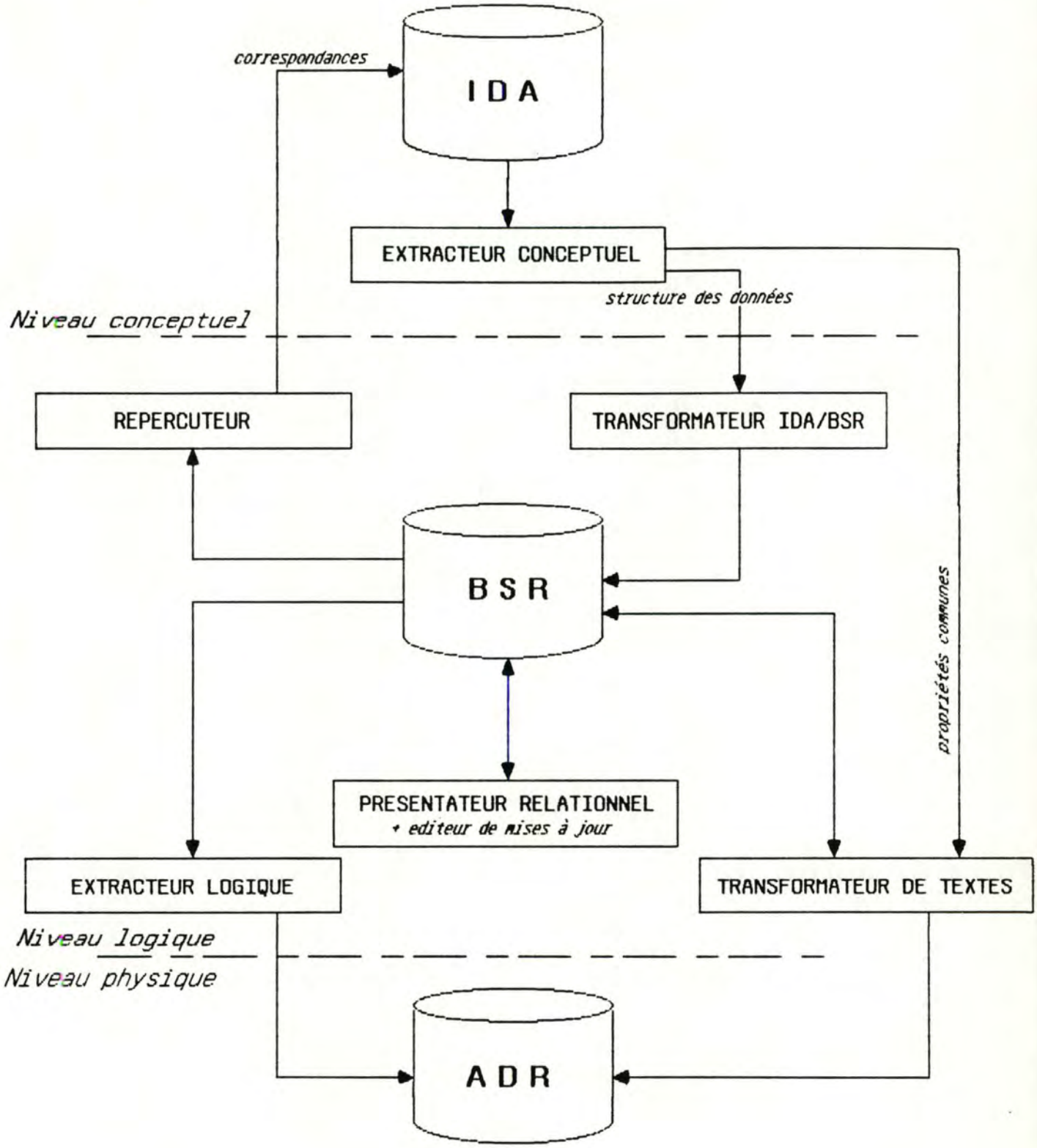


Figure II.4.1 : Architecture de l'Interface

II.4.2 La base de spécifications relationnelles (BSR)

Cette base de données est le coeur de l'atelier de manipulation organique. Elle est décrite en III.3.1.1 et permet de stocker les définitions des schémas de relations, de leurs constituants, identifiants et clés d'accès. On y stocke également les définitions d'index et les correspondances entre occurrences d'objets DSL et occurrences d'objets ADR.

DEFINITIONS

Le modèle relationnel pris en compte au niveau logique diffère du modèle relationnel proposé par Codd [CODD 70]. Les définitions qui suivent illustrent notre modèle relationnel.

Domaine :

Un Domaine est un ensemble de valeurs.

Constituant :

Un Constituant d'un Schéma de Relations est une donnée de ce Schéma caractérisée par un nom et un domaine.

De plus le Constituant est l'Image ou l'Implémentation d'un Attribut conceptuel. Ce Constituant hérite du domaine sur lequel cet Attribut a été défini. On dira que le Constituant est défini sur un Attribut. Etant donné que plusieurs Constituants peuvent être définis sur le même Attribut au sein d'un Schéma de Relations (II.4.3.1 exemple du Schéma de Relations représentant l'Association récursive "Responsabilité"), il est nécessaire de ce donner un mécanisme pour distinguer ces Constituants. C'est pour cela que le Constituant sera identifié par l'Attribut sur lequel il est défini et le Rôle que joue ce Constituant au sein du Schéma de Relations.

Tout comme l'Attribut sur lequel il est défini, le Constituant peut être simple ou répétitif, élémentaire ou décomposable.

Schéma de Relations :

Un Schéma de Relations est un ensemble de relations définies sur les mêmes domaines de valeurs et ayant même identifiant. Il est caractérisé par son nom suivi de la liste de ses Constituants.

Relation :

Une Relation est un sous ensemble du produit cartésien d'un ensemble de domaines. Elle est une occurrence d'un Schéma de Relations

Identifiant :

Un Identifiant d'un Schéma de Relations est un sous-ensemble des Constituants d'un Schéma de Relations, qui pour un ensemble de valeurs de ces Constituants n'identifie qu'une seule Relation.

Identifiant Principal :

Un Identifiant Principal d'un Schéma de Relations est un Identifiant qui est également une Clé. Il n'en existe qu'un seul par Schéma de Relations.

Clé :

Une Clé est un mécanisme d'accès à une ou Plusieurs Relations d'un Schéma de Relations sur base d'un ensemble de valeurs des Constituants formant la Clé. Une Clé peut être Principale ou Primaire (elle est la clé correspondant à l'Identifiant Principal) ou Secondaire (tout autre clé).

NOTATION

S(a, b, c, d(e, f), g(3)) (1)

où S est un Schéma de Relations ayant a, b, c, d et g comme Constituants définis respectivement sur les Attributs conceptuels a, b, c, d et g et identifié par les Constituants a et b. Le Constituant d est un Constituant simple et décomposable, formé des Constituants e et f définis respectivement sur les Attributs conceptuels e et f. Le Constituant g est un Constituant élémentaire de répétitivité 3 et les Constituants a, b, c, e et f sont des Constituants simples et élémentaire.

Un Constituant i d'un Schéma de Relations S peut également être noté de la façon suivante :

i:ri

où i est l'Attribut conceptuel sur lequel ce Constituant est défini et ri est le Rôle que joue ce Constituant au sein du Schéma de Relations S.

La Notation (1) devient :

S(a:ra, b:rb, c:rc, d(e:re, f:rf):rd, g(3):rg)

II.4.3 Les automates de transformations de schémas

II.4.3.1 Le Transformateur IDA/BSR

OBJECTIFS

Le Transformateur IDA/BSR est un processeur qui doit effectuer la transformation d'un schéma conceptuel de données décrit sous le modèle Entité-Association, mis sous forme canonique, en un schéma relationnel brut (I.2). Ce schéma conceptuel est décrit dans un des deux fichiers fournis par l'Extracteur Conceptuel (II.4.5.1). De plus il doit garnir la BSR (II.4.2). Ces transformations doivent se faire automatiquement.

REGLES DE CORRESPONDANCES

La correspondance entre les objets de ces deux modèles est la suivante :

- A une Entité correspond un ou plusieurs Schéma(s) de Relations.
- A une Association correspond un ou plusieurs Schéma(s) de Relations.
- A un Elément correspond un ou plusieurs Constituant(s).
- A un Group correspond un ou plusieurs Constituant(s).
- A un Identifiant correspond un Identifiant et peut correspondre une Clé.

REGLES DE TRANSFORMATIONS

Transformation d'une Entité :

- Une Entité sera transformée en un Schéma de Relations.
- Un Groupe de l'Entité sera transformé en un Constituant décomposable du Schéma de Relations correspondant à l'Entité.
- Un Elément de l'Entité sera transformé en un Constituant du Schéma de Relations correspondant à l'Entité ssi cet Elément n'a pas été déclaré comme étant dérivable[en].

Transformation d'une Association :

- Une Association sera transformée en un Schéma de Relations.
- Un Groupe de l'Association sera transformé en un Constituant décomposable du Schéma de Relations correspondant à l'Association.
- Un Elément de l'Association sera transformé en un Constituant du Schéma de Relations correspondant à l'Association ssi cet Elément n'a pas été déclaré comme étant dérivable.

[en] Un Elément est dérivable si on lui a associé un Attribut DSL "Dérivable" qui a pour valeur "Oui".

Transformation d'un Groupe :

- Un Groupe d'une Association ou d'une Entité sera transformé en un Constituant décomposable du Schéma de Relations correspondant à l'Entité ou l'Association.
- Un Elément d'un Groupe sera transformé en un Constituant du Constituant correspondant au Groupe ssi cet Elément n'a pas été déclaré comme étant dérivable.
- Un Groupe d'un Groupe sera transformé en un Constituant décomposable du Constituant correspondant au Groupe.

Transformation d'un Identifiant :

- Un Identifiant d'une Entité ou Association sera transformé en un Identifiant du Schéma de Relations correspondant à l'Entité ou l'Association.
- Un Identifiant d'une Entité ou Association sera transformé en une Clé ssi il répond aux critères de Clé du SGBD d'ADR (en l'occurrence, maximum 4 Constituants d'une longueur totale de maximum 180 positions (II.1.2.2)).

Pour chaque Schéma de Relations, on définit un Identifiant Principal ou Clé Principale (II.4.2). Le choix de cet identifiant est, malheureusement, purement aléatoire étant donné qu'il n'y a pas moyen de définir un Identifiant Principal au niveau du schéma conceptuel.

- Si un Rôle intervient dans la composition d'un Identifiant d'une Entité ou d'une Association, alors le Schéma de Relations correspondant à l'Entité ou l'Association hérite des Constituants formant l'Identifiant Principal du Schéma de Relations correspondant à l'Entité sur lequel porte le Rôle.
- Chaque Schéma de Relations correspondant à une Association hérite des Constituants formant les Identifiants Principaux des Schémas de Relations correspondant aux Entités que chacune de ces Associations relie.

RESULTATS

Outre la génération du schéma relationnel brut dans la BSR, ce processeur générera également un fichier de diagnostics des transformations. On y trouvera une description du schéma relationnel brut généré. Ce fichier permettra de vérifier si chaque Schéma de Relations a fait l'objet d'une définition d'un Identifiant Principal. Ceci est une condition nécessaire au bon fonctionnement du Présentateur Relationnel. Si cette condition n'est pas vérifiée, il est nécessaire d'adapter le schéma conceptuel.

EXEMPLE

Etant donné le schéma conceptuel de données représenté graphiquement par la Figure II.4.3.1.1.

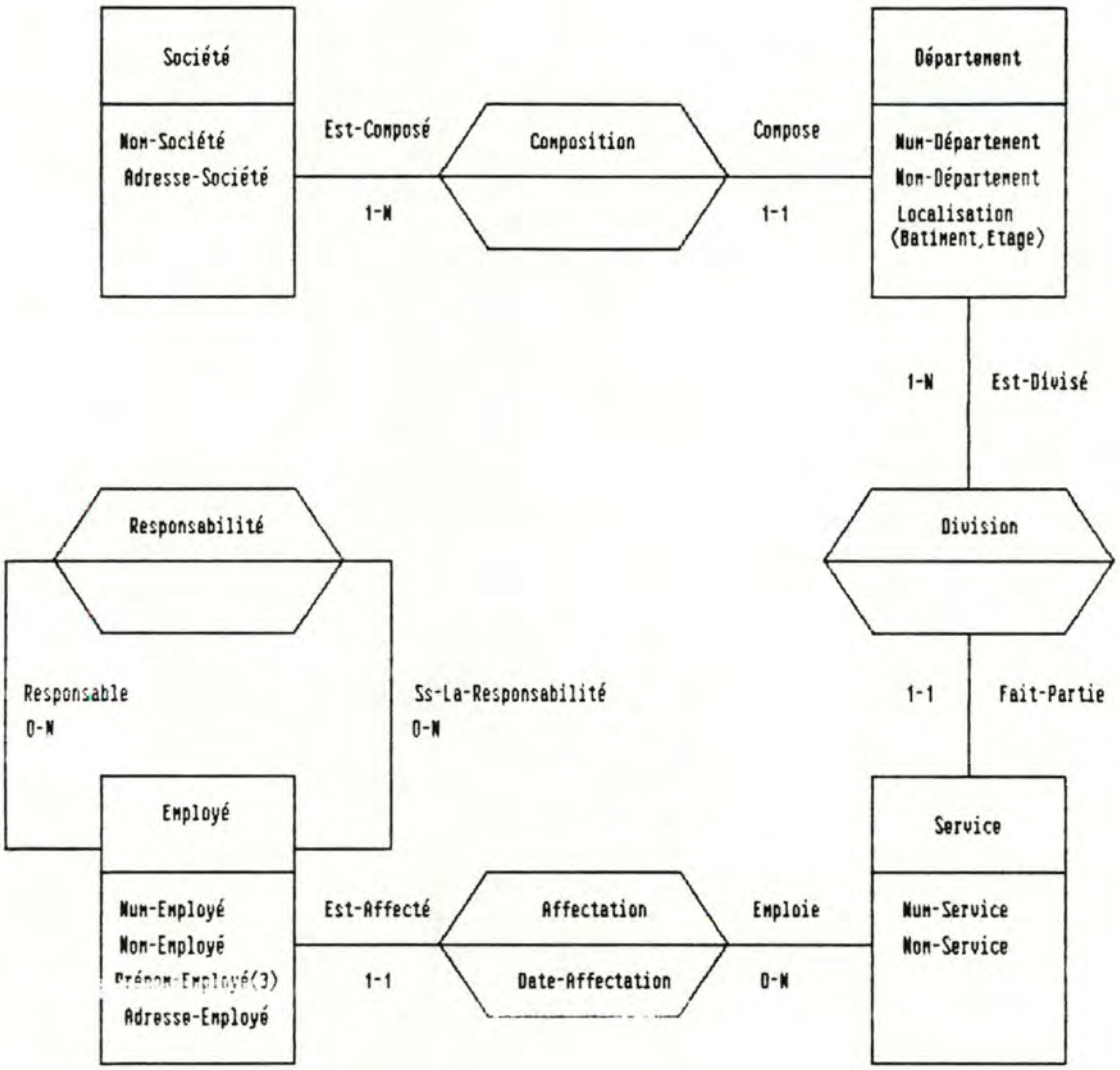


Figure II.4.3.1.1 : Schéma Conceptuel de Données

Dont les identifiants sont :

pour Société	:	Nom-Société	(element)
pour Département	:	Num-Département	(element)
pour Composition	:	Compose	(rôle)
pour Service	:	Est-Divisé	(rôle)
	:	Num-Service	(element)
pour Division	:	Fait-Parti-De	(rôle)
pour Employé	:	Num-Employé	(element)
pour Affectation	:	Est-Affecté	(rôle)
pour Responsabilité	:	Responsable	(rôle)
		Ss-La-Responsabilité	(rôle)

Le Schéma Relationnel Brut sera composé des Schémas de Relations suivants :

Société(Nom-Société:, Adresse-Société:)

Département(Num-Département:, Nom-Département:,
Localisation(Batiment:, Etage:):)

Composition(Num-Département:Compose, Nom-Société:Est-Composé)

Service(Num-Département:Est-Divisé, Num-Service:, Nom-Service:)

Employé(Num-Employé:, Nom-Employé:, Prénom-Employé(3):,
Adresse-Employé:)

Affectation(Num-Employé:Est-Affecté, Num-Département:Emploie,
Num-Service:Emploie, Date-Affectation:)

Responsabilité(Num-Employé:Responsable, Num-Employé:Ss-La-
Responsabilité)

II.4.3.2 Le Transformateur BSR/ADR

OBJECTIFS

Le Transformateur BSR/ADR ou Extracteur Logique est le processeur responsable de la création du schéma logique ADR. Le schéma physique sera défini par l'intermédiaire du DATADICTIONARY ON LINE d'ADR. Nous nous limitons donc à la définition des objets ADR suivants : Record, Field, Element, Key, Dataview (II.1.2.2). La génération de ces deux derniers objets correspond à la génération du schéma externe. Cette transformation doit se dérouler de manière automatique.

REGLES DE CORRESPONDANCES

- A un Schéma de Relations correspond un Record.
- A un Constituant correspond un Field.
- A une Clé correspond une Key.

REGLES DE TRANSFORMATIONS

Il n'y a pas à proprement parler de transformations entre le modèle de la BSR et le modèle d'ADR, tous deux d'inspiration relationnelle.

Transformations d'un Schéma de Relations :

- Un Schéma de Relations sera transformé en un Record.

Transformations d'un Constituant :

- Un Constituant d'un Schéma de Relations sera transformé en un Field du Record correspondant au Schéma de Relations.
- Un Constituant d'un Constituant décomposable sera transformé en un Field du Field correspondant au Constituant décomposable.

Transformations d'une Clé :

- Une Clé Principale d'un Schéma de Relations sera transformée en une Master et Native Key (II.1.2.2) du Record correspondant au Schéma de Relations.
- Une Clé Secondaire d'un Schéma de Relations sera transformée en une Key (qui est ni Master, ni Native) du Record correspondant au Schéma de Relations.

De plus le Transformateur BSR/ADR génèrera pour chaque Record :

- Un Element défini sur l'ensemble des Fields du Record.
- Deux Dataviews définies sur l'Element, l'une en écriture, l'autre en lecture.

OPTIONS

Lors de l'extraction, l'utilisateur aura plusieurs choix quant aux transactions ADR (II.1.2.3) générées pour chaque Schéma de Relations. L'utilisateur pourra choisir une combinaison des différentes options.

1e Option (Choix des Schémas de Relations) :

- Soit on génère les descriptions de tous les Schémas de Relations décrits dans la BSR.
- Soit on génère les descriptions de certains Schémas de Relations sélectionnés.

De toute manière, l'on ne pourra procéder à la génération de ces descriptions que pour les Schémas de Relations qui ont fait l'objet d'une définition de Noms ADR pour les Schémas de Relations ainsi que pour tous leurs Constituants. Ceci peut se faire par l'intermédiaire des fonctions de Paramétrages du Présentateur Relationnel (II.4.4.1.5).

2e Option (Option de Connection) :

Les descriptions générées doivent-elles oui ou non être reliées à des spécifications existantes dans le DATADICTIONARY d'ADR, tels que File, Area ou Database (II.1.2.2) ?

- Si l'option "non" est choisie pour un Schéma de Relations, alors les descriptions correspondantes se trouveront dans le Dictionnaire d'ADR indépendamment de toute autre description.
- Si l'option "oui" est choisie pour un Schéma de Relations et que la description d'une occurrence de File de même Nom que le Nom de Record du Schéma de Relations existe dans le Dictionnaire d'ADR, alors les descriptions de ce Schéma de Relations seront reliées à l'occurrence de File en question.

3e Option (Option de Suppression) :

Les occurrences d'objets décrites doivent-elles oui ou non exister sous un nouveau numéro de version (II.1.2.1) à côté de descriptions existantes des mêmes occurrences, ou doivent-elles entraîner la suppression de toutes descriptions préalables de ces occurrences?

- Si l'option "non" est choisie pour un Schéma de Relations, alors le dictionnaire contiendra la description du Record correspondant au Schéma de Relations sous différents numéros de version.
- Si l'option "oui" est choisie pour un Schéma de Relations, alors le dictionnaire ne contiendra que la description du Record correspondant à ce Schéma de Relations. Il n'existera plus de description du même Record sous un autre numéro de version.

RESULTATS

Le transformateur BSR/ADR génère un fichier qui contient les transactions ADR décrites ci-dessous. De plus amples informations concernant ces transactions ADR, peuvent être trouvées dans [DD2G-TX] et [DD4G-MG].

(Sous la transaction ADR, nous décrivons la convention de nom ADR prise chez Charles Veillon S.A.)

Pour un Record :

-ADD RECORD, <Nom-Record-ADR>(TEST)
-END

Où <Nom-Record-ADR> := <Code-Base><Nom-Record>

Pour un Field :

-ADD FIELD, <Nom-Record-ADR>.<Nom-Field>(TEST)
 4000 <after><parent>
 4001 <class><type> <length><decimals><repeat>
-END

Valeur de <after> :

 START (si c'est le premier Field d'un Record)
 * (si le Field fait partie d'un Field décomposable)
 <Nom-Field> du Field qui le précède (sinon)

Valeur de <parent> :

 <Nom-Field> du Field Décomposable (si le Field appartient à un Field décomposable)
 PARENT (sinon)

Valeur de <class> :

 S (si c'est un Field élémentaire (SIMPLE))
 C (si c'est un Field décomposable (COMPOUND))

Valeur de <type> :

 C (si c'est un Field de type alphanumérique)
 N (si c'est un Field de type numérique)

Valeur de <length> := Longueur du Field.

Valeur de <decimals> := Nombre de positions décimales du Field numérique.

Valeur de <repeat> := Répétitivité du Field.

Pour un Element :

```
-ADD ELEMENT, <Nom-Record-ADR>. <Nom-Défaut-Element> (TEST)
 6000<Nom-Element-BD>
(6001 ADD <Nom-Field>)
-END
```

Où <Nom-Défaut-Element> := TOTAL-1
Où <Nom-Element-BD> := <Code-Base><Code-Record>
Il y aura autant de clauses 6001 qu'il y a de Field dans le Record correspondant.

Pour une Key :

```
-ADD KEY, <Nom-Record-ADR>. <Nom-Key> (TEST)
 5000 <Nom-Key-BD><Key-Id><master-key><nativ-seq>
      <Key-include>
 5001 <Nom-Record-ADR>. <Nom-Field>       (1e Field de la Clé)
 5002 <Nom-Record-ADR>. <Nom-Field>       (2e Field de la Clé)
 5003 <Nom-Record-ADR>. <Nom-Field>       (3e Field de la Clé)
 5004 <Nom-Record-ADR>. <Nom-Field>       (4e Field de la Clé)
-END
```

Où <Nom-Key-BD> := <Code-Base><Code-Record>-<Numéro>
Où <Numéro> := 0 pour une Master Key ou Clé primaire.
1-9 pour les autres.

Valeur de <master-key> :

```
 Y   (si c'est une Master Key)
 N   (sinon)
```

Valeur de <native-seq> :

```
 Y   (si c'est une Native Key)
 N   (sinon)
```

Valeur de <Key-Id> : Numéro quelconque mais identique pour chaque Clé définie dans le même Index.

Pour une Dataview :

```
-ADD DATAVIEW, <Nom-Dataview>. <Nom-Record-ADR> (TEST)
               DVW-ELM-ACCESS SUBJECT
```

Où <Nom-Dataview> := <Record-Name>-<Numéro>
Où <Numéro> := 0 pour une Dataview en lecture seulement.
1 pour une Dataview en mise-à-jour.

Option de Connection

Dans le cas ou l'Option de Connection = "oui" le fichier contiendra également la clause suivante au sein de chaque section de Record :

```
1000 CONNECT, <Nom-Fichier> (TEST)
```

Où <Nom-Fichier> := <Nom-Record-ADR>

Option de Suppression

Dans le cas ou l'Option de Suppression = "oui" le fichier contiendra avant toute section de Record la section suivante :

```
-DEL RECORD,<Nom-Record-ADR>(TEST)
-END
```

Des erreurs pourront se produire lors de l'introduction en "batch" du fichier dans le DATADICTIONARY d'ADR. Elles sont dues à un choix d'options impossibles avec les définitions courantes du dictionnaire (Ex. : Choix de l'option de Connection = "oui" alors qu'aucune occurrence File de même nom que le Record n'existe dans le dictionnaire). Ces erreurs ne peuvent être vérifiées au niveau du poste de travail.

II.4.4 Editeurs de Schémas Relationnels

II.4.4.1 Le Présentateur Relationnel

OBJECTIFS

Ce processeur interactif est l'outil de base de l'utilisateur. A l'aide du Présentateur Relationnel, il va pouvoir transformer son Schéma Relationnel Brut en un Schéma Relationnel Logique en passant par un Schéma Relationnel Optimalisé. Pour ce faire, ce Présentateur doit offrir les fonctions nécessaires pour réaliser cette tâche. L'ensemble des fonctions offertes doit permettre de passer par ces différentes étapes, sans rendre cette séquence obligatoire par le Présentateur lui même.

FONCTIONS

Les différentes fonctions peuvent être classées en 5 classes :

- Les fonctions de Transformation de Schémas, permettant de transformer le schéma relationnel brut à l'aide d'opérateurs définis sur les Schemas de Relations, et ses Constituants.
- Les fonctions d'Aide à l'Optimisation, offrant la possibilité d'évaluer les Schémas de Relations et les Index quant à la place nécessaire pour leur stockage sur disque.
- Les fonctions de Définition des Accès permettant de doter les Schémas de Relations de mécanismes d'accès.
- Les fonctions de Services offrant la possibilité d'extraire différentes données de la base sous forme de listes.
- Les fonctions de Paramétrage permettant de définir les caractéristiques de l'interface et des disques du système, ainsi qu de définir les Noms ADR de la Base, des Schémas de Relations, et des Constituants.

II.4.4.1.1. Transformations de Schémas

On distinguera deux types de transformation :

- Les transformations sur les Schémas de Relations.
  - Collage de deux Schémas de Relations.
  - Eclatement d'un Schéma de Relations.
- Les transformations sur les Constituants.
  - Regroupement de Constituants.
  - Décomposition d'un Constituant décomposable.
  - Modification de la répétitivité d'un Constituant.

Certains exemples sont tirés de II.4.3.1. La notation utilisée est définie en II.4.2.

II.4.4.1.1.1 Transformations sur les Schémas de Relations

COLLAGE

Définition :

Le Collage ou Fusion de deux Schémas de Relations R et S consiste à créer un Schéma de Relations RS composé de l'union des ensembles de Constituants de R et de S et à supprimer les Schémas de Relations R et S.

Démarche et règles à suivre :

- Sélectionner deux Schémas de Relations R et S que l'on désire coller (1).
- Modifier les rôles que jouent les Constituants au sein des deux Schémas de Relations R et S afin d'avoir des Constituants identiques dans les deux Schémas de Relations. Les Attributs sur lesquels ces Constitants sont définis ne pouvant être modifiés, cette opération n'autorise la fusion de deux Schémas de Relations que sur base de Constituants définis sur les même Attributs, et a fortiori sur les mêmes Domaines de valeurs (2).
- Ajouter, si nécessaire, un Constituant au nouveau Schéma de Relations, pouvant éventuellement faire partie de l'identifiant de ce nouveau Schéma de Relations qui est composé de l'union des Identifiants Principaux (II.4.2) des deux Schémas de Relations R et S. Donner un nom au nouveau Schéma de Relations RS (3).

Exemples :

A (1) Considérons les Schémas de Relations suivants :

Département(Num-Département:, Nom-Département:,
Localisation(Batiment:, Etage:):)
Composition(Num-Département:Compose,
Nom-Société:Est-Composé)

(2) En supprimant le rôle "Compose" de "Num-Département" dans "Composition" on peut Coller les deux Schémas de Relations sur base de "Num-Département".
(3) En donnant au nouveau Schéma de Relations le nom "Département", on dispose du Schéma de Relations suivant :

Département(Num-Département:, Nom-Département:,
Nom-Societé:Est-Composé, Localisation
(Batiment:, Etage:):)

B (1) Considérons les Schémas de Relations suivants :

Commande(Num-Commande:, Date:, Prix-Commande:)
Ligne-Cde(Num-Commande:, Num-Produit:, Quantité:)

(2) On peut Coller les deux Schémas de Relations sur "Num-Commande".
(3) En donnant au nouveau Schéma de Relations le nom "Ligne-Commande", on dispose du Schéma de Relations

suivant :

Ligne-Commande(Num-Commande:, Num-Produit:, Date:,
Prix-Commande:, Quantité:)

C (1) Considérons les Schémas de Relations suivants :

Pièce-Voiture(Num-Pièce:, Prix:)
Pièce-Camion(Num-Pièce:, Prix:)

(2) On peut Coller les deux Schémas de Relations sur "Num-Pièce".
(3) En donnant au nouveau Schéma de Relations le nom "Pièce-Véhicule", et en ajoutant un nouveau Constituant "Type-Pièce", j'ai encore deux possibilités quant à l'identifiant :

- Si Num-Pièce identifie une Pièce au sein de toutes les Pièces, alors on a le Schéma de Relations suivant :

  Pièce-Véhicule(Num-Pièce:, Type-Pièce:, Prix:)

- Si Num-Pièce n'identifie qu'une Pièce au sein des Pièce-Voiture ou Pièce-Camion, alors Num-Pièce fait partie de l'identifiant et on a le Schéma de Relations suivant :

  Pièce-Véhicule(Num-Pièce:, Type-Pièce:, Prix:)

Justification :

1. Permet la suppression d'un Schéma de Relations regroupant un ensemble de mécanismes d'accès à d'autres Schémas de Relations, en intégrant ces mécanismes au sein d'un de ces Schémas de Relations. Ainsi dans l'exemple A, le Schéma de Relations "Composition" regroupe les mécanismes d'accès à "Société" et "Département" et a été Collé à "Département" (Le Collage avec "Société" étant moins efficace comme on peut facilement s'en rendre compte).
2. Permet une dénormalisation d'un Schéma de Relations.
3. Permet l'aggrégation de plusieurs Schémas de Relations.
4. Permet l'opération inverse de l'Eclatement.

Remarque :

On peut remarquer que l'opération de Collage peut s'effectuer sur n'importe quel Constituant des Schémas de Relations. Néanmoins le Collage sur des Constituants n'intervenant pas dans l'identifiant d'un des deux Schémas de Relations peut générer un Schéma de Relations gravement dénormalisé. Ce genre d'opération sera effectué avec grande prudence.

ECLATEMENT

Définition :

L'Eclatement d'un Schéma de Relations RS consiste à créer deux Schémas de Relations R et S composés chacun d'un sous-ensemble (non nécessairement distinct) des Constituants de RS et à supprimer le Schéma de Relations RS.

Démarche et règles à suivre :

- Sélectionner un Schéma de Relations RS à Eclater (1).
- Définir des nouveaux noms des Schémas de Relations qui vont résulter de l'Eclatement (2).
- Sélectionner pour chaque Schéma de Relations R et S les Constituants de RS dont ils doivent être composés. Chaque Constituant de RS doit intervenir soit dans R, soit dans S, soit dans les deux. Seul un Constituant créé par l'opération de Collage peut ne pas être repris. Il est également obligatoire de sélectionner une partie ou la totalité des Constituants formant l'identifiant principal de RS pour les Schémas de Relations R et S (3).

Exemples :

A (1) Considérons le Schéma de Relations suivant :

Client(Num-Client:, Données-Pers:, Données-Stat:)

(2) Les deux Schémas de Relations que l'on désire créer sont "Client-Pers" et "Client-Stat".
(3) On sélectionne pour "Client-Pers" les Constituants "Num-Client" et "Données-Pers" et pour "Client-Stat" les Constituants "Num-Client" et Données-Stat". Les Schémas de Relations résultant sont :

Client-Pers(Num-Client:, Données-Pers)
Client-Stat(Num-Client:, Données-Stat)

B (1) Considérons le Schéma de Relations suivant :

Ligne-Commande(Num-Commande:, Num-Produit:, Quantité:, Prix:)

(2) Les deux Schémas de Relations que l'on désire créer sont "Ligne-Cde" et "Produit".
(3) On sélectionne pour "Ligne-Cde" les Constituants "Num-Commande","Num-Produit" et "Quantité" et pour "Produit" les Constituants "Num-Produit" et "Prix". Les Schémas de Relations résultants sont :

Ligne-Cde(Num-Commande:, Num-Produit:, Quantité:)
Produit(Num-Produit:, Prix:)

C (1) Considérons le Schéma de Relations suivant :

Pièce-Véhicule(Num-Pièce:, Type-Pièce:, Prix:)

(2) Les deux Schémas de Relations que l'on désire créer sont "Pièce-Voiture" et "Pièce-Camion".
(3) On sélectionne pour "Pièce-Voiture" et "Pièce-Camion" les Constituants "Num-Pièce" et "Prix". "Type-Pièce" n'est pas sélectionné (en considérant que "Type-Pièce" provient d'une opération de Collage). Les Schémas de Relations résultant sont :

Pièce-Voiture(Num-Pièce:, Prix:)
Pièce-Camion(Num-Pièce:, Prix:)

Justification :

1. Permet de créer un Schéma de Relations jouant le rôle d'Index en sortant un ensemble de mécanismes d'accès entre Schémas de Relations par Eclatement du Schéma de Relations possédant ces mécanismes.
2. Permet une normalisation des Schémas de Relations en supprimant des dépendances fonctionnelles entre Constituants.
3. Permet la dé-aggrégation d'un Schéma de Relations.
4. Permet l'opération inverse du Collage.

II.4.4.1.1.2 Transformations sur les Constituants

REGROUPEMENT

Définition :

Le Regroupement de Constituants consiste à créer un Constituant décomposable au sein d'un Schéma de Relations. Ce Constituant sera composé de Constituants de ce Schéma de Relations.

Démarche et règles à suivre :

- Sélectionner un Schéma de Relations (1)
- Définir le nom du Constituant décomposable à créer (2).
- Sélectionner les Constituants dont ce Constituant décomposable devra être composé (3).

Exemple :

(1) Considérons le Schéma de Relations suivant :

Personne(Nom-Personne:,Commune:,Rue:,Numéro:)

(2) On désire créer le Constituant décomposable "Adresse".
(3) Si "Adresse" doit être composé de "Commune", "Rue" et "Numero", on obtient le Schéma de Relations suivant:

Personne(Nom-Personne:,Adresse(Commune:,Rue:,Numéro:):)

Justification :

1. Une Clé d'accès à un Schéma de Relations ne pouvant être composée que de maximum 4 Constituants (II.4.2), cet opérateur permet de créer un Constituant décomposable et de pallier à cette contrainte.
2. Permet une dénormalisation d'un Schéma de Relations.
3. Permet l'opération inverse de la Décomposition.

DECOMPOSITION

Définition :

La Décomposition d'un Constituant décomposable permet de supprimer un Constituant décomposable en ne gardant que les Constituants qui le composent

Démarche et règles à suivre :

- Sélectionner un Schéma de Relations (1).
- Sélectionner un Constituant décomposable (2).

Exemple :

(1) Considérons le Schéma de Relations suivant :

Département(Num-Département:, Nom-Département:,
Localisation(Batiment:, Etage:):)

(2) Si nous désirons Décomposer le Constituant "Localisation", on obtient le Schéma de Relations suivant :

Département(Num-Département:, Nom-Département:,
Batiment:, Etage:)

Justification :

1. Permet de supprimer un Constituant décomposable afin de normaliser le Schéma de Relations.
2. Permet l'opération inverse du Regroupement.

MODIFICATION DE LA REPETITIVITE

Définition :

La Modification de la Reptitivité d'un Constituant consiste à (dé-) normaliser un Schéma de Relations en (augmentant) supprimant la répétitivité d'un Constituant (faisant) ne faisant pas partie d'un Identifiant, entraînant également (la suppression) l'introduction de ce Constituant dans l'Identifiant principal.

Démarche et règles à suivre :

- Sélection d'un Schéma de Relations (1).
- Sélection d'un Constituant dont on doit modifier la répétitivité et modification de cette dernière (2).

Exemple :

Deux cas se présentent :

A Soit on veut modifier la répétitivité d'un Constituant de répétitivité > 1 et ne faisant pas partie de l'Identifiant Principal en un Constituant de répétitivité = 1 et faisant partie de cet Identfiant.

(1) Considérons le Schéma de Relations suivant :

Employé(Num-Employé:, Nom-Employé:, Prénom-Employé(3):,
Adresse_Employé:)

(2) Si on désire supprimer la répétitivité de "Prénom-Employé", l'on obtient le Schéma de Relations suivant :

Employé(Num-Employé:, Prénom-Employé:, Nom-Employé:,
Adresse-Employé:)

B Soit on veut modifier la répétitivité d'un Constituant de répétitivité = 1 et faisant partie de l'Identifiant Principal en un Constituant de répétitivité > 1 et ne faisant pas partie de cet Identfiant.

(1) Considérons le Schéma de Relations suivant :

Employé(Num-Employé:, Prénom-Employé:, Nom-Employé:,
Adresse-Employé:)

(2) Si on désire rendre le Constituant "Prénom-Employé" répétitif on obtient le Schéma de Relations suivant :

Employé(Num-Employé:, Nom-Employé:, Prénom-Employé(*):,
Adresse_Employé:)

où * est le nombre maximum de "Prénom-Employé" par "Num-Employé".

Justification du cas A :

- Permet de supprimer la répétitivité d'un Constituant répétitif afin de normaliser le Schéma de Relations.

Justification du cas B :

- Permet de réaliser l'opération inverse du cas A, rendant la transformation réversible.

Les autres cas de modifications de répétitivité ne sont pas admis. Elles entraînent une modification de la sémantique.

II.4.4.1.1.3 Justifications des opérateurs de transformations

1. Tous les opérateurs ont leurs inverses, nous assurons ainsi la réversibilité des transformations.

2. De plus les contraintes d'utilisation de ces opérateurs, qui sont :

   Pour le Collage :
   Interdiction de Coller deux Schémas de Relations sur des Constituants n'étant pas définis sur les mêmes Attributs.
   Pour l'Eclatement :
   Interdiction de laisser tomber des Constituants autre que ceux crées par l'opération de Collage (et ne provenant donc pas du schéma conceptuel),
   Pour la Modification de la Répétitivité :
   Modification restreinte de la répétitivité des Constituants.

   doivent assurer qu'il n'y pas de perte de sémantique lors des transformations.

3. La combinaison des différents opérateurs permet de normaliser, à n'importe quel niveau (1FN, 2FN, 3FN, FNBC, 4FN ou 5FN), tous les Schémas de Relations. Ceci est du au fait que nous disposons d'opérateurs de suppression d'un Constituant décomposable (l'opérateur Décomposition), de modification de la répétitivité (l'opérateur Modification de la répétitivité), et de suppression de dépendances fonctionnelles (l'opérateur Eclatement).
4. Un opérateur de duplication de Constituants n'a pas été réalisé, mais cette duplication peut être obtenue par combinaison des opérateurs de Collage et d'Eclatement.

II.4.4.1.2. Définition des Accès

Plusieurs fonctions permettent la définition des mécanismes d'accès aux Schémas de Relations :

On distingue deux types d'objets :
- Les Clés d'accès.
- Les Index.

II.4.4.1.2.1. Gestion des Clés

Les Clés d'accès définies sur les Schémas de Relations doivent répondre aux contraintes de Clé de DATACOM/DB, la base de données d'ADR. Ces Contraintes sont les suivantes :

- Maximum 4 Constituants (qui peuvent être décomposables).
- Taille maximum de la Clé : 180 bytes.

Il existe deux types de Clé (II.4.2):

- Clé Principale ou Primaire (Une seule)
- Clé Secondaire (0, 1 ou plusieurs)

Chaque Schéma de Relations doit avoir au moins une Clé.

CREATION D'UNE CLE

Définition :

La fonction "Création de Clé" permet de désigner un ensemble de Constituants devant former une Clé Primaire ou Secondaire d'un Schéma de Relations.

Démarche à suivre pour la création d'une Clé Secondaire :

(1) Définition du nom de la clé.
(2) Spécification de la caractéristique "Clé-Vide" (oui ou non). Cette caractéristique spécifie que si une Clé faisant partie d'un Index a une valeur nulle, elle doit entraîner la création d'une entrée dans l'index (Clé-Vide = oui) ou pas (Clé-Vide = non).
(3) Sélection des Constituants devant former la Clé.

Démarche à suivre pour la création d'une Clé Principale :

(1) Nom de la Clé = IDENTIFIANT.
(2) "Clé-Vide" = oui.
(3) Sélection des Constituants devant former la Clé.

Les étapes (1) et (2) sont réalisées automatiquement.

Remarque:

Une Clé Principale peut être créée de deux façons :

- Explicitement : par utilisation de cette fonction.
- Implicitement : lors de la définition automatique d'un Identifiant principal réalisée par les transformations décrites en II.4.4.1.1.1 (Collage et Eclatement) et par les transformations réalisées par le Transformateur IDA/BSR (II.4.3.1).

SUPPRESSION D'UNE CLE

Définition :

La fonction "Suppression de Clé" permet de supprimer une Clé d'un Schéma de Relations.

Remarque :

Si la Clé supprimée fait partie d'un Index, cet Index sera également supprimé si ce dernier n'est pas défini sur d'autres Clés.

Une Clé peut être supprimée de deux façons :

- Explicitement : par utilisation de cette fonction.
- Implicitement : lors de l'utilisation des transformations décrites en II.4.4.1.1.2. (Modification de la Répétitivité et Décomposition) sur un des Constituants d'une Clé.

II.4.4.1.2.2. Gestion des Index

Un Index DATACOM/DB peut être un index multi-fichiers, c-à-d qu'il peut regrouper des clés d'accès à différents records appartenant à des fichiers différents. Ceci permet un accès rapide lorsque l'on désire accéder d'un record à un autre via les clés d'accès définies au sein du même index (Ex. : Lors d'opérations de "join")

Par analogie nous pourrons définir un Index regroupant des Clés définies sur des Schémas de Relations différents. Néanmoins, il est nécessaire que ces Clés soient semblables, c-à-d définies sur des Constituants qui, chacuns sont définis sur le même Attribut, et ceci dans le même ordre.

CREATION D'UN INDEX

Définition :

La fonction "Création d'un Index" permet de définir un Index et de lui associer des Clés.

SUPPRESSION D'UN INDEX

La fonction "Suppression d'un Index" permet de supprimer un Index.

Remarque :

Un Index peut être supprimé de deux façon :

- Explicitement : par utilisation de cette fonction.
- Implicitement : Si l'Index n'est défini que sur une seule Clé et que celle-ci est supprimée.

MODIFICATION D'UN INDEX

La fonction "Modification d'un Index" permet de supprimer la participation d'une Clé dans un Index.

Remarque :

Un Index peut être modifié de deux façons :

- Explicitement : par utilisation de cette fonction.
- Implicitement : Si une Clé de cette Index est supprimée.

II.4.4.1.3. Aides à l'Optimalisation

Une aide à l'optimalisation des Schémas de Relations est offerte par le biais de deux fonctions de calcul de l'espace disque nécessaire pour le stockage des données ou des index, selon les caractéristiques d'un support physique et de l'organisation physique des données propres au SGBD cible utilisé (en l'occurrence ADR/DATACOM/DB). Ce SGBD a la particularité qu'il effectue une compression des données et des index (Optionnelle pour les données, automatique pour les index). Les compressions suivantes sont d'application :

- Compression horizontale :

  Si dans une chaîne de caractères plus de trois caractères semblables se suivent, alors cette suite est compressée (Ex. : "Marcel Durant" suivit de 40 blancs sera stocké comme "Marcel Durant" avec 1 blanc).

- Compression verticale :

  Si à l'intérieur d'un bloc de données les chaînes "Smith John" et "Smith Johnny" se suivent, alors la seconde valeur est stockée comme "ny".

La compression permet un gain de 20 à 40% selon les données compressées. Etant donné que le gain dû à la compression ne peut être évalué qu'en comparant deux échantillons de données, l'un compressé, l'autre pas, on ne peut en tenir compte ici.

Une troisième fonction, non encore réalisée, devrait fournir des informations statistiques sur les Schémas de Relations et leurs Constituants.

CALCUL VOLUME DONNEES

Objectif :

Evaluation de l'espace disque pour stocker les relations d'un Schéma de Relations.

Structure des fichiers de Données DATACOM/DB ([DB2G-IN] et [DB2G-DS]) :

Les données sont rangées les unes après les autres dans des blocs de données. En début de chaque bloc se trouve un ensemble d'informations de contrôle (4 bytes). Si le mécanisme de LOG (caractéristique physique de stockage de données [DB2G-DS]) est utilisé, alors l'ensemble des informations de contrôle est défini sur 12 bytes.

Formule de calcul :

Espace utile par bloc :

esp-util = (bs -cb) x pb

bs : Taille du bloc (en nombre de bytes) utilisé sur le disque.
cb : 12 ou 4 bytes d'informations de contrôle.
pb : Taux de remplissage du bloc.

Nombre total de bytes de données :

bytes = lr x oc

lr : Longueur d'un record (ou relation).
oc : Nombre de relations du Schéma de Relations.

Nombre total de blocs de données :

bloc = bytes / esp-util (arrondi sup)

Nombre total de cylindres de données :

cyl = bloc /cs

cs : Nombre de blocs par cylindre

Démarche et règles à suivre :

- Sélection d'un Schéma de Relations (1).
- Sélection d'un Disque (2). Les caractéristiques de ce disque représentées par les paramètres bs, cs, pb peuvent être modifiées par l'utilisateur.
- La cardinalité d'un Schéma de Relations peut être modifiée par l'utilisateur. L'option de LOG doit être choisie par ce dernier (oui ou non) (3).

Résultats :

Les valeurs des paramètres lr, bloc, cyl et bytes sont données comme résultat à l'utilisateur.

Exemple :

(1) Soit un Schéma de Relations de 2000 relations d'une longueur de 150 bytes chacun.
(2) Soit un disque ayant des blocs de 1954 bytes, 150 blocs par cylindre et un taux de remplissage des blocs de 65%.
(3) L'option LOG = NON.

esp-util = (1954 - 4) x 0.65 = 1267

bytes = 150 x 2000 = 300000

bloc = 300000 / 1267 = 236

cyl = 236 / 150 = 2

CALCUL VOLUME INDEX

Objectif :

Evaluation de l'espace disque pour un Index.

L'espace disque nécessaire pour stocker un index (les index de DATACOM/DB ont une structure d'arbre) peut être évalué à 10% de l'espace disque nécessaire pour les données. Néanmoins, une formule permet de calculer cet espace de manière plus fine.

Structure des Index DATACOM/DB ([DB2G-IN] et [DB2G-DS]) :

L'Index Area est composé d'un Control Block (bloc de contrôle) et d'un ou plusieurs Index Blocks (blocs d'index).

L'Index Block est composé d'un Block Control Information (information de contrôle d'une longueur de 32 bytes + la longueur maximum d'une valeur de clé) et d'une ou plusieurs Index Entries (entrées d'index).

L'Index Entry est composé d'un Control Information (information de contrôle d'une longueur de 5 bytes), d'une Key Value (valeur de clé, de 1 à 180 bytes), d'une longueur de 7 bytes chacun).

Formule de calcul :

Espace utile par bloc :

esp-util = (bs - cb -lk) x pb

bs : Taille du bloc (en nombre de bytes) utilisé sur le disque.
cb : 32 premier bytes du Block Control Information.
lk : Longueur de la clé.
pb : Taux de remplissage du bloc.

Taille moyenne d'une entrée pour les valeurs de clé uniques.

tail-uniq = ce + lk + pl

ce : 5 bytes d'information de contrôle pour l'entrée d'index.
lk : Longueur de la clé.
pl : 7 bytes de pointeur.

Taille totale des entrées d'index :

tail-ent = (tail-uniq x nk) + (ek x pl)

nk : Nombre de records (ou relation) avec une valeur de clé unique.
ek : Nombre de records (ou relation) avec une valeur de clé non unique.
pl : 7 bytes de pointeur.

Nombre de blocs d'index de bas niveau (feuilles de l'arbre d'index) :

ind-bn = tail-ent / esp-util (arrondi sup)

Nombre total de blocs d'index :

bloc = ind-bn x (1 + hp) + cn (arrondi sup)

hp : Nombre de blocs de haut niveau (ceux qui ne sont pas feuilles de l'arbre d'index) exprimé comme % du nombre de blocs de bas niveau (5%).
cn : Nombre de blocs de contrôle (1).

Nombre total de bytes d'index :

bytes = tail-ent

Nombre total de cylindres d'index :

cyl = bloc / cs (arrondi sup)

cs : Nombre de blocs par cylindre.

Démarche et règles à suivre :

- Sélection d'un Index (1).
- Sélection d'un Disque (2). Les caractéristiques de ce disque représentées par les paramètres bs, cs, pb peuvent être modifiées par l'utilisateur.
- Les paramètres nk et ek doivent être donnés par l'utilisateur (3).

Résultat :

Les valeurs des paramètres lk, bloc, cyl et bytes sont données comme résultat à l'utilisateur.

Exemple :

(1) Soit un Index défini sur deux Clés semblables et d'une longueur de 50 bytes définies sur un Schéma de Relations de 2000 relations et un autre de 8000 relations.
(2) Soit un disque ayant des blocs de 1954 bytes, 150 blocs par cylindre et un taux de remplissage des blocs de 65%.
(3) Pour chaque relation du premier Schéma de Relations, il y correspond 4 relations du second Schéma de Relations ayant la même valeur de clé.

esp-util = (1954 - 32 - 50) x 0.65 = 1216

tail-uniq = 5 + 50 + 7 = 62

tail-ent = (62 x 2000) + (8000 x 7) = 180000

ind-bn = 180000 / 1216 = 149

bloc = 149 x (1 + 0.05) + 1 = 157

bytes = 180912

cyl = 157 / 150 = 2

II.4.4.1.4 Services

Un ensemble de fonctions générales est également offert à l'utilisateur.

LISTE DES SCHEMAS

La fonction Liste des Schémas donne la liste des Schémas de Relations existant dans la Base.

LISTE DES INDEX

La fonction Liste des Index donne la liste des Index existant dans la Base.

LISTE DES CLES

La fonction Liste des Clés donne la liste des Clés existant dans la Base ainsi que les Schémas de Relations et éventuellement les Index auquelles elles appartiennent.

CONSTITUANTS D'UN SCHEMA

La fonction Constituants d'un Schéma permet de connaître l'ensemble des Constituants d'un Schéma de Relations.

IDENTIFIANTS D'UN SCHEMA

La fonction Identifiants d'un Schéma donne, pour chaque Identifiant d'un Schéma de Relations, la liste des Constituants dont il est formé.

CLES D'UN SCHEMA

La fonction Clés d'un Schéma donne, pour chaque Clé d'un Schéma de Relations, la liste des Constituants dont elle est formée.

CHANGEMENT D'ORDRE ET DE ROLE

La fonction Changement d'Ordre et de Nom de Rôle permet de modifier l'ordre d'apparition des Constituants dans un Schéma, ainsi que la modification du Nom de Rôle des Constituants. (De par sa caractéristique de changer l'ordre des Constituants, cette fonction pourrait se trouver dans les fonctions de transformation de Schémas décrites en II.4.4.1.1. Elles se trouvent ici pour des raisons historiques.)

II.4.4.1.5 Paramétrages

Sont regroupées ici les fonctions permettant de changer les valeurs par défaut du Présentateur Relationnel ainsi que de préparer l'Extraction Logique (II.4.3.2).

COULEURS

La fonction Couleurs permet d'adapter la présentation des écrans du Poste de Travail aux désirs de l'utilisateur.

CARACTERISTIQUES DISQUES

La fonction Caracteristiques Disques permet de visualiser les caractéristiques (taille des blocs, taille des cylindres, taux de remplissage) des disques définis et de modifier ces caractéristiques ainsi que d'ajouter de nouveaux disques.

NOMS ADR

La fonction Noms ADR permet de définir les Noms ADR des Records et Fields correspondant respectivement aux Schémas de Relations et aux Constituants ainsi que le Code de la Base et les Codes des Records. Ceci doit être réalisé pour tous les Schémas de Relations et Constituants dont on désire effectuer l'extraction logique (II.4.3.2).

II.4.4.2 Editeur des Mises-à-jour

OBJECTIFS

Cet éditeur devrait permettre de modifier le contenu de la BSR pour y répercuter des modifications faites au niveau conceptuel, dont on aimerait tenir compte dans les niveaux inférieurs, et ceci sans devoir répercuter tout le schéma conceptuel entraînant la perte de toutes transformations effectuées préalablement. Ici se pose le problème des répercussions des mises-à-jour d'un niveau à un autre. Une répercussion automatique de ces mises-à-jour, devant tenir compte de l'impact de ces modifications, nous semble trop lourde (Ex. : Que ce passe-t-il si je change l'Identifiant d'une Entité ? cfr. II.4.3.1). La solution d'un éditeur offre des possibilités plus souples, mais n'assure en aucun cas la correspondance entre les schémas des différents niveaux. La responsabilité de la correspondance entre le schéma conceptuel et le schéma logique appartient donc à l'utilisateur qui effectue ces modifications. Tout dépend bien entendu des possibilités qui seront offertes par l'éditeur et qui doivent encore faire l'objet d'une étude approfondie.

II.4.5 Utilitaires

II.4.5.1 L'Extracteur Conceptuel

OBJECTIFS

L'Extracteur IDA ou Extracteur Conceptuel est un processeur qui extrait les schémas conceptuels de données mis sous forme canonique. Il est destiné à produire un lien entre IDA et d'autres logiciels nécessitant la description des schémas conceptuels définis au sein d'IDA. Cet extracteur a fait l'objet d'une spécification complète dans [GAT-BER 86].

RESULTATS

Deux fichiers, l'un contenant la description du schéma conceptuel, l'autre contenant la description des propriétés communes des occurrences d'objets décrites dans le premier fichier, sont produits par l'Extracteur Conceptuel. Ces fichiers ne sont pas destinés à être utilisés comme rapports documentaires, mais pour être traités par programme.

II.4.5.2 Le Répercuteur des Correspondances

OBJECTIFS

Ce processeur a pour objectif de réaliser la répercussion de la correspondance qui peut exister entre les occurrences d'objets DSL et les occurrences du dictionnaire de données d'ADR.

REGLES

Cette correspondance peut se faire avec les informations contenues dans la BSR et ceci pour trois raisons :

- A chaque Schéma de Relations, Constituant ou Clé décrit dans la BSR ne correspond qu'une et une seule occurrence d'objet ADR.
- Les Schéma de Relations et Constituants correspondant respectivement aux Entités et Associations d'une part et aux Eléments et Groupes d'autre part, peuvent avoir fait l'objet d'une définition de leurs noms ADR au sein même du Présentateur Relationnel à l'aide des fonctions de paramétrage (II.4.4.1.5).
- Les fonctions de transformation de Schéma du Présentateur Relationnel (II.4.4.1.1) effectuent le suivi de la correspondance entre occurrences d'objets DSL et occurrences d'objets BSR.

Cette correspondance se fera au sein d'IDA, et non dans l'environnement ADR. Ce premier dispose de moyens plus puissants pour définir cette correspondance et pour l'analyser via DSL-SPEC. De plus, vouloir stocker cette correspondance dans le dictionnaire d'ADR de manière intelligente (c-à-d. pas sous forme de texte libre qui ne se prête pas à l'analyse) nécessite de définir le modèle Entité-Association au sein même du dictionnaire de données d'ADR, entraînant une redondance accrue et un problème de maintenance des deux dictionnaires.

RESULTATS

Un fichier DSL sera créé, contenant les sections DSL décrites ci-dessous et pouvant être introduites dans la base de données d'IDA via DSL-SPEC [DSL-SPEC 84].

Pour chaque Entité décrite dans la BSR :

```
DEFINIR ENTITE <Nom-Entité>;
[IMAGE <Nom-Record> TYPE record;]*
```

Pour chaque Association décrite dans la BSR :

```
DEFINIR ASSOCIATION <Nom-Association>;
[IMAGE <Nom-Record> TYPE record;]
```

---

* la clause IMAGE peut être répétitive

Pour chaque Groupe décrit dans la BSR :

```
DEFINIR GROUPE <Nom-Groupe>;
[IMAGE <Nom-Field> TYPE field;]
```

Pour chaque Elément décrit dans la BSR :

```
DEFINIR ELEMENT <Nom-Elément>;
[IMAGE <Nom-Field> TYPE field;]
```

En fin de fichier une remarque indique le nombre d'occurrences d'objets DSL décrites dans ce fichier n'ayant pas fait l'objet d'une spécification de nom ADR. Ces occurrences d'objets sont ceux qui ont fait l'objet d'une entête de section DSL dans ce fichier, mais n'ont pas de clause IMAGE.

II.4.5.3 Le Transformateur de Textes

OBJECTIFS

Ce processeur devrait permettre d'adapter les descriptions faites sur des occurrences d'objets DSL pour les introduire dans le DATADICTIONARY d'ADR.

REGLES

Ces descriptions proviennent d'un des deux fichiers en provenance de l'Extracteur Conceptuel. L'utilisateur devra définir les occurrences d'objets ADR qui devront hériter de la Description portant sur une occurrence d'objet DSL. L'utilisateur disposera des correspondances entre occurrences d'objets pour faire son choix. Une fois la transformation effectuée, le processeur génèrera un fichier de transactions ADR destinées à être introduit en "batch" dans le DATADICTIONARY d'ADR.

RESULTATS

Le fichier généré contiendra les transactions suivantes :

(Sous la transaction ADR, nous décrivons la convention de nom ADR prise chez Charles Veillon S.A.)

Pour un Record :

```
-UPD RECORD,<Nom-Record-Adr>(TEST)
 1001 '<Nom-Auteur>'
 1200 TEXT ADD
      <Description>
 1200 TEXT END
-END
```

Où <Nom-Record-Adr> := <Code-Base><Nom-Record>

Pour un Field :

```
-UPD FIELD,<Nom-Record-Adr>.<Nom-Field>(TEST)
 1001 '<Nom-Auteur>'
 1200 TEXT ADD
      <Description>
 1200 TEXT END
-END
```

Des erreurs pourront se produire lors de l'introduction en batch du fichier dans le DATADICTIONARY d'ADR. Elles sont dues à des modifications dans le dictionnaire (Ex. : Le Record dont on désire ajouter la description n'existe plus ou l'Extraction Logique n'a pas été effectuée.). Ces erreurs ne peuvent être vérifiées au niveau du poste de travail.

TROISIEME PARTIE

REALISATION DE L'INTERFACE

III.1 Environnement

Poste de travail

L'ensemble des processeurs décrits au niveau logique de la figure II.4.1. forme le poste de travail qui est développé sur PC. L'environnement IDA (y inclus l'Extracteur Conceptuel (II.4.5.1)) et l'environnement ADR, se situant respectivement au niveau conceptuel et au niveau physique, se trouvent sur mainframe.

Pour le bon fonctionnement du poste de travail, il est conseillé d'utiliser un PC doté d'un disque dur, d'une mémoire centrale de 640K, et fonctionnant sous MS-DOS version 2.0 ou supérieure. Un logiciel (software et hardware) de communication avec le mainframe permettant le transfert de fichier ASCII est également nécessaire.

Communications avec le mainframe

Le poste de travail doit pouvoir recevoir (en entrée) ainsi que fournir (en sortie) des fichiers de ou vers le mainframe. Ces fichiers sont :

en entrée :

- Les deux fichiers produits par l'Extracteur Conceptuel (II.4.5.1).

en sortie :

- le fichier des correspondances, produit par le Répercuteur (II.4.5.2) et à destination de l'environnement IDA.
- le fichier contenant le schéma logique DATACOM/DB, produit par l'Extracteur Logique (II.4.3.2) et à destination de l'environnement ADR,
- le fichier contenant les descriptions des occurrences d'objets ADR/IDEAL, produit par le transformateur de textes (II.4.5.3) et à destination de l'environnement ADR.

Aucune autre communication n'est nécessaire entre le PC et le mainframe. Celle-ci se réduit donc à un simple transfert de fichiers entre machines.

III.2 Interface homme/machine

III.2.1 Le dialogue

Le contexte d'utilisation de l'interface IDA/ADR doit permettre de déterminer les propriétés du dialogue entre l'utilisateur humain et ce programme. En fait, le contexte d'utilisation de l'interface est un contexte tout à fait particulier. En effet, on ne conçoit pas une base de données par jour et donc la fréquence d'utilisation de l'interface est très faible bien que l'utilisation en elle-même de l'interface soit intensive pendant la période de conception. Etant donné cette fréquence d'utilisation, il existe une forte probabilité pour que l'utilisateur soit différent à chaque utilisation. De plus, une base de données est souvent associée à un projet, et à un groupe de personnes travaillant sur ce projet. Par projet, il y aura donc un ou plusieurs utilisateurs, ce qui augmente encore la diversité de ceux-ci. Les utilisateurs seront donc en général peu au courant de la procédure d'utilisation et l'utilisation, quand à elle, sera souvent occasionnelle.

De ce contexte particulier, découle un dialogue tout aussi particulier. Celui-ci sera en tout cas à l'initiative du programme c.a.d. qu'il présentera à l'utilisateur toutes les alternatives possibles au moyen de menus. Un tel type de dialogue permet de s'assurer de la bonne qualité des entrées et par là d'éviter de nombreuses procédures de vérification. Il présente cependant l'inconvénient, pour l'utilisateur expérimenté, de ne pas pouvoir sauter d'étapes. Cet inconvénient n'est pas pour autant majeur puisqu'il permettra d'assurer plus facilement une guidance méthodologique lors des transformations des schémas de relations.

En ce qui concerne la complexité du dialogue (nombre d'alternatives), elle est réduite au maximum malgré la redondance de certaines commandes qui favorise une utilisation plus rationnelle des alternatives. Comme l'utilisateur est souvent peu expérimenté, la charge informationnelle est assez elevée (beaucoup d'écrans explicatifs) quoi que, si l'aide n'est pas sollicitée, elle n'entrave pas l'utilisation courante. Il est toutefois souhaitable, que l'utilisateur ait de légers prérequis quant à la manière d'utiliser les outils de base du dialogue (menus, fenêtres et grilles dont les différentes utilisations sont décrites au début du mode d'emploi).

III.2.2 Les procédures de dialogue (c.a.d. la façon d'utiliser un type d'entrée ou de sortie particulier)

Les outils permettant d'effectuer les procédures de dialogue entre l'utilisateur et le programme ont été construits de manière à permettre au premier de donner au second des informations de types différents. Ces types sont les suivants : des alternatives pour orienter le programme vers l'une ou l'autre de ses fonctions; une fois choisie, une fonction sera réalisée en déterminant des actions à effectuer; ces actions peuvent être alors précisées par des paramètres que l'on choisit dans un ensemble; enfin l'utilisateur peut fournir au programme des données dont seule la syntaxe sera déterminée, la valeur sémantique d'une donnée ne peut être appréciée par le programme que si la syntaxe ou la situation induit cette valeur; ces données devront être stockées à la demande de l'utilisateur.

Les outils ont été aussi construits de manière à permettre à l'ordinateur de donner à l'utilisateur des informations de types différents. Ces types sont les suivants : des listes pouvant être vues soit comme un ensemble de paramètres soit comme un ensemble de données selon l'utilisation qu' on leur assigne; des données isolées ou regroupées (sous une autre forme que celle d'une liste) selon leur sémantique ; des messages d'erreur ou de guidance; et enfin des textes qui permettent d'informer l'utilisateur de ce qu'il doit faire.

Les outils permettant de désigner des alternatives ou des actions sont appelés des menus. Ils sont composés de propositions qui représentent les alternatives ou les actions à choisir. La différence entre un menu proposant des actions par rapport à un menu d'alternatives, c'est que ce menu est toujours accompagné d'autres outils de saisie d'informations qui permettrons de définir cette action. Le choix d'une proposition se fait par pointage direct au moyen des touches de direction [↑], [↓], [→], [←], [Home], [End], [PgUp], [PgDn] (voir le mode d'emploi).

Les outils permettant de présenter une liste à l'utilisateur sont appelés des fenêtres. Ces fenêtres sont en fait des ouvertures à l'écran sur cette liste généralement trop longue pour être présentée complètement à l'écran. La différence entre une liste de données et une liste de paramètres est que l'utilisateur peut pointer dans la liste de paramètres un ou plusieurs élément de cette liste pour définir, par exemple, une action. La gestion d'une fenêtre est définie dans le mode d'emploi.

Les messages et les textes, quand à eux, sont gérés également par des fenêtres mais des fenêtres particulières à cet emploi. Ces fenêtres sont toujours situées au même endroit. Elles contiennent respectivement, soit un message selon qu'il s'agit de la fenêtre dédiée aux messages, soit un texte selon qu'il s'agit de la fenêtre dédiée à l'information de l'utilisateur.

Les outils permettant de gérer l'entrée ou la sortie des données sous une autre forme que celle d'une liste sont appelés des grilles. Ces grilles sont composées de champs protégés ou de champs non protégés. Les champs non protégés sont des zones où l'utilisateur peut faire un échange de données avec le programme, les zones protégées servent à reconnaitre et à qualifier l'endroit de la grille où sont définies les zones non protégées. La gestion d'une grille est décrite dans le mode d'emploi.

Ces outils sont gérés par des modules de programmation (cfr description des modules). Ces modules offrent des primitives qui permettent au concepteur de l'interface de réaliser les procedures de dialogue. De plus ces modules offrent, outre la possibilité de définir le dialogue homme/machine, la possibilité de paramètrer les outils qu'ils gérent. Par exemple de définir les messages et les textes dans la langue de l'utilisateur, de redessiner une grille selon le désir d'une catégorie d'utilisateurs ou encore de déterminer la couleur à l'écran des composants des outils du dialogue.

III.3 Architecture de programmation

Le programme est composé d'un ensemble de modules. Ces différents modules sont décrits dans une architecture logique selon une hiérarchie "Utilise". Deux grandes filières sont à distinguer : la filière "accès aux données", et la filière "gestion du dialogue", la première s'occupant de l'accès aux données et la deuxième s'occupant de la gestion de l'interface avec l'utilisateur pour réaliser les traitements (figures III.3.1 et III.3.2). Chacune des filières a fait l'objet d'une découpe en niveaux et en modules selon la hiérarchie "Utilise" qui veut qu'un module A de niveau i utilise un module B de niveau i+1 ssi le fonctionnement correct de A dépend de la disponibilité d'une version correcte de B. Chaque module offre un service d'un type bien déterminé qui peut être obtenu à l'aide d'une des primitives de ce module. La spécification de chaque module ainsi que de ses primitives est donnée ci-dessous, en commançant par les modules du bas de la hiérarchie (ayant le niveau le plus élevé) et en remontant dans cette hiérarchie.

La filière "Accès aux Données" est assuré par les modules "Accès aux Données", "Extraction-listes" et "Traitements-Listes".

La filière "Gestion du Dialogue" est assuré par les modules "Dial", "Initcolor", "Menu", "Wind", "Grid", "Help", "Wup", "Head", "Dimwind", "Buildbuf" et "Color".

Les traitements sont assurés par les modules "Etablissement-Transformations", "Calcul-Volume", "Transformations-Schémas", "Aide-Optimalisation", "Définition-Accès", "Services", "Paramétrages".

Le module "Verification-Transformation" est un module appelé par le module "Gestion du Dialogue" pour contrôler les actions de l'utilisateur.

Logiciels utilisés :

- Lattice C-Compiler.
- Lattice C-Food Smorgasbord
- Plib-86
- MS-DOS 2.11.
- Base de données de l'atelier de conception de base de données [CADELLI 86]

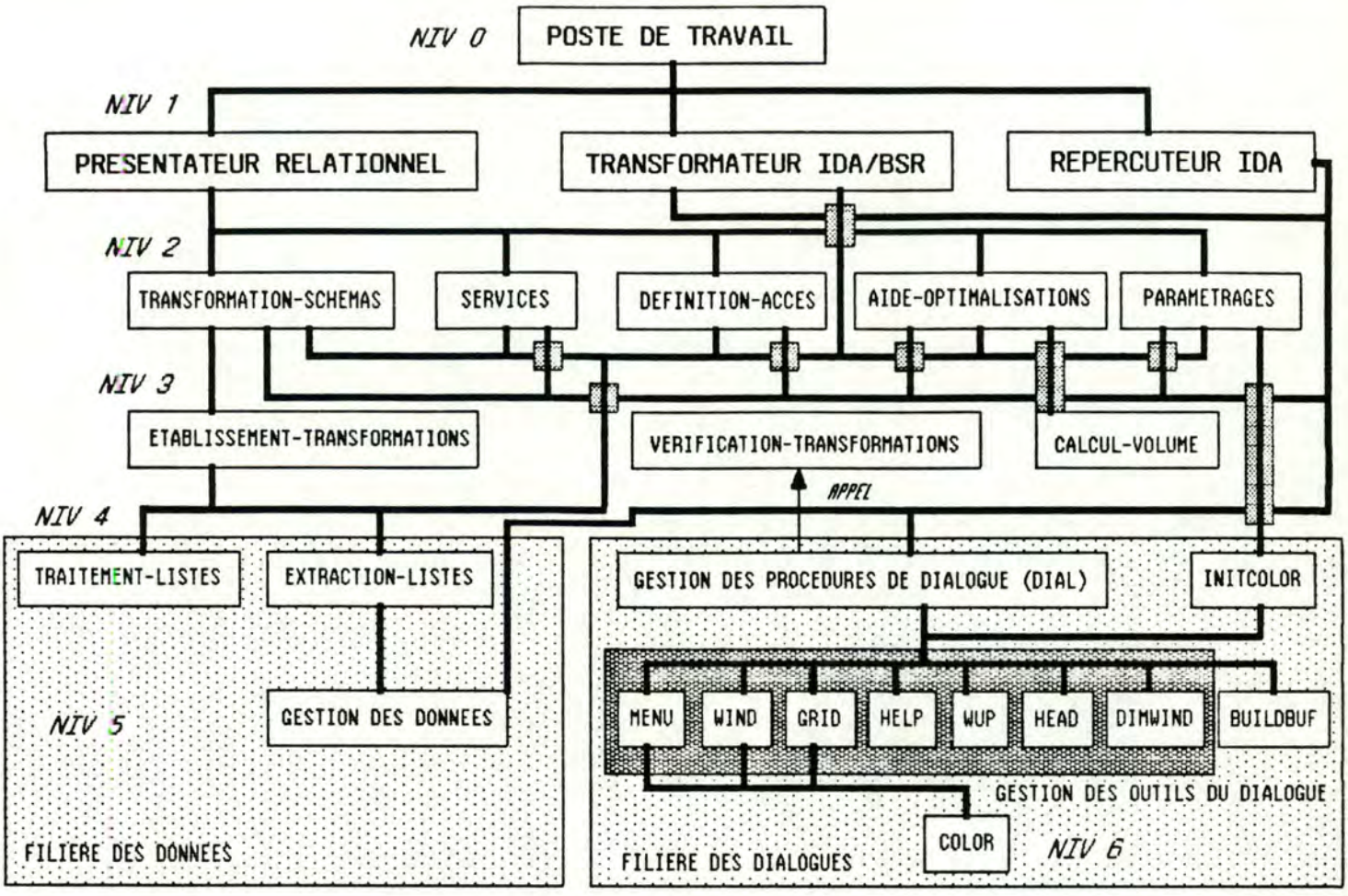


Figure III.3.1 : Hiérarchie des modules

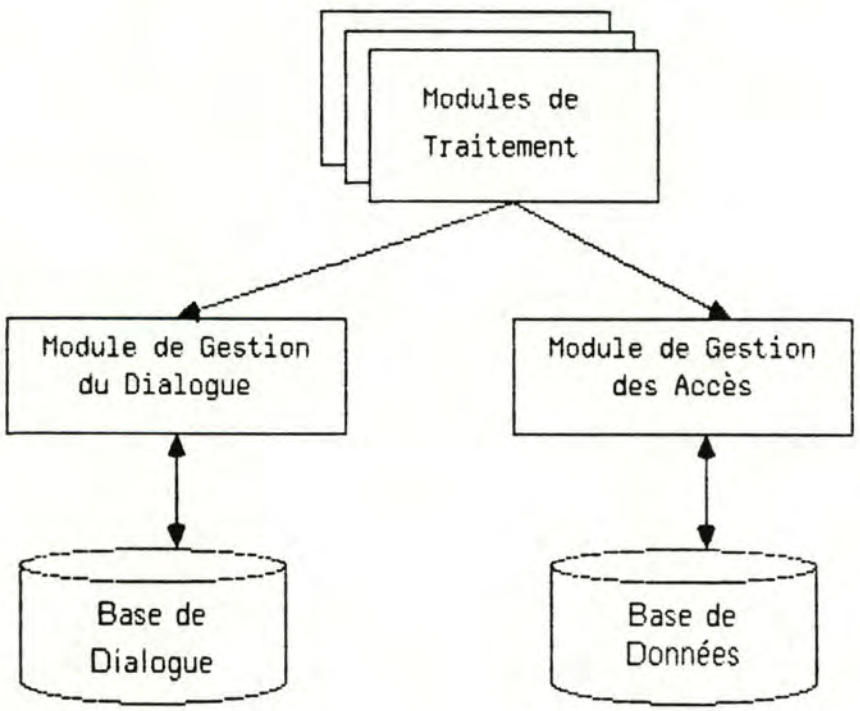


Figure III.3.2 : Architecture de Programmation

III.3.1 NIveaux 4, 5 et 6

III.3.1.1 Filière des dialogues

III.3.1.1.1 Introduction

Ce paragraphe regroupe la description des modules des niveaux 6, 5 et 4 de la filière des dialogues. Cette filière permet de décrire les modules de gestion des outils du dialogue (niveau 5) et les modules de gestion des procédures de dialogue entre l'homme et la machine. Ces modules sont décrits dans le paragraphe suivant selon l'ordre indiqué dans le schéma.

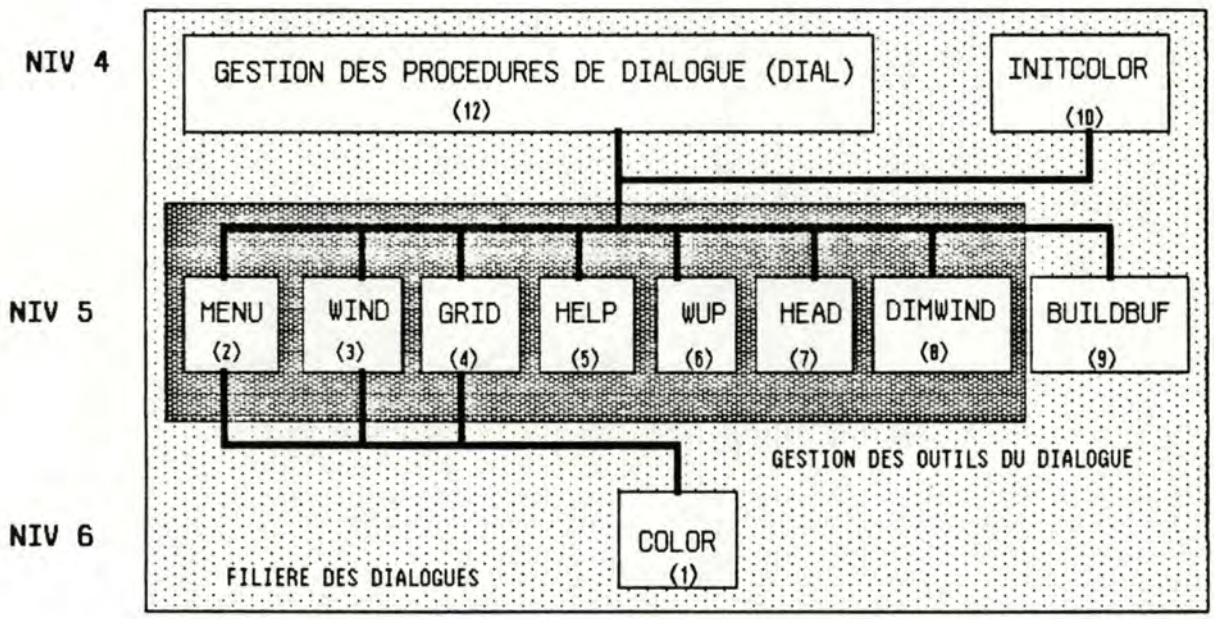


Figure 3.1.1.1 : La filière des dialogues

III.3.1.1.2 Description des modules

Gestion de la couleur [COLOR]

Niveau : 6
Fichier : "color.c"
Bibliothèque : "toollib.lib"

Description :

Le module COLOR propose deux primitives pour définir la couleur courante du curseur. Si la couleur courante du curseur est bleue, alors les caractères affichés à l'écran prendront la même couleur que celle du curseur c.a.d. la couleur bleue.
Les couleurs possibles et leur numéro correspondant sont les suivantes :

0	NORMAL	32	VERT
1	GRAS	33	JAUNE
4	SOULIGNE	34	BLEU SOMBRE
7	REVERSE	35	MAGENTA
8	INVISIBLE	36	BLEU CLAIR
30	NOIR	37	BLANC
31	ROUGE		

Initialisation d'une couleur courante

Définition :
Cette primitive permet d'initialiser une couleur courante pour le curseur.
Paramètres en entrée :
c : numéro de la couleur à initialiser
Préconditions :
c est nécessairement un des numéros spécifiés ci-dessus.
Fonction :
Cette primitive est utilisée par presque toutes les primitives du niveau supérieur. Elle ne produit ses effets que si l'ordinateur est muni d'une carte couleur ou alors si c est plus petit que 30.
Appel :

```
color(c)
int c;
```

Verification de l'existence d'une carte couleur

Définition :
Cette primitive permet de verifier si l'ordinateur est muni d'une carte couleur.
Paramètres en sortie :
testcolor(): numero d'erreur
Postconditions :
testcolor() == 1 si le test a réussit
Fonction :
Cette primitive est utilisée par la primitive de gestion des couleurs.
Appel :

```
testcolor();
```

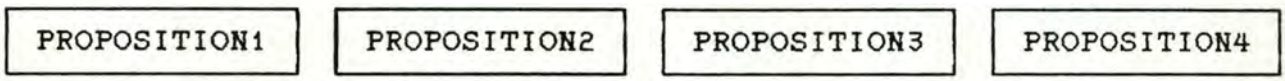
Gestion des menus [MENU]

Niveau : 5
Fichier : "menu.c"
Bibliothèque : "toollib.lib"

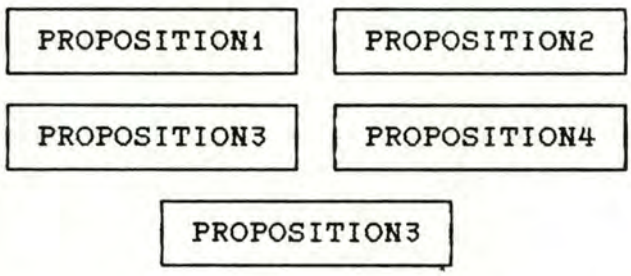
Description :

Le module MENU propose toutes les primitives pour travailler sur l'objet structuré menu. Cet objet qui est décrit dans le fichier "tool.h", représente un menu au sens traditionnel du terme composé de propositions que l'utilisateur peut choisir en fonction de ce qu'il désire voir être executé. Un menu se présente des deux manières suivantes :

MENU horizontal



MENU vertical



Chaque menu possède sa proposition activée qui indique la proposition que l'utilisateur veut choisir. Cette proposition est soit plus grasse que les autres soit clignotante (au choix de l'utilisateur, défini au moyen du module INITCOLOR). Pour faire changer la proposition activée il suffit d'utiliser les touches:

[→],[←] pour un MENU horizontal

[home],[↑],[end],[→],[←],
[pgup],[↓],[pgdn] pour un MENU vertical

Ainsi, pour choisir la PROPOSITION3 il suffit de rendre cette proposition active en la pointant au moyen des touches décrites ci-dessus ensuite de confirmer le choix en utilisant soit la touche [CR], soit la touche [SPACE].

Le MENU vertical prend aussi en considération les touches [↑], [↓], [pgdn], [pgup] mais elles seront sans effet, en tout cas pour la gestion du menu. L'utilité de ces touches est expliquée dans le module WIND.

Tous les menus sont décrits dans le fichier "menu.txt" de la manière et selon la syntaxe suivante :

...

```
BEGIN[CR]
<nom de menu>[CR]
<nombre de propositions>[CR]
<numero de colonne du debut du texte de la proposition>[CR]
<numero de ligne du debut du texte de la proposition>[CR]
<texte de la proposition>[CR]
<proposition à activer si [home]>[CR]
<proposition à activer si [←]>[CR]
<proposition à activer si [end]>[CR]
<proposition à activer si [↑]>[CR]
<proposition à activer si [↓]>[CR]
<proposition à activer si [pgup]>[CR]
<proposition à activer si [→]>[CR]
<proposition à activer si [pgdn]>[CR]
```

*)

...

** à répéter pour chaque proposition

EXEMPLE

```
BEGIN
menuexemple
2
3
23
PROPOSITION1
1
2
1
1
1
1
2
1
19
23
PROPOSITION2
2
1
2
2
2
2
1
2
```

Pour des raisons de rapidité, le fichier "menu.txt" est compilé pour produire deux nouveaux fichiers "menu.dat" et "menu.idx" qui seront utilisés par les primitives du module MENU. Il suffit pour cela de lancer le programme de compilation qui s'appelle "getmenu.exe"

Pour utiliser correctement les primitives du module MENU, il faut inclure les fichiers "tool.h" et "var.h" dans le fichier où l'on fera appel a ces primitives.

Ouverture d'un menu

Définition :
Cette primitive permet d'initialiser un menu et de l'afficher à l'ecran.
Paramètres en entrée :
name : nom du menu dans le fichier "menu.txt".
Précondition :
taille(name) < CAR_NAME (cfr "tool.h") et name est défini dans "menu.txt".
Paramètres en sortie :
m : menu initialisé
Postconditions :
m est initialisé par la description du menu correspondant à name dans "menu.txt".
Appel :

```
openmenu(name,m)
char name[];
struct MENU *m;
```

Initilisation d'un menu

Définition :
Cette primitive permet d'initialiser un menu c.a.d. de lire l'enregistrement du menu dans un fichier et de le mettre dans une variable de travail.
Paramètres en entrée :
name : nom du menu dans le fichier "menu.txt"
Précondition :
taille(name) < CAR_NAME (cfr "tool.h") et name est défini dans "menu.txt".
Paramètres en sortie :
m : menu initialisé
Postconditions :
m est initialisé par la description du menu correspondant à name dans "menu.txt".
Appel :

```
initmenu(name,m)
char name[];
struct MENU *m;
```

Affichage du menu à l'écran

Définition :
Cette primitive permet d'afficher un menu à l'écran c.a.d. de dessiner à l'écran les propositions du menu.
Paramètres en entrée :
m : menu à afficher
Préconditions :
m est correctement initialisé.
Appel :

```
dispmenu(m)
struct MENU *m;
```

Gestion du menu

Définition :

Cette primitive gère le mécanisme de passage de l'activation d'une proposition à une autre ainsi que la confirmation du choix.

Paramètres en entrée :

m	:	menu à gérer
ud	:	indique si il s'agit d'un menu vertical ou non. On le remplacera par la valeur UPDOWN (voir "tool.h") si il s'agit d'un menu vertical, la valeur NOUPDOWN sinon.
first	:	détermine la première proposition à activer

Préconditions :

m est correctement initialisé et le menu est déjà affiché à l'écran.

Paramètres en sortie et Postconditions :

mvt	:	si ud == UPDOWN mvt == PD ou PU ou LD ou LU ("tool.h") sinon mvt == 0
mgtmenu()	:	si ud == UPDOWN mgtmenu() == - (proposition courante) sinon mgtmenu() == proposition choisie

Appel :

```
mgtmenu(m, ud, first, mvt)
struct MENU *m;
int ud;
int first;
int *mvt;
```

Ecrire le texte d'une proposition

Définition :

Cette primitive permet d'écrire le texte d'une propositon à l'écran.

Paramètres en entrée :

x	:	abcisse + 1 du début du texte de la proposition sur l'écran.
y	:	ordonnée + 1 du début du texte de la proposition sur l'écran.
s	:	contenu du texte de la proposition.

Fonction :

Cette primitive est appelée par la primitive d'affichage.

Appel :

```
drawtxt(x, y, s)
int x, y;
char s[];
```

Dessiner la limte d'un proposition

Définition :

Ces deux primitives permettent de dessiner le contour d'une proposition d'un menu à l'écran. Soit il s'agit d'un contour d'une proposition activée (drawlmt) , soit d'un contour d'une proposition normale (drawlmtb).

Paramètres en entrée :

x	:	abcisse + 1 du début du texte de la propostion sur l'écran.
y	:	ordonnée + 1 du début du texte de la proposition sur l'écran.
n	:	longueur du texte de la proposition.

Fonction :

Cette primitive est appelée par les primitives d'affichage et de gestion du menu.

Appel :

```
drawlmt(x, y, n)
int x, y, n;
int n;

drawlmtb(x, y, n)
int x, y, n;
```

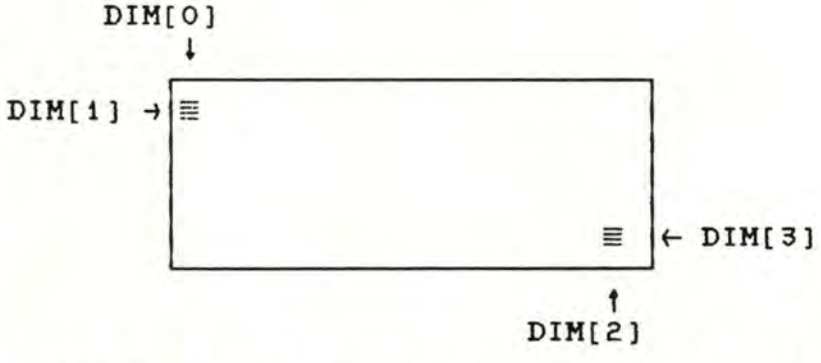
Gestion des fenêtres [WIND]

Niveau : 5
Fichier : "wind.c"
Bibliothèque : "toollib.lib"

Description :

Le module WIND propose toutes les primitives pour travailler sur l'objet structuré window. Cet objet qui est décrit dans le fichier "tool.h", représente une fenêtre qui est en fait une vue ouverte à l'écran sur une page de texte. Cette page de texte est représentée par un tableau de lignes que l'on appelera buffer. La fenêtre a nécessairement la même largeur que le buffer sur lequel elle est ouverte, mais pas nécessairement la même longueur. Du fait de cette dernière propriété, la fenêtre doit pouvoir voyager de haut en bas du buffer pour permettre à l'utilisateur de lire la totalité de son contenu.

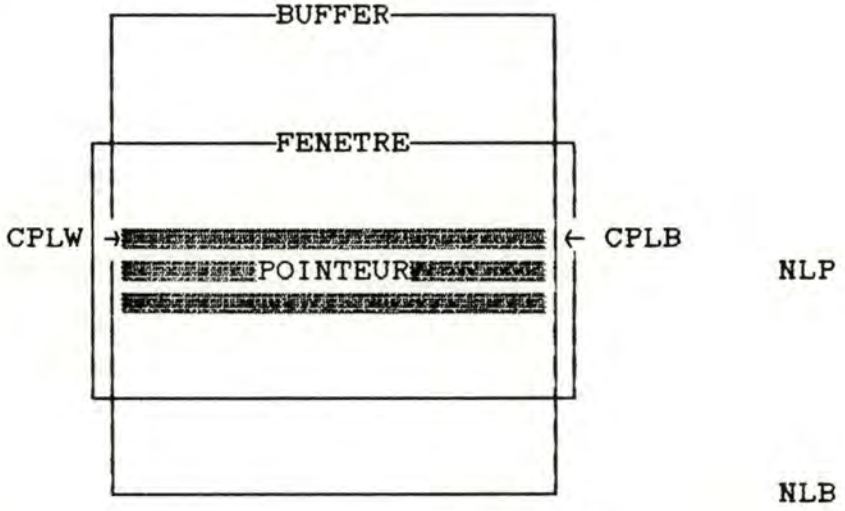
Physiquement à l'écran, la fenêtre est représentée par un cadre. A l'intérieur de ce cadre se trouvent des lignes de caractères représentant le contenu du buffer visé par la fenêtre. Lorsque l'utilisateur fait voyager la fenêtre dans le buffer, c'est en fait l'intérieur de la fenêtre qui changera puisque la fenêtre a hérité d'un emplacement fixe sur l'écran. Une fenêtre logique se définit comme ceci :



La position et la grandeur de la fenêtre se définissent au moyen d'un tableau comportant quatre positions. Les deux premières positions déterminent la position du coin supérieur gauche de la fenêtre sur l'écran (abscisse,ordonnée) tandis que les deux dernières déterminent le coin inférieur droit. Ce tableau dim fait partie de l'objet structuré window. La couleur du cadre qui entoure la fenêtre peut être définie au moyen du module INITCOLOR.

La fenêtre possède en outre un pointeur qui est représenté par une ou plusieurs lignes que l'on distingue des autres car elle(s) possède(nt) une couleur différente (ces couleurs sont définies au moyen du module INITCOLOR). Ce pointeur voyage dans la fenêtre et quand le pointeur arrive soit à l'extrémité supérieure soit à l'extrémité inférieure de la fenêtre, il permet de faire voyager la fenêtre dans le buffer. En effet, si le pointeur est à l'extrémité inférieure de la fenêtre et que l'utilisateur pousse sur [↓], la fenêtre va descendre de nlp lignes dans le buffer. Si celui-ci pousse sur [PgDn], la fenêtre descendra de longueur(fenêtre) lignes dans le buffer. La méthode est la même dans l'autre sens.

Pour pouvoir gérer le mouvement du pointeur dans la fenêtre il a été nécessaire de définir d'autres attributs à l'objet structuré window :



CPLW : (Current Pointer Line in Window) c.a.d. le numéro dans la fenêtre de la première ligne du pointeur.

CPLB : (Current Pointer Line in Buffer) c.a.d. le numéro dans le buffer de la ligne qui correspond à la première ligne du pointeur dans la fenêtre.

NLB : (Number of Lines in Buffer) c.a.d. le nombre de lignes que le buffer contient.

NLP : (Number of Lines for Pointer) c.a.d. le nombre de lignes qui composent le pointeur.

Pour faire voyager le pointeur dans la fenêtre, l'utilisateur peut utiliser les touches suivantes [↑], [↓], [PgDn], [PgUp]. (cfr la primitive "scrollwind")

Le module WIND offre aussi des primitives qui permettent à l'utilisateur de marquer une ligne dans le buffer.

Pour utiliser correctement les primitives du module WIND, il

faut inclure les fichiers "tool.h" et "var.h" dans le fichier où l'on fera appel a ces primitives.

Certaines fenêtres sont déjà définies (cfr module DIMWIND).

Ouvertures d'une fenêtre

Définition :
Ces primitives permettent d'ouvrir une fenêtre à l'écran et d'initialiser son contenu avec le début (openwind) du buffer que l'on désire afficher ou la partie courante (reopenwind) de ce buffer (c.a.d. la partie pointée par cplb).
Paramètres en entrée :
wind : fenêtre à ouvrir
buf : buffer à parcourir au moyen de la fenêtre
Préconditions :
buf est un tableau de lignes et longueur(ligne) < MAXCBWIND et wind doit être complètement initialisé.
Paramètres en sortie :
wind : fenêtre ouverte
Postconditions :
La primitive openwind initialise cplb et cplw à 0 tandis que la primitive reopenwind laisse ces deux variables à leur valeur initiale.
Appel :

```
(re)openwind(wind,buf)
struct WIND *wind;
char (*buf)[MAXCBWIND]
```

Fermetures d'une fenêtre

Définition :
Ces primitives permettent de fermer une fenêtre c.a.d. d'effacer à l'écran le contenu (clearwind) ou le contenu et le cadre d'une fenêtre (closewind).
Paramètres en entrée :
wind : la fenêtre à fermer
Préconditions :
La fenêtre doit déjà être ouverte et les dimensions de wind doivent être initialisées.
Appel :

```
clearwind(wind)
struct WINDOW *wind;

closewind(wind)
struct WINDOW *wind;
```

Dessin du cadre d'une fenêtre

Définition :
Cette primitive permet de dessiner à l'écran un cadre autour d'une fenêtre.

Paramètres en entrée :
wind : la fenêtre à encadrer
Préconditions :
Les dimensions de wind sont initialisées. La fenêtre ne possède pas de limites communes avec l'écran.
Fonction :
Cette primitive est utilisée par les primitives d'ouverture de la fenêtre.
Appel :

```
drawwind(wind)
struct WINDOW *wind;
```

Initialisation de la fenêtre

Définition :
Cette procédure permet d'initialiser le contenu de la fenêtre avec le contenu du buffer visé par cette fenêtre.
Paramètres en entrée :
wind : fenêtre à initialiser
buf : buffer visé par la fenêtre
Préconditions :
buf est un tableau de lignes et longueur(ligne) < MAXCBWIND et wind est correctement initialisé.
Fonction :
Cette primitive est utilisée par les primitives d'ouverture de la fenêtre.
Appel :

```
initwind(wind, buf)
struct WINDOW *wind;
char (*buf)[MAXCBWIND];
```

Gestion du déplacement du pointeur et de la fenêtre

Définition :
Cette primitive permet à l'utilisateur de déplacer le pointeur dans la fenêtre et par conséquent la fenêtre dans le buffer.
Paramètres en entrée :
wind : fenêtre à gérer
mvt : mouvement demandé par l'utilisateur c.a.d. LU, LUE, PU, PUE, LD, LDE, PD, PDE (voir "tool.h")
buf : buffer visé par la fenêtre
Préconditions :
buf est un tableau de lignes et longueur(ligne) < MAXCBWIND et wind est correctement initialisé.
Paramètres en sortie et Postconditions :
wind : fenêtre modifiée (cplb, cplw)
scrollwind(): si le mouvement du pointeur est impossible c.a.d. la fenêtre dépasse les limites du buffer.
alors == LUE si mvt est LU
== LDE si mvt est LD
== PUE si mvt est PU
== PDE si mvt est PD
sinon == 0
Si le mouvement est impossible alors le pointeur est désactivé.

Fonction :

Cette procédure permet à l'utilisateur de gérer lui-même les limites du parcours de la fenêtre dans le buffer. En effet, il peut s'agir d'un buffer alimenté par une liste. Dans ce cas, il faut laisser à l'utilisateur le soin de gérer lui-même la fin du buffer car cette fin n'est pas nécessairement la fin de la liste et donc il lui faut réinitialiser le buffer avec la suite de la liste.
Si ces limites (début et fin du buffer) ne sont pas atteintes la gestion du mouvement du pointeur et de la fenêtre s'effectuent correctement.

Pour obtenir correctement le mouvement **mvt** que l'utilisateur désire on peut employer un menu (cfr le module MENU). En effet lorsque l'on gère un menu vertical, le programme prend en compte les touches [↑], [↓], [PgDn], [PgUp] et renvoit, alors, en sortie LU, LD, PD, PU au moyen du troisième paramètre de mgtmenu. Voici alors comment structurer son programme.

```
while(1)
switch(x = mgtmenu(&menu, UPDOWN, &mvt)
{
  case 1 : ....

  ...                  /*  gestion des propositions du menu */


  default: switch(scrollwind(&wind, mvt, buf))
           {
            case LDE :
            case PDE :              /* gestion des limites */
            case LUE :
            case PUE :
            default  :
           }
           x = -x;
           break;
}
```

Avec cet algorithme, on peut utilser à la fois un menu et une fenêtre sans le moindre problème.

Appel :

```
mgtwind(wind, mvt, buf)
struct WINDOW *wind;
int mvt;
char (*buf)[MAXCBWIND];
```

Gestion du pointeur aux limites du buffer

Définition :

Cette primitive permet à l'utilisateur de gérer la fenêtre lorsque celle-ci atteind les limites du buffer.

Paramètres en entrée :

wind	:	fenêtre à gérer
mvt	:	mouvement litigieux envoyé par scrollwind c. a. d. LUE, PUE, LDE, PDE (voir "tool.h")
buf	:	buffer visé par la fenêtre

Préconditions :

buf est un tableau de lignes et longueur(ligne) < MAXCBWIND et wind est correctement initialisé.

Paramètres en sortie :
wind : fenêtre modifiée
Postconditions :
Si mvt est différent de LUE, LDE, PDE, PUE la primitive est inopérante sinon cplb et cplw ont été réajustés.
Fonction :
Cette primitive offre une solution à l'utilisateur pour gérer les limites du déplacement de la fenêtre. Il faut l'utiliser juste après scrollwind de cette manière :

```
while(1)
switch(x = mgtmenu(&menu, UPDOWN, &mvt)
{
  case 1 : ....

  ...                    /*  gestion des propositions du menu */


  default: mvt = scrollwind(&wind, mvt, buf))
           mgtwend(&wind, mvt, buf);
           x = -x;
           break;
}
```

Appel :

```
mgtwend(wind, mvt, buf)
struct WINDOW *wind;
int mvt;
char (*buf)[MAXCBWIND];
```

Pointer des lignes du buffer

Définition :
Ces primitives permettent de marquer nlp lignes dans le buffer (pointwind) au moyen de la fenêtre ou d'effacer cette marque (nopointwind).
Paramètres en entrée :
wind : fenêtre à utiliser
buf : buffer visé par la fenêtre
Préconditions :
buf est un tableau de lignes et longueur(ligne) < MAXCBWIND et wind est correctement initialisé.
Paramètres en sortie et Postconditions :
buf : buf[i][0] 0 <= i <= nlb modifié c.a.d. initialisé avec les valeurs soit [ ] soit [▶] soit [*] soit [@].
Fonction :
Permet de marquer des lignes dans le buffer. Grâce à ce système on peut choisir des lignes du buffer au moyen de la fenêtre. Il existe un double marquage. En effet buf[i][0] peut valoir [▶], [*], [@] ou [ ], pointwind le marquera de [▶] si il vaut [ ], si il est déjà marqué par [*] alors pointwind le marquera [@], si il déjà marqué par [▶] ou par [@] alors pointwind sera inopérant. Nopointwind rendra buf[i][0] == [ ] si il valait [▶] ou [ ], == [*] si il valait [@]. Ces marques sont invisibles dans la fenêtre sauf cas précis (cfr windptr).
Appel :

```
(no)pointwind(wind, buf)
struct WINDOW *wind;
char (*buf)[MAXCBWIND];
```

Ecriture d'une ligne

Définition :

Ces primitives permettent d'écrire des lignes distinctives. La distinction est la suivante : ligne de texte dans la fenêtre (windtxt), ligne de pointeur dans la fenêtre (windptr), ligne de contour de la fenêtre (windlmt). La distinction est matérialisée par leur couleur respective (cfr INITCOLOR).

Paramètres en entrée :

line : la ligne à écrire

Préconditions :

La ligne ne doit pas excéder la grandeur de la fenêtre où elle est écrite (windtxt, windptr).

Fonction :

La primitive wintxt :

```
si line[0] == [*]     line est souligné
si line[0] == [@]     line est souligné et a la couleur
                      tabcolor[5]
si line[0] == [▶]     line a la couleur tabcolor[5]
si line[0] == [ ]     line a la couleur tabcolor[4]
```

line[0] vaut à l'impression [ ]

La primitive windptr :

```
si line[0] == [*]     line est souligné
si line[0] == [@]     line est souligné
```

La couleur de line vaut en tous cas à l'impression tabcolor[5].

line[0] vaut à l'impression [▶] si line[0] == [▶] ou [@].
line[0] vaut à l'impression [ ] sinon.

La primitive windlmt :

La couleur de line vaut en tous cas à l'impression tabcolor[3].

Appel :

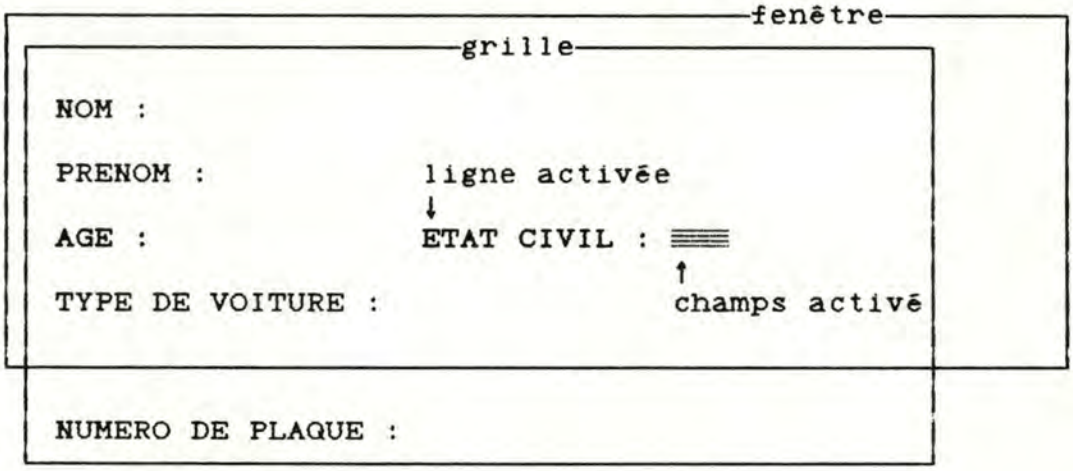
```
windtxt(line)
char line[];

windptr(line)
char line[];

windlmt(line)
char line[];
```

Gestion des grilles [GRID]

Niveau : 5
Fichier : "grid.c"
Bibliothèque : "toollib.lib"

Description :

Le module GRID propose toutes les primitives pour travailler sur l'objet structuré grid. Cet objet qui est décrit dans le fichier "tool.h", représente une grille qui est composée d'un fond (ou "layout") qui est un ensemble de lignes de caractères et d' un ensemble de champs répartis dans cette grille, accessibles en écriture et en lecture par l'utilisateur. Une grille en utilisation possède une ligne activée et dans cette ligne, un champs activé. Ne peuvent être activées que les lignes qui ont un champs décrit sur elles. Une grille doit être définie dans une fenêtre (cfr module WIND). Le coin supérieur gauche du fond de la grille coincidera avec le coin supérieur gauche de la fenêtre définie pour elle. Le fond de la grille joue le même rôle que le buffer pour la fenêtre. Le fond peut donc être plus long que la fenêtre qui le vise. Attention la gestion du déplacement de la fenêtre sur le fond est assuré ici par la primitive de gestion de la grille (mgtgrid).



Pour changer de ligne activée il suffit de pousser sur les touches [↑] et [↓] et ainsi faire bouger la ligne activée vers le haut ou vers le bas. Dans la ligne activée, pour changer de champs activé, il suffit d'utiliser la touche [CR]. Pour remplir un champs, il suffit de l'activer et de remplir celui-ci au moyen du clavier. Une fois sur un champs activé les touches [→] et [←] permettent de voyager dans ce champs. La touche [CR] permet aussi de parcourir tous les champs de la grille l'un après l'autre. Enfin, pour sortir de la grille il suffit de pousser sur la touche [Home].

Toutes les grilles sont décrites dans le fichier "grid.txt" de la manière et selon la syntaxe suivante :

```
BEGIN DESIGN

       Dessin du fond de la grille

END DESIGN

BEGIN DESCRIPTION
<nom de la grille>[CR]
<nombre de champs>[CR]
                                                  ligne de commentaire
<numéro de ligne du champs>[CR]
<numéro de colonne du début du champs>[CR]
<ligne à activer si [↑]>[CR]
<ligne à activer si [↓]>[CR]
<champs suivant sur la ligne>[CR]                                    **
<champs précedant sur la ligne>[CR]
<type du champs>[CR]
<longueur du champs>[CR]

END DESCRIPTION

              ** à répéter pour chaque champs
```

EXEMPLE

```
BEGIN DESIGN
NOM :                    PRENOM :
AGE :
ETAT CIVIL :
END DESIGN

BEGIN DESCRIPTION
test
3
         ceci est la description du champs nom
1
6
1
2
2
1
3
10
        ceci est la description du champs prénom
1
33
1
2
2
1
3
10
        ceci est la description du champs age
2
6
1
3
3
3
2
```

```
3
        ceci est la description du champs etat civil
3
14
2
3
4
4
3
3
END DESCRIPTION
```

Remarques :

(1) Lorsqu'un champs est seul sur une ligne, son précédent et son suivant sur la ligne n'existent pas, alors on met son propre numéro. Il en est de même avec la première et la dernière ligne. Ces lignes n'ont pas de ligne précedente ou suivante.

(2) Il faut mettre des blancs à la place des champs dans le fond de la grille, sinon il faut s'attendre à des problèmes.

(3) Pour le type du champs cfr "tool.h"

Pour des raisons de rapidité, le fichier "grid.txt" est compilé pour produire deux nouveaux fichiers "grid.dat" et "grid.idx" qui seront utilisés par les primitives du module GRID. Il suffit pour cela de lancer le programme de compilation qui s'appelle "getgrid.exe"

Pour utiliser correctement les primitives du module GRID, il faut inclure les fichiers "tool.h" et "var.h" dans le fichier où l'on fera appel a ces primitives.

Initialisation d'une grille

Définition :
Cette primitive permet d'initialiser une grille c.a.d. de lire l'enregistrement de la grille dans un fichier et de le mettre dans une variable de travail.

Paramètres en entrée :
name : nom de la grille dans le fichier "grid.txt"

Préconditions :
taille(name) < CAR_NAME (cfr "tool.h") et name est décrit dans le fichier "grid.txt".

Paramètres en sortie :
grid : grille initialisée

Postconditions :
grid est initialisé par la description de la grille correspondant à name dans "grid.txt".

Appel :

```
initgrid(name, grid)
char name[];
struct GRID *grid;
```

Affichage d'une grille à l'écran

Définition :
Cette primitive permet de dessiner à l'écran une grille dans une fenêtre c.a.d. d'afficher cette fenêtre et d'initialiser son contenu au moyen d'un buffer qui est le "layout" de la grille.

Paramètres en entrée :
grid : grille à afficher
wind : fenêtre visant cette grille

Préconditions :
La largeur de la grille est inférieure à celle de la fenêtre, grid et wind sont correctement initialisés.

Paramètres en sortie :
wind : fenêtre ouverte

Postconditions :
cplb == 0 et cplw est égale au numéro de la ligne qui contient le premier champs.

Fonction :
Cette primitive fait les appels suivants drawwind(wind) et initwind(wind, grid.cont).

Appel :
```
drawgrid(grid, wind)
struct GRID *grid;
struct WINDOW *wind;
```

Lecture ou écriture d'un champs de la grille

Définition :
Ces primitives permettent à l'utilisateur de lire un champs de la grille (readfield) ou de donner une valeur à celui-ci (writefield).

Paramètres en entrée :
numfield : numéro du champs à lire ou à écrire c.a.d le numéro d'ordre de sa définition dans "grid.txt".
grid : La grille dans lequel se trouve le champs.
line : La variable permettant d'initialiser le champs (writefield).

Préconditions :
numfield <= grid.nfield et grid est correctement initialisé.

Paramètres en sortie :
line : La variable à initialiser par la valeur du champs (readfield).

Postconditions :
line est copie conforme du champs de numéro numfield.

Appel :
```
(write)readfield(numfield, grid, line)
int numfield;
struct GRID *grid;
char line[];
```

Activation ou désactivation d'un champs

Définition :
Ces primitives permettent d'activer (setcuron) ou de désactiver un champs dans une grille.

Paramètres en entrée :

field	:	le champs à activer ou à désactiver.
grid	:	la grille à laquelle appartient le champs.
wind	:	la fenêtre visant la grille.

Préconditions :

field, grid et wind sont correctement initialisés.

Fonction :

Ces primitives sont appelées par la primitive de gestion de la grille.

Appel :

```
setcuron(field, grid, wind)
struct FIELD *field;
struct GRID *grid;
struct WINDOW *wind;

setcurof(field, grid, wind)
struct FIELD *field;
struct GRID *grid;
struct WINDOW *wind;
```

Analyse d'un caractère d'un champs

Définition :

Cette primitive permet d'analyser un caractère d'un champs c.a.d. voir si ce carcactère appartient ou non à un ensemble de caractères déterminé par le type du champs.

Paramètres en entrée :

c	:	le caractère en question
field	:	le champs auquel appartient ce caractère
curcar	:	le numéro du caractère courant dans le champs

Préconditions :

field est correctement initialisé et 0 <=curcar <=field.lgf

Paramètres en sortie et Postconditions :

numerror	:	le numéro du message d'erreur à afficher
analyse()	:	si c satisfait au type de field alors analyse()==0 sinon analyse()==-1 et numerror indique le message d'erreur correspondant.

Fonction :

Cette primitive est appelée par la primitive de gestion de la grille. Elle permet de gérer les erreurs de syntaxe lors de cette gestion. Les types des champs sont définis dans le fichier "tool.h". L'utilisateur peut en définir de nouveaux, mais il faut pour cela changer le contenu de cette primitive. Pour les messages d'erreur cfr le module ERROR.

Appel :

```
analyse(c, numerror, field, curcar)
char c;
char numerror[];
struct FIELD *field;
int curcar;
```

Ecriture d'un caractère dans un champs

Définition :

Cette primitive permet d'écrire un caractère dans un champs.

Paramètres en entrée :

c	:	le caractère à écrire
curcar	:	le numéro du caractère courant dans le champs
field	:	le champs auquel appartient ce caractère
grid	:	la grille à laquelle appartient ce champs

Préconditions :
field et grid sont correctement initialisés et 0 <= curcar <= field.lgf
Paramètres en sortie et Postconditions :
grid : le fond de la grille modifiée par l'inscription du caractère à l'endroit désigné par field et par curcar.
writecar() : le nouveau numéro du caractère courant dans le champs
Fonction :
Cette primitive est appelée par la primitive de gestion de la grille. Elle permet d'inscrire dans la grille le caractère qui a déjà été inscrit à l'écran.
Appel :

```
writecar(c, curcar, grid, field)
char c;
int curcar;
struct GRID *grid;
struct FIELD *field;
```

Gestion de la grille

Définition :
Cette primitive permet de gérer une grille c.a.d. gérer la ligne et le champs activé, gérer les entrées et gérer les erreurs concernant ces entrées.
Paramètres en entrée :
grid : la grille à gérer
wind : la fenêtre visant cette grille
first : le premier champs à gérer
Préconditions :
grid et wind sont correctement initialisés et first <= grid.nfield. La grille est déjà affichée à l'écran.
Paramètres en sortie et Postconditions :
first : le dernier champs à avoir été géré.
Appel :

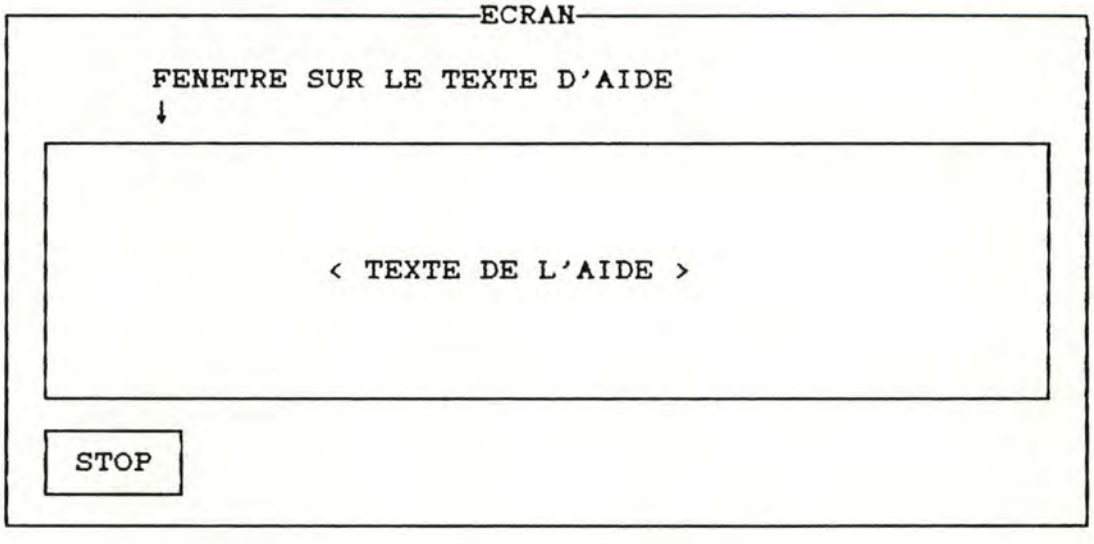
```
mgtgrid(grid, wind, first)
struct GRID *grid;
struct WIND *wind;
int *first;
```

Gestion de l'aide à l'utilisateur [HELP]

Niveau : 5
Fichier : "help.c"
Bibliothèque : "toollib.lib"

Description :

Le module HELP propose toutes les primitives pour travailler sur l'objet structuré help. Cet objet qui est décrit dans le fichier "tool.h", représente un ensemble de lignes constituant un texte. Ce texte est destiné à informer l'utilisateur sur la démarche à suivre à tout moment. Il est présenté à l'écran au moyen d'une fenêtre prédéfinie ("windhelp" cfr module DIMWIND) et est géré au moyen d'un menu particulier ("menuhelp" cfr fichier "menu.txt"). A l'écran cela se présente de la manière suivante :



Pour obtenir ce résultat, le concepteur du programme doit utiliser les primitives qui sont décrites ci-dessous.

Les textes qui constituent l'aide à l'utilisateur sont décrits dans le fichier "help.txt" selon la manière et la syntaxe suivante :

```
...
BEGIN[CR]
<nom du texte d'aide>[CR]
...
<ligne de texte>[CR]          ] contenu du texte
...
END[CR]
...
```

EXEMPLE

```
BEGIN
aidetest

        Ceci est un texte qui sert
        d'exemple pour le module HELP

END
```

Remarques :

(1) Le nombre maximum de lignes du texte est déterminé par LINE_HELP (cfr "tool.h") et le nombre maximun de caractères par ligne est déterminé par MAXCBHELP.

(2) Actuellement, LINE_HELP ne peut excéder le nombre de lignes disponibles dans la fenêtre mais des aménagements sont possibles pour permettre d'avoir un texte plus long que la fenêtre.
MAXCBHELP ne peut en tous cas pas dépasser la largeur de la fenêtre.

(3) Pour connaître LINE_HELP, calculer windhelp.dim[3] - windhelp.dim[1] + 1.

Pour des raisons de rapidité, le fichier "help.txt" est compilé pour produire deux nouveaux fichiers "help.dat" et "help.idx" qui seront utilisés par les primitives du module HELP. Il suffit pour cela de lancer le programme de compilation qui s'appelle "gethelp.exe"

Pour utiliser correctement les primitives du module HELP, il faut inclure les fichiers "tool.h" et "var.h" dans le fichier où l'on fera appel a ces primitives.

Gestion d'une demande d'aide

Définition :
Cette primitive permet de répondre entièrement à une demande d'aide c.a.d. qu'elle initialise le texte d'aide demandé et ensuite gère la fenêtre et le menu de l'aide.

Paramètres en entrée :
name : nom du texte d'aide à charger et à gérer

Préconditions :
taille(name) < CAR_NAME (cfr "tool.h") et name doit être définit dans le fichier "help.txt".

Fonction :
Cette primitive est la seule à être utilisée par un niveau supérieur sauf si le concepteur veut une gestion différente de l'aide.

Appel :

```
helpme(name)
char name[];
```

Initialisation d'un texte d'aide

Définition :
Cette primitive permet d'initialiser un texte d'aide c.a.d. de lire l'enregistrement de ce texte dans un fichier et de le mettre dans une variable de travail.
Paramètres en entrée :
name : nom du texte à lire
Préconditions :
taille(name) < CAR_NAME (cfr "tool.h") et name doit être définit dans le fichier "help.txt".
Paramètres en sortie :
help : texte d'aide initialisé
Postconditions :
help est initialisé par la description du texte d'aide correspondant à name dans "help.txt".
Fonction :
Cette primitive est utilisée par la primitive de gestion d'une demande d'aide.
Appel :

```
inithelp(name,help)
char name[];
struct HELP *help;
```

Gestion du mécanisme d'aide

Définition :
Cette primitive permet de gérer le mécanisme d'aide c.a.d. de gérer la fenêtre contenant le texte d'aide et le menu associé.
Paramètres en entrée :
help : texte d'aide à gérer
wind : fenêtre visant ce texte
Préconditions :
help et wind sont correctement initialisés et wind == "windhelp" (cfr dimwind).
Fonction :
Cette primitive est utilisée par la primitive de gestion d'une demande d'aide avec la fenêtre prédéfinie "windhelp". Il est cependant possible de l'utiliser séparément avec une autre fenêtre.
Appel :

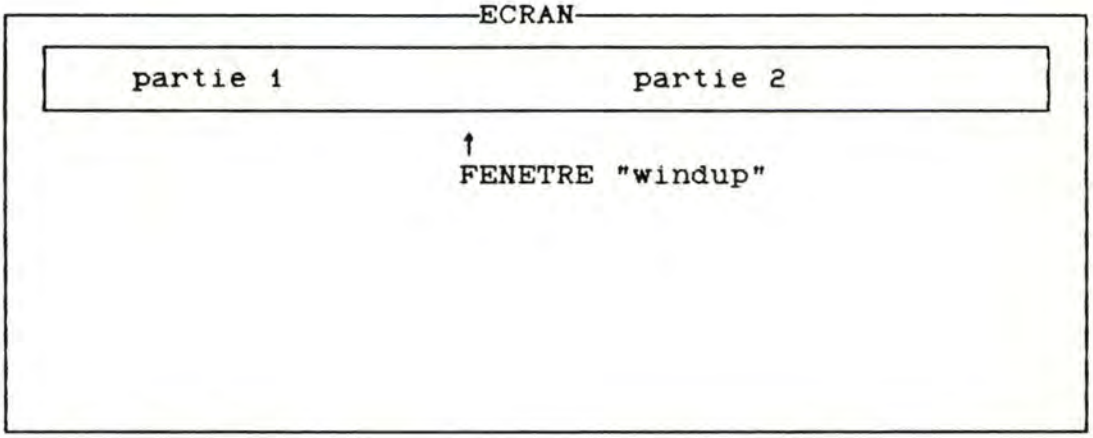
```
mgthelp(help,wind)
struct HELP *help;
struct WINDOW *wind;
```

Gestion de la fenêtre "windup" [WUP]

Niveau : 5
Fichier : "wup.c"
Bibliothèque : "toollib.lib"

Description :

Le module WUP propose deux primitives pour permettre au concepteur d'une procédure de dialogue d'afficher des messages d'information au moyen de la fenêtre prédéfinie "windup" (cfr module DIMWIND). Une de ces primitives permet d'ouvrir cette fenêtre, l'autre permet d'afficher un message composé de deux parties. Cela se présente à l'écran comme ceci :



Pour utiliser correctement les primitives du module WUP, il faut inclure les fichiers "tool.h" et "var.h" dans le fichier où l'on fera appel a ces primitives. Il faut aussi exécuter la primitive "dimwind" du module DIMWIND avant d'exécuter ces primitives.

Ouverture de la fenêtre "windup"

Définition :
Cette primitive permet d'ouvrir la fenêtre "windup".
Fonction :
Il s'agit d'une primitive sans paramètre local. Elle utilise une fenêtre "windup" et un buffer "bufwup" définis comme paramètres globaux. Cela permet une économie de variable.
Appel :
openwup()

Affichage d'un message en deux partie

Définition :
Cette primitive permet l'affichage d'un message qui est composé de deux "strings".

Paramètres en entrée :

ini : premier "string"
in2 : deuxième "string"
lgth : longueur maximum du deuxième "string"

Préconditions :

lgth < S_WIDTH - 4 (cfr "tool.h")

Fonction :

Si taille(ini) > (S_WIDTH - lgth - 4) alors ini est tronqué, in2 commence à l'emplacement S_WIDTH - lgth - 1 de la fenêtre.

Appel :

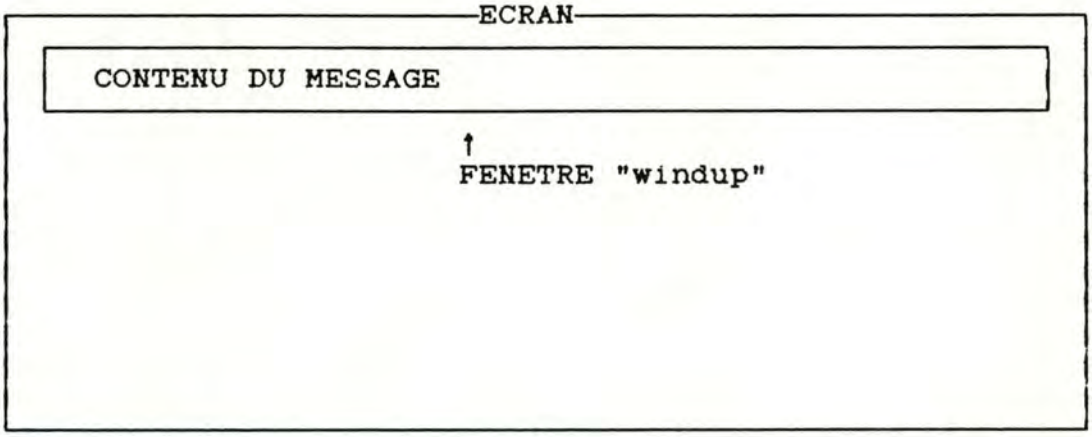
```
fillwup(ini, in2, lgth)
char ini[];
char in2[];
int lgth;
```

Gestion des messages d'information [HEAD]

Niveau : 5
Fichier : "head.c"
Bibliothèque : "toollib.lib"

Description :

Le module WUP propose une primitive pour initialiser le contenu de l'objet structuré head. Cet objet qui est décrit dans le fichier "tool.h", représente un message d'information ou de guidance constitué d'une ligne de caractères. Ce message est destiné à informer l'utilisateur de l'endroit où il se trouve dans le programme et quelle est la fonction courante ou l'action qu'il effectue. Ce message est présenté à l'écran au moyen d'une fenêtre prédéfinie ("windup" cfr module DIMWIND) et il se présente de la manière suivante :



Le contenu des différents messages d'information est décrit dans le fichier "head.txt" selon la manière et la syntaxe suivante :

```
...
<nom du message>[CR]
[space]<contenu du message>[CR]
...
```

EXEMPLE

```
...
messagetest
 Ceci est le premier message d'information
messagebis
 Deuxième message d'information
...
```

Remarques :

(1) Le nombre maximum de caractères d'un message d'information est déterminé par MAXCBWUP (cfr "tool.h") qui dépend lui-même de la taille de la fenêtre "windup".

Pour des raisons de rapidité, le fichier "head.txt" est compilé pour produire deux nouveaux fichiers "head.dat" et "head.idx" qui seront utilisés par les primitives du module HEAD. Il suffit pour cela de lancer le programme de compilation qui s'appelle "gethead.exe"

Pour utiliser correctement la primitives du module HEAD, il faut inclure les fichiers "tool.h" et "var.h" dans le fichier où l'on fera appel a cette primitive. Il faut aussi exécuter la primitive "dimwind" du module DIMWIND avant d'exécuter ces primitives.

Initialisation d'un message

Définition :
Cette primitive permet d'initialiser un message c.a.d. de lire l'enregistrement de ce message dans un fichier et de le mettre dans un variable de travail.

Paramètres en entrée :
name : nom du message dans le fichier "head.txt"

Préconditions :
taille(name) < CAR_NAME (cfr "tool.h") et name est décrit dans le fichier "head.txt".

Paramètres en sortie :
inithead() : message initialisé

Postconditions :
inithead() est initialisé par le contenu du message correspondant au nom name dans le fichier "head.txt"

Fonction :
Cette primitive est souvent utilisée de pair avec la primitive "fillwup" du module WUP de la manière suivante :

```
fillwup(inithead(...),...,...);
```

Appel :

```
inithead(name)
char name[];
```

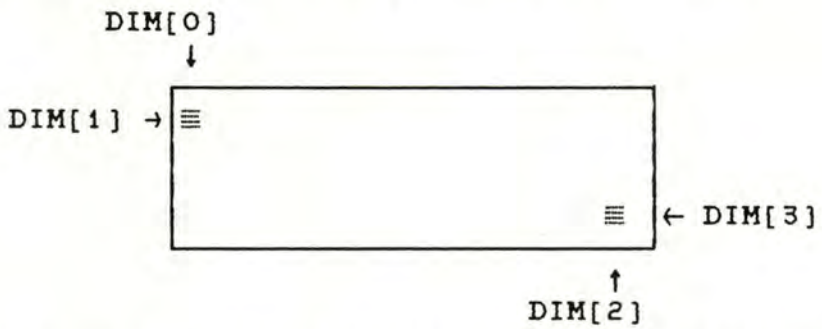
Initialisation des fenêtres prédéfinies [DIMWIND]

Niveau : 5
Fichier : "dimwind.c"
Bibliothèque : "toollib.lib"

Description :

Le module DIMWIND propose une primitive pour définir des fenêtres dont l'emplacement et la taille sont déjà prédéfinis. Il s'agit de quatre fenêtres appelées **windup**, **windmid**, **winddown** et **windhelp**. Ces fenêtres sont souvent utilisées par les modules qui gèrent les outils du dialogue, c'est pourquoi ce module offre la possibilité au concepteur des procédures de dialogue de les créer automatiquement au moyen d'une primitive. Ces fenêtres sont utilisées en tant que paramètres globaux pour l'ensemble des primitives de la bibliothèque "toollib.lib".

Les dimensions des fenêtres sont les suivantes :



FENETRE	DIM[0]	DIM[1]	DIM[2]	DIM[3]	NLB	NLP
windup	1	1	78	1	1	0
windmid	0	3	79	21		
winddown	0	22	79	24		
windhelp	1	6	78	20	*	0

* == LINE_HELP (cfr "tool.h")

Pour utiliser correctement la primitive du module DIMWIND, il faut inclure les fichiers "tool.h" et "var.h" dans le fichier où l'on fera appel a cette primitive.

Initialisation des fenêtres prédéfinies

Définition :
Cette primitive initialise les fenêtres prédéfinies.
Fonction :
Cette primitive est utilisée en premier lieu avant l'appel aux autres primitives de la bibliothèque "toollib.lib" et ceci dans le programme qui fait appel à ces primitives.
Appel :

```
dimwind();
```

Création des buffers pour la gestion du dialogue [BUILBUF]

Niveau : 5
Fichier : "buildbuf.c"
Bibliothèque : "toollib.lib"

Description :

Le module BUILDBUF propose des primitives pour définir les buffers qui seront visés par les fenêtres des primitives du module de gestion du dialogue. En effet les buffers ont une taille que l'on doit déterminer à l'avance, tandis que les listes comportant les données que l'on veut afficher au moyen des fenêtres ont une taille que l'on ne peut prévoir à l'avance. Par conséquent il est possible que cette taille dépasse celle du buffer associé à la fenêtre. Il faut donc s'y reprendre à plusieurs fois pour afficher l'entieretê de des données. Et donc, il faut réinitialiser plusieurs fois le buffer avec des parties différentes de la liste. C'est pourquoi ce module existe. De plus, les données de ces listes doivent être présentées à l'écran dans un agencement bien précis. Cet agencement doit donc être executé lors de de la construction du buffer. Les primitives du module BUILDBUF permettent de définir cet agencement.

Ces primitives de construction des buffers sont exclusivement utilisées par les primitives du module de gestion des procédures du dialogue. Il s'agit par conséquent de primitives techniques dont la description n'éclairera en rien le fonctionnement de l'interface. L'utilisation de ces primitives sera expliquée dans le module de gestion du dialogue.

Initialisation de la couleur pour les outils du dialogue [INITCOLOR]

Niveau : 4
Fichier : "initcolor.c"
Bibliothèque : "toollib.lib"

Description :

Le module INITCOLOR propose une primitive pour définir les couleurs des différents outils du dialogue. Grâce à cette primitive, l'utilisateur de ces outils peut lui-même définir leurs couleurs. Les couleurs des outils sont mémorisées dans un tableau d'entiers à sept positions portant le nom tabcolor. Ce tableau est utilisé comme variable externe pour cette primitive et toutes les primitives qui gèrent les outils du dialogue. Voici la description de ce tableau :

INDICE	DESCRIPTION
0	Couleur des caractères qui délimitent les propositions d'un menu.
1	Couleur des caractères qui composent le texte des propositions d'un menu.
2	Couleur des caractères qui délimitent la proposition activée d'un menu.
3	Couleur des caractères qui délimitent une fenêtre.
4	Couleur des caractères qui composent le texte à l'intérieur d'une fenêtre.
5	Couleur des caractères qui composent le pointeur dans le texte de la fenêtre.
6	Couleur des caractères qui composent le texte d'une grille.

Quand au choix des couleurs en lui-même, il est limité par un ensemble de couleurs disponibles pour les outils. Ces ensembles ne sont pas nécessairement identiques pour chaque outils. Ces ensembles sont déterminés par la primitive qui suit.

Pour utiliser correctement la primitive du module INITCOLOR, il faut inclure les fichiers "tool.h" et "var.h" dans le fichier où l'on fera appel a cette primitive.

Initialisation des couleurs des outils

Définition :
Cette primitive gère le dialogue avec l'utilisateur pour lui permettre de définir les couleurs de ces outils.

Fonction :
Cette primitive est généralement utilisée avant l'appel des primitives qui permettent de se servir des outils du dialogue.

Appel :

```
initcolor();
```

Gestion des procédures du dialogue [DIAL]

Niveau : 4
Fichiers : "<nom de primitive>.c"
Bibliothèque : "diallib.lib"

Description :

Le module DIAL propose des primitives pour définir les procédures de dialogue entre l'homme et le programme. Ces procédures permettent à l'utilisateur de donner et de recevoir les informations nécessaires à la bonne éxecution des fonctions de l'interface. Ces primitives travaillent avec les outils du dialogue (menu , fenêtres, grilles) en utilisant les primitives des modules qui permettent de gérer ces outils. Ces primitives appelent aussi des primitives permettant de vérifier les données qui sont introduites par l'utilisateur (module vérification).

Le type des paramètres des primitives de ce module est décrit dans le fichier "bsr.h" qu'il faut évidemment inclure au début du fichier où l'on appelle ces primitives. Il faut également y inclure les fichiers "tool.h" et "var.h" (cfr les modules de gestion des outils du dialogue).

Remarques :

(1) Certaines de ces primitives recoivent comme paramètres des listes de variables. Ces listes sont utilisées pour construire des buffers (cfr module BUILDBUF) qui seront visés par des fenêtres. Or, comme ces buffers ne sont pas extensibles (il ont une taille maximum, qui dans le cas présent, est égale à la taille de la fenêtre qui le vise), ils doivent être constamment réinitialisés par la liste des variables lorsqu'il n'est pas possible d'y mettre en une fois toute l'information que cette liste contient. Ces réinitialisations sont effectuées selon une procédure bien déterminée. Cette procédure est réalisée au sein des primitives de la manière suivante.

```
...
switch(x = mgtmenu(&menu, UPDOWN, x, &mvt))
{
 case 1 :
 case 2 :    Gestion du menu qui accompagne la fenêtre
 ...
 default: switch(scrollwind(&wind, mvt, buf))
          {
          case LDE :
          case PDE :
          case LUE :     Gestion de la fin du buffer
          case PUE :     (cfr module WIND)
          default  :
          }
          x = - x;
          break;
}
...
```

Quatre cas litigieux se présentent donc.

(A) Le pointeur est en dessous de la fenêtre (et par conséquent à la fin du buffer) et l'utilisateur demande [↓] (c'est le cas LDE). Alors, on descend dans la liste de telle manière que l'on puisse remplir tout le buffer. Si on ne peut pas descendre assez pour remplir le buffer en entier, on remonte dans la liste de telle manière que l'on puisse remplir le buffer en entier. On descend alors la fenêtre d'autant de lignes qu'il y en a dans le pointeur.

(B) Le pointeur est en dessous de la fenêtre et l'utilisateur demande [PgDn] (c'est le cas PDE). Même scénario que (A). Une fois le buffer initialisé, on descend la fenêtre d'autant de lignes qu'il y en a dans celle-ci.

(C) Le pointeur est au dessus de la fenêtre et l'utilisateur demande [↑]. Alors, on remonte dans la liste de manière à pouvoir remplir le buffer avec de nouvelles données. Si on atteind le début de la liste, on initialise le buffer avec le début de la liste. On remonte alors la fenêtre d'autant de lignes qu'il y en a dans le pointeur.

(D) Le pointeur est au dessus de la fenêtre et l'utilisateur demande [PgUp]. Même scénario que (C). Une fois le buffer réinitialisé, on remonte la fenêtre d'autant de lignes qu'il y en a dans celle-ci.

L'initialisation ou la réinitialisation du buffer s'effectue grâce à une primitive du module BUILDBUF. Dans les descriptions des primitives qui suivent, sera indiqué le nom de la primitive de ce module qui est utilisée.

De même on indiquera le nom de la primitive de vérification qui est appelée.

Les primitives qui sont décrites ci-dessous utilisent donc des menus, des grilles, des messages d'erreur ou d'information et des écrans d'aide qui sont décrits dans des fichiers que l'on peut éditer et compiler. La description de ces primitives comprend des références à ces objets. Il s'agit de leur nom qui permet de les identifier dans leur fichier respectif et de les appeler au moyen des primitives d'initialisation. Les menu sont contenus dans le fichier "menu.txt", les grilles dans le fichier "grid.txt", les messages d'erreur dans le fichier "error.txt", les messages d'information dans le fichier "head.txt" et les textes d'aide dans le fichier "help.txt".

Définition du synonyme ADR pour la base

Définition :
Cette primitive permet gérer le dialogue pour la définition du code de la base pour le dictionnaire DATACOM/DD.
Paramètres en entrée :
base : base dont on doit modifier le code.
Préconditions :
base est correctement initialisé (même vide).
Paramètres en sortie :
base : base
cbase() : code de retour
Postconditions :
La base est modifiée ou pas.
cbase() == 0 STOP
1 ACCEPT
Utilisation des outils du dialogue :
MENU : cbase
GRILLE : cbase
ERREUR : /
INFO : /
AIDE : cbase
BUILDBUF : /
Appel :

```
cbase(base)
R_BASE *base;
```

Définition d'un synonyme ADR pour un constituant

Définition :
Cette primitive permet de gérer le dialogue pour la définition des synonymes ADR pour tous les constituants d'un schéma de relations (nom de field).
Paramètres en entrée :
schrval : nom du schéma de relation
schr : schéma de relation
lst : liste de constituants
dec : variable de réentrance
Préconditions :
schrval, schr et lst sont correctement initialisés, dec == 0 et lst est la liste de l'ensemble des constituants de schr.
Paramètres en sortie :
lst : liste de constituants
ccnst() : code de retour
Postconditions :
Les synonymes ADR des constituants de lst sont modifiés ou non. Leur nom ADR est unique dans le schéma.
ccnst() == 1 ACCEPT
0 STOP
Utilisation des outils du dialogue :
MENU : ccnst
GRILLE : ccnst
ERREUR : /
INFO : /
AIDE : ccnst
BUILDBUF : lstbuf3()
Utilisation de procédures de vérification :
field_ident()

Appel :

```
ccnst(schrval, schr, lst, dec)
char schrval[];
R_SCHR *schr;
struct tab_cnat *lst;
int dec;
```

Collage de deux schéma de relations

Définition :

Cette primitive permet de gérer le dialogue pour le collage de deux schémas de relations.

Paramètres en entrée :

schr1	:	schéma de relation
lst1	:	liste de constituants
schr2	:	schéma de relation
lst2	:	liste de constituants

Préconditions :

schr1, lst1, schr2, lst2 sont correctement initialisés. lst1 est la liste de tous les constituants de schr1 et lst2 est la liste de tous les constituants de schr2.

Paramètres en sortie :

lst1	:	liste de constituants
lst2	:	liste de constituants
collage()	:	code de retour

Postconditions :

lst1 et lst2 avec des constituants sélectionnés pour le collage et avec le qualificatif de ces constituants éventuellemnt modifié. Le nombre de constituants sélectionnés dans les deux listes sont identiques.

collage() ==	0	STOP
	1	ACCEPT

Utilisation des outils du dialogue :

MENU	:	collage
GRILLE	:	collage
ERREUR	:	/
INFO	:	/
AIDE	:	collage
BUILDBUF	:	lstbuf13()

Utilisation de procédures de vérification :

verif_col()

Appel :

```
collage(schr1, lst1, schr2, lst2)
R_SCHR *lst1;
struct tab_cnat *lst1;
R_SCHR *lst2;
struct tab_cnat *lst2;
```

Affichage du contenu d'une clé

Définition :

Cette primitive permet de gérer le dialogue de l'affichage du contenu d'une clé.

Paramètres en entrée :

schr	:	schéma de relations
lst	:	liste de constituants

Préconditions :

schr et lst sont correctement initialisées et lst est la liste des constituants qui forment une des clés de schr.

Paramètres en sortie :
    contkey() : code de retour
Postconditions :
    contkey() == 0   STOP
Utilisation des outils du dialogue :
    MENU   : contkey
    GRILLE   : /
    ERREUR   : /
    INFO   : contkey
    AIDE   : contkey
    BUILDBUF   : lstbuf9
Appel :
    contkey(schr, lst)
    R_SCHR *schr;
    struct tab_cnat *lst:

Ajout d'un constituant et édition d'un schéma de relations

Définition :
    Cette primitive permet de gérer le dialogue pour l'ajout d'un constituant à un schéma de relation ou pour lui donner une nouvelle cardinalité et un nouveau nom de schéma.
Paramètres en entrée :
    schrval   : nom de schéma de relations
    schr   : schéma de relations
    lst   : liste de constituants
    attr   : attribut
    cnst   : constituant
    dec   : variable de réentrance
Préconditions :
    schrval, schr, lst, attr, cnst sont correctement initialisés et dec == 0. lst est la liste des constituants de schr et attr et cnst se rapporte au nouveau constituant.
Paramètres en sortie :
    schr   : schéma de relations
    attr   : attribut
    cnst   : constituant
    contschi() : code de retour
Postconditions :
    attr et cnst contiennent les caractéristiques du nouveau constituant, si il existe. schr est modifié ou non.
    contschi() == 0   STOP
                   1   ACCEPT
Utilisation des outils du dialogue :
    MENU   : contschi
    GRILLE   : contschi, contsch2
    ERREUR   : /
    INFO   : contschi, contsch2
    AIDE   : contschi
    BUILDBUF   : lstbuf14()
Utilisation de procédures de vérification :
    Non encore réalisée.

```
Appel :
     contschi(schrval, schr, lst, attr, cnst, dec)
     char schrval[];
     R_SCHR *schr;
     struct tab_cnat *lst;
     R_ATTR *attr;
     R_CNST *cnst;
     int dec;
```

Affichage du contenu d'un schéma de relations

```
Définition :
     Cette primitive permet de gérer le dialogue pour l'affichage
     du contenu d'un schéma de relation.
Paramètres en entrée :
     schrval     :   nom de schéma de relations
     lst         :   liste de constituants
     dec         :   variable de réentrance
Préconditions :
     schrval  et lst sont correctement initialisés et dec  ==  0.
     lst  est la liste des constituants du schéma de relations de
     nom schrval.
Paramètres en sortie :
     contsch2() :   code de retour
Postconditions :
     contsch2() ==  0    STOP
Utilisation des outils du dialogue :
     MENU        :   contsch2
     GRILLE      :   /
     ERREUR      :   /
     INFO        :   contschi, contsch2
     AIDE        :   contsch2
     BUILDBUF    :   lstbufi4()
Appel :
     contsch2(schrval, lst, dec)
     char schrval[];
     struct tab_cnat *lst;
     int dec;
```

Affichage du contenu d'un schéma de relation (bis)

```
Définition :
     Cette primitive permet de gérer le dialogue pour l'affichage
     du contenu d'un schéma de relation.
Paramètres en entrée :
     schrval     :   nom de schéma de relations
     lst         :   liste de constituants
     dec         :   variable de réentrance
Préconditions :
     schrval  et lst sont correctement initialisés et dec  ==  0.
     lst  est la liste des constituants du schéma de relations de
     nom schrval.
Paramètres en sortie :
     contschema():  code de retour
```

```
Postconditions :
     contschema() == 0    STOP
                     1    ACCEPT
                     2    CLES
Utilisation des outils du dialogue :
     MENU         :   contschema
     GRILLE       :   /
     ERREUR       :   /
     INFO         :   /
     AIDE         :   contschema
     BUILDBUF     :   lstbuf6()
Appel :
     contschema(schrval, lst, dec)
     char schrval[];
     struct tab_cnat *lst;
     int dec;
```

Changement d'ordre des constituants

```
Définition :
     Cette primitive permet gérer le dialogue pour le changement
     d'ordre des constituants d'un schéma de relations et/ou le
     changement du nom de qualificatif de ces constituants.
Paramètres en entrée :
     schrval      :   nom de schéma de relations
     schr         :   schéma de relations
     lst          :   liste de constituants
     dec          :   variable de réentrance
Préconditions :
     schrval, schr et lst sont correctement initialisés et dec ==
     0.   lst est la liste des constituants du schéma de
     relations de nom schrval.
Paramètres en sortie :
     lst          :   liste de constituants
     cqualord()   :   code de retour
Postconditions :
     L'ordre des constituants de lst est modifié ou pas. Le nom
     de qualificatif peut être également modifié ou non mais il
     est en tous cas unique dans le schéma de relations.
     cqualord() ==  0    STOP
                    1    ACCEPT
Utilisation des outils du dialogue :
     MENU         :   cqualord
     GRILLE       :   cqualord
     ERREUR       :   /
     INFO         :   /
     AIDE         :   cqualord
     BUILDBUF     :   lstbuf9()
Utilisation de procédures de vérification :
     qual_ident(), sort_lst()
Appel :
     cqualord(schrval, schr, lst, dec)
     char schrval[];
     R_SCHR *schr;
     struct tab_cnat *lst;
     int dec;
```

Définition d'un synonyme ADR pour les schémas de relations

Définition :
Cette primitive permet de gérer le dialogue pour définir le nom de record des schémas de relations d'une base de données.
Paramètres en entrée :
base : base de données
lst : liste de schémas de relations
Préconditions :
base et lst sont correctement initialisés et lst est la liste de tous les schémas de relations de base.
Paramètres en sortie :
lst : liste de schémas de relations
cschr() : code de retour
Postconditions :
Le synonyme ADR des schémas de relations de la liste lst sont modifiés ou non mais en tous cas ils sont uniques dans la base.
cschr() == 0 STOP
1 ACCEPT
Utilisation des outils du dialogue :
MENU : cschr
GRILLE : cschr
ERREUR : /
INFO : /
AIDE : cschr
BUILDBUF : lstbuf2()
Utilisation de procédures de vérification :
record_ident()
Appel :

```
cschr(base, lst)
R_BASE *base;
struct tab_schr *lst;
```

Définition d'un groupe

Définition :
Cette primitive permet de gérer le dialogue pour la définition d'un groupe de constituants dans un schéma de relations.
Paramètres en entrée :
schr : schéma de relations
lst : liste de constituants
Préconditions :
schr et lst sont correctement initialisés et lst est la liste des constituants du schéma de relations schr.
Paramètres en sortie :
group : nom de groupe
qual : qualificatif
defgroup() : code de retour
Postconditions :
group est initialisé par le nom du groupe que l'on veut créer ou non mais group est en tous cas unique dans schr.
qual est le nom de qualificatif du groupe si il existe.
defgroup() == 0 STOP
1 ACCEPT

Utilisation des outils du dialogue :

MENU	:	defgroup
GRILLE	:	defgroup
ERREUR	:	/
INFO	:	/
AIDE	:	defgroup
BUILDBUF	:	/

Utilisation de procédures de vérification :

group_ident()

Appel :

```
defgroup(schr, group, qual, lst)
R_SCHR *schr;
char group[];
char qual[];
struct tab_cnat *lst;
```

Définition d'une clé

Définition :

Cette primitive permet de gérer le dialogue pour définir une clé appartenant à un schéma de relations.

Paramètres en entrée :

schr	:	schéma de relations
lst	:	listes de constituants
k_lst	:	liste de clés

Préconditions :

schr, k_lst et lst sont correctement initialisés et lst est la liste des constituants du schéma de relations schr. k_lst est la liste des clés du schéma de relations schr.

Paramètres en sortie :

keid	:	clé
defkey()	:	code de retour

Postconditions :

keid est initialisé par les caractéristiques d'une clé ou non mais en tous cas le nom de la clé est unique dans le schéma schr.

defkey() ==	0	STOP
	1	ACCEPT

Utilisation des outils du dialogue :

MENU	:	defkey
GRILLE	:	defkey
ERREUR	:	/
INFO	:	defkey
AIDE	:	defkey
BUILDBUF	:	/

Utilisation de procédures de vérification :

verif_key()

Appel :

```
defkey(schr, lst, keid, k_lst)
R_SCHR *schr;
struct tab_cnat *lst;
R_KEID *keid;
struct tab_keid *k_lst;
```

Définition des caractéristiques des disques

Définition :

Cette primitive permet de gérer le dialogue pour définir les caractéristiques des disques associés à une base de données.

Paramètres en entrée :

base	:	base de données
dsk	:	disque
first	:	variable d'appel

Préconditions :

base et dsk sont correctement initialisés et first == 1 si c'est le premier appel de cette primitive. dsk est un disque associé à la base base.

Paramètres en sortie :

dsk	:	disque
dfndsk()	:	code de retour

Postconditions :

dsk est modifié ou non par ses nouvelles caractéristiques.

dfndsk() ==

0	STOP
1	ACCEPT
3	PREC
4	SUIV

Utilisation des outils du dialogue :

MENU	:	dfndsk
GRILLE	:	dfndsk
ERREUR	:	/
INFO	:	/
AIDE	:	dfndsk
BUILDBUF	:	/

Appel :

```
dfndsk(base, dsk, first)
R_BASE *base;
R_DSK *dsk;
int first;
```

Calcul du volume d'un index

Définition :

Cette primitive permet de gérer le dialogue pour le calcul du volume d'un l'index.

Paramètres en entrée :

dsk	:	disque
indx	:	index
ukey	:	clé unique (?)
dkey	:	clé double (?)
keysize	:	taille de la clé
volbl	:	volume en blocs
volby	:	volume en bytes
volcy	:	volume en cylindres
first	:	variable d'appel

Préconditions :

Toutes les variables sont correctement initialisées. dsk est le disque qui devra stocker indx. ukey, dkey et keysize sont les caractéristiques de la clé de indx. first == 1 pour le premier appel de la primitive.

Paramètres en sortie :

dsk	:	disque
ukey	:	clé unique (?)
dkey	:	clé double (?)
dskvilindx():		code de retour

Postconditions :

Les données de ces variables sont modifiées ou non pour le calcul de volume.

dskvolindx() ==

0	STOP
1	ACCEPT
2	CONTENU

Utilisation des outils du dialogue :

MENU	:	dskvolindx
GRILLE	:	volindx1, volindx2, volindx3
ERREUR	:	/
INFO	:	dskvolindx
AIDE	:	dskvolindx
BUILDBUF	:	/

Appel :

```
dskvolindx(dsk, indx, ukey, dkey, volbl, volby, volcy, keysize, first)
R_DSK *dsk;
R_INDX *indx;
char ukey[], dkey[], volbl[], volby[], volcy[], keysize[];
int first;
```

Calcul du volume d'un schéma de relations

Définition :

Cette primitive permet de gérer le dialogue pour le calcul du volume d'un schéma de relation.

Paramètres en entrée :

dsk	:	disque
log	:	logging (?)
schr	:	schéma de relations
volbl	:	volume en blocs
volby	:	volume en bytes
volcy	:	volume en cylindres
recsize	:	taille du record
first	:	variable d'appel

Préconditions :

Toutes les variables sont correctement initialisées. dsk est le disque qui devra stocker schr. recsize est la taille du record synonyme de schr. first == 1 pour le premier appel de la primitive. log est associé à dsk.

Paramètres en sortie :

dsk	:	disque
schr	:	schéma de relations
log	:	logging (?)
dskvolsch():		code de retour

Postconditions :

Les données de ces variables sont modifiées ou non pour le calcul de volume.

dskvolsch() ==

0	STOP
1	ACCEPT
2	CONTENU

Utilisation des outils du dialogue :

MENU	:	dskvolsch
GRILLE	:	volsch1, volsch2, volsch3
ERREUR	:	/
INFO	:	dskvolsch
AIDE	:	dskvolsch
BUILDBUF	:	/

Appel :

```
dskvolsch(dsk, log, schr, volbl, volby, volcy, recsize, first)
R_DSK *dsk;
char log[];
R_SCHR *schr;
char volbl[], volby[], volcy[], recsize[];
int first;
```

Eclatement d'un schéma de relations

Définition :

Cette primitive permet de gérer le dialogue pour éclater un schéma de relations.

Paramètres en entrée :

s_lst	:	liste de schémas de relations
schr1	:	schéma de relations
lst1	:	liste de constituants
schr2	:	schéma de relations
lst2	:	liste de constituants

Préconditions :

Toutes les variables sont correctement initialisées. schr1 == schr2 et lst1 == lst2. s_lst est la liste des schémas de relations de la base qui contient schr1.

Paramètres en sortie :

schr1	:	schéma de relations
lst1	:	liste de constituants
schr2	:	schéma de relations
lst2	:	liste de constituants
eclat()	:	code de retour

Postconditions :

Si l'éclatement est effectué, schr1 est différent de schr2 par son nom de schéma et les constituants de lst1 + lst2 forment au moins l'ensemble de départ lst1.

eclat() ==	0	STOP
	1	ACCEPT

Utilisation des outils du dialogue :

MENU	:	eclat
GRILLE	:	eclat
ERREUR	:	/
INFO	:	eclat
AIDE	:	eclat
BUILDBUF	:	lstbuf13()

Utilisation de procédures de vérification :

verif_eclat()

Appel :

```
eclat(s_lst, schr1, lst1, schr2, lst2)
struct tab_schr *s_lst;
R_SCHR *schr1;
struct tab_cnat *lst1;
R_SCHR *schr2;
struct tab_cnat *lst2;
```

Affichage des identifiants d'un schéma de relations

Définition :
Cette primitive permet de gérer le dialogue pour afficher les identifiants d'un schéma de relations.

Paramètres en entrée :

schr	:	schéma de relations
lst	:	liste de constituants
first	:	variable d'appel

Préconditions :
Toutes ces variables sont correctement initialisées. lst est la liste des constituants d'un identifiant de schr. first == 1 au premier appel.

Paramètres en sortie :
idntschema(): code de retour

Postconditions :

idntschema() ==	0	STOP
	1	CONTENU
	2	CLES
	3	PREC
	4	SUIV

Utilisation des outils du dialogue :

MENU	:	idntschema
GRILLE	:	/
ERREUR	:	/
INFO	:	/
AIDE	:	idntschema
BUILDBUF	:	lstbuf8()

Appel :

```
idntschema(schr, lst, first)
R_SCHR *schr;
struct tab_cnat *lst;
int first;
```

Affichage des clés d'un schéma de relations

Définition :
Cette primitive permet de gérer le dialogue pour l'affichage des clés d'un schéma de relations.

Paramètres en entrée :

schr	:	schéma de relations
keid	:	clé
lst	:	liste de constituants
first	:	variable d'appel

Préconditions :
Toutes les variables sont correctement initialisées. keid est une des clés associées au schéma de relations schr. lst est la liste des constituants qui forment keid. first == 1 au premier appel.

Paramètres en sortie :
keyschema() : code de retour

Postconditions :

keyschema() ==	0	STOP
	1	CONTENU
	2	IDENT
	3	PREC
	4	SUIV

```
Utilisation des outils du dialogue :
     MENU        :   keyschema
     GRILLE      :   keyschema
     ERREUR      :   /
     INFO        :   /
     AIDE        :   keyschema
     BUILDBUF    :   lstbuf7()
Appel :
     keyschema(schr,keid,lst,first)
     R_SCHR *schr;
     R_KEID *keid;
     struct tab_cnat *lst;
     int first;
```

Affichage des index

```
Définition :
     Cette primitive permet de gérer le dialogue pour l'affichage
     des index d'une base de données.
Paramètres en entrée :
     base        :   base de données
     lst         :   liste d'index
Préconditions :
     Toutes les variables sont correctement initialisées. lst est
     la liste des index de base.
Paramètres en sortie :
     lindxi()    :   code de retour
Postconditions :
     lindxi() ==     0    STOP
Utilisation des outils du dialogue :
     MENU        :   lindxi
     GRILLE      :   /
     ERREUR      :   /
     INFO        :   /
     AIDE        :   lindxi
     BUILDBUF    :   lstbuf4()
Appel :
     lindxi(base,lst)
     R_BASE *base;
     struct tab_indx *lst;
```

Affichage des clés

```
Définition :
     Cette primitive permet de gérer le dialogue pour l'affichage
     des clés d'une base de données.
Paramètres en entrée :
     base        :   base de données
     lst         :   liste de clés
Préconditions :
     Toutes les variables sont correctement initialisées. lst est
     la liste des clés de base.
Paramètres en sortie :
     lkeyi()     :   code de retour
Postconditions :
     lkeyi() ==      0    STOP
```

```
Utilisation des outils du dialogue :
     MENU        :   lkey1
     GRILLE      :   /
     ERREUR      :   /
     INFO        :   lkey1
     AIDE        :   lkey1
     BUILDBUF    :   lstbuf5()
Appel :
     lkey1(base, lst)
     R_BASE *base;
     struct tab_ksi *lst;
```

Affichage des schémas de relations

```
Définition :
     Cette primitive permet de gérer le dialogue pour l'affichage
     des schémas de relations d'une base de données.
Paramètres en entrée :
     base        :   base de données
     lst         :   liste de schémas de relations
Préconditions :
     Toutes les variables sont correctement initialisées. lst est
     la liste des schémas de relations de base.
Paramètres en sortie :
     lschri()    :   code de retour
Postconditions :
     lschri() ==     0    STOP
Utilisation des outils du dialogue :
     MENU        :   lschri
     GRILLE      :   /
     ERREUR      :   /
     INFO        :   lschri
     AIDE        :   lschri
     BUILDBUF    :   lstbufi()
Appel :
     lschri(base, lst)
     R_BASE *base;
     struct tab_schr *lst;
```

Menu "DEFINITION DES SYNONYMES ADR"

```
Définition :
     Cette  primitive  permet de gérer le dialogue pour  le  menu
     proposant  les sous-fonctions  de la fonction de  définition
     des synonymes ADR.
Paramètres en entrée :
     base        :   base de données
Paramètres en sortie :
     menuchadr():    code de retour
Postconditions :
     menuchadr() == 0    STOP
                    1    POUR LA BASE
                    2    POUR UN SCHEMA
                    3    POUR LES CONSTITUANTS D'UN SCHEMA
```

```
Utilisation des outils du dialogue :
     MENU        :   menuchadr
     GRILLE      :   /
     ERREUR      :   /
     INFO        :   /
     AIDE        :   menuchadr
     BUILDBUF    :   /
Appel :
     menuchadr(base)
     R_BASE *base;
```

Menu "DEFINITION DES ACCES"

```
Définition :
     Cette primitive permet de gérer le dialogue pour le menu
     proposant les sous-fonctions de la fonction de définition
     des accès.
Paramètres en entrée :
     base        :    base de données
Paramètres en sortie :
     menudefa():      code de retour
Postconditions :
     menudefa() ==  0    STOP
                    1    GESTION DES INDEX
                    2    CREATION D'UNE CLE PRIMAIRE
                    3    CREATION D'UNE CLE SECONDAIRE
                    5    GESTION DES CLES
Utilisation des outils du dialogue :
     MENU        :   menudefa
     GRILLE      :   /
     ERREUR      :   /
     INFO        :   menudefa
     AIDE        :   menudefa
     BUILDBUF    :   /
Appel :
     menudefa(base)
     R_BASE *base;
```

Menu "CHOIX DES OPTIMISATIONS"

```
Définition :
     Cette primitive permet de gérer le dialogue pour le menu qui
     propose les sous-fonctions de la fonction de l'optimisation.
Paramètres en entrée :
     base        :    base de données
Paramètres en sortie :
     menuopt()   :    code de retour
Postconditions :
     menuopt() ==   0    STOP
                    1    CALCUL DU VOLUME D'UN SCHEMA
                    2    CALCUL DU VOLUME D'UN INDEX
                    3    INFORMATIONS STATISTIQUES
```

```
Utilisation des outils du dialogue :
     MENU        :   menuopt
     GRILLE      :   /
     ERREUR      :   /
     INFO        :   /
     AIDE        :   menuopt
     BUILDBUF    :   /
Appel :
     menuopt(base)
     R_BASE *base;
```

Menu "PARAMETRAGES"

```
Définition :
     Cette primitive permet de gérer le dialogue pour le menu qui
     propose les sous-fonctions de la fonction de paramètrage.
Paramètres en entrée :
     base        :   base de données
Paramètres en sortie :
     menupar()   :   code de retour
Postconditions :
     menupar() ==   0    STOP
                    1    COULEURS
                    2    NOMS ADR
                    3    CARACTERISTIQUES DISQUE
Utilisation des outils du dialogue :
     MENU        :   menupar
     GRILLE      :   /
     ERREUR      :   /
     INFO        :   /
     AIDE        :   menupar
     BUILDBUF    :   /
Appel :
     menupar(base)
     R_BASE *base;
```

Menu "PRESENTATEUR RELATIONNEL"

```
Définition :
     Cette primitive permet de gérer le dialogue pour le menu qui
     propose les sous-fonctions de la fonction présentateur
     relationnel.
Paramètres en entrée :
     base        :   base de données
Paramètres en sortie :
     menupres()  :   code de retour
Postconditions :
     menupres() ==  0    STOP
                    1    TRANSFORMATION DE SCHEMA
                    2    DEFINITION DES ACCES
                    3    AIDE A L'OPTIMISATION
                    4    SERVICES
                    5    PARAMETRAGES
```

Utilisation des outils du dialogue :

MENU	:	menupres
GRILLE	:	/
ERREUR	:	/
INFO	:	/
AIDE	:	menupres
BUILDBUF	:	/

Appel :

```
menupres(base)
R_BASE *base;
```

Menu principal

Définition :

Cette primitive permet de gérer le dialogue pour le menu qui propose les fonctions de l'interface.

Paramètres en sortie :

menuprinc(): code de retour

Postconditions :

menuprin()	==	0	STOP
		1	PRESENTATEUR RELATIONNEL
		2	CHARGEMENT DE LA BD
		3	REPERCUTIONS IDA
		4	REPERCUTIONS ADR

Utilisation des outils du dialogue :

MENU	:	menuprinc
GRILLE	:	/
ERREUR	:	/
INFO	:	/
AIDE	:	menuprinc
BUILDBUF	:	/

Appel :

```
menuprinc()
```

Menu des transformations

Définition :

Cette primitive permet de gérer le dialogue pour le menu qui propose les sous-fonctions de la fonction de transformation de schémas de relations.

Paramètres en entrée :

base : base de données

Paramètres en sortie :

menutrans(): code de retour

Postconditions :

menutrans()	==	0	STOP
		1	COLLAGE
		2	ECLATEMENT
		3	CREATION D'UN GROUPE
		4	SUPPRESSION D'UN GROUPE
		5	MODIFICATION DE LA REPETITIVITE

Utilisation des outils du dialogue :

MENU	:	menutrans
GRILLE	:	/
ERREUR	:	/
INFO	:	/
AIDE	:	menutrans
BUILDBUF	:	/

Appel :

```
menutrans(base)
R_BASE *base;
```

Sélection de constituants pour un groupe

Définition :

Cette primitive permet de gérer le dialogue pour sélectionner des constituants dans une liste pour former un groupe.

Paramètres en entrée :

group	:	nom d'un groupe
schr	:	schéma de relations
lst	:	liste de constituants

Préconditions :

Toutes les variables sont correctement initialisées. lst est la liste des constituants de schr qui servent à constituer group.

Paramètres en sortie :

lst	:	liste de constituants
selcnsti()	:	code de retour

Postconditions :

Des élements de la liste lst sont pointés ou non.

selcnsti()	==	0 STOP
		1 ACCEPT

Utilisation des outils du dialogue :

MENU	:	selcnsti
GRILLE	:	/
ERREUR	:	/
INFO	:	/
AIDE	:	selcnsti
BUILDBUF	:	lstbufii()

Appel :

```
selcnsti(group, schr, lst)
char group[];
R_SCHR *schr;
struct tab_cnat *lst;
```

Sélection d'un groupe pour éclatement

Définition :

Cette primitive permet de gérer le dialogue pour la sélection d'un groupe dans une liste pour l'éclater.

Paramètres en entrée :

schrval	:	nom de schéma de relations
lst	:	liste de constituants

Préconditions :

Toutes les variables sont initialisées. lst est la liste de tous les constituants du schéma de relations de nom schrval.

```
Paramètres en sortie :
     lst          :   liste de constituants
     selcnst2()   :   code de retour
Postconditions :
     Un élement de la liste est sélectionné si ACCEPT.
     selcnst2()  ==  0    STOP
                     1    ACCEPT
Utilisation des outils du dialogue :
     MENU         :   selcnst2
     GRILLE       :   /
     ERREUR       :   /
     INFO         :   selcnst2
     AIDE         :   selcnst2
     BUILDBUF     :   lstbuf12()
Appel :
     selcnst2(schrval, lst)
     char schrval[];
     struct tab_cnat *lst;
```

Modification de la répétitivité

```
Définition :
     Cette primitive permet de gérer le dialogue qui permet à
     l'utilisateur de modifier la répétitivité d'un constituant.
Paramètres en entrée :
     schr         :   schéma de relations
     lst          :   liste de constituants
Préconditions :
     Toutes ces variables sont correctement initialisées. lst est
     la liste des constituants de schr.
Paramètres en sortie :
     lst          :   liste de constituants
     selcnst3()   :   code retour
Postconditions :
     lst est modifié ou non.
     selcnst3()  ==  0    STOP
                     1    ACCEPT
Utilisation des outils du dialogue :
     MENU         :   selcnst3
     GRILLE       :   selcnst3
     ERREUR       :   /
     INFO         :   selcnst3, selcnst31
     AIDE         :   selcnst3
     BUILDBUF     :   lstbuf12()
Appel :
     selcnst3(schr, lst)
     R_SCHR *schr;
     struct tab_cnat *lst;
```

Sélection d'un disque

```
Définition :
     Cette primitive permet de gérer le dialogue pour la
     sélection dans une liste d'un disque d'une base de données.
Paramètres en entrée :
     base         :   base de données
     lst          :   liste de disques
```

```
Préconditions :
     Toutes les variables sont correctement initialisées. lst est
     la liste des disques définis pour base.
Paramètres en sortie :
     lst          :   liste de disques
     seldsk()     :   code de retour
Postconditions :
     Un élément de lst est sélectionné si ACCEPT.
     seldsk() ==      0    STOP
                      1    ACCEPT
Utilisation des outils du dialogue :
     MENU         :   seldsk
     GRILLE       :   /
     ERREUR       :   /
     INFO         :   /
     AIDE         :   seldsk
     BUILDBUF     :   lstbuf10()
Appel :
     seldsk(base, lst)
     R_BASE *base;
     struct tab_dsk *lst;
```

Sélection et modification d'un index

```
Définition :
     Cette  primitive  permet  de  gérer  le  dialogue  pour  la
     sélection dans une liste et la modification d'un index.
Paramètres en entrée :
     base         :   base de données
     lst          :   liste d'index
     first        :   variable d'appel
Préconditions :
     Toutes ces variables sont correctement initialisées. lst est
     la liste des index de la base de données base. first == 1 au
     premier appel.
Paramètres en sortie :
     lst          :   liste d'index
     selindx1()   :   code de retour
Postconditions :
     lst modifié ou pas.
     selindx1() ==    0    STOP
                      1    ACCEPT
                      2    SUPPRIMER
                      3    CONTENU
Utilisation des outils du dialogue :
     MENU         :   selindx1
     GRILLE       :   selindx1
     ERREUR       :   /
     INFO         :   selindx1, selindx12
     AIDE         :   selindx1
     BUILDBUF     :   lstbuf5()
Appel :
     selindx1(base, lst, first)
     R_BASE *base
     struct tab_Ksi *lst;
     int first;
```

Sélection et modification d'une clé

Définition :
Cette primitive permet de gérer le dialogue pour la sélection dans une liste et la modification d'une clé.

Paramètres en entrée :
base : base de données
lst : liste de clés
first : variable d'appel

Préconditions :
Toutes ces variables sont correctement initialisées. lst est la liste des clés de la base de données base. first == 1 au premier appel.

Paramètres en sortie :
lst : liste de clés
selkey1() : code de retour

Postconditions :
lst modifié ou pas.
selkey1() == 0 STOP
1 ACCEPT
2 SUPPRIMER
3 CONTENU

Utilisation des outils du dialogue :
MENU : selkey1
GRILLE : selkey1
ERREUR : /
INFO : selkey1, selkey12
AIDE : selkey1
BUILDBUF : lstbuf5()

Appel :
```
selkey1(base, lst, first)
R_BASE *base
struct tab_Ksi *lst;
int first;
```

Sélection d'un schéma de relation

Définition :
Cette primitive permet de gérer le dialogue pour sélectionner un schéma de relation d'une base de données dans une liste.

Paramètres en entrée :
base : base de données
lst : liste de schémas de relations

Préconditions :
Toutes ces variables sont correctement initialisées. lst est la liste des schémas de relations de la base de données base.

Paramètres en sortie :
lst : liste de schémas de relations
selschema(): code de retour

Postconditions :
Un élément de lst est sélectionné si ACCEPT.
selschema() == 0 STOP
1 ACCEPT

```
Utilisation des outils du dialogue :
     MENU        :   selschema
     GRILLE      :   /
     ERREUR      :   /
     INFO        :   /
     AIDE        :   selschema
     BUILDBUF    :   lstbuf1()
Appel :
     selschema(base,lst)
     R_BASE *base;
     struct tab_schr *lst;
```

III.3.1.2 Filière des Données

Module de Gestion de la Base de Données

Niveau : 5

Description :

Ce module reprend les primitives d'accès aux données. Elles permettent l'accès au données de la BSR (cfr. II.4.2)

Une Base est composé d'un ensemble de Schémas de Relations qui sont l'Implémentation d'Objets Conceptuels. Un Objet Conceptuel est de type Entité ou Association. Les liens existant entre les Entités et Associations sont à l'origine des Liens entre Schémas de Relations. Un Schéma de Relations est composé de Constituants, pouvant être décomposables, définis sur des Domaines. Pour pouvoir distinguer plusieurs Constituants définis sur le même Domaine et appartenant au même Schéma de Relations, nous qualifions ces Constituants par le Rôle joué par le Domaine correspondant dans le Schéma de Relations. Un Constituant est l'implémentation d'un Attribut pouvant être de type Elément ou Groupe, provenant du Schéma Conceptuel (Attribut Conceptuel), ou créé lors des transformations de Schémas (Attribut Technique). Les Schémas de Relations peuvent avoir des Identifiants et des Clés définis sur des Constituants du même Schéma de Relations. Enfin, des Clés semblables (c.à.d. définies sur les mêmes Domaines) peuvent être Implémentées sous un Index.

Le modèle Entité-Association décrivant cette structure de données est représenté par la figure III.3.3.

Base

Une Base est un ensemble de Schémas de Relations et d'Index et regroupe diverses informations d'utilité générale.

Nom-Base : Nom de la Base provenant du Schéma Conceptuel correspondant.
Nom-Auteur : Nom de l'auteur du Schéma Conceptuel.
Date : Date d'extraction du Schéma Conceptuel.
Code-Base : Code de la Base.
Element-Def : Nom par défaut d'un Element ADR.
MasterKey-Def: Nom par défaut d'une Clé Primaire ou Principale.
Couleur : Paramètres par défaut du Présentateur Relationnel.
Caractéristiques-Disques : Caractéristiques des disques sur le site de l'utilisateur.

Nom-Disque : Nom du disque.
Bloc : Taille des blocs en nombre de caractères.
Cylindre : Nombre de blocs par cylindre de disque.
Taux : Taux de remplissage d'un bloc (en nombres de bytes).

Identifiant : Nom-Base.

Schéma de Relations

Un Schéma de Relations est un ensemble de relations définis sur les mêmes domaines de valeurs et ayant mêmes identifiants.

Nom-Schéma : Nom du Schéma de Relations.
Nom-Record : Nom du Record ADR correspondant au Schéma.
Code-Record : Code du Record ADR correspondant au Schéma.
Cardinalité : Nombre de relations du Schéma.

Identifiant : Nom-Schéma.

Objet Conceptuel

Objet DSL correspondant au Schéma de Relations.

Nom-Objet : Nom de l'Objet Conceptuel.
Type-Objet : Type de l'Objet (Entité ou Association).
Date : Date de création de l'Objet.

Identifiant : Nom-Objet.

Constituant

Caractéristique d'une relation pouvant prendre une valeur dans un ou plusieurs Domaines de valeurs.

Nom-Rôle : Rôle que joue le Constituant au sein d'un Schéma de Relations.
Nom-Field : Nom du Field ADR correspondant au Constituant.
Ordre : Ordre d'apparition d'un Constituant au sein d'un Schéma de Relations ou d'un autre Constituant.
Répétitivité : Répétitivité d'un Constituant au sein d'un Schéma de Relations ou d'un autre Constituant.
Densité : Probabilité de ne pas avoir une valeur "nulle" ou "inconnue".

Identifiant : Nom-Rôle + Nom-Attribut (via Implémentation-Const)
+ Nom-Schema (via Composition-Schéma).

Attribut-Domaine

Attribut DSL correspondant au Constituant. Domaine de valeurs du Constituant, qui de plus est celui de l'Attribut.

Nom-Attribut : Nom de l'Attribut.
Format : Format DSL-Proto de l'Attribut.
Date : Date de création de l'Attribut.

Identifiant : Nom-Attribut.

Attribut-Conceptuel

Sous-type de l'Attribut. Il regroupe les Attributs provenant du Schéma Conceptuel.

Type : Type de l'Attribut (Element ou Groupe).

Attribut-Technique

Sous-type de l'Attribut. Il regroupe les Attributs créés lors des transformations de Schémas.

Identifiant

Mécanisme regroupant un ensemble de Constituants d'un Schéma de Relations qui, pour un ensemble de valeurs données, n'identifie qu'une et une seule relation de ce Schéma de Relations.

Code-Identifiant : Code permettant d'identifier un Identifiant parmi les Identfiants d'un Schéma de Relations.

Identifiant : Code-Identifiant + Nom-Schéma (via Identification).

Clé

Mécanisme regroupant un ensemble de Constituants d'un Schéma de Relations, qui permet l'accès à une ou plusieurs relations d'un Schéma de Relations.

Nom-Clé : Nom de la Clé.
Type : Type de la Clé (Clé Principale ou Primaire et Clé Secondaire).
Clé-Vide : Code définissant si une occurrence de Clé ayant une valeur nulle doit être reprise dans l'Index ou pas.

Identifiant : Nom-Clé + Nom-Schéma (via Accès).

Index

Implémentation physique d'un mode d'accès à un ou plusieurs Records, définis sur des Clés formées de Constituants définis sur les mêmes domaines de valeurs et dans le même ordre d'apparition.

Nom-Index : Nom de l'Index.
Numéro-Index : Numéro de l'Index.

Identifiant : Nom-Index.

Lien-Schémas

Lien qui peut exister entre deux Schémas de Relations, provenant du fait que deux Schémas sont la représentation d'une Entité et une Association liées entre elles.

Densité : Probabilité d'accès à partir d'un Schéma de Relations vers l'autre.

Composition-Clé

Définit la composition d'une Clé.

Ordre : Ordre d'apparition d'un Constituant au sein d'une Clé.

Composition-Ident

Définit la composition d'un Identifiant.

Décomposition

Définit les Constituants d'un Constituant décomposable.

Implémentation-Const

Définit l'Attribut et le Domaine sur lesquels sont définis les Constituants. Définit ainsi la correspondance entre Constituant BSR, Field ADR et Element ou Group DSL.

Composition-Schéma

Définit la composition d'un Schéma de Relations.

Implémentation-Schéma

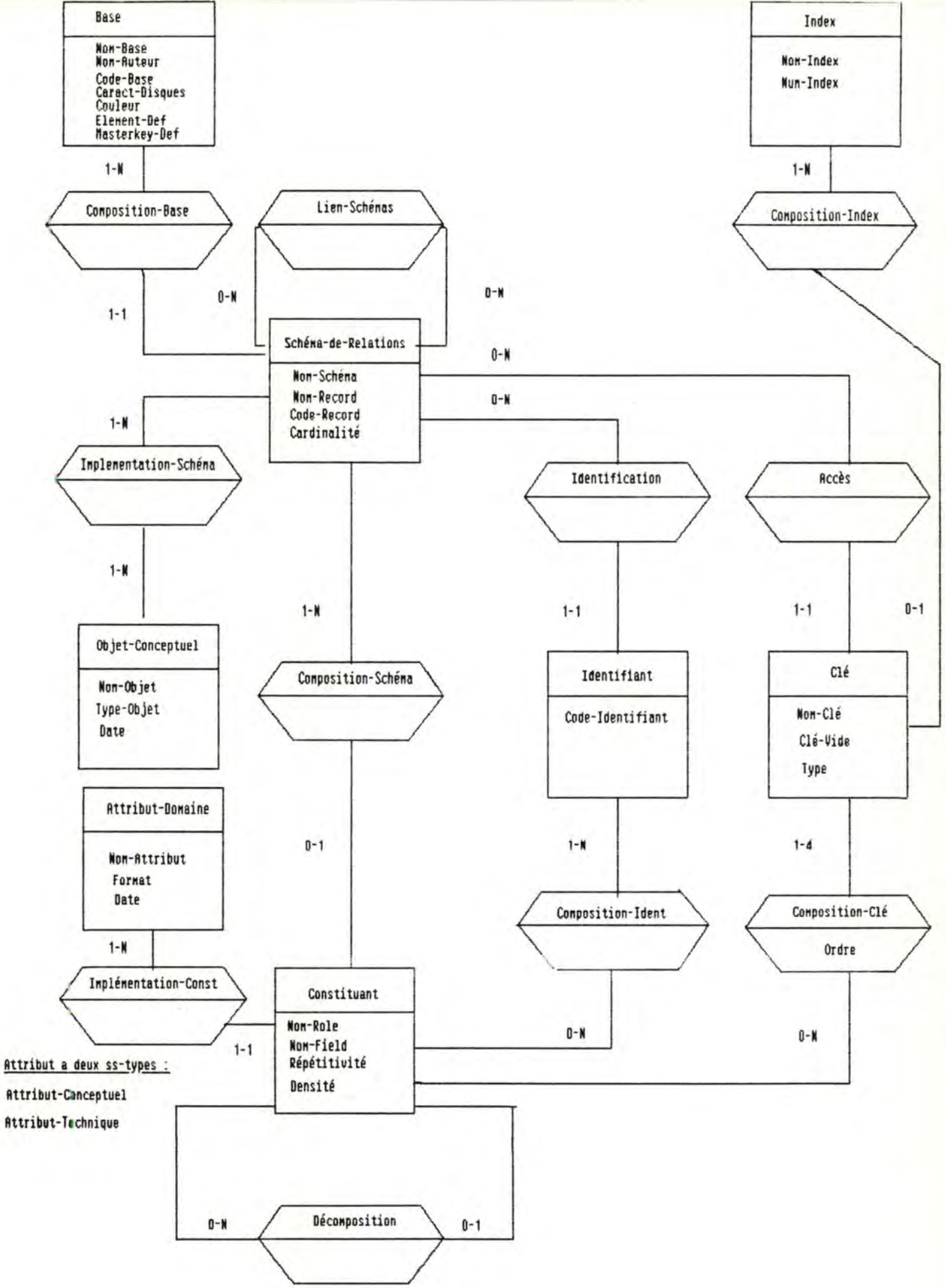
Définit la correspondance entre Schéma de Relations BSR, Record ADR et Entité ou Association DSL.

Composition-Index

Définit la composition d'un Index.

Les contraintes d'integrité sont les suivantes (notation tirée de [HAINAUT 85] :

1) # Base = 1.
2) Constituant(:Schéma-de-Relations) U
   Constituant(Décomposition:Constituant) = Constituant
3) Constituant(Décomposition:Constituant) =
   Constituant(:Attribut(:Type='GROUP'))
4) # Attribut((:Type='ELEMENT') ET
   (:Constituant(Décomposition:Constituant))) = 0
5) Constituant(:Identifiant(:Schéma-de-Relations
   (:Nom-Schéma='X'))) C
   Constituant(:Schéma-de-Relations(:Nom-Schéma='X'))
6) Constituant(:Clé(:Schéma-de-Relations(:Nom-Schéma='X')))C
   Constituant(:Schéma-de-Relations(:Nom-Schéma='X'))
7) Pour tout X ∈ Nom-Record ET <> ∅
   # Schéma-de-Relations(:Nom-Record='X') = 1
8) Pour tout X ∈ Code-Record ET <> ∅
   # Schéma-de-Relations(:Code-Record='X') = 1
9) Pour tout X ∈ Nom-Field ET <> ∅
   Pour tout Y ∈ Nom-Schéma
   # Constiuant((:Nom-Field='X') ET
   (:Schéma-de-Relations(:Nom-Schéma='Y'))) <= 1



Structure de la Base de Données (1)
Figure III.3.3

La base de données cible qui a été utilisée pour le poste de travail est celle développée dans le cadre de l'Atelier de Conception de Base de Données de l'Institut d'Informatique de Namur [CADELLI 86]. Le schéma décrit ci-dessus a été transformé en un schéma MAG (I.1.3) compatible aux restrictions de la base de données utilisées (Figure III.3.4). C'est ce dernier schéma qui est à la base des primitives d'accès qui sont décrites dans ce module.

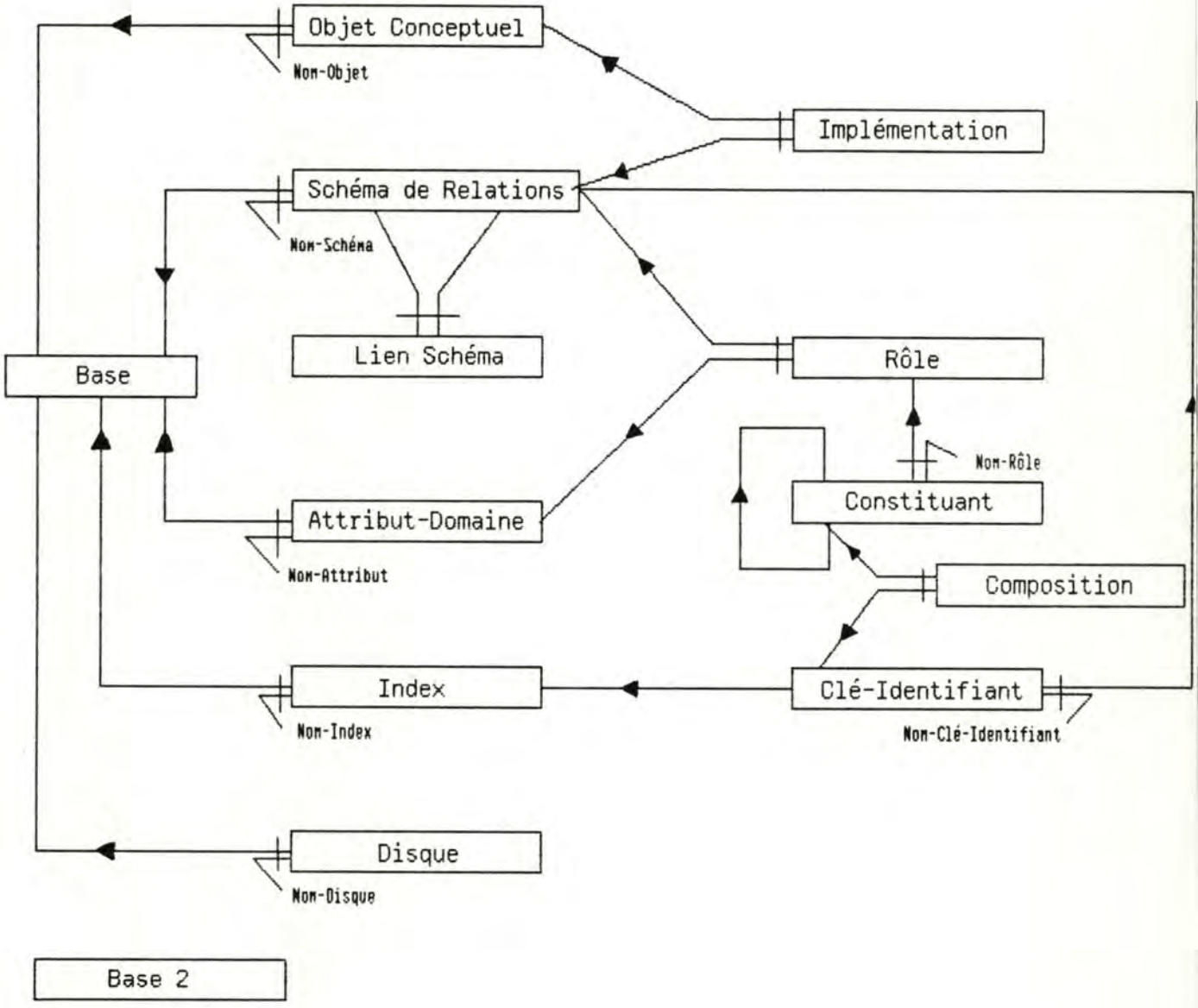
Les schémas autorisés par cette base sont des schémas MAG, restreints à des chemins 1-N ou N-1. On admet une clé par type de chemins et tout type de chemins dispose de son inverse. Il est également admis d'avoir des chemins multi-origines 1-N.

Les items sont élémentaires, simples, obligatoires, de type caractères. Les articles ont une longueur logique maximum de 87 caractères. Les spécifications complètes de ces primitives d'accès se trouvent décrites dans [CADELLI 86].

Le programmeur dispose essentiellement de deux types d'objets, les constantes de codes et les variables de référence . Les constantes de codes sont des entiers identifiant un type de chemins, un type d'articles, un type de clés ou un type d'erreurs. Une variable de référence est une structure C permettant d'accueillir les données d'un article de la base. Elle contient une référence à l'article, son type, la longueur totale de ses items ainsi que les valeurs de ces derniers.

Toutes les primitives de ce module renvoient un code d'erreur dans la variable dbstatus. Les codes d'erreurs sont les suivants:

ER_OK : L'opération est réussie.
ER_NOF : L'accès demandé a echoué.
ER_UNQ : La modification de l'article entraîne la violation d'une contrainte d'identification.
ER_MAND : L'article va être détaché alors qu'il est cible obligatoire du chemin dont on veut le détacher.
ER_NOP : La base est fermée.
ER_AOP : La base est déjà ouverte.
ER_PATH : Le code du chemin est invalide.
ER_RECT : Le code de l'article est invalide.
ER_REF : La variable de référence de l'article concerné par la primitive est invalide.
ER_LREF : Une variable de référence autre que celle concernée par la primitive est invalide.
ER_DBMS : Erreur grave devant provoquer l'arrêt du programme.



Contraintes d'intégrité :

1) Pour tout X ∈ Clé-Identifiant
   # Constituant(:Composition(:Clé-Identifiant((:Clé="OUI")
   ET (:Nom-Clé-Identifiant="X")))) ⩽ 4
2) Index = Index(:Clé-Identifiant(:Clé="OUI"))
3) # Index(:Clé-Identifiant(:Clé="NON")) = 0
4) # Base = # Base-2 = 1
5) Objet-Conceptuel = Objet-Conceptuel(:Implémentation)
6) Schéma-de-Relations = Schéma-de-Relations(:Implémentation)
7) Clé-Identifiant = Clé-Identifiant(:Composition)
8) Attribut = Attribut(:Rôle)
9) Rôle = Rôle(:Constituant)
10) Clé-Identifiant = Clé-Identifiant(:Schéma-de-Relations)

Structure de la Base de Données (2)
Figure III.3.4

Ces codes d'erreurs, ainsi que les codes des types d'articles, de clés et de chemins et les différents type d'articles sont décrits dans le fichier bsr.h.

L'ordre selon lequel on accède aux articles est aléatoire.

L'accès par clé n'etant pas implémenté nous l'avons simulé par accès séquentiel en attendant que les primitives correspondantes soient à notre disposition. Les codes de clé sont les suivants :

ATTR_NAME : Clé définie sur Nom-Attribut d'Attribut.
CNST_QUAL : Clé définie sur Nom-Role de Constituant.
SCHR_NAME : Clé définie sur Nom-Schéma de Schéma de Relations.
COBJ_NAME : Clé définie sur Nom-Objet de l'Objet Conceptuel.
REC_NAME : Clé definie sur Nom-Record de Schéma de Relations.
FLD_NAME : Clé définie sur Nom-Field de Constituant.
DISK_NAME : Clé définie sur Nom-Disque de Disque.

De plus comme dans le schéma final de la base Clé et Identifiant sont réunis au sein du même type d'article, 4 autres clés d'accès ont été définies :

KEID_NAME : Clé définie sur Nom-Clé de Clé ou Code-Identifiant d'Identifiant.
KEID_KEY : Clé permettant d'accéder aux Clés d'un Clé-Identifiant.
KEID_IDNT : Clé permettant d'accéder aux Identifiants d'un Clé-Identifiant.
KEID_MSTR : Clé permettant d'accéder à un Clé-Identifiant, candidat à être Clé principale.

Il est également conseillé d'utiliser le plus souvent les accès par chemins, plus rapide que les accès séquentiel à un type d'article.

Ouverture de la Base

Définition :
Ouverture de la Base de Données.
Paramètres en entrée :
dbname : Nom d'une Base de Données.
Préconditions :
taille(dbname) < 6
Fonction :
La primitive a pour effet d'ouvrir la Base de Données de nom dbname.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_AOP, ER_NOF.
Appel :
opendb(dbname);
char *dbname;
Fichier : sem.c

Fermeture de la Base

Définition :
Fermeture de la Base de Données.
Paramètres en entrée :
dbname : Nom d'une Base de Données.
Préconditions :
taille(dbname) < 6
Fonction :
La primitive a pour effet de fermer la Base de Données de nom dbname.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_NOP
Appel :
closedb(dbname);
char *dbname;
Fichier : sem.c

Accès direct à un article

Définition :
Accès à un article sur base de sa référence.
Paramètres en entrée :
reccode : Code d'un article.
rec.ref : Référence d'un article.
Préconditions :
reccode doit être le code de l'article référencé par rec.ref.
Paramètres en sortie :
rec : Variable de référence.
Postconditions :
Si ER_OK alors rec contient la référence, la taille, le type, la longueur et les valeurs d'items d'un article.
Fonction :
Fournit dans rec la taille, le type, la longueur et les valeurs d'items de l'article référencé par rec.ref de type reccode.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF.

Appel :
direct(reccode,rec)
int reccode;
R_RECORD *rec;
Fichier : sem.c

Accès séquentiel au premier article

Définition :
Accès au premier article d'un type donné.
Paramètres en entrée :
reccode : Code d'un article.
Préconditions :
reccode doit être un code d'article.
Paramètres en sortie :
rec : Variable de Référence.
Postconditions :
Si ER_OK alors rec contient la référence, la taille, le type, la longueur et les valeurs d'items d'un article.
Fonction :
Fournit dans rec la taille, le type, la longueur, la référence et les valeurs d'items du premier article de type reccode.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF, ER_RECT.
Appel :
first(reccode,rec)
int reccode;
R_RECORD *rec;
Fichier : sem.c

Accès séquentiel à l'article suivant

Définition :
Accès à l'article suivant un article d'un type donné.
Paramètres en entrée :
reccode : Code d'un article.
rec.ref : Référence d'un article.
Préconditions :
reccode doit être le code de l'article référence par rec.ref.
Paramètres en sortie :
rec : Variable de Référence.
Postconditions :
Si ER_OK alors rec contient la référence, la taille, le type, la longueur et les valeurs d'items d'un article.
Fonction :
Fournit dans rec la taille, le type, la longueur, la référence et les valeurs d'items de l'article de type reccode suivant celui référencé par rec.ref.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF, ER_RECT.
Appel :
next(reccode,rec)
int reccode;
R_RECORD *rec;
Fichier : sem.c

Accès au premier article dans un chemin

Définition :
Accède au premier article cible d'un chemin à partir d'un article origine donné.
Paramètres en entrée :
reccode : Code d'un article.
reco : Variable de Référence.
pathc : Code d'un chemin.
Préconditions :
reco.ref est la référence d' un article qui peut être origine d'un chemin de type pathc qui a pour cible des articles de type reccode.
Paramètres en sortie :
recc : Variable de Référence.
Postconditions :
Si ER_OK alors recc contient la référence, la longueur, le type et les valeurs d'items d'un article de la Base de Données.
Fonction :
Accède au premier article de type reccode, cible d'un chemin de type pathc qui a pour origine l'article référencé par reco.ref et stocke la longueur, le type, la référence et les valeurs d'items de cet article dans recc.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF, ER_LREF, ER_PATH.
Appel :

```
first_path(reccode,recc,reco,pathc)
int reccode,pathc;
R_RECORD *recc,*reco;
```

Fichier : sem.c

Accès à l'article suivant dans un chemin

Définition :
Accède à l'article cible d'un chemin suivant un article cible donné à partir d'un article origine donné.
Paramètres en entrée :
reccode : Code d'un article.
reco : Variable de Référence.
recc : Variable de Référence.
pathc : Code d'un chemin.
Préconditions :
reco.ref est la référence d'un article origine d'un chemin de type pathc qui a pour cible des articles de type reccode dont celui référencé par recc.ref.
Paramètres en sortie :
recc : Variable de Référence.
Postconditions :
Si ER_OK alors recc contient la référence, la longueur, le type et les valeurs d'items d'un article de la Base de Données.
Fonction :
Accède à l'article suivant l'article référencé par recc.ref de type reccode, cible d'un chemin de type pathc, qui a pour origine l'article référencé par reco.ref et stocke la longueur, le type, la référence et les valeurs d'items de cet article dans recc.

Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF, ER_LREF, ER_PATH.

Appel :

```
next_path(reccode, recc, reco, pathc)
int reccode, pathc;
R_RECORD *recc, *reco;
```

Fichier : sem.c

Accès par clé au premier article dans un chemin

Définition :
Accède au premier article cible d'un chemin ayant une valeur de clé donnée et un article origine du chemin donné.

Paramètres en entrée :
reccode : Code d'un article.
keycode : Code d'une clé.
reco : Variable de Référence.
pathc : Code d'un chemin.
val : Valeur d'un item.

Préconditions :
reco.ref référence un article qui peut être origine d'un chemin de type pathc qui a pour cible des articles de type reccode.
val est la valeur d'un item d'un type d'item appartenant à un article de type reccode et qui est clé de type keycode.

Paramètres en sortie :
recc : Variable de Référence.

Postconditions :
Si ER_OK alors recc contient la référence, la longueur, le type et les valeurs d'items d'un article de la Base de Données.

Fonction :
Accède au premier article recc de type reccode, cible d'un chemin de type pathc qui a pour origine l'article référencé par reco.ref et dont la valeur de clé de type keycode égale val. Stocke le type, la longueur, la référence et les valeurs d'items de cet article dans recc.

Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF, ER_LREF, ER_PATH.

Appel :

```
first_key_path(reccode, keycode, recc, reco, pathc, val)
int reccode, pathc, keycode;
R_RECORD *recc, *reco;
char *val;
```

Fichier : sem.c

Accès par clé à l'article suivant dans un chemin

Définition :
Accède à l'article cible d'un chemin suivant un article cible donné ayant une valeur de clé donnée à partir d'un article origine du chemin donné.

Paramètres en entrée :
reccode : Code d'un article.
Keycode : Code d'une clé.
reco : Variable de Référence.
recc : Variable de Référence.
pathc : Code d'un chemin.
val : Valeur d'un item
Préconditions :
reco.ref référence un article origine d'un chemin de type pathc qui a pour cible des articles de type reccode, dont celui référencé par recc.ref.
val est la valeur d'un item d'un type d'item appartenant à un article de type reccode et qui est clé de type Keycode.
Paramètres en sortie :
recc : Variable de Référence.
Postconditions :
Si ER_OK alors recc contient la référence, la longueur, le type et les valeurs d'items d'un article de la Base de Données.
Fonction :
Accède à l'article suivant celui référencé par recc.ref de type reccode, cible d'un chemin de type pathc ayant une clé de type Keycode d'une valeur val et pour origine l'article référencé par reco.ref. Stocke la longueur, le type, la référence et les valeurs d'items de cet article dans recc.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF, ER_LREF, ER_PATH.
Appel :
```
next_key_path(reccode,keycode,recc,reco,pathc,val)
int reccode,pathc,keycode;
R_RECORD *recc,*reco;
char *val;
```
Fichier : sem.c

Accès par clé au premier article

Définition :
Accède à l'article ayant une valeur de clé donnée.
Paramètres en entrée :
reccode : Code d'un article.
Keycode : Code d'une clé.
val : Valeur d'un item
Préconditions :
val est la valeur d'un item d'un type d'item appartenant à un article de type reccode et qui est clé de type Keycode.
Paramètres en sortie :
recc : Variable de Référence.
Postconditions :
Si ER_OK alors recc contient la référence, la longueur, le type et les valeurs d'items d'un article de la Base de Données.
Fonction :
Accède à l'article de type reccode, ayant une clé de type Keycode d'une valeur val. Stocke la longueur, le type, la référence et les valeurs d'items de cet article dans recc.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF.

Appel :

```
next_acces_key(reccode,keycode,recc,val)
int reccode,keycode;
R_RECORD *recc;
char *val;
```

Fichier : sem.c

Accès par clé à l'article suivant

Définition :
Accède à l'article ayant une valeur de clé donnée et suivant un article donnée.

Paramètres en entrée :
reccode : Code d'un article.
recc : Variable de Référence.
keycode : Code d'une clé.
val : Valeur d'un item

Préconditions :
val est la valeur d'un item d'un type d'item appartenant à un article de type reccode et qui est clé de type keycode.
recc.ref est la référence d'un article de type reccode.

Paramètres en sortie :
recc : Variable de Référence.

Postconditions :
Si ER_OK alors recc contient la référence, la longueur, le type et les valeurs d'items d'un article de la Base de Données.

Fonction :
Accède à l'article de type reccode, ayant une clé de type keycode d'une valeur val et qui suit l'article référencé par recc.ref. Stocke la longueur, le type, la référence et les valeurs d'items de cet article dans recc.

Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF.

Appel :

```
next_acces_key(reccode,keycode,recc,val)
int reccode,keycode;
R_RECORD *recc;
char *val;
```

Fichier : sem.c

Accès au premier article dans un double chemin

Définition :
Accède au premier article cible d'un chemin N-1 qui a pour origine un article cible d'un chemin 1-N à partir d'un article origine donné.

Paramètres en entrée :
reccode,recicode : Codes d'articles.
reco : Variable de Référence.
pathic,pathoi : Codes de chemins.

Préconditions :
reco.ref est la référence d'un article origine d'un chemin de type pathoi qui a pour cible des articles de type recicode, qui sont origines de chemins pathic qui ont pour cibles des articles de type reccode.

Paramètres en sortie :
reci, recc : Articles.

Postconditions :
Si ER_OK alors recc et reci contiennent la référence, la longueur, le type et les valeurs d'items d'articles de la Base de Données.
Fonction :
Accède au premier article de type recicode, cible d'un chemin de type pathoi qui a pour origine l'article référencé par reco.ref. Stocke la longueur, le type, la référence et les valeurs d'items de cet article dans reci.
Accède au premier article de type reccode, cible d'un chemin de type pathic qui a pour origine l'article référencé par reci.ref. Stocke la longueur, le type, la référence et les valeurs d'items de cet article dans recc.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF, ER_LREF, ER_PATH.
Appel :
```
firstiNi(reco,pathoi,recicode,reci,pathic,reccode,recc)
int reccode,recicode,patoi,pathic;
R_RECORD *recc,*reco,*reci;
```
Fichier : sem.c

Accès à l'article suivant dans un double chemin

Définition :
Accède au premier article cible d'un chemin N-1 qui a pour origine un article cible d'un chemin 1-N, suivant un article donnée à partir d'un article origine donné.
Paramètres en entrée :
reccode, recicode : Codes d'articles.
reco, reci, recc : Variables de Références.
pathic, pathoi : Codes de chemins.
Préconditions :
reco.ref est la référence d'un article origine d'un chemin de type pathoi qui a pour cible des articles de recicode (dont celui référencé par reci.ref), qui est origine d'un chemin de type pathic qui a pour cibles des articles de type reccode (dont celui référencé par recc.ref).
Paramètres en sortie :
reci, recc : Article.
Postconditions :
Si ER_OK alors recc et reci contiennent la référence, la longueur, le type et les valeurs d'items d'articles de la Base de Données.
Fonction :
Accède à l'article suivant l'article référencé par reci.ref de type recicode, cible d'un chemin de type pathoi qui a pour origine l'article référencé par reco.ref. Stocke la longueur, le type, la référence et les valeurs d'items de cet article dans reci.
Accède au premier article de type reccode, cible d'un chemin de type pathic qui a pour origine l'article référencé par reci.ref. Stocke la longueur, le type, la référence et les valeurs d'items de cet article dans recc.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_NOF, ER_LREF, ER_PATH.

Appel :

```
nextiN1(reco, pathoi, recicode, reci, pathic, reccode, recc)
int reccode, recicode, patoi, pathic;
R_RECORD *recc, *reco, *reci;
```

Fichier : sem.c

Création d'un article

Définition :

Crée un article dans la Base de Données.

Paramètres en entrée :

record : Variable de Référence.

reco i : Variable(s) de Référence.

Préconditions :

Les reco.ref i sont des références d'articles origines de chemins obligatoires pour l'article de type record.type.

Paramètres en sortie :

record : Variable de Référence.

Postconditions :

Si ER_OK record contient le type et la référence de l'article créé.

Fonction :

Crée un article ayant les valeurs d'items données dans record et le relie à tous les articles origines référencés par reco.ref i qui doivent lui être lié par des chemins obligatoires et renvoie le type et la référence de l'article crée dans record.

Erreurs :

dbstatus : ER_OK, ER_DBMS, ER_REF, ER_UNQ, ER_NOP, ER_LREF, ER_PATH, ER_RECT.

Appel :

Pour la création d'un article BASE (Base) :

```
c_base(baseref);
R_BASE *baseref;
```

Pour la création d'un article BASBIS (Base 2) :

```
c_basbis(basbisref);
R_BASBIS *basbisref;
```

Pour la création d'un article DISK (Disque) :

```
c_disk(diskref, baseref);
R_DISK *diskref;
R_BASE *baseref;
```

Pour la création d'un article COBJ (Objet Conceptuel) :

```
c_cobj(cobjref, baseref);
R_COBJ *cobjref;
R_BASE *baseref;
```

Pour la création d'un article SCHR (Schéma de relations) :

```
c_schr(schrref, baseref);
R_SCHR *cobjref;
R_BASE *baseref;
```

Pour la création d'un article ATTR (Domaine ou Atttribut) :

```
c_attr(attrref, baseref);
R_ATTR *attrref;
R_BASE *baseref;
```

Pour la création d'un article INDX (Index) :

```
c_indx(indxref, baseref);
R_INDX *attrref;
R_BASE *baseref;
```

Pour la création d'un article CNST (Constituant) :
```
c_cnst(cnstref,roleref);
R_CNST *cnstref;
R_ROLE *roleref;
```
Pour la création d'un article KEID (Clé-Identifiant) :
```
c_keid(keidref,schrref);
R_KEID *keidref;
R_SCHR *schrref;
```
Pour la création d'un article ROLE (Rôle) :
```
c_role(roleref,attrref,schrref);
R_ROLE *roleref;
R_ATTR *attrref;
R_SCHR *schrref;
```
Pour la création d'un article IMPL (Implémentation) :
```
c_impl(implref,cobjref,schrref);
R_IMPL *implref;
R_COBJ *cobjref;
R_SCHR *schrref;
```
Pour la création d'un article APOR (Composition) :
```
c_apor(aporref,keidref,cnstref);
R_APOR *aporref;
R_KEID *keidref;
R_CNST *cnstref;
```
Pour la création d'un article SCHL (Lien Schéma) :
```
c_schl(schlref,schr1ref,schr2ref);
R_SCHL *schlref;
R_SCHR *schr1ref;
R_SCHR *schr2ref;
```
Fichier : sem.c

Suppression d'un article

Définition :
Supprime un article de la Base de Données.
Paramètres en entrée :
reccode : Code d'un article.
rec.ref : Référence d'un article.
Préconditions :
reccode doit être le code de l'article référencé par rec.ref.
Paramètres en sortie :
rec.ref : Référence d'un article.
Postconditions :
Si ER_OK alors rec.ref est une référence nulle.
Fonction :
Supprime de la Base de Données l'article de type reccode référencé par rec.ref et supprime la référence rec.ref et ceci recursivement pour toutes les cibles obligatoires de l'article supprimé.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_RECT.
Appel :
```
ddelete(reccode,rec)
int reccode;
R_RECORD *rec;
```
Fichier : sem.c

Modifications de valeurs d'items

Définition :
Modifie les valeurs d'items n'intervenant pas dans un identifiant.
Paramètres en entrée :
reccode : Code d'un article.
rec : Variable de Référence.
Préconditions :
reccode doit être le code de l'article référencé par rec.ref.
Fonction :
Modifie toutes les valeurs d'items différents de l'identifiant de l'article référencé par rec.ref de type reccode par leurs nouvelles valeurs définies dans rec.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_RECT.
Appel :

```
modify(reccode,rec)
int reccode;
R_RECORD *rec;
```

Fichier : sem.c

Modifications de valeurs d'ikos

Définition :
Modifie les valeurs d'items intervenant dans un identifiant.
Paramètres en entrée :
reccode : Code d'un article.
rec : Variable de Référence.
Préconditions :
reccode doit être le code de l'article référencé par rec.ref.
Fonction :
Modifie toutes les valeurs d'items intervenant dans l'identifiant de l'article référencé par rec.ref de type reccode par leurs nouvelles valeurs définies dans rec.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_NOP, ER_RECT, ER_UNQ.
Appel :

```
modiko(reccode,rec)
```

Fichier : sem.c

Attacher un article à un chemin

Définition :
Insère ou transfère un chemin entre deux articles.
Paramètres en entrée :
reccode : Code d'un article.
reco : Variable de Référence.
recc : Variable de Référence.
pathc : Code d'un chemin.
Préconditions :
pathc est le type d'un chemin qui a pour origine des articles de même type que celui référencé par reco.ref et pour cible des articles de type reccode, type de l'article référencé par recc.ref.

Fonction :
Relie l'article référencé par recc.ref de type reccode via un chemin de type pathc à l'article origine référencé par reco.ref. Si l'article référencé par recc.ref est déjà cible d'un chemin de type pathc, alors cette insertion correspond à un transfert.
Erreurs :
dbstatus : ER_OK, ER_DBMS, ER_REF, ER_UNQ, ER_NOP, ER_LREF, ER_PATH, ER_RECT.
Appel :
```
attach(reccode,recc,reco,pathc)
int reccode,pathc;
R_RECORD *recc,*reco;
```
Fichier : sem.c

Détacher un article d'un chemin

Définition :
Supprime un chemin entre deux articles.
Paramètres en entrée :
reccode : Code d'un article.
reco : Variable de Référence.
recc : Variable de Référence.
pathc : Code d'un chemin.
Préconditions :
reco.ref est la référence d'un article origine d'un chemin de type pathc qui à pour cible l'article référencé par recc.ref de type reccode. pathc est un chemin non obligatoire.
Fonction :
Supprime le chemin de type pathc entre l'article origine référencé par reco.ref et l'article cible référencé par recc.ref de type reccode.
Erreurs :
ER_OK, ER_DBMS, ER_REF, ER_UNQ, ER_NOP, ER_MAND, ER_LREF, ER_PATH, ER_RECT.
Appel :
```
detach(reccode,recc,pathc)
int reccode,pathc;
R_RECORD *recc;
```
Fichier : sem.c

Création d'une référence

Définition :
Donne le statut de référence à une référence.
Paramètres en entrée :
ref : Entier.
Paramètres en sortie :
ref : Entier.
Postconditions :
ref est une référence reconnue par la Base de Données.
Fonction :
Donne le statut de référence à la référence ref.
Erreurs :
ER_OK, ER_DBMS, ER_REF.
Appel :
```
get_var(ref)
int *ref;
```

Fichier : sem.c.

Suppression d'une référence

Définition :
    Supprime le statut de référence d'une référence.
Paramètres en entrée :
    ref : Entier.
Préconditions :
    ref est une référence reconnue par la Base de Données.
Paramètres en sortie :
    ref : Entier.
Postconditions :
    ref n'a plus le statut de référence.
Fonction :
    Supprime le statut de référence de la référence ref.
Erreurs :
    ER_OK, ER_DBMS, ER_REF.
Appel :
    free_var(ref)
    int *ref;
Fichier : sem.c.

Annulation d'une référence

Définition :
    Annule une référence.
Paramètres en entrée :
    ref : Entier.
Préconditions :
    ref est une référence reconnue par la Base de Données.
Paramètres en sortie :
    ref : Entier.
Postconditions :
    ref est une référence ne référencant aucun article de la Base.
Fonction :
    Annule la référence ref.
Erreurs :
    ER_OK, ER_DBMS, ER_REF.
Appel :
    null(ref)
    int *ref;
Fichier : sem.c.

Assignation d'une référence à une autre

Définition :
    Assigne la valeur d'une référence à une autre.
Paramètres en entrée :
    ref1 : Entier.
Préconditions :
    ref1 est une référence reconnue par la Base de Données.
Paramètres en sortie :
    ref2 : Entier.
Postconditions :
    ref2 est une référence reconnue par la Base de Données.

Fonction :
Assigne la valeur de la référence ref1 à la référence ref2 de telle manière que ref1 et ref2 référencent le même article de la Base de Données.
Erreurs :
ER_OK, ER_DBMS, ER_REF, ER_LREF.
Appel :
assign(ref1,ref2)
int *ref1,*ref2;
Fichier : sem.c.

Test d'égalité de deux références

Définition :
Teste l'égalité de deux références.
Paramètres en entrée :
ref1,ref2 : Entiers.
Préconditions :
ref1 et ref2 sont des références reconnues par la Base de Données.
Paramètres en sortie :
res : Entier.
Postconditions :
res vaut 0 ou 1.
Fonction :
Renvoie res = 1 si ref1 et ref2 référencent le même article de la Base de Données, 0 sinon.
Erreurs :
ER_OK, ER_DBMS, ER_REF, ER_LREF.
Appel :
res = equal(ref1,ref2)
int res,*ref1,*ref2;
Fichier : sem.c.

Test de la nullité d'une référence

Définition :
Teste si une référence est nulle.
Paramètres en entrée :
ref : Entier.
Préconditions :
ref est une référence reconnue par la Base de Données.
Paramètres en sortie :
res : Entier.
Postconditions :
res vaut 0 ou 1.
Fonction :
Renvoie res = 1 si ref ne référence aucun article de la Base de Données, 0 sinon.
Erreurs :
ER_OK, ER_DBMS, ER_REF.
Appel :
res = isnul(ref)
int res,*ref;
Fichier : sem.c.

Transfert entre variables de références

Définition :
Transfère la valeur d'une variable de référence à une autre.
Paramètres en entrée :
rec1: Variable de Référence.
reccode : Code d'un Article.
Préconditions :
rec1 référence un article de type reccode.
Paramètres en sortie :
rec2 : Variable de Référence.
Postconditions :
rec2 est une Variable de Référence.
Fonction :
Transfère les valeurs d'items de la variable de référence rec1, de type reccode, à la variable rec2, de même type.
Appel :

```
transfert(rec1,rec2,reccode)
R_RECORD *rec1,*rec2;
int reccode;
```

Fichier : sem.c.

Module d'Extractions de Listes

Niveau : 4

Description :

Ce module offre des primitives d'extractions d'articles de la Base de Données sous forme de listes bidirectionnelles. Chaque élément de la liste est composé des champs suivants :

- Une ou plusieurs variables de référence d'article(s)
- Un pointeur vers l'élément qui le précède,
- Un pointeur vers l'élément qui le suit,
- Dans le cas d'une liste de Constituant-Attribut, un pointeur vers une liste de Constituant-Attribut lorsque l'élément contient les informations d'un Constituant décomposable.
- Dans le cas d'une liste de Constituant-Attribut, un code definissant l'appartenance à une clé primaire.
- Une variable act permettant de définir un code de traitement pour l'élément.

Ces listes sont dynamiques et doivent dès lors être supprimées lorsque l'on ne les utilise plus à l'aide d'une primitive adéquate définie dans le module "Traitements de Listes".

Liste d'articles cibles d'un chemin

Définition :
Extrait la liste de tous les articles cibles d'un chemin, ou d'une suite de chemins à partir d'une origine donnée.

Paramètres en entrée :
orgval : Variable de Référence.
val : entier.

Préconditions :
orgval.ref doit être la référence d'un article de la base qui peut être origine d'un chemin ou d'une suite de chemins quelconque (Voir type d'appel).

Paramètres en sortie :
lst : Liste.

Postconditions :
lst est une liste d'articles de la Base de Données.

Fonction :
Fournit lst qui est la liste de tous les articles cibles d'un chemin, ou d'une suite de chemins qui a l'article référencé par orgval.ref comme origine. Chaque élément de la liste aura comme valeur de lst.act la valeur définie par val.

Appel :
Pour la liste des Schémas de Relations de la Base :

```
lst = *schr_lst(orgval,val)
tab_schr *lst;
R_BASE *orgval;
int val;
```

```
Pour la liste des Disques de la Base :
     lst = *disk_lst(orgval,val)
     tab_disk *lst;
     R_BASE *orgval;
     int val;
Pour la liste des Objets Conceptuels de la Base :
     lst = *cobj_lst(orgval,val)
     tab_cobj *lst;
     R_BASE *orgval;
     int val;
Pour la liste des Index de la Base :
     lst = *indx_lst(orgval,val)
     tab_indx *lst;
     R_BASE *orgval;
     int val;
Pour la liste des Attributs (ou Domaines) de la Base :
     lst = *attr_lst(orgval,val)
     tab_cnat *lst;
     R_BASE *orgval;
     int val;
Pour la liste des Constituants d'un Attribut (ou Domaine) :
     lst = *cnstattr_lst(orgval,val)
     tab_cnat *lst;
     R_ATTR *orgval;
     int val;
Pour la liste de Constituant & Attributs (-Domaines) d'un
Schéma de Relations :
     lst = *cnatsch_lst(orgval,val)
     tab_cnat *lst;
     R_SCHR *orgval;
     int val;
(De plus chaque élément de cette liste aura le champs
id==YES si le Constituant intervient dans la Clé Principale)
Pour la liste des Constituants & Attributs (-Domaines) d'un
Constituants décomposable :
     lst = *cnatcnst_lst(orgval,val)
     tab_cnat = *lst;
     R_CNST *orgval;
     int val;
Pour la liste des Constituants & Attributs (-Domaines)
d'une Clé ou d'un Identifiant :
     lst = *cnatkeid_lst(orgval,val)
     tab_cnat = *lst;
     R_KEID *orgval;
     int val;
Pour la liste des Clés d'un Index :
     lst = *keyindx_lst(orgval,val)
     tab_keid *lst;
     R_INDX *orgval;
     int val;
Pour la liste des Clés d'un Schéma de Relations :
     lst = *keysch_lst(orgval,val)
     tab_keid *lst;
     R_SCHR *orgval;
     int val;
Pour la liste des Identifiants d'un Schéma de Relations :
     lst = *idntsch_lst(orgval,val)
     tab_keid *lst;
     R_SCHR *orgval;
     int val;
```

Pour la liste des Schémas de Relations correspondant à un Objet Conceptuel :

```
lst = *schrcobj_lst(orgval,val)
tab_schr *lst;
R_COBJ *orgval;
int val;
```

Pour la liste des Clés avec leurs Index et Schémas de Relations correspondant :

```
lst = *Ksi_lst(orgval,val)
tab_Ksi *lst;
R_BASE *orgval;
int val;
```

Traitement des listes

Niveau : 4

Fichier : traitlist.c

Description :

Ce module offre des primitives de traitement de listes fournies par le module "Extractions de listes" ou par lui-même.

Libération de mémoire de liste

Définition :
Libère la place mémoire réservée par une liste.
Paramètres en entrée :
lst : Liste.
Préconditions :
lst est une liste créée par le module "Extractions de listes" ou par le module "Traitements de listes".
Fonction :
Libère la place mémoire réservée par la liste lst.
Appel :
Pour libérer une liste de Schémas de Relations :

```
free_slst(lst)
tab_schr *lst;
```

Pour libérer une liste d'Index :

```
free_ilst(lst)
tab_indx *lst;
```

Pour libérer une liste d'Objets Conceptuels :

```
free_olst(lst)
tab_cnat *lst;
```

Pour libérer une liste de Clés-Identifiants :

```
free_klst(lst)
tab_keid *lst;
```

Pour libérer une liste de Constituants-Attributs :

```
free_clst(lst)
tab_cnat *lst;
```

Pour libérer une liste de Clés, Schémas de Relations et Index :

```
free_xlst(lst)
tab_ksi *lst;
```

Pour libérer une liste de Disques :

```
free_qlst(lst)
tab_disk *lst;
```

Pour libérer une liste de travail DBSS :

```
free_dlst(lst)
tab_dbss *lst;
```

Fichier : traitlist.c

Compte le nombre d'éléments d'une liste

Définition :
Compte le nombre d'éléments d'une liste ayant une valeur de traitement donnée.
Paramètres en entrée :
lst : Liste.
val : entier.
Préconditions :
lst est une liste créée par le module "Extractions de listes" ou par le module "Traitements de listes".
Paramètres en sortie :
i : entier.
Postconditions :
0 <= i <= taille de la liste.
Fonction :
Compte le nombre d'élément d'une liste lst ayant une valeur de lst.act égale à val et renvoit le résultat dans i.
Appel :
Pour compter les éléments d'une liste de Schémas de Relations :

```
i = count_sact(lst,val)
tab_schr *lst;
int val,i;
```

Pour compter les éléments d'une liste d'Index :

```
i = count_iact(lst,val)
tab_indx *lst;
int val,i;
```

Pour compter les éléments d'une liste d'Objets Conceptuels :

```
i = count_oact(lst,val)
tab_cnat *lst;
int val,i;
```

Pour compter les éléments d'une liste de Clés-Identifiants :

```
i = count_kact(lst,val)
tab_keid *lst;
int val,i;
```

Pour compter les éléments d'une liste de Constituants-Attributs :

```
i = count_cact(lst,val)
tab_cnat *lst;
int val,i;
```

Pour compter les éléments d'une liste de Clés, Schémas de Relations et Index :

```
i = count_kact(lst,val)
tab_ksi *lst;
int val,i;
```

Pour compter les éléments d'une liste de Disques :

```
i = count_qact(lst,val)
tab_disk *lst;
int val,i;
```

Pour compter les éléments d'une liste de travail DBSS :

```
i = count_dact(lst,val)
tab_dbss *lst;
int val,i;
```

Fichier : traitlist.c

Assignation d'une valeur à une liste

Définition :
Assigne une valeur d'action donnée à tous les éléments d'une liste.

Paramètres en entrée :
lst : Liste.
val : entier.

Préconditions :
lst est une liste créée par le module "Extractions de listes" ou par le module "Traitements de listes".

Paramètres en sortie :
lst : Liste

Postconditions :
Tous les éléments de la liste lst ont la même valeur dans le champs lst->act.

Fonction :
Assigne la valeur val à tous les éléments de la liste lst et renvoie cette liste lst.

Appel :

Pour assigner une valeur à une liste de Schémas de Relations :

```
mact_sval(lst,val)
tab_schr *lst;
int val;
```

Pour assigner une valeur à une liste d'Index :

```
mact_ival(lst,val)
tab_indx *lst;
int val;
```

Pour assigner une valeur à une liste d'Objets Conceptuels :

```
mact_oval(lst,val)
tab_cnat *lst;
int val;
```

Pour assigner une valeur à une liste de Clés-Identifiants :

```
mact_kval(lst,val)
tab_keid *lst;
int val;
```

Pour assigner une valeur à une liste de Constituants-Attributs :

```
mact_cval(lst,val)
tab_cnat *lst;
int val;
```

Pour assigner une valeur à une liste de Clés, Schémas de Relations et Index :

```
mact_xval(lst,val)
tab_ksi *lst;
int val;
```

Pour assigner une valeur à une liste de Disques :

```
mact_qval(lst,val)
tab_disk *lst;
int val;
```

Pour assigner une valeur à une liste de travail DBSS :

```
mact_qval(lst,val)
tab_dbss *lst;
int val;
```

Fichier : traitlist.c

Recherche d'un élément dans une liste

Définition :
Recherche un élément ayant une valeur d'action donnée dans une liste.

Paramètres en entrée :
lst : Liste.
val : entier.
i : entier.

Préconditions :
lst est une liste créée par le module "Extractions de listes" ou par le module "Traitements de listes".

Paramètres en sortie :
elmt : Adresse d'un élément de la liste.

Fonction :
Recherche le ième élément de la liste lst ayant une valeur d'action égale à val. Renvoie l'adresse de cette élément dans elmt.

Appel :
Pour rechercher un élément dans une liste de Schémas de Relations :

```
elmt = *search_sval(lst,i,val)
tab_schr *lst,elmt;
int val,i;
```

Pour rechercher un élément dans une liste d'Index :

```
elmt = *search_ival(lst,i,val)
tab_indx *lst,elmt;
int val,i;
```

Pour rechercher un élément dans une liste d'Objets Conceptuels :

```
elmt = *search_oval(lst,i,val)
tab_cnat *lst,elmt;
int val,i;
```

Pour rechercher un élément dans une liste de Clés-Identifiants :

```
elmt = *search_kval(lst,i,val)
tab_keid *lst,elmt;
int val,i;
```

Pour rechercher un élément dans une liste de Constituants-Attributs :

```
elmt = *search_cval(lst,i,val)
tab_cnat *lst,elmt;
int val,i;
```

Pour rechercher un élément dans une liste de Clés, Schémas de Relations et Index :

```
elmt = *search_kval(lst,i,val)
tab_ksi *lst,elmt;
int val,i;
```

Pour rechercher un élément dans une liste de Disques :

```
elmt = *search_qval(lst,i,val)
tab_disk *lst,elmt;
int val,i;
```

Pour rechercher un élément dans une liste de travail DBSS :

```
elmt = *search_dval(lst,i,val)
tab_dbss *lst,elmt;
int val,i;
```

Fichier : traitlist.c

Trier une liste

Définition :
Trie une liste.
Paramètres en entrée :
lst : Liste.
Préconditions :
lst est une liste créée par le module "Extractions de listes" ou par le module "Traitements de listes".
Paramètres en sortie :
lst : Liste.
Fonction :
Trie une liste lst sur une valeur de clé prédéterminée et par ordre croissant.
Appel :
Pour trier une liste de Schémas de Relations sur le Nom du Schéma :

```
lst = *sort_slst(lst)
tab_schr *lst;
```

Pour trier une liste d'Index sur le Nom de l'Index :

```
lst = *sort_ilst(lst)
tab_indx *lst;
```

Pour trier une liste d'Objets Conceptuels sur le Nom de l'Objet :

```
lst = *sort_olst(lst)
tab_cnat *lst;
```

Pour trier une liste de Clés-Identifiants sur le Nom de la Clé-Identifiant :

```
lst = *sort_klst(lst)
tab_keid *lst;
```

Pour trier une liste de Constituants-Attributs sur le Nom de l'Attribut :

```
lst = *sort_clst(lst)
tab_cnat *lst;
```

Pour trier une liste de Clés, Schémas de Relations et Index sur le Nom du Schéma :

```
lst = *sort_xlst(lst)
tab_ksi *lst;
```

Pour trier une liste de Disques sur le Nom du Disque :

```
lst = *sort_qlst(lst)
tab_disk *lst;
```

Pour trier une liste de travail DBSS sur le Nom du Schéma :

```
lst = *sort_dlst(lst)
tab_dbss *lst;
```

Fichier : traitlist.c

Ajouter un élément à une liste

Définition :

Ajoute un élément en début de liste.

Paramètres en entrée :

lst : Liste.
val : entier.
record i : Variables de Références.

Préconditions :

lst est une liste créée par le module "Extractions de listes" ou par le module "Traitements de listes". record i sont les Variables de Références qui forment un élément de la liste.

Paramètres en sortie :

lst : Liste.

Fonction :

Ajoute un élément formé des Variables de Références record i avec la valeur d'action égale val au début de la liste lst.

Appel :

Pour ajouter un élément à une liste de Schémas de Relations :

```
lst = *adelm_slst(lst, schrval, val)
tab_schr *lst;
R_SCHR *schrval;
int val;
```

Pour ajouter un élément à une liste d'Index :

```
lst = *adelm_ilst(lst, indxval, val)
tab_indx *lst;
R_INDX *indxval;
int val;
```

Pour ajouter un élément à une liste d'Objets Conceptuels :

```
lst = *adelm_olst(lst, cobjval, val)
tab_cnat *lst;
R_COBJ *cobjval;
int val;
```

Pour ajouter un élément à une liste de Clés-Identifiants :

```
lst = *adelm_klst(lst, keidval, val)
tab_keid *lst;
R_KEID *keidval;
int val;
```

Pour ajouter un élément à une liste de Constituants-Attributs :

```
lst = *adelm_clst(lst, attrval, cnstval, val)
tab_cnat *lst;
R_ATTR *attrval;
R_CNST *cnstval;
int val;
```

Pour ajouter un élément à une liste de Clés, Schémas de Relations et Index :

```
lst = *adelm_xlst(lst, keidval, schrval, indxval)
tab_ksi *lst;
R_SCHR *schrval;
R_KEID *keidval;
R_INDX *indxval;
int val;
```

Pour ajouter un élément à une liste de Disques :

```
lst = *adelm_qlst(lst, diskval, val)
tab_disk *lst;
R_DISK *diskval;
int val;
```

Pour ajouter un élément à une liste de travail DBSS :

```
lst = *adelm_dlst(lst, schrval, val)
tab_dbss *lst;
R_SCHR *schrval;
int val;
```

Fichier : traitlist.c

Concaténer deux listes

Définition :

Concatène deux listes de même type.

Paramètres en entrée :

lst1, lst2 : Listes.

Préconditions :

lst1 et lst2 sont deux listes créées par le module "Extractions de listes" ou par le module "Traitements de listes" et qui sont du même type.

Paramètres en sortie :

lst1 : Liste.

Fonction :

Concatène la liste lst2 au bout de la liste lst1 et renvoie lst1.

Appel :

Pour concaténer deux listes de Schémas de Relations :

```
lst1 = *concat_slst(lst1, lst2)
tab_schr *lst1, *lst2;
```

Pour concaténer deux listes d'Index :

```
lst1 = *concat_ilst(lst1, lst2)
tab_indx *lst1, *lst2;
```

Pour concaténer deux listes d'Objets Conceptuels :

```
lst1 = *concat_olst(lst1, lst2)
tab_cnat *lst1, *lst2;
```

Pour concaténer deux listes de Clés-Identifiants :

```
lst1 = *concat_klst(lst1, lst2)
tab_keid *lst1, *lst2;
```

Pour concaténer deux listes de Constituants-Attributs :

```
lst1 = *concat_clst(lst1, lst2)
tab_cnat *lst1, *lst2;
```

Pour concaténer deux listes de Clés, Schémas de Relations et Index :

```
lst1 = *concat_xlst(lst1, lst2)
tab_ksi *lst1, *lst2;
```

Pour concaténer deux listes de Disques :

```
lst1 = *concat_qlst(lst1, lst2)
tab_disk *lst1, *lst2;
```

Pour concaténer deux listes de travail DBSS :

```
lst1 = *concat_dlst(lst1, lst2)
tab_dbss *lst1, *lst2;
```

Fichier : traitlist.c

III.3.2 Niveau 3

Module de Vérification des Transformations

Niveau : 3

Description :

Ce module est responsable de la vérification des actions de l'utilisateur sur la Base de Données. Il vérifie que ces actions n'entraîne aucune incohérence de la Base.

Nom de Field valide

Définition :
Vérifie qu'un Nom de Field pour un Constituant n'identifie pas plus d'un élément dans une liste de Constituants-Attribut.
Paramètres en entrée :
field_name : String
lst : Liste
Postconditions :
lst est une liste de Constituants-Attributs.
Paramètres en sortie :
res : Entier.
Postconditions :
res = 1 ou = 0.
Fonction :
Renvoie res = 1 si field_name n'apparaît pas plus d'une fois dans la liste lst, 0 sinon.
Appel :

```
res = field_ident(field_name, lst)
char *field_name;
struct tab_cnat *lst;
int res:
```

Fichier : verif.c

Nom de Rôle et Attribut valide

Définition :
Vérifie qu'un Nom de Rôle et d'Attribut n'apparaissent pas plus d'une fois dans une liste de Constituants-Attributs.
Paramètres en entrée :
qual_name : String
attr_name : String
lst : Liste
Postconditions :
lst est une liste de Constituants-Attributs.
Paramètres en sortie :
res : Entier.
Postconditions :
res => 1.
Fonction :
Renvoie res = 1 si qual_name et attr_name n'identifie qu'un seul Constituant-Attribut de la liste lst et res > 1 sinon.
Appel :

```
res = qual_ident(qual_name, attr_name, lst)
char *qual_name, *attr_name;
struct tab_cnat *lst;
```

```
     int res;
Fichier : verif.c
```

Nom de Schéma inexistant

```
Définition :
     Vérifie qu'un Nom de Schéma n'apparaît pas dans une liste de
     Schémas.
Paramètres en entrée :
     schema_name : String
     lst : Liste
Postconditions :
     lst est une liste de Schémas de Relations.
Paramètres en sortie :
     res : Entier.
Postconditions :
     res = 0 ou 1.
Fonction :
     Renvoie res = 1 si schema_name n'identifie aucun élément de
     la liste lst, 0 sinon.
Appel :
     res = schema_ident(schr_name, lst)
     char *schr_name;
     struct tab_schr *lst;
     int res;
Fichier : verif.c
```

Nom et Code et de Record valide

```
Définition :
     Vérifie qu'un Nom et un Code de Record n'apparaissent pas
     plus d'une fois dans une liste de Schémas.
Paramètres en entrée :
     record_name : String
     code : String
     lst : Liste
Postconditions :
     lst est une liste de Schémas de Relations.
Paramètres en sortie :
     res : Entier.
Postconditions :
     res = 0 ou 1.
Fonction :
     Renvoie res = 1 si record_name et code n'apparaissent pas
     plus d'une fois dans la liste lst, 0 sinon.
Appel :
     res = record_ident(record_name, code, lst)
     char *record_name, code;
     struct tab_schr *lst;
     int res;
Fichier : verif.c
```

Nom de Clé valide

```
Définition :
     Vérifie qu'un Nom de Clé n'apparaît pas plus d'une fois dans
     une liste de Clés-Identifiants.
Paramètres en entrée :
     key_name : String
     lst : Liste
```

Postconditions :
    lst est une liste de Clés-Identifiants.
Paramètres en sortie :
    res : Entier.
Postconditions :
    res = 0 ou 1.
Fonction :
    Renvoie res = 1 si key_name n'apparait pas plus d'une fois dans la liste lst, 0 sinon.
Appel :
    res = key_ident(key_name, lst)
    char *key_name;
    struct tab_keid *lst;
    int res;
Fichier : verif.c

Nom d'Index valide

Définition :
    Vérifie qu'un Nom d'Index n'apparaît pas plus d'une fois dans une liste d'Index.
Paramètres en entrée :
    index_name : String
    lst : Liste
Postconditions :
    lst est une liste d'Index.
Paramètres en sortie :
    res : Entier.
Postconditions :
    res = 0 ou 1.
Fonction :
    Renvoie res = 1 si index_name n'apparait pas plus d'une fois dans la liste lst, 0 sinon.
Appel :
    res = idx_ident(index_name, lst)
    char *index_name;
    struct tab_indx *lst;
    int res;
Fichier : verif.c

Nom de Groupe et de Rôle inexistant

Définition :
    Vérifie qu'un Nom de Groupe et de Rôle n'apparaissent pas plus d'une fois dans une liste de Constituants-Attributs.
Paramètres en entrée :
    group_name : String
    role_name : String
    lst : Liste
Postconditions :
    lst est une liste de Constituants-Attributs.

Paramètres en sortie :
    res : Entier.
Postconditions :
    res = 0 ou 1.
Fonction :
    Renvoie res = 1 si group_name et role_name n'identifie aucun élément de la liste lst, 0 sinon.

Appel :
```
res = group_ident(group_name,role_name,lst)
char *group_name,*qual_name;
struct tab_cnat *lst;
int res;
```
Fichier : verif.c

Elément sélectionné est une Groupe ?

Définition :
Vérifie que le premier élément sélectionné d'une liste de Constituants-Attributs est un Constituant de type Groupe.
Paramètres en entrée :
lst : Liste
Postconditions :
lst est une liste de Constituants-Attributs.
Paramètres en sortie :
res : Entier.
Postconditions :
res = 0 ou 1.
Fonction :
Renvoie res = 1 si le premier élément sélectionné dans une liste lst (lst->act==YES) est de type GROUP, 0 sinon.
Appel :
```
res = verif_group(lst)
struct tab_cnat *lst;
int res;
```
Fichier : verif.c

Répétitivité modifiée valable ?

Définition :
Vérifie que le premier élément sélectionné d'une liste de Constituants-Attribut à une répétitivité valable.
Paramètres en entrée :
lst : Liste
Postconditions :
lst est une liste de Constituants-Attributs.
Paramètres en sortie :
res : Entier.
Postconditions :
res = 0 ou 1.
Fonction :
Renvoie res = 1 si le premier élément sélectionné dans une liste lst (lst->act==YES) a une répétitivité = 1 s'il ne fait pas parti de l'identifiant principal (lst->id==YES) ou > 1 s'il en fait parti, res = 0 sinon.
Appel :
```
res = verif_repet(lst)
struct tab_cnat *lst;
int res;
```
Fichier : verif.c

Vérifie l'éclatement

Définition :
Verifie que lors de l'opération d'éclatement il n'y a pas perte de Constituants-Attributs qui ne soit pas de type technique.

Paramètres en entrée :
lst1 et lst2 : Liste

Postconditions :
lst1 et lst2 sont deux listes de Constituants-Attributs identiques à la valeur de traitement près.

Paramètres en sortie :
res : Entier.

Postconditions :
res = 0 ou 1.

Fonction :
Renvoie res = 1 si le pour chaque élément de la liste lst1 avec une valeur d'action lst1->act = NO, l'élément correspondant dans la liste lst2 à une valeur d'action lst2->act = YES (sauf si l'Attribut est de type technique (lst1->attr.type = TECHNICAL), res = 0 sinon.

Appel :
```
res = verif_eclat(lst1,lst2)
struct tab_cnat *lst1,*lst2;
int res;
```

Fichier : verif.c

Module de l'Etablissement des Transformations

Niveau : 3.

Description :

Ce module offre des primitives aux modules "Transformations-Schémas" et "Définition des Accès" allégeant ces modules des tâches de mises-à-jour de la base de données.

Création d'un Groupe

Définition :
Création d'un Constituant Décomposable.
Paramètres en entrée :
inschr : Variable de Référence d'un Schéma de Relations.
inattr : Variable de Réference d'un Attribut.
incnst : Variable de Référence d'un Constituant.
lst : Liste de Constituants-Attributs.
Préconditions :
inschr.ref référence un Schéma de Relations auquel sont attachés tous les Constituants-Attributs de la liste lst. inattr et incnst contiennent la description du Constituant décomposable à créer. inattr et incnst ne peuvent pas identifier un Constituant du Schéma de Relations référencé par inschr.
Paramètres en sortie :
inattr : Variable de Réference à un Attribut.
incnst : Variable de Référence à un Constituant.
Postconditions :
inattr.ref et incnst.ref référencent le Constituant décomposable crée.
Fonction :
Crée le Constituant décomposable, du Schéma de Relations référencé par inschr.ref, et défini par inattr et incnst et lui attache comme composants tous les Constituants de la liste lst pour lesquels lst->act == YES.
Appel :

```
creat_group(inschr, inattr, incnst, lst)
R_SCHR *inschr;
R_ATTR *inattr;
R_CNST *incnst;
struct tab_cnat *lst;
```

Fichier : transche.c

Décomposition d'un Groupe

Définition :
Suppression d'un Constituant décomposable en gardant les Constituants dont il est composé.
Paramètres en entrée :
inschr : Variable de Référence d'un Schéma de Relations.
incnst : Variable de Référence d'un Constituant.
lst : Liste de Constituants-Attributs.
Précondtions :
incnst.ref référence un Constituant décomposable composé des Constituants de la liste lst et appartenant au Schéma de Relations référencé par inschr.ref.
Paramètres en sortie :
incnst : Variable de Référence.
Postconditions :
incnst.ref ne référence aucun article de la base.
Fonction :
Supprime le Constituant décomposable référencé par incnst.ref. Si ce Constituant fait partie d'une Clé, supprime la Clé. Si cette Clé était le composant unique d'un Index, supprime cette Index. Si ce Constituant fait partie d'un Identifiant, alors cet Identifiant hérite des Constituants composant ce Constituant décomposable. Redéfinit un Identifiant Principal, si nécessaire et si possible. Prévient l'utilisateur de toutes modifications de l'Identifiant principal.
Appel :

```
eclat_group(inschr, incnst, lst)
R_SCHR *inschr;
R_CNST *incnst;
struct tab_cnat *lst;
```

Fichier : transche.c

Modification de la Répétitivité (1)

Définition :
Met la répétitivité d'un Constituant à 1 et ajoute ce Constituant à l'Identifiant Principal.
Paramètres en entrée :
inschr : Variable de Référence d'un Schéma de Relations.
inkeid : Variable de Référence d'un Identifiant-Clé.
lst : Liste de Constituants-Attributs.
Préconditions :
inschr.ref référence un Schéma de Relations qui possède un Identifiant Principal référencé par inkeid.ref. lst est la liste des Constituants-Attributs de ce Schéma de Relations.
Fonction :
Modifie la répétitivité du premier élément de la liste lst ayant lst->act == YES et ajoute ce Constituant à l'Identifiant référencé par inkeid.ref. Supprime tous les autres Identifiants du Schéma de Relations référencé par inschr.ref. Prévient l'utilisateur s'il n'y a plus d'Identifiant Principal.
Appel :

```
mod_irepet(inschr, inkeid, lst)
R_SCHR *inschr;
R_KEID *inkeid;
struct tab_cnat *lst;
```

Fichier : transche.c

Modification de la Répétitivité (2)

Définition :
Modifie la répétitivité d'un Constituant faisant partie d'un Identifiant Principal et supprime ce Constituant de cet Identifiant.

Paramètres en entrée :
inschr : Variable de Référence d'un Schéma de Relations.
inkeid : Variable de Référence d'un Identifiant-Clé.
lst : Liste de Constituants-Attributs.

Préconditions :
inschr.ref référence un Schéma de Relations qui possède un Identifiant Principal référencé par inkeid.ref. lst est la liste des Constituants-Attributs de ce Schéma de Relations.

Fonction :
Modifie la répétitivité du premier élément de la liste lst ayant lst->act == YES et supprime ce Constituant de l'Identifiant référencé par inkeid.ref. Supprime tous les autres Identifiants du Schéma de Relations référencé par inschr.ref. Prévient l'utilisateur s'il n'y a plus d'Identifiant Principal.

Appel :
```
mod_orepet(inschr, inkeid, lst)
R_SCHR *inschr;
R_KEID *inkeid;
struct tab_cnat *lst;
```

Fichier : transche.c

Modification après éclatement

Définition :
Crée les deux Schémas de Relations obtenus après éclatement.

Paramètres en entrée :
inschr1 : Variable de Référence d'un Schéma de Relations.
inschr2 : Variable de Référence d'un Schéma de Relations.
lst1 : Liste de Constituants-Attributs.
lst2 : Liste de Constituants-Attributs.

Préconditions :
inschr1.ref référence un Schéma de Relations ayant comme Constituants les éléments de la liste lst1 == lst2 à la valeur d'act près. Les description de inschr1 et inschr2 ne peuvent entraîner des violation d'identification par rapport aux Schémas de Relations existant dans la base.

Paramètres en sorties :
inschr2 : Variable de Référence d'un Schéma de Relations.

Postconditions :
inschr2.ref référence un Schéma de Relations.

Fonction :
Eclate le Schéma de Relations référencé par inschr1.ref en supprimant tous ses Constituants ayant une valeur de lst1->act = NO. Modifie le nom de ce Schéma de Relations. Crée le Schéma de Relations défini par inschr2 composé des Constituants ayant une valeur de lst2->act = YES et compose l'Identifiant Principal des Constituants ayant une valeur de lst2->id = YES. Définit la correspondance de inschr2 avec les objets conceptuels.

Appel :

```
mod_eclat(inschr1, lst1, inschr2, lst2)
R_SCHR *inschr1, *inschr2;
struct tab_cnat *lst1, *lst2;
```

Fichier : transche.c

Modification après collage

Définition :

Crée le Schéma de Relations obtenu après collage.

Paramètres en entrée :

inschr1 : Variable de Référence d'un Schéma de Relations.
inschr2 : Variable de Référence d'un Schéma de Relations.
lst1 : Liste de Constituants-Attributs.
lst2 : Liste de Constituants-Attributs.
lst3 : Liste de Constituants-Attributs.

Préconditions :

inschr1.ref référence un Schéma de Relations ayant comme Constituants les éléments de la liste lst1. lst3 est la liste des Constituants de ce Schéma avant collage. inschr2.ref référence un Schéma de Relations ayant comme Constituants les éléments de la liste lst2.

Paramètres en sorties :

inschr2 : Variable de Référence d'un Schéma de Relations.

Postconditions :

inschr2.ref ne référence aucun Schéma de Relations.

Fonction :

Modifie tous les Constituants du Schéma de Relations référencé par inschr1.ref par leurs nouvelles valeurs définies dans lst3. Colle les Schémas de Relations référencés par inschr1.ref et inschr2.ref en ajoutant tous les Constituants de lst2 au Schéma de Relations référencé par inschr1.ref. Définit la correspondance de inschr1 avec les objets conceptuels. Supprime le Schéma de Relations référencé par inschr2.ref. Supprime les anciennes Clés (éventuellement Index) et Identifiant du Schéma de Relations référencé par inschr1.ref. Définit un Identifiant Principal formé des Constituants apparaissant dans les listes lst1 et lst2 avec une valeur de lst->id = YES. Prévient l'utilisateur si l'Identifiant Principal n'a pas pu être défini.

Appel :

```
mod_col(inschr1, lst1, lst3, inschr2, lst2)
R_SCHR *inschr1, *inschr2;
struct tab_cnat *lst1, *lst2, *lst3;
```

Fichier : transche.c

Suppression de clé

Définition :
Permet la suppression d'une Clé.
Paramètres en entrée :
lst : Liste de Clés, Schémas de Relations et Index.
Fonction :
Supprime toutes les clés de la liste lst ayant lst->act == YES. Previent l'utilisateur s'il supprime une Clé Principale.
Appel :

```
geskey(lst)
struct tab_Ksi *lst;
```

Fichier : transche.c

Gestion des Index

Définition :
Permet la suppression d'un Index, la création d'un Index ou la modification de ces Composants.
Paramètres en entrée :
lst1 : Liste de Clés, Schémas de Relations, Index.
lst2 : Liste de Clés, Schémas de Relations, Index.
Préconditions :
lst1 = lst2 avant l'action de suprression, modification ou ajout d'index.
Fonction :
Tout index apparaissant dans la liste lst1 et n'apparaissant pas dans lst2 est supprimé. Tout index apparaissant dans lst2 et n'apparaissant pas dans lst1 est crée. Toute Clé auquel a été ajouté un Index dans la liste lst2 sera relié à ce dernier.
Appel :

```
gesindex(lst1, lst2)
struct tab_Ksi *lst1, *lst2;
```

Fichier : transche.c

Module de Calcul de Volume

Niveau : 3.

Description :

Ce module offre les deux primitives de calcul du volume des données et des index selon les formules définis en II.4.4.1.3.

Volume des données

Définition :
Calcule la place disque nécessaire pour stocker les relations d'un Schéma de Relations.

Paramètres en entrée :
lst : Liste de Constituants-Attributs.
occ : Entier.
bloc_size : Entier.
cyl-size : Entier.
rate : Entier.
log : Booléen (O/N).

Paramètres en sorties :
vol_bloc :Entier.
vol_cyl : Entier.
vol_bytes : Entier.
record_size : Entier.

Fonction :
Calcule le nombre blocs (vol_bloc) de taille bloc_size (en bytes) avec un taux de remplissage rate %, le nombre de cylindre (vol_cyl) de taille cyl_size (en nombre de blocs), le nombre de bytes (vol_bytes) nécessaire pour stocker occ relations de taille record_size formées des Constituants de lst, et ceci selon le choix physique log.

Appel :

```
vol_data(lst, occ, bloc_size, cyl_size, rate, log, vol_bloc,
         vol_bytes, vol_cyl, record_size)
struct tab_cnat *lst;
char *occ;
char *vol_bloc;
char *record_size;
char *vol_bytes;
char *vol_cyl;
char *bloc_size;
char *cyl_size;
char *log;
char *rate;
```

Fichier : calcvol.c

Volume des Index

Définition :
Calcule la place disque nécessaire pour stocker un Index.
Paramètres en entrée :
lst : Liste de Constituants-Attributs.
bloc_size : Entier.
cyl-size : Entier.
rate : Entier.
uni_key : Entier.
dup_key : Entier.
Paramètres en sorties :
vol_bloc :Entier.
vol_cyl : Entier.
vol_bytes : Entier.
key_size : Entier.
Fonction :
Calcule le nombre blocs (vol_bloc) de taille bloc_size (en bytes) avec un taux de remplissage de rate %, le nombre de cylindre (vol_cyl) de taille cyl_size (en nombre de blocs), le nombre de bytes (vol_bytes) nécessaire pour stocker uni_key valeurs de clés uniques et dup_key valeurs de clés non unique de taille key_size formées des Constituants de lst.
Appel :

```
vol_indx(lst,bloc_size,cyl_size,rate,uni_key,dup_key,
         vol_bloc,vol_bytes,vol_cyl,key_size)
struct tab_cnat *lst;
char *bloc_size;
char *cyl_size;
char *vol_bloc;
char *key_size;
char *vol_bytes;
char *vol_cyl;
char *rate;
char *uni_key;
char *dup_key;
```

Fichier : calcvol.c

III. 3. 3 **Niveaux 0, 1 et 2**

Description :

Les modules des niveaux 0, 1 et 2 sont des automates. Chaque état de ces automates correspond soit à un dialogue (niveau 2), soit à un programme.

Chaque module du niveau 2 réalise une des classes de fonctions du Présentateur Relationnel. Ces classes de fonctions et ces fonctions elles-mêmes sont décrites en II. 4. 4. 1. Chaque module du niveau 1 correspond à un processeur décrit dans l'architecture fonctionnelle (II. 4). Le module du niveau 1 est le Poste de Travail.

Le passage d'un état de l'automate à un autre est dicté par les choix de l'utilisateur et est généralement accompagné d'un traitement. Ces modules s'occupe des traitements en laissant la gestion du dialogue aux modules de la filière "Gestion du Dialogue". Pour chaque état, il est indiqué quel est la référence du dialogue (entre parenthèses sur les figures). Les passages entre états numérotés sont accompagnés d'un traitement décrit avec chaque automate.

Les paramètres de ces automates sont etat (état courant de l'automate) et chemin (dernier chemin parcouru entre deux états). Ces deux paramètres permettent d'identifier le traitement à éffectuer par l'automate.

A chaque module ne correspond qu'une seule primitive.

Les descriptions de ces différents modules sont données à titre illustratif et doivent en aucun cas être considérées comme spécifications de ces modules. Ceux-ci ne font que réaliser les processeurs dont la spécification est donnée en II. 4. De plus seul les modules réalisés effectivement sont décrits ici.

Transformations de Schémas

Niveau : 2.

Description :

(Consultez également : II.4.4.1.1 Transformations de Schémas)

(1) Choix de la Transformation -> ... :

Recherche de la liste de tous les Schémas de Relations triée sur le Nom des Schémas de Relations.

(2) Collage Schémas -> Ajout d'un Constituant + Définition du Nom du Schéma :

Crée un nouveau Schéma de Relations et supprime les anciens, avec mise-à-jour de la correspondance, des identifiants et des clés, si l'action n'a pas été abandonnée par l'utilisateur.

(3) Ajout d'un Constituant + Définition du Nom de Schéma -> Choix de la Transforation :

Ajout éventuel d'un Constituant au sein du nouveau Schéma de Relations (l'ajoutant éventuellement à l'Identifiant Principal) et modification du nom du Schéma de Relations, si l'action n'a pas été abandonnée par l'utilisateur.

(4) Eclatement Schémas -> Choix de la Transformation :

Crée les deux nouveaux Schémas de Relations et supprime l'ancien, avec mise-à-jour de la correspondance, des identifiants et des clés, si l'action n'a pas été abandonnée par l'utilisateur.

(5) Sélection d'un Schéma -> ... :

Recherche de la liste de tous les Constituants du Schéma de Relations sélectionné. Liste triée sur le Nom des Attributs sur lesquels sont définis les Constituants.

(6) Modification de la répétitivité -> Choix de la Transformation :

Modifie la répétitivité du Constituant édité, supprime les anciens identifiants et crée le nouveau qui en découle, si l'action n'a pas été abandonnée par l'utilisateur.

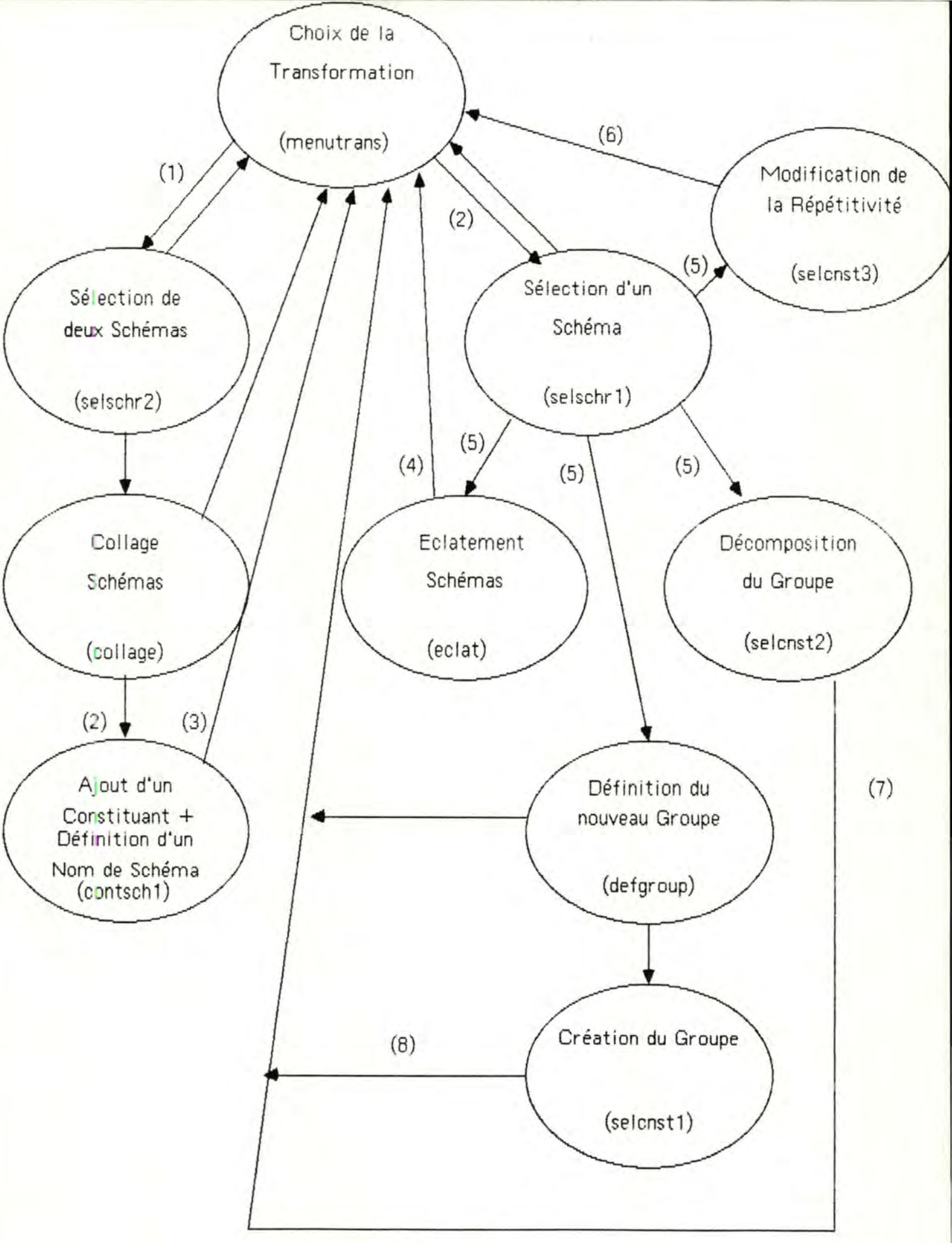


Figure III.3.3.1

(7) Décomposition du Groupe -> Choix de la Transformation :

Supprime le Constituant décomposable sélectionné et les Clés dans lesquelles il intervenait et rattache ses composants au Schéma de Relations et aux Identifiants dans lesquels il intervenait. Ces opérations sont réalisées si l'action n'a pas été abandonnée par l'utilisateur.

(8) Création du Groupe -> Choix de la Transformation :

Crée le nouveau Constituant décomposable composé des Constituants sélectionnés, qui sont rattachés à ce nouveau Constituant, si l'action n'a pas été abandonnée par l'utilisateur.

Appel :

```
transfo(etat,chemin)
int *etat,*chemin;
```

Fichier : scenario.c

Définition des Accès

Niveau : 2.

Description :

(Consultez également : II.4.4.1.2 Définition des Accès)

(1) Choix de l'Accès -> Sélection d'un Schémaa :

Recherche de la liste de tous les Schémas de Relations triée sur le Nom des Schémas de Relations.

(2) Choix de l'Accès -> ... :

Recherche de la liste de toutes les Clés, et leur Schéma de Relations et Index respectif, triée sur le Nom des Schémas des Relations.

(3) Sélection d'un Schéma -> Sélection de Constituants pour Clé :

Recherche de la liste de tous les Constituants du Schéma de Relations sélectionné. Liste triée sur le Nom des Attributs sur lesquels sont définis les Constituants.

(4) Sélection de Constituants pour Clé -> Choix de l'Accès :

Crée la Clé pour le Schéma de Relations sélectionné avec les Constituants sélectionnés, si l'action n'a pas été abandonnée par l'utilisateur.

(5) Suppression de la Clé -> Choix de l'Accès :

Supprime les Clés sélectionnées si l'action n'est pas abandonnée par l'utilisateur et prévient ce dernier s'il n'y a plus de Clé Principale.

(6) Modif + Suppr + Création d'Index -> Choix de l'Accès :

Modifie les composants d'un Index, supprime, ou crée un Index (selon les actions effectuées par l'utilisateur), si l'action n'est pas abandonnée par l'utilisateur.

(7) ... -> Contenu de la Clé :

Recherche la liste des Constituants formant la Clé sélectionnée, triée sur le Nom des Attributs sur lesquels sont définis ces Constituants.

Appel :

```
defacc(etat, chemin)
int *etat, *chemin;
```

Fichier : scenario.c

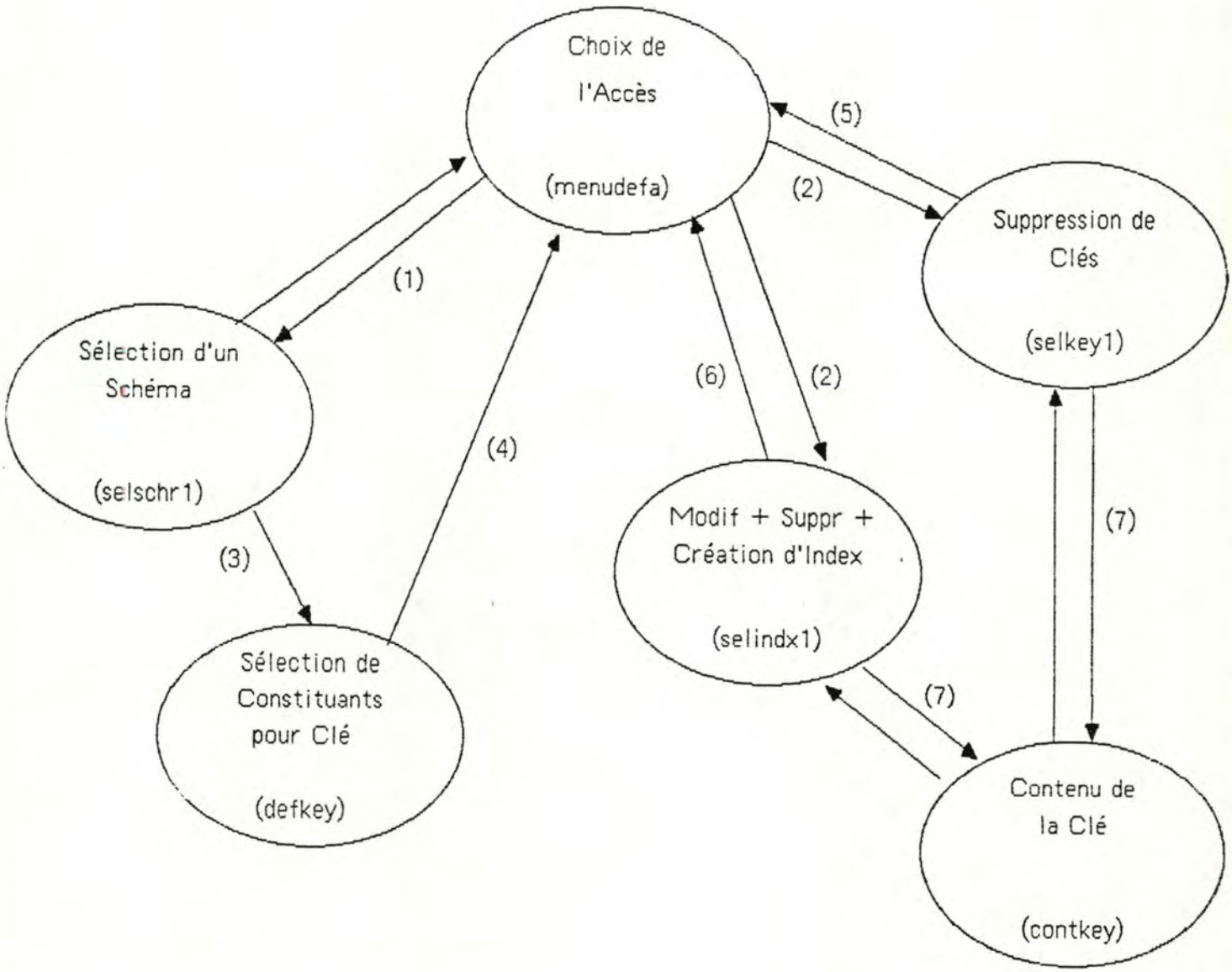


Figure III. 3. 3. 2

Aide à l'Optimalisation

Niveau : 2.

Description :

(Consultez également : II.4.4.1.3 Aides à l'Optimalisation)

(1) Choix de l'Aide -> Sélection d'un Schéma

Recherche de la liste de tous les Schémas de Relations triée sur le Nom des Schémas de Relations.

(2) Choix de l'Aide -> Sélection d'un Index :

Recherche de la liste de toutes les Clés, et leur Schéma de Relations et Index respectif, triée sur le Nom des Schémas de Relations.

(3) ... -> Sélection d'un Disque :

Recherche de la liste de tous les Disques, triée sur le Nom des Disques.

(4) Sélection d'un Disque -> Volume des données du Schéma :

Recherche de la liste de tous les Constituants du Schéma de Relations sélectionné. Liste triée sur le Nom des Attributs sur lesquels sont définis les Constituants.

(5) Sélection d'un Disque -> Volume des données de l'Index :

Recherche de la liste de tous les Constituants d'une des Clés de l'Index sélectionné. Liste triée sur le Nom des Attributs sur lesquels ces Constituants sont définis.

(6) Volume des données du Schéma -> Volume des données du Schéma :

Calcule l'espace disque nécessaire pour stocker le Schéma de Relations sélectionné sur le Disque sélectionné.

(7) Volume des données de l'Index -> Volume des donnnées de l'Index :

Calcule l'espace disque nécessaire pour stocker l'Index sélectionné sur le Disque sélectionné.

Appel :

```
aidop(etat, chemin)
int *etat, *chemin;
```

Fichier : scenario.c

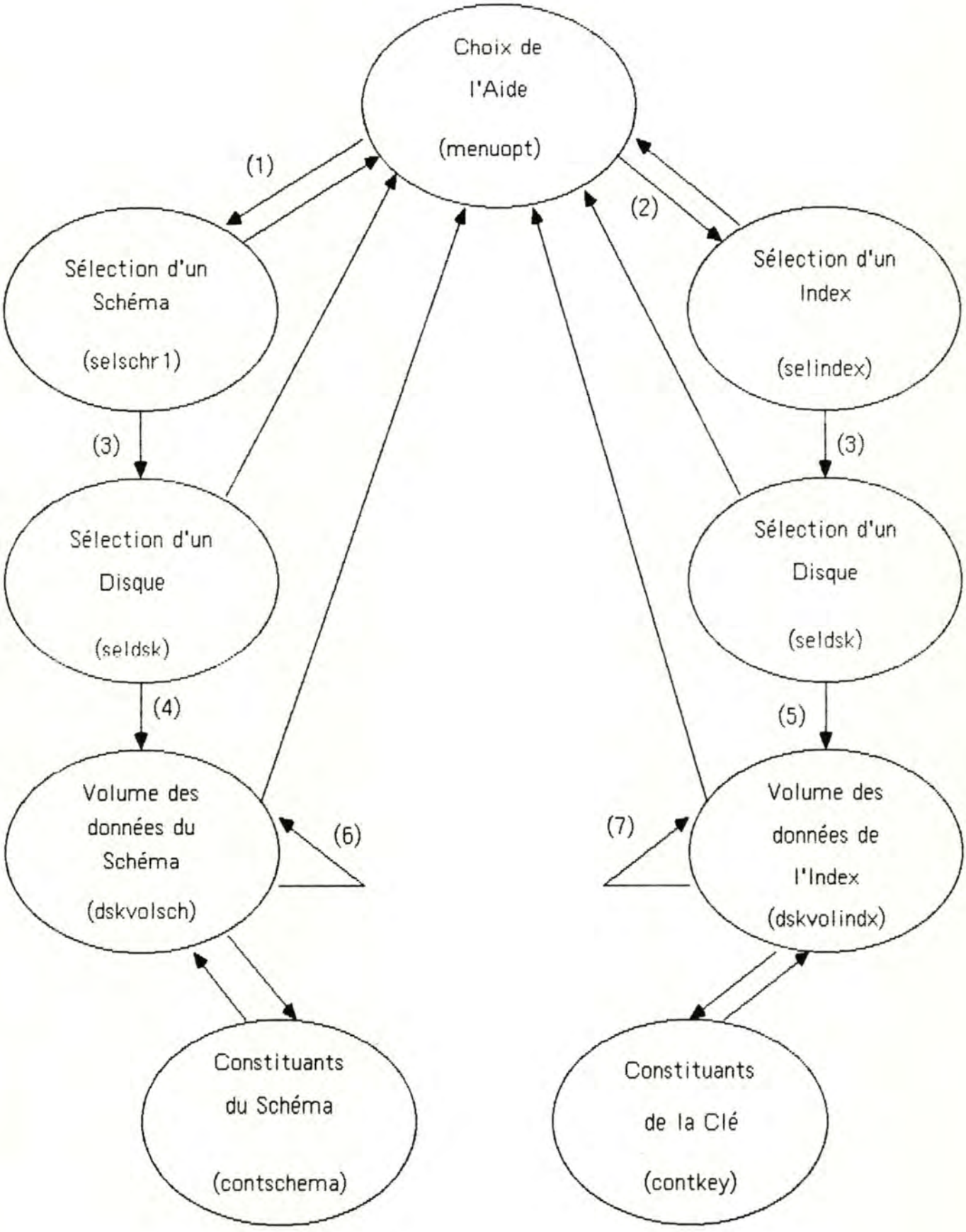


Figure III.3.3.3

Services

Niveau : 2.

Description :

(Consultez également : II.4.4.1.4 Services)

(1) Choix d'un Service -> Liste des Schémas :

Recherche de la liste de tous les Schémas de Relations triée sur le Nom des Schémas de Relations.

(2) Choix d'un Service -> Liste des Index :

Recherche de la liste de tous les Index triée sur le Nom des Index.

(3) Choix d'un Service -> Liste des Clés :

Recherche de la liste de toutes les Clés, et leur Schéma de Relations et Index respectif, triée sur le Nom des Schémas de Relations.

(4) Choix d'un Service -> Liste des Schémas :

Recherche de la liste de tous les Schémas de Relations triée sur le Noms des Schémas de Relations.

(5) Sélection d'un Schéma -> ... :

Recherche de la liste de tous les Constituants du Schéma de Relations sélectionné. Liste triée sur le Nom des Attributs sur lesquels sont définis les Constituants.

Recherche de la liste de tous les Identifiants du Schéma de Relations sélectionné, triée sur le Code des l'Identifiants.

Recherche de la liste de toutes les Clés du Schéma de Relations sélectionné, triée sur le Nom des Clés.

(6) Changement d'ordre + rôle des Constituants -> Choix d'un Service :

Mise-à-jour de la base de données des Constituants modifiés, si l'action n'est pas abandonnée par l'utilisateur.

Appel :

```
services(etat, chemin)
int *etat, *chemin;
```

Fichier : scenario.c

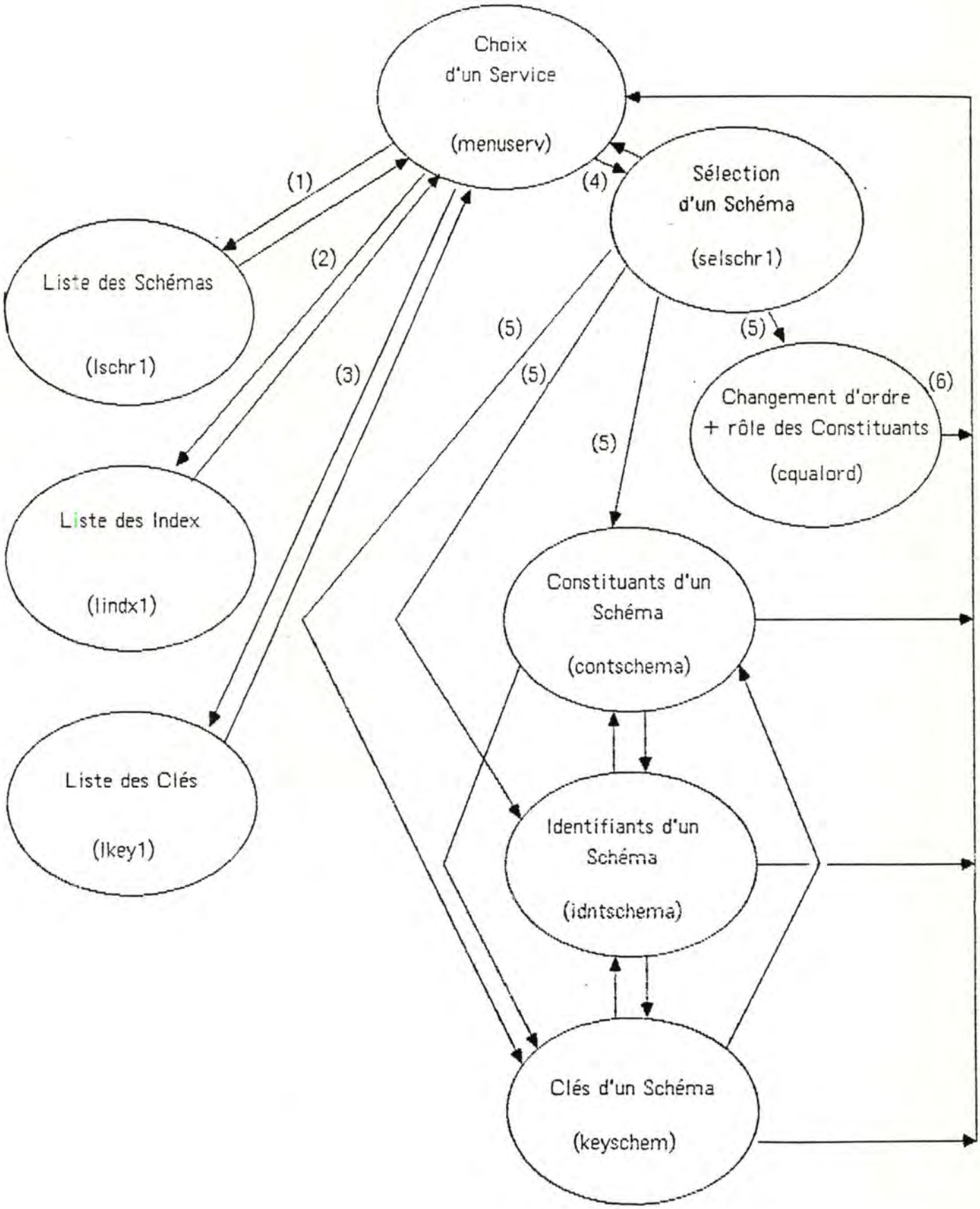


Figure III. 3. 3. 4

Paramétrages

Niveau : 2.

Description :

(Consultez également : II.4.4.1.4 Paramétrages)

(1) Modification des caractéristiques de l'interface -> Choix du Paramétrage :

Mise-à-jour de la base de données des caractéristiques modifiées, si l'action n'est pas abandonnée par l'utilisateur.

(2) Choix du Paramétrage -> Choix de l'objet ADR à définir :

Recherche de la liste de tous les Schémas de Relations triée sur le Nom des Schémas de Relations.

(3) Sélection d'un Schéma -> Définition des Noms de Fields :

Recherche de la liste de tous les Constituants du Schéma de Relations sélectionné. Liste triée sur le Nom des Attributs sur lesquels sont définis les Constituants.

(4) Définition des Noms de Fields -> Choix de l'objet ADR à définir :

Mise-à-jour de la base de données des Constituants ayant fait l'objet d'une définition de Nom de Field, si l'action n'est pas abandonnée par l'utilisateur.

(5) Définition de Codes et Noms de Records -> Choix de l'objet ADR à définir :

Mise-à-jour de la base de données des Schémas de Relations ayant fait l'objet d'une définition de Code ou Nom de Record, si l'action n'est pas abandonnée par l'utilisateur.

(6) Définition d'un Code de Base -> Choix de l'objet ADR à définir :

Mise-à-jour de la base de données de l'article Base, si l'action n'est pas abandonnée par l'utilisateur.

(7) Choix du Paramétrage -> Modification + Ajout Disques :

Recherche de la liste de tous les Disques, triée sur le Nom des Disques.

(8) Modification + Ajout Disques -> Choix du Paramétrage :

Mise-à-jour de la base de données des Disques modifiés et ajout des Disques créés, si l'action n'est pas abandonnée par l'utilisateur.

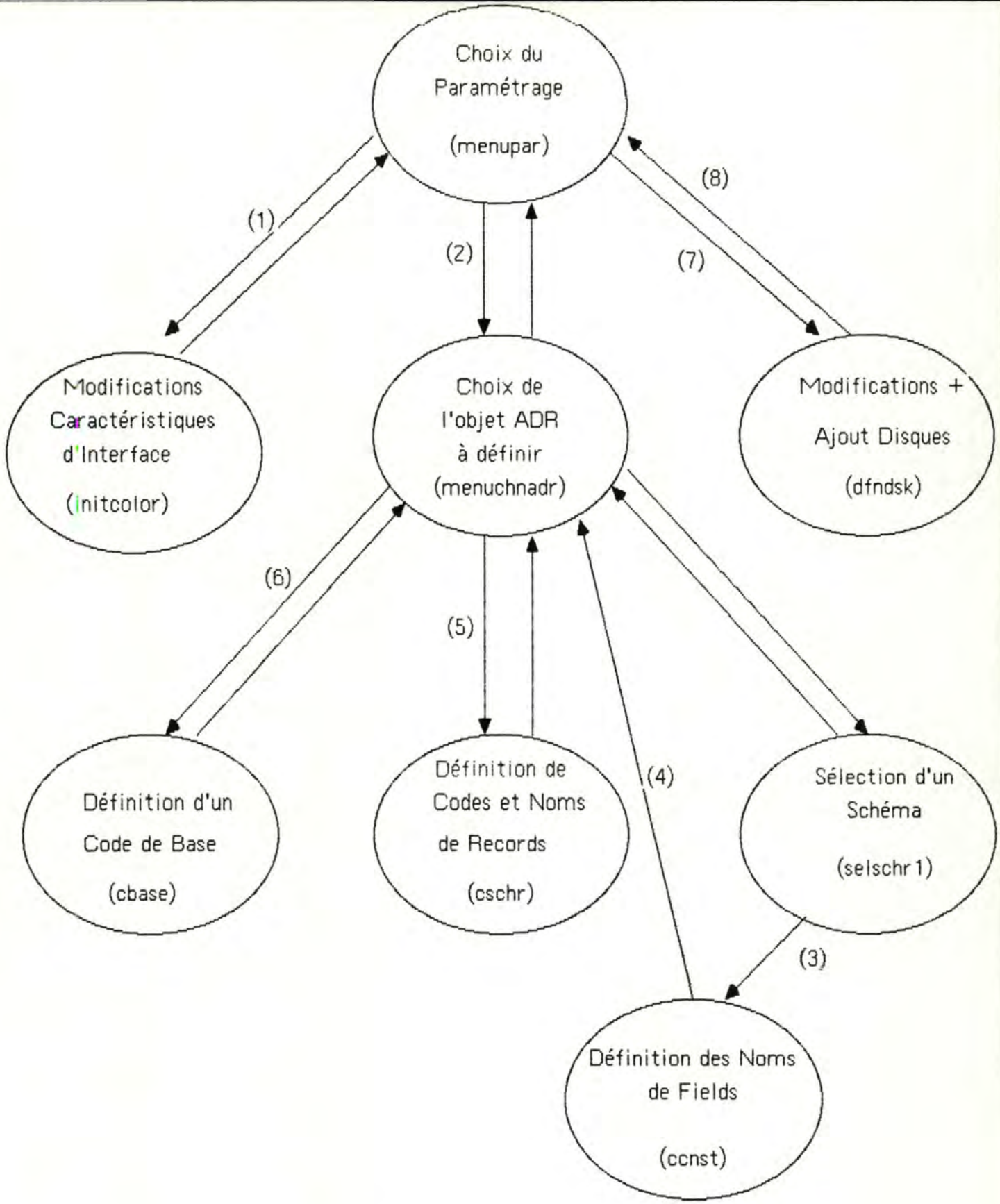


Figure III.3.3.5

Appel :

```
parametrages(etat, chemin)
int *etat, *chemin;
```

Fichier : scenario.c

Présentateur Relationnel

Niveau : 1.

Description :

Ce module est composé d'une seule primitive qui a pour but de réaliser les fonctions qui ont été assigné au Présentateur Relationnel (II.4.4.1) en utilisant les primitives des modules du niveau 1. La figure (III.3.3.6) décrit l'aiguillage vers les états de départ des différents modules du niveau 2. Ce module s'intègre dans le Poste de Travail.

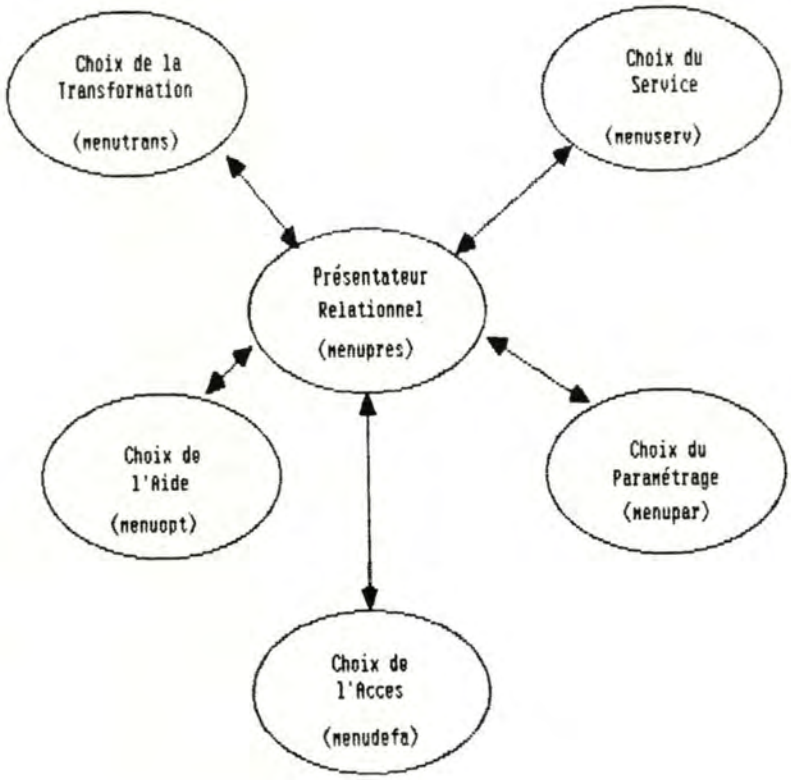


Figure III.3.3.6

Appel :

```
presrel(etat, chemin)
int *etat, *chemin;
```

Fichier : scenario.c

Répercuteur des Correspondances

Niveau : 1.

Description :

Ce module s'occupe de la répercussion des correspondances décrite en II.4.5.2. dans le fichier de nom fichier. Si un fichier de ce nom n'existe pas, il sera créé, sinon il sera ecrasé. A cette primitive correspond un programme ne nécessitant pas l'intervention de l'utilisateur. Ce programme ira lire les correpondances des objets DSL dans la BSR pour écrire ce fichier.

Appel :

```
repercut(etat, chemin, fichier)
int *etat, *chemin;
char *fichier;
```

Fichier : repercut.c

Transformateur IDA/BSR

Niveau : 1.

Description :

Ce module s'occupe de la transformation décrite en II.4.3.1 à partir d'un fichier de nom fichier1. A cette primitive correspond un programme ne nécessitant pas l'intervention de l'utilisateur. Le fichier sera lu séquentiellement et en une seule fois pour réaliser les opérations suivantes (respectant les transformations décrites en II.4.3.1) :

- transformations des Entités et Associations ainsi que leurs Elements et Groupes respectifs,
- stockage des identifiants des Entités dans une liste,
- interprétation de cette liste pour définir les identifiants, clés et nouveaux attributs de chacune de ces Entités,
- définition des identifiants, clés et nouveaux attributs de chaque Association.

Le diagnostic (II.4.3.1) de ces transformations sera décrit dans le fichier fichier2. Si un fichier de ce nom n'existe pas, il sera créé, sinon il sera ecrasé.

Appel :

```
charg_bsr(etat, chemin, fichier1, fichier2)
int *etat, *chemin;
char *fichier1, fichier2;
```

Fichier : charg.c

Poste de Travail

Niveau : 0.

Description :

Ceci est le programme principal à disposition de l'utilisateur. Il permet l'utilisation des différents programmes du niveau 1. Les fichiers utilisés sont définis sous des noms standards qui sont les suivants :

Fichier provenant de l'Extracteur Conceptuel devant être traité par le Transformateur IDA/BSR : IP.DSL

Fichier de diagnostics de cette transformation : IP.ERR

Fichier provenant du Répercuteur des Correspondances : REP.DSL

Fichier de la base de données (13) : BSR0.DBF, BSR1.DBF, BSR2.DBF, BSRT.DBF, BSR0C.NDX, BSR0T.NDX, BSR1C.NDX, BSR2C.NDX, BSR21.NDX, BSR22.NDX, BSRT.NDX, BSR1.NDX.

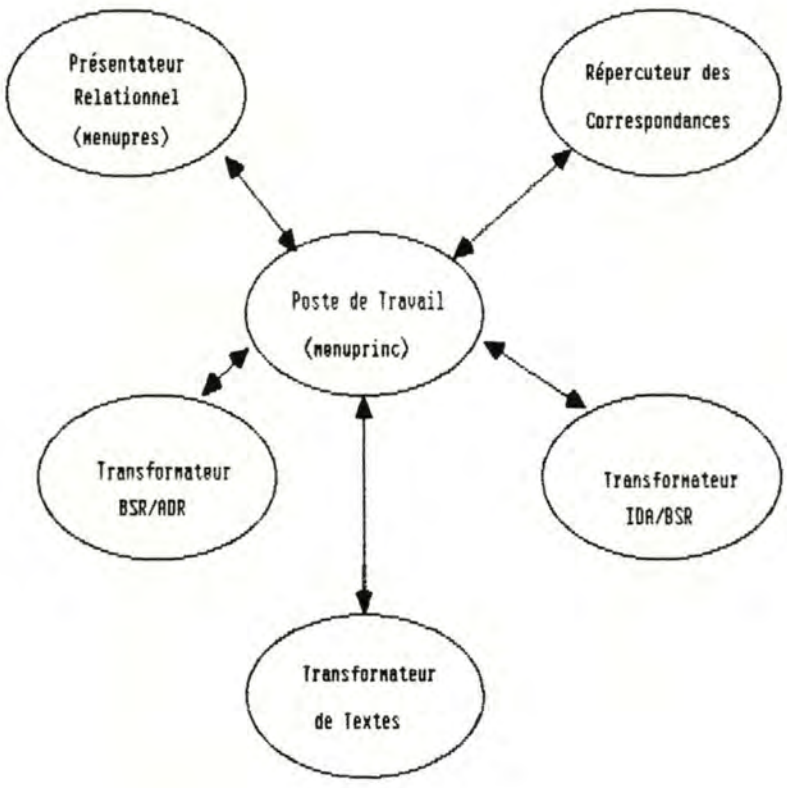


Figure III.3.3.7

une erreur s'est produite lors de l'impression de cette section
veuillez tenir compte de celle-ci. Merci pour votre compréhension

Conclusion

Ce mémoire nous a permis de toucher du doigt les problèmes cruciaux de linterfaçage de logiciels. Nous avons en particulier pu étudier les problèmes provenant du fait que les modèles de représentations des données des deux environnements logiciels à intégrer ne sont pas les mêmes. Ces différences de modèle nécessitent l'utilisation de règles de transformations précises permettant de récuperer les structures de données d'un environnement pour un autre. De plus, comme nous nous situons dans une démarche de conception d'une base de données relationnelle, partant de l'environnement IDA pour réaliser l'analyse conceptuelle et utilisant l'environnement ADR pour la réalisation de la solution adoptée et ce, avec les outils que cet environnement offre, il était également nécessaire d'offrir un environnement qui permette d'adapter la solution conceptuelle en une solution logique.

L'interface qui a été réalisé est encore au stade du prototype et n'est donc pas complet. Il faut en particulier vérifier, au niveau d'un projet pilote, que l'ensemble des fonctions offertes permettent de :

- Réaliser correctement l'interface entre les deux environnement,
- Elaborer correctement une solution logique à partir d'une solution conceptuelle développée sous IDA.

L'extension du mémoire consisterait en premier lieu à terminer les processeurs qui ne l'on pas été :

- l'Extracteur Logique (II.4.3.2),
- l'Editeur des Mises-à-jour (II.4.4.2) et
- le Transformateur de Textes (II.4.5.3).

Une extension supplémentaire comporterait l'adaptation du Transformateur IDA/BSR (II.4.3.1) de manière à ce qu'il fournisse un schéma relationnel moins "brut". On pourrait ainsi tenir compte des transformations proposées par [TE-YA-FR 82] (I.2.6) qui sont nettement plus fines que celles réalisées par ce transformateur et qui permettent une économie de transformations restant à effectuer à l'aide du Présentateur Relationnel.

Une troisième extension consisterait à la prise en compte du schéma conceptuel des traitements. On pourrait ainsi offrir un rapport concernant la spécification des traitements non plus en termes d'occurrences d'objets DSL, mais en termes des occurrences d'objets ADR correspondants. Une aide à la transformation du schéma relationnel, basée sur le schéma conceptuel des traitements, permettrait de prendre en compte les accès aux données lors du choix des transformations. Nous pourrions alors envisager qu'une des fonctions de l'interface permette de définir le nombre d'accès logique nécessaire pour la réalisation par exemple, d'un traitement donné. De même des informations statistiques sur les données pourraient être offertes.

Conclusion

Ce mémoire nous a permis de toucher du doigt les problèmes cruciaux d'interface entre logiciels. Nous avons en particulier pu étudier les problèmes provenant du fait que les modèles de représentations des données des deux environnements logiciels à intégrer ne sont pas les mêmes. Ces différences de modèles nécessitent l'utilisation de règles de transformations précises permettant de récuperer les structures de données d'un environnement pour un autre. De plus, comme nous nous situons dans une démarche de conception d'une base de données relationnelle, partant de l'environnement IDA pour réaliser l'analyse conceptuelle et utilisant l'environnement ADR pour la réalisation de la solution adoptée et ce, avec les outils que cet environnement offre, il était également nécessaire d'offrir un environnement qui permette d'adapter la solution conceptuelle en une solution logique.

L'interface qui a été réalisé est encore au stade du prototype et n'est donc pas complet. Il faut en particulier vérifier que l'ensemble des fonctions offertes permettent de :

- Réaliser correctement l'interface entre les deux environnement,
- Elaborer correctement une solution logique à partir d'une solution conceptuelle développée sous IDA.

L'extension du mémoire consisterait en premier lieu à terminer les processeurs qui ne l'on pas été :

- l'Extracteur Logique (II.4.3.2),
- l'Editeur des Mises-à-jour (II.4.4.2) et
- le Transformateur de Textes (II.4.5.3).

Une extension supplémentaire comporterait l'adaptation du Transformateur IDA/BSR (II.4.3.1) de manière à ce qu'il fournisse un schéma relationnel moins "brut". On pourrait ainsi tenir compte des transformations proposées par [TE-YA-FR 82] (I.2.6) qui sont nettement plus fines que celles réalisées par ce transformateur.

Une troisième extension consisterait à offrir une aide à la transformation du schéma relationnel qui serait basée sur le schéma conceptuel des traitements, permettant de tenir compte des types d'accès aux différentes données lors des transformations. Nous pourrions alors envisager qu'une des fonctions de l'interface permette de définir le nombre d'accès logique nécessaire pour la réalisation par exemple, d'un traitement donné. De même des informations statistiques sur les données pourraient être offertes.

BIBLIOGRAPHIE

[ANT-LEV 84] DE ANTONELLIS, DI LEVA "DATAID-1 : A Database Design Methodology" Information Systems (Vol. 10, No. 2, pp. 181-195) 85.

[BERN 76] BERNSTEIN, "Synthesizing Third Normal Form Relations from Functional Dependencies", ACM Transaction on Database Systems 1.4, 76.

[BOD-PIG 83] BODART, PIGNEUR, "Conception Assistée des Applications Informatiques. 1. Etude d'Opportunité et Analyse Fonctionnelle", Masson 83.

[CADELLI 86] CADELLI, "Atelier de Conception de Base de Données. SPEC-86/7-3 Programmation sur la Base de Données de l'Atelier", FNDP Juil 86.

[CERI 82] CERI, "Methodology and Tools for Data Base Design", North-Holland 83.

[CHA-MUL 86] CHARLOT, MULLER, "Contribution à l'Atelier Logiciel de Conception de Base de Données : Etude de Transformations de Schémas", FNDP Juin 86.

[CHEN 76] CHEN, "The Entity-Relationship Model, Toward a Unified View of Data", ACM TODS (Vol. 1, No. 1), 76.

[CHUNG 81] CHUNG, NAKAMURA, CHEN, "A Decomposition of Relations Using the Entity Relationship Approach to Information Modeling and Analysis", CHEN (ed.) ER Institute, 81.

[CODD 70] CODD, "A Relational Model of Data for Large, Shared, Data Banks", ACM Communications 13,6 p. 377-387 Juin 70.

[DB2G-DS] ADR/DATACOM/DB, "Data Base Design", ADR Inc. Sept 82.

[DB2G-IN] ADR/DATACOM/DB, "Introduction to DATACOM/DB", ADR Inc. Sept 82.

[DD2G-IN] ADR/DATADICTIONARY, "Introduction to DATADICTIONARY", ADR Inc. Sept 83.

[DD2G-TX] ADR/DATADICTIONARY, "Transaction Reference", ADR Inc. Sept 83.

[DD2G-US] ADR/DATADICTIONARY, "Maintaining DATADICTIONARY", ADR Inc. Sept 83.

[DD4G-MG] ADR/DATADICTIONARY, "System Management", ADR Inc. Sept 83.

[DEL-ADI 82] DELOBEL, ADIBA, "Bases de Données et Systèmes Relationnels", Dunod Informatique 82.

[DSL-SPEC 84] IDA "Atelier Logiciel IDA d'Aide à la Conception de Système d'Information Manuel de Référence DSA", FNDP Nov 84.

[FAGIN 77] FAGIN, "Multivalued Dependencies and a new Normal Form for Relational Databases", ACM Transactions on Database Systems 2.3, Sept 77.

[GAT-BER 86] GATELLIER, BERTINCHAMPS, "Système de Pilotage pour la Conception d'un Schéma Conceptuel d'Information", FNDP Sept 86.

[HAINAUT 80] HAINAUT, "Un Modèle Relationnel Généralisé", FNDP Oct 80.

[HAINAUT 85] HAINAUT, "Conception Assistée des Applications Informatiques. 2. Conception de la Base de Données", Masson 86.

[LING 85] LING, "A Normal Form For Entity-Relationship Diagrams", The 4th International Conference on the Entity-Relationship Approach, IEEE Computer Society 85.

[MICHEL 85] MICHEL, "Modèle Relationnel et Conception de Base de Données", Charles Veillon S.A. Avril 85.

[SI2G-01] ADR/IDEAL, "Application Development Reference Manual", ADR Inc. Oct 84.

[TEO-FRY 82] TEORY, FRY, "Design of Database Structures", Prentice Hall 82.

[TE-YA-FR 85] TEOREY, YANG, FRY "Relational Database Design using the ER Model. A Practical Methodology", Computing Research Laboratory, University of Michigan Oct 85.

QUATRIEME PARTIE

ANNEXE

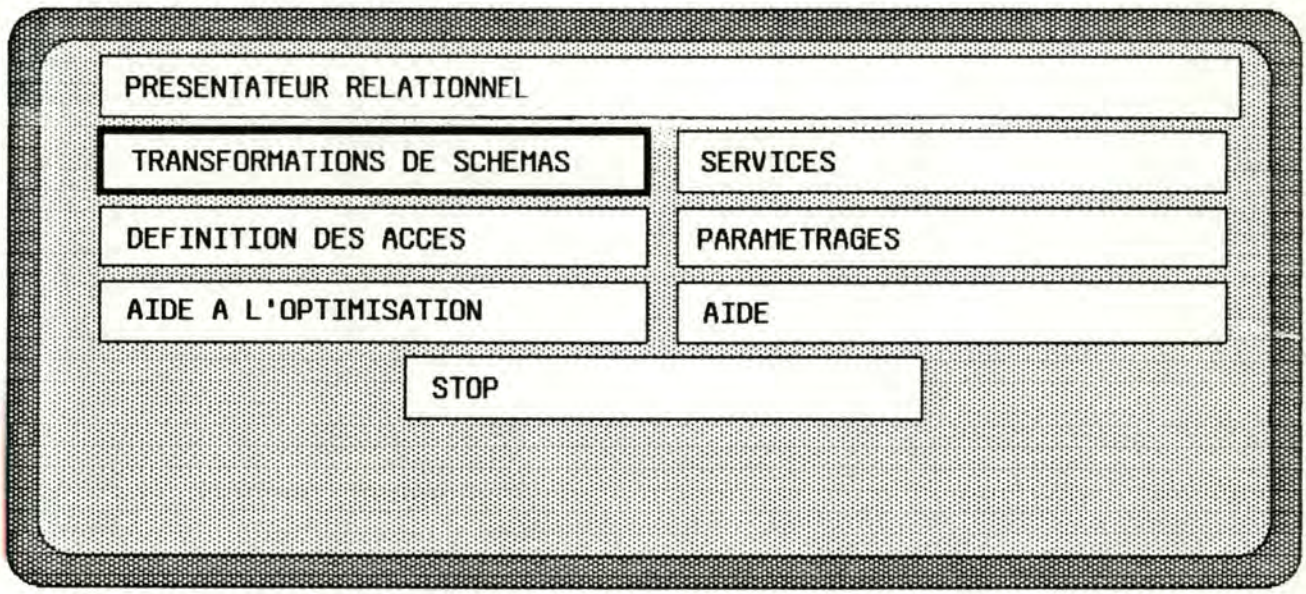
MODE D'EMPLOI

Ce mode d'emploi présente des exemples types qui permettent à l'utilisateur de comprendre les procédures d'utilisation des outils du dialogue. Ces outils sont les menus, les fenêtres et les grilles (cfr III.2.2). La manière d'utiliser et d'interpréter les procédures de dialogue relatives aux fonctions de l'interface ne sera pas expliquée dans ce chapitre. En effet, plutôt que de définir un type d'utilisation arbitraire, il est préférable de laisser le soin à l'utilisateur (ou à un groupe d'utilisateurs) d'interpréter lui-même ces procédures de dialogue et ainsi de le laisser déterminer une utilisation plus fidèle aux notions et au vocabulaire qu'il manie tous les jours. Le contenu informationnel de l'interface étant paramètrable, l'utilisateur peut donc parfaitement y intégrer ses notions et son vocabulaire. Il peut par exemple définir ses propres écrans d'aide. Il peut encore changer le contenu des propositions des menus ou celui des messages d'erreur et de guidance. Ainsi, si l'utilisateur le veut, il peut utiliser le mot "table" plutôt que celui de "schéma de relations" ou encore le mot "fusion" pour le mot "collage".

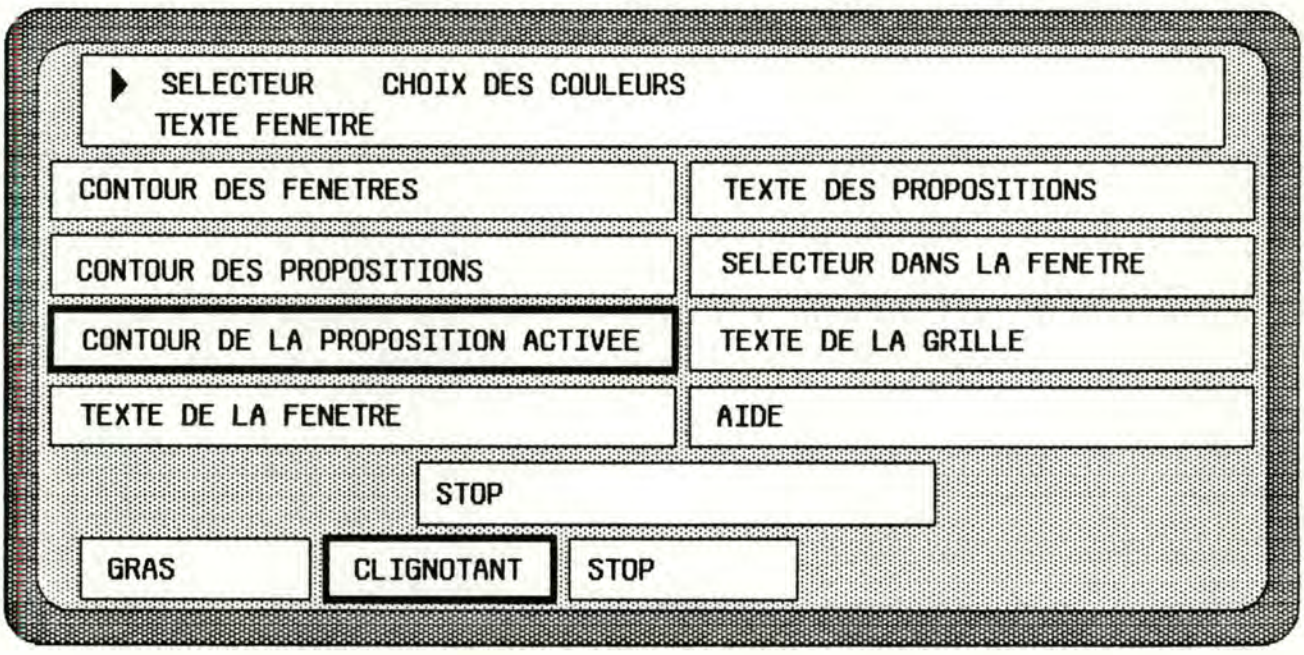
I Utilisation des menus

Les menus sont utilisés pour permettre à l'utilisateur d'orienter l'interface vers l'une ou l'autre de ses fonctions ou pour désigner une action à effectuer dans le cadre de cette fonction.

Exemple I.1 :



Exemple I.2 :



L'exemple I.1 nous montre un menu composé d'alternatives qui permettent de définir la fonction de l'interface vers laquelle l'utilisateur veut s'orienter. Ce menu possède une proposition activée qui est différents des autres soit parce que son contour est plus éclairé, soit parce qu'il clignote. Si l'utilisateur veut faire des transformations de schémas de relations, il doit activer la proposition du menu contenant "TRANSFORMATION DE SCHEMAS". Par contre si il veut utiliser les fonctions de service il doit activer la proposition du menu contenant "SERVICES".

Pour activer une proposition du menu, il faut déplacer le pointeur d'activation vers la proposition voulue au moyen des touches [home], [↑], [end], [→], [←], [pgup], [↓], [pgdn] et ensuite de confirmer son choix au moyen de la touche [CR] ou de la touche [SPACE].

Ces touches permettent de faire voyager le pointeur d'activation dans la direction indiquée par leur position par rapport au centre du clavier numérique. Ainsi, la touche [PgUp] qui se trouve au coin supérieur droit du clavier numérique permet de faire déplacer le pointeur d'activation sur la première proposition située en haut à droite (si elle existe !) par rapport à la proposition activée actuelle.

L'exemple I.2 nous montre quant à lui un menu représentant la fonction du choix des couleurs composé d'actions permettant de réaliser cette fonction. Pour parvenir à ce menu l'utilisateur à du passer par toute une série de menus proposant des alternatives. Si on choisit dans ce menu la proposition "CONTOUR DE LA PROPOSITION ACTIVEE", un deuxième menu s'affichera à l'écran qui permettra à l'utilisateur de choisir une couleur pour la proposition activée c.a.d. "GRAS" ou "CLIGNOTANT" dans le cas présent.

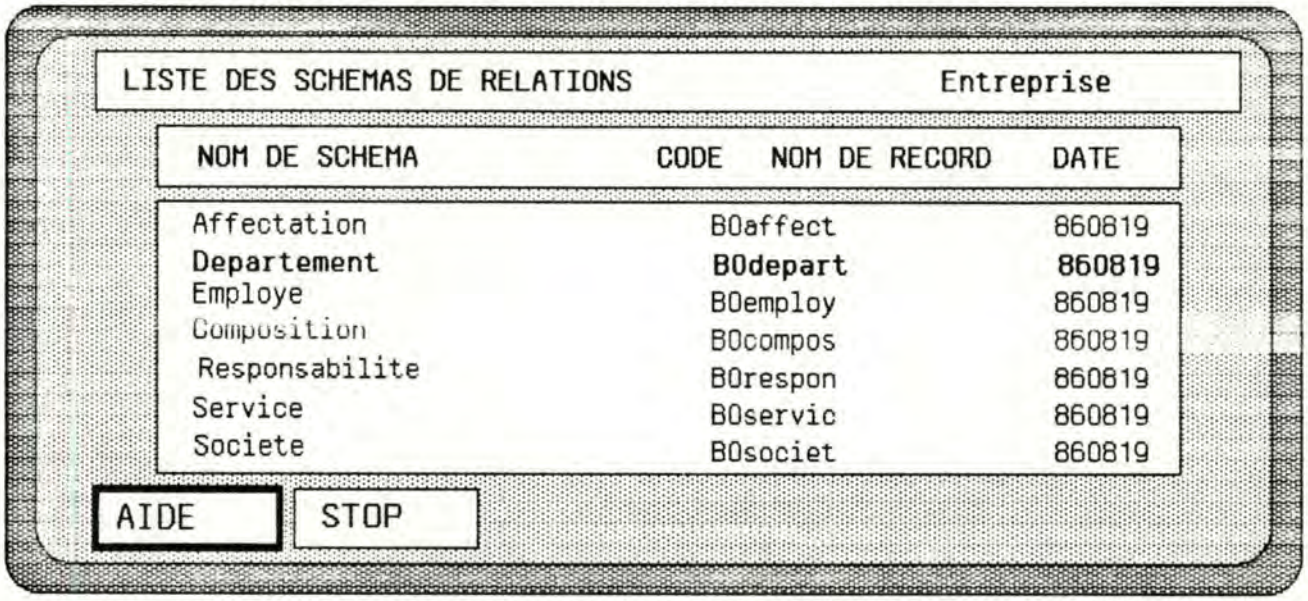
Chaque menu dispose d'une proposition contenant "AIDE". Cette proposition permet de faire afficher l'écran d'aide associé à ce menu. Comme nous l'avons dit ci-dessus le contenu de cet écran doit être définit par l'utilisateur. Il permettra d'expliquer plus complètement chacune des propositions du menu.

Chaque menu dispose aussi d'une proposition contenant "STOP". Cette proposition permet de revenir à tout moment au menu précédent.

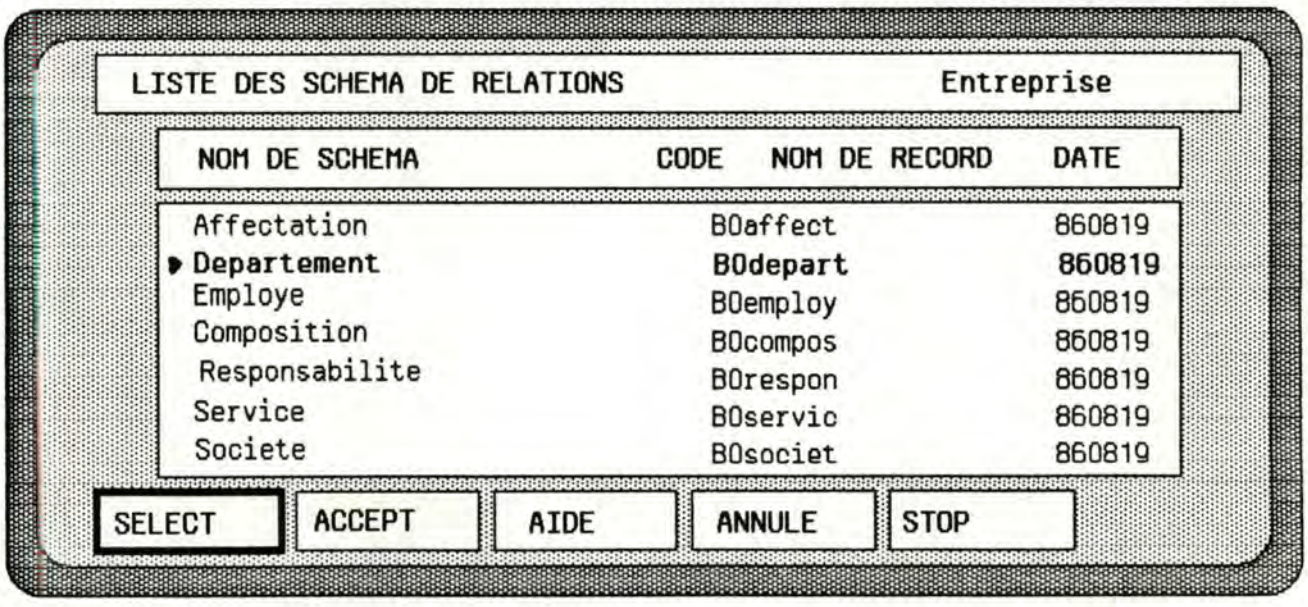
II Utilisation des fenêtres

Les fenêtres sont généralement utilisées pour afficher une liste de données ou de paramètres à l'écran. Cette liste étant généralement trop longue pour la présenter entièrement à l'écran, elle doit être parcourue au moyen d'une fenêtre ouverte sur une partie cette liste. Cette fenêtre peut voyager dans cette liste et permettre ainsi de présenter l'entièrté de celle-ci. A l'écran, une fenêtre est représentée par un cadre. A l'intérieur de ce cadre se trouvent le contenu d'une partie ou de l'entièrté de la liste visée par la fenêtre. Lorsque la fenêtre se déplace sur la liste, c'est en fait l'intérieur de la fenêtre qui change puisque la fenêtre hérite d'un emplacement fixe sur l'écran.

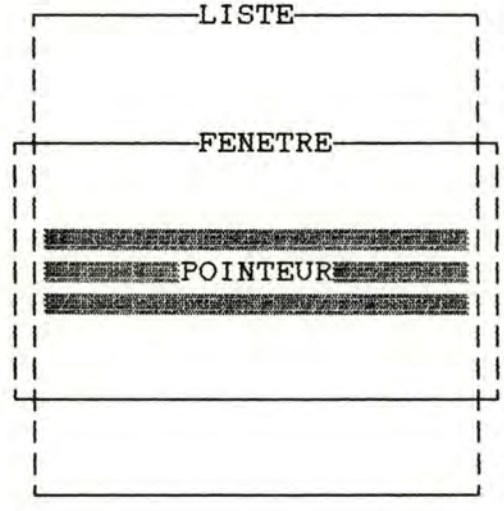
Exemple II.1 :



Exemple II.2 :



Une fenêtre possède un pointeur qui est représenté par une ou plusieurs lignes que l'on distingue des autres car elle(s) possède(nt) une couleur différente. Ce pointeur voyage dans la fenêtre au moyen des touches [↑], [↓], [PgUp] et [PgDn] et quand le pointeur arrive soit à l'extrémité supérieure soit à l'extrémité inférieure de la fenêtre, il permet de faire voyager la fenêtre dans la liste. En effet, si le pointeur est à l'extrémité inférieure de la fenêtre et que l'utilisateur pousse sur [↓], la fenêtre va descendre dans la liste d'un nombre de lignes égal au nombre de lignes composant le pointeur. Si celui-ci pousse sur [PgDn], la fenêtre descendra d'un nombre de lignes équivalent à la taille de la fenêtre. La méthode est la même dans l'autre sens.



L'exemple II.1 nous montre une fenêtre qui est utilisée pour obtenir la liste des schémas de relations d'une base de données. Les occurrences des schémas de relations sont vues ici comme des données car la liste ne sert pas à déterminer une action. Cette liste est plutôt le résultat de l'action qui consiste à afficher tous les schémas de relations d'une base de données déterminée. Dans ce cas le pointeur a la seule fonction de faire voyager la fenêtre sur cette liste pour permettre à l'utilisateur de lire l'entièreté de cette liste. Ce pointeur est manipulé au moyen des touches définies ci-dessus. Les touches [→] et [←] quand à elles sont utilisées pour changer la proposition active du menu qui accompagne toujours une fenêtre et qui permet de définir les actions à effectuer dans ce contexte. Dans cet exemple le menu possède deux propositions. La première permet d'afficher l'écran d'aide par dessus la fenêtre. La seconde permet de revenir au menu précédent.

L'exemple II.2 nous montre quand à lui, une fenêtre qui est utilisée pour présenter à l'utilisateur un ensemble de paramètres. Cette fenêtre est le complément du menu qui se trouvent en dessous. Ce menu propose des actions à effectuer, par exemple "SELECT" qui veut dire "selectionner

un schéma de relations", quand à la la fenêtre, elle propose l'ensemble dans lequel il faut choisir ce schéma de relations. Le pointeur de la fenêtre permet de désigner le schéma de relation dans cet ensemble. Si le pointeur se trouve sur le schéma de relations "Employé" et que l'utilisateur choisit la proposition "SELECT", l'action résultante est celle-ci: "L'utilisateur a sélectionné le schéma de relations Employé". L'utilisateur doit ensuite confirmer ce choix au moyen de la proposition "ACCEPT". La proposition "ANNULE" sert faire l'action inverse de "SELECT".

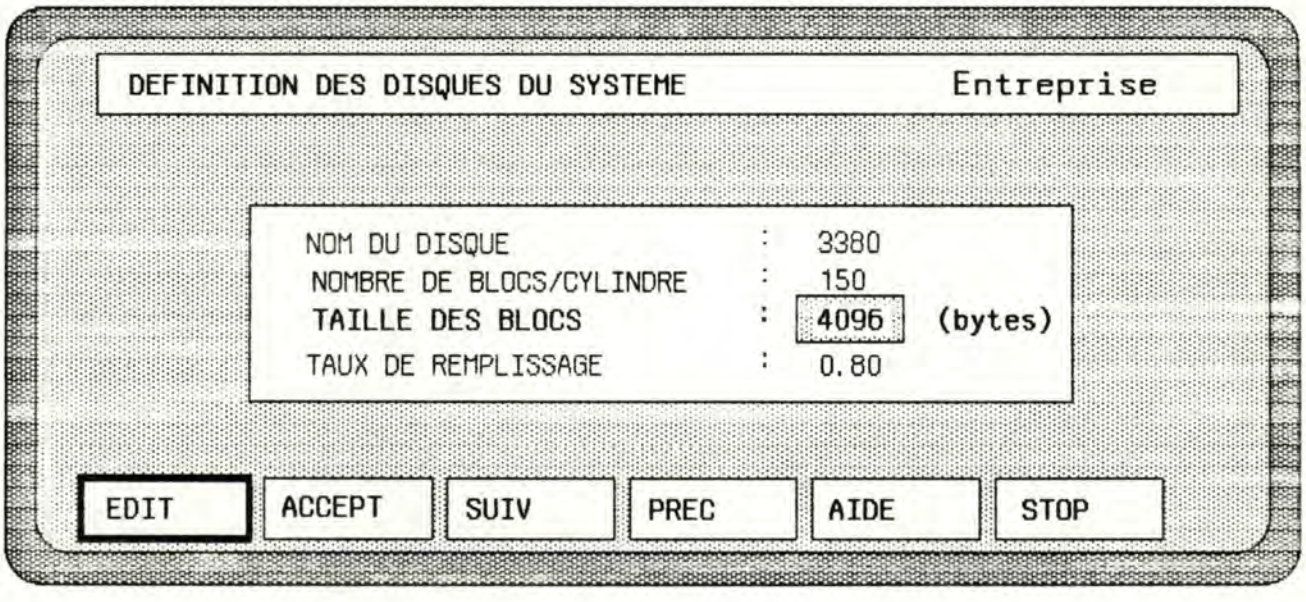
Cet écran est utilisé pour désigner un schéma de relations pour lequel ont veut afficher ses composants.

L'interface utilise aussi des fenêtres pour afficher des messages d'information, de guidance ou d'erreur. Ces fenêtres sont toujours situées en haut de l'écran ou au dessus d'une fenêtre servant à afficher une liste. Ce sont des fenêtres passives, l'utilisateur ne peut que les lire.

III Utilisation des grilles

Les grilles sont utilisées pour permettre à l'utilisateur de rentrer ou de recevoir des données isolées (ou regroupées par thèmes sous un autre forme que celle d'une liste). Elles proposent des zones appelées champs qui servent de zones d'échange de données entre l'utilisateur et l'interface.

Exemple III.1 :



Une grille est composée d'un fond (ou "layout") qui est un ensemble de lignes de caractères et d' un ensemble de champs répartis dans cette grille, accessibles en écriture et en lecture par l'utilisateur. Une grille en utilisation possède une ligne activée et dans cette ligne, un champs activé. Ne peuvent être activées que les lignes qui ont un champs décrit sur elles.

Pour changer de ligne activée il suffit de pousser sur les touches [↑] et [↓] et ainsi faire bouger la ligne activée vers le haut ou vers le bas. Dans la ligne activée, pour changer de champs activé, il suffit d'utiliser la touche [CR]. Pour remplir un champs, il suffit de l'activer et de remplir celui-ci au moyen du clavier. Une fois sur un champs activé les touches [→] et [←] permettent de voyager dans ce champs. La touche [CR] permet aussi de parcourir tous les champs de la grille l'un après l'autre. Enfin, pour sortir de la grille il suffit de pousser sur la touche [Home].

L'exemple III.1 montre une grille qui permet de définir les caractéristiques physiques d'un disque. Cette grille sert à la fois à afficher les données existantes, à changer

ces données ou à en créer de nouvelles. Elle est accompagnée d'un menu qui propose ces différentes actions. En effet, si l'utilisateur choisit l'action "EDIT", il pourra alors effectuer des changements sur les données affichées dans la grille de la manière décrite ci-dessous. Il pourra aussi créer de nouvelles données si ces données sont absentes. Si il choisit l'action "ACCEPT" il confirme les changements qu'il a effectué. Si il choisit les actions "SUIV" ou "PREC", il demande d'initialiser la grille avec les caractéristiques du disque suivant ou du disque précédent.