FUNDP
Institut d'Informatique
Rue Grandgagnage, 21
B-5000 Namur
Belgique

EVOLUTION D'APPLICATIONS DE BASES DE DONNÉES RELATIONNELLES
MÉTHODES ET OUTILS

Jean-Marc Hick

Thèse soumise en vue de l'obtention du grade
de Docteur en Sciences (orientation Informatique)

Jury : Professeur Jean Fichefet, Institut d'informatique (Président)
Professeur Jean-Luc Hainaut, Institut d'informatique (Promoteur)
Professeur Naji Habra, Institut d'informatique
Professeur Jacques Kouloumdjian, INSA de Lyon
Professeur Esteban Zimányi, Université Libre de Bruxelles

26 septembre 2001

RÉSUMÉ

Si les nouvelles technologies en matière de SGBD envisagent le problème de l'évolution du schéma d'une base de données, les systèmes d'information actuels posent des problèmes particulièrement ardus lors des phases d'évolution.

Ce travail étudie ces problèmes dans un contexte pratique tel qu'il est vécu actuellement par les développeurs. Il propose une typologie complète des modifications possibles ainsi qu'une analyse du phénomène de l'évolution et de son impact sur les structures de données, les données et les programmes d'application au travers de stratégies typiques.

Le document décrit ensuite l'environnement de génie logiciel DB-Main avec lequel a été construit un prototype d'outil d'aide à l'évolution. Cet outil permet notamment de générer automatiquement les programmes de conversion de la base de données à partir de la trace des opérations de modification des schémas conceptuel ou logique de la base de données. Il aide aussi le programmeur à modifier les programmes d'application en conséquence.

ABSTRACT

If the recent DBMS technologies consider the problem of databases schema evolution, standard Information Systems in use raise hard problems when evolution is concerned.

This work studies these problems in the current developer context. It gives a complete typology of modifications and analyses the evolution of systems and its impact on the data structures, data and programs through typical strategies.

The document introduces the DB-Main CASE environment, with which an evolution tool prototype has been developed. This tool can automatically generate the conversion programs for the database from the operational trace of the conceptual and logical schema modifications. It can also help the programmer to modify the application programs.

REMERCIEMENTS

Comme tous les travaux de longue haleine, celui-ci n'aurait pas pu être réalisé sans l'aide de certaines personnes.

Tout d'abord, je suis reconnaissant au professeur Jean-Luc Hainaut pour sa constante disponibilité et ses conseils judicieux qui m'ont permis d'éviter de nombreuses voies de recherche sans issue. Nos discussions fertiles ont toujours été une source d'inspiration et de remise en question. Il a toujours su placer ses chercheurs dans un cocon scientifique pour leur donner les meilleures conditions de travail.

J'ai la chance de travailler depuis plusieurs années au sein d'une équipe de recherche spécialisée dans le domaine des bases de données. Tous ces chercheurs ont contribué à la mise en place de méthodes et de technologies, sans lesquelles ce travail n'aurait pu voir le jour. Je profite de l'occasion, pour remercier plus spécialement Vincent Englebert, Jean Henrard et Didier Roland pour leurs avis pertinents, leurs nombreuses compétences et le travail de très haut niveau qu'ils ont accompli dans le développement d'un outil ayant acquis une reconnaissance internationale.

Je remercie également les professeurs Jean Fichefet, Naji Habra, Jacques Kouloumdjian et Esteban Zimányi d'avoir accepter d'être membres du jury de cette thèse. Leurs remarques constructives m'ont permis d'améliorer sensiblement la qualité de ce travail.

Les problèmes administratifs et d'organisation ont été rapidement aplani grâce au savoir-faire de Babette, Christiane, Gisèle et Laura. Je tiens à les remercier pour cela et tout le reste.

Ma famille a également joué un rôle important tout au long de ces cinq années de labeur. Sandra a toujours été là pour me soutenir et me conseiller. Même si le fond lui échappait, elle a su mettre ses dons en exergue pour la forme. Avec Corentin et Nathan, ils ont fait preuve d'une patience largement supérieure aux normes généralement admises. Je leur en serai longtemps reconnaissant et j'espère que l'avenir nous permettra de rattraper le temps perdu. Je remercie également mes parents pour leur soutien affectueux, leur patience et leur dévouement. Sans eux, ce travail n'aurait jamais vu le jour. Grâce à leurs compétences respectives, Liliane, Maria, Marie-Anne, José et Christophe m'ont apporté un soutien essentiel qui n'a fait que renforcer la douce affection qui nous lie.

Enfin, je pense à tous ceux qui m'ont soutenu dans l'élaboration de ce travail et, dont il m'est impossible de donner une énumération sans crainte d'en omettre l'un ou l'autre.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

AVANT-PROPOS

Plus que jamais, les changements sont dans l'air du temps, plus que jamais les réformes, les innovations, les modifications foisonnent dans la vie de tous les jours, plus que jamais les organisations doivent s'adapter à de nouvelles techniques, à de nouvelles circonstances et à de nouveaux défis. La place stratégique occupée par l'informatique dans bon nombre d'entreprises nécessite une répercussion immédiate des changements sur les applications informatiques.

Actuellement, le processus de maintenance est souvent négligé dans le développement d'un nouveau système. Il constitue, avec l'insatisfaction des utilisateurs, les coûts et les délais de développement, un des principaux symptômes de ce qu'on appelle communément la crise du logiciel. Les problèmes de maintenance ne sont pas uniquement dus à des contraintes techniques. La complexité de la réalité organisationnelle d'un système est aussi la cause de nombreuses difficultés. De même, certaines propriétés inhérentes aux logiciels peuvent être à l'origine du problème.

Le constat est sévère : les aspects évolutifs des systèmes d'information automatisés sont insuffisamment couverts par les approches et les outils traditionnels. On ne dispose pas de règles systématiques de traduction des modifications des besoins des utilisateurs en modifications des composants techniques de l'application. Au niveau des outils, les concepteurs se voient offrir des ateliers de génie logiciel qui ignorent, pour la plupart, les phases de maintenance et d'évolution.

«*Aider l'ingénieur dans sa tâche de maintenance d'un système informatique !*» : voilà l'objectif ambitieux de ce travail et la principale motivation qui nous a poussé à choisir ce sujet de recherche. Pour atteindre cet objectif, il a fallu préciser le problème tant le sujet est vaste. D'abord, on s'attaque aux applications de bases de données dans lesquelles le composant données persistantes (ou base de données) est central. Ensuite, on considère que l'évolution de ces applications provient de changements dans les besoins de l'entreprise ou de l'apparition de nouveaux besoins qui se traduisent par des modifications techniques de la base de données. Tout le problème réside dans la propagation des modifications dans les bases de données et les programmes. Finalement, pour développer des outils concrets, la démarche adoptée est basée sur le modèle de données relationnel. Elle est applicable à des systèmes développés à l'aide de langages standards de troisième génération tels que COBOL/SQL ou C/SQL.

L'objectif a-t-il été atteint ? La thèse propose un contexte de travail, une étude typologique des modifications, une méthodologie basée sur des stratégies de propagation des modifications, des recommandations pour la conception de systèmes flexibles et des outils qui visent à supporter la maintenance et l'évolution d'une application. Malgré une instabilité inhérente à toutes nouvelles méthodologies et un manque de mises à l'épreuve des outils dans des situations réelles, la solution théorique et le prototype développés permettent de soulager autant que possible le travail des développeurs dans la phase de maintenance, ce qui m'autorise à penser que l'objectif fixé initialement a été atteint.

«Varium et mutabile.»

Virgile, l'Enéide, IV, 569

CHAPITRE 1 INTRODUCTION GÉNÉRALE

«*Les temps changent. Le monde bouge*». Combien de fois n'a-t-on pas entendu ces paroles ?

Les organisations subissent également la pression de changements produits par des forces environnementales. La plupart des organisations utilisent des systèmes pour gérer les informations qu'elles brassent. Ces systèmes sont supportés par un ensemble d'applications indépendantes ou intégrées. Ces applications doivent évoluer très rapidement pour satisfaire les nouveaux besoins et continuer à aider les organisations à maintenir voire même étendre leurs activités.

Or, une crise majeur révèle, depuis de nombreuses années, les problèmes liés au développement et à la maintenance dans l'ingénierie du logiciel. Le constat est grave : les aspects évolutifs des systèmes d'information automatisés sont insuffisamment couverts par les approches et outils traditionnels.

La section 1.1 de ce chapitre rappelle les enjeux de la maintenance des systèmes d'information. Elle précise les tenants et aboutissants de la maintenance, aborde la crise du logiciel révélée par des symptômes aussi disparates que les dépassements de délais et de coûts de développement ou l'insatisfaction des utilisateurs et constate l'impuissance des chercheurs à résoudre le problème.

La section 1.2 propose un état des recherches dans le domaine. Elle se focalise sur la recherche dans la maintenance que ce soit au niveau des logiciels (point 1.2.1), des bases de données traditionnelles (point 1.2.2) et orientées objets (point 1.2.3). Le dernier point s'intéresse brièvement au domaine de la rétro-ingénérie qui est intimement lié à celui de la maintenance.

La contribution de la thèse vis-à-vis des problèmes rencontrés est précisée dans la section 1.3. Pour mieux cerner le problème, la section 1.4 propose un exemple intuitif qui met en avant les points délicats du processus de maintenance d'une application de base de données relationnelles. Et finalement, la section 1.5 donne un aperçu du contenu de ce travail.

1.1 Maintenance des systèmes d'information

1.1.1 Monde en perpétuelle évolution

Le monde dans lequel nous vivons est en perpétuelle évolution. Personne ne peut nier cette affirmation. Pour s'en convaincre, il suffit d'observer les choses qui nous entourent. L'émergence des nouvelles technologies (notamment dans le domaine des communications), les modifications sociales (la structure familiale, l'évolution démographique, ...), les changements comportementaux des consommateurs sont autant de facteurs qui ont des influences directes ou indirectes sur la société et les organisations qui en dépendent.

Dans ce contexte, les organisations doivent s'adapter aux modifications de leurs environnements. La théorie organisationnelle sépare généralement les forces qui induisent des changements dans une organisation en deux catégories : celles provenant d'environnements internes et celles provenant d'environnements externes. Krogstie [Kro95] propose une typologie assez complète des changements environnementaux qui ont des répercussions sur une organisation (figure 1.1). Au delà de la décomposition interne/externe, il dégage d'autres catégories.

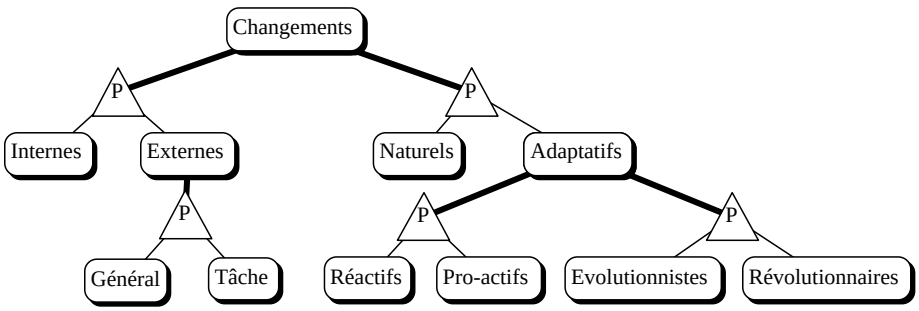


Figure 1.1 - Typologie des changements : les catégories forment des partitions représentées par un triangle marqué de la lettre P.

Tout d'abord, les *changements internes* proviennent d'un milieu interne à une organisation (les catégories de travailleurs, la direction, le personnel, ...). Alors que les *changements externes* proviennent d'un milieu externe à une organisation (la société de consommation, les instances politiques, les groupes de pression, ...). Cette catégorie est également divisée en deux. Les *changements concernant l'environnement général* contiennent les composants d'un environnement externe qui ont des effets indirects sur une organisation. La globalisation de l'économie, le vieillissement de la population, le changement de politique (consommation durable) peuvent influencer indirectement une organisation. Les *changements concernant l'environnement de la tâche* contiennent les composants d'un environnement externe qui interagissent directement avec une organisation. Les clients, les concurrents, les fournisseurs sont des composants importants de cet environnement. Un changement dans les habitudes de consommation, une concurrence acharnée ou l'apparition de nouveaux produits peuvent avoir des impacts considérables sur une organisation.

Ensuite, les changements, qu'ils soient externes et internes, peuvent être divisés en deux autres groupes. Les *changements naturels* apparaissent soit parce qu'ils sont engendrés par des acteurs organisationnels soit parce qu'ils sont dans l'ordre des choses. L'augmentation de la taille d'une organisation est un changement naturel interne. Les *changements adaptatifs* sont exécutés consciemment pour s'adapter aux changements des environnements externes ou internes. Ces derniers changements sont soit réactifs, soit pro-actifs. Les *changements réactifs* tentent de répondre aux changements adaptatifs qui surviennent. Tandis que les *changements pro-actifs* ont pour but d'atteindre de nouveaux objectifs en saisissant par exemple de nouvelles

opportunités. Les changements adaptatifs sont aussi évolutionnistes ou révolutionnaires. Les *changements évolutionnistes* sont dans le cadre de l'évolution normale du fonctionnement d'une organisation. Alors que les *changements révolutionnaires* sont le résultat d'une réévaluation en profondeur des grandes lignes des activités d'une organisation.

Depuis leur apparition dans les années 50, les ordinateurs sont devenus des outils-clés pour la gestion des informations dans une organisation. Les progrès technologiques phénoménaux réalisés depuis un demi-siècle en ont fait des outils de travail incontournables. La plupart des organisations ont des systèmes d'information plus ou moins complets pour supporter leurs tâches et, notamment, distribuer les données par le biais de l'informatique. Les changements survenant sur l'environnement d'une organisation entraînent obligatoirement des modifications au niveau des informations et de leur gestion. Les modifications organisationnelles vont de pair avec la maintenance et l'évolution des systèmes d'information.

Dans un contexte de société en perpétuelle évolution, les organisations doivent s'adapter aux changements en répercutant les modifications au niveau des applications [1] qui composent les systèmes d'information. La place stratégique occupée par l'informatique dans bon nombre d'entreprises engendre une répercussion immédiate des changements (de leur environnement) aussi bien sur le matériel que sur les applications informatiques.

1.1.2 Maintenance des logiciels

1.1.2.1 Cycle de vie d'un logiciel

Avant d'analyser la maintenance, il est utile de la replacer dans le cycle de vie complet d'un logiciel. La figure 1.2 illustre un modèle classique de développement d'un logiciel ([Bat92], [Som92]) dont les principales étapes sont :

– l'analyse et la définition des besoins qui établissent les services, les contraintes et les objectifs du système en collaboration avec les utilisateurs;

– la conception du système qui répartit les besoins entre les parties logicielles et matérielles et qui définit une architecture générale;

– l'implémentation qui réalise la conception du logiciel comme un ensemble de programmes testés séparément afin de vérifier le respect des spécifications;

– l'intégration des programmes dans un système complet et les tests qui vérifient l'intégrité globale;

– la mise en route du système et sa maintenance qui impliquent la correction des erreurs non détectées dans les étapes précédentes, l'amélioration du système et le développement de nouveaux services.

En pratique, les différentes étapes du développement se chevauchent. Le processus n'est pas un modèle linéaire, il implique des retours en arrière et des itérations sur les phases de développement. Ces fréquentes itérations compliquent la planification et la gestion des tâches. Pour éviter ces complications, les concepteurs figent certaines parties du développement après quelques itérations, ce qui engendre parfois des systèmes mal structurés (par exemple suite à des besoins insuffisamment spécifiés).

Le logiciel devient opérationnel dans la dernière phase du processus durant laquelle des erreurs de spécification, de programmation ou de conception sont découvertes et de nouvelles fonctionnalités sont identifiées. Des modifications sont alors nécessaires pour que le logiciel soit utilisable. La réalisation de ces modifications constitue la *maintenance logicielle*. Le processus de développement est répété complètement ou partiellement pendant la phase de maintenance.

1. La littérature utilise régulièrement le terme de "Computerized Information System" (CIS) que nous traduirons par Système d'Information Automatisé.

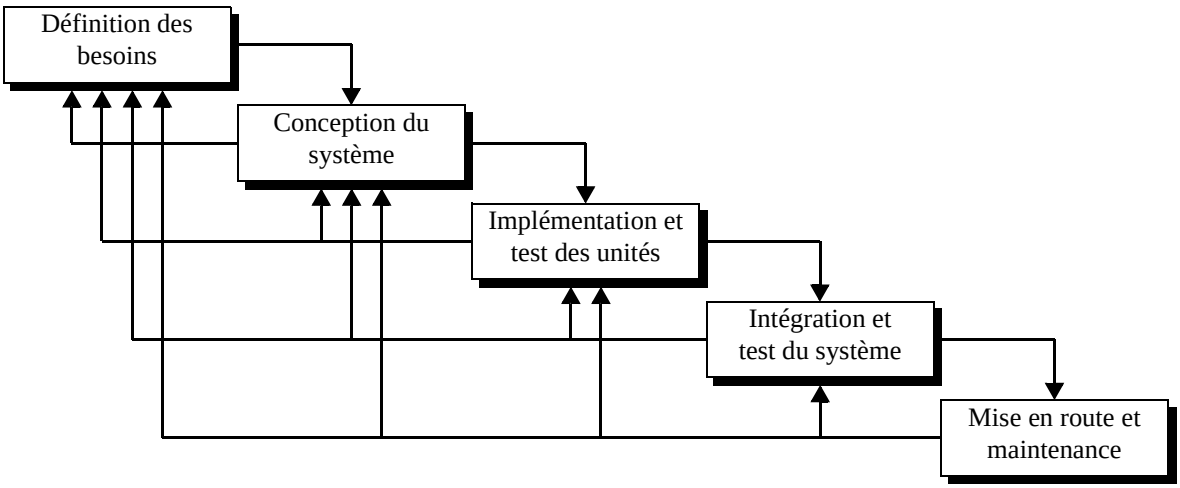


Figure 1.2 - Modèle classique du cycle de vie d'un logiciel.

Ce travail s'intéressant plus particulièrement à l'évolution d'applications de bases de données, il faut également présenter le cycle de vie de conception des bases de données qui constituent les parties centrales des applications considérées. Le cycle présenté à la figure 1.3 s'inscrit également dans le cadre d'une modélisation classique qui envisage la conception d'une base de données comme un processus transformant l'expression des besoins en une collection de spécifications et de programmes. La démarche de conception se décompose en :

- l'étude des besoins qui délimite le domaine d'application;
- l'analyse conceptuelle qui donne une spécification formelle et non technique des informations;
- la conception logique qui élabore les structures de données en fonction du modèle logique choisi et la conception physique qui spécifie les caractéristiques techniques d'implémentation de la base de données et produit du code exécutable;
- le développement des procédures d'exploitation qui produit les procédures d'aide au développement, de démarrage et d'arrêt du système, de protection contre les incidents, ...;
- l'implantation qui construit les structures de données de la base et qui charge les données initiales;
- l'utilisation qui met le système en exploitation;
- la maintenance qui modifie les structures de données et convertit les données et programmes;
- le recyclage qui migre, intègre ou recycle les composants du système. La base de donnée peut aussi être abandonnée.

Comme dans le cycle de vie d'un logiciel, la maintenance d'une base de données est l'ultime phase de la conception qui entraîne une révision totale ou partielle du processus. La maintenance constitue une étape primordiale du processus de conception d'un logiciel comme la suite de ce chapitre va le montrer.

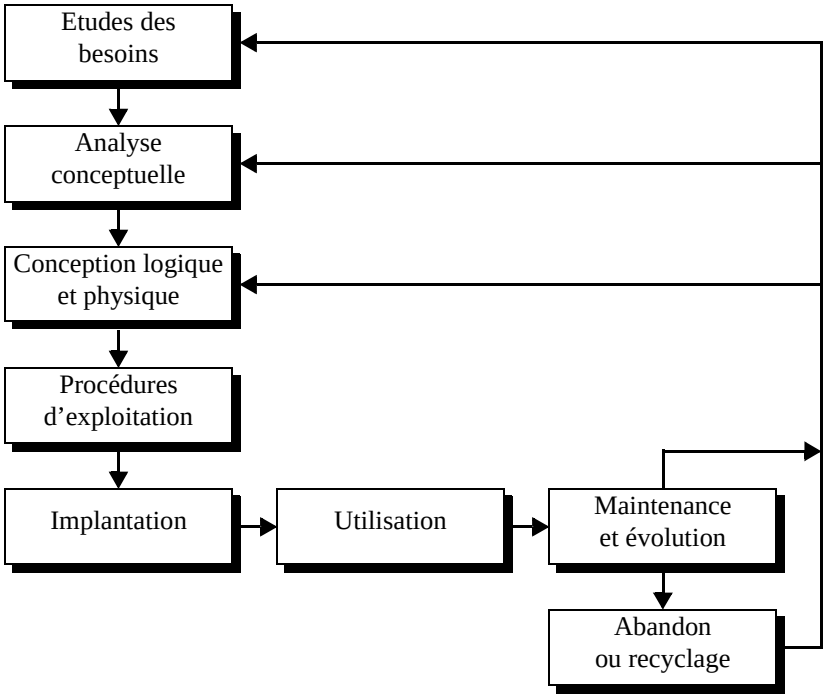


Figure 1.3 - Modèle classique du cycle de vie d'une base de données.

1.1.2.2 Définition

Les systèmes d'information doivent être maintenus pour qu'une organisation puisse assumer les objectifs qu'elle s'est fixés. Historiquement, le terme de maintenance a été utilisé pour désigner le processus de modification d'un programme après sa mise en œuvre. Schneidewind [Sch87] définit la maintenance des logiciels comme la «*modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment*» [2].

Habituellement, la maintenance des logiciels se décompose en trois catégories ([Swa76], [Som92], [Kro95], [Bur00]) :

1. La *maintenance corrective* identifie et corrige les erreurs de conception ou d'implémentation d'un programme.
2. La *maintenance adaptative* permet d'adapter un programme aux changements techniques de son environnement (nouvelles versions de logiciels, renouvellement ou amélioration du matériel informatique, ...).
3. La *maintenance perfective* augmente les performances, ajoute de nouvelles fonctionnalités ou améliore la flexibilité du système. Krogstie [Kro95] distingue deux sous-catégories :
   a) La *maintenance perfective fonctionnelle* implique des modifications relatives à des fonctionnalités du système de manière à augmenter le champ d'action des utilisateurs.
   b) La *maintenance perfective non fonctionnelle* améliore la qualité du système en termes de flexibilité ou toutes autres caractéristiques importantes aux yeux du développeur et de l'utilisateur.

Certains auteurs ([Bur00], [Kem97]) utilisent le terme de *maintenance préventive* pour désigner l'amélioration et la restructuration d'un programme en vue d'une meilleure flexibilité, ce qui peut être classé dans la maintenance perfective non fonctionnelle.

2. Il s'agit de la définition standard de ANSI/IEEE.

Selon des études réalisées par Lientz et Swanson [Lie80] fin des années septante, 65 pour-cent des coûts de maintenance sont destinés à la maintenance perfective, 18 pour-cent à la maintenance adaptative et 17 pour-cent à la maintenance corrective. Au-delà des chiffres qui dépendent essentiellement du type des applications prises en compte, on constate que les erreurs de codage sont généralement faciles à corriger, les problèmes de conception sont plus lourds à résoudre car ils impliquent la réécriture de certains composants des applications et les problèmes d'interprétation au niveau des besoins sont les plus difficiles à réparer car ils nécessitent une nouvelle conception.

1.1.2.3 Modèle de la maintenance

La figure 1.4 propose un modèle classique pour le processus de maintenance qui est adapté du modèle de Sommerville [Som92]. Sur la base de changements des besoins engendrés par l'évolution des environnements d'une organisation, le processus de maintenance est mis en œuvre. Après une analyse des impacts et des coûts de la maintenance, sous réserve d'acceptation des modifications proposées, une nouvelle version du système est planifiée. Les changements sont implémentés et validés de manière à créer une nouvelle version du système intégrant les changements demandés.

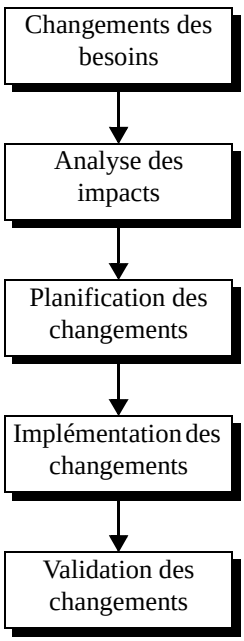


Figure 1.4 - Un modèle pour le processus de maintenance des logiciels.

1.1.2.4 Coût

Le coût de la maintenance d'un système informatique est difficile à estimer. Il varie fortement en fonction du domaine d'application. En règle générale, le coût de la maintenance représente approximativement entre 50 et 80 pour-cent du coût total du système [Som92] [Wie95]. La maintenance de logiciel est une activité très coûteuse qui est souvent mal comprise, sous-estimée et sous-évaluée. Pour s'en convaincre, il suffit de revenir deux ans en arrière et de se souvenir des problèmes posés par la problématique de l'an 2000. Au départ, un problème qui semblait relativement simple s'est transformé en une crise logicielle majeure comme le souligne à juste titre [Ulr97].

Plusieurs facteurs classés en deux grandes familles influencent les coûts investis dans la maintenance. La première famille concerne les facteurs indépendants des techniques utilisées pour le développement du système comme :

- la stabilité du staff technique : le concepteur d'un programme le maintient mieux qu'une personne extérieure aussi expérimentée soit-elle;
- la durée de vie du programme : plus un programme est âgé, plus il a été maintenu, plus sa structure a été dégradée et plus il est difficile à maintenir;
- la dépendance vis-à-vis de l'environnement extérieur : il est clair qu'un programme de gestion des clients est plus difficile à maintenir qu'un programme de traitement de texte,
- la stabilité du matériel : les programmes sont conçus pour une configuration particulière, tout changement de matériel entraîne des modifications.

La deuxième famille concerne les facteurs dépendants de l'approche adoptée et des techniques utilisées pour le développement d'un programme. Citons par exemple :

- l'indépendance modulaire qui autorise la modification de modules spécifiques sans toucher aux autres modules;
- la qualité de la documentation du programme et des mises à jour qui a un impact déterminant sur la maintenance des programmes;
- les langages de programmation de haut niveau qui sont plus faciles à maintenir car ils sont plus facilement compréhensibles;
- le style de programmation qui influence la compréhension des programmes et leurs modifications;
- la validation et les tests qui permettent de minimiser la maintenance corrective.

1.1.3 Crise du logiciel

Les progrès technologiques réalisés sur les composants matériel ont augmenté la puissance des machines ce qui a permis la résolution de problèmes organisationnels de plus en plus complexes. Parallèlement, on a constaté que la mise en œuvre et le support des systèmes d'information automatisés posent d'énormes problèmes. Les coûts et les délais de développement, les efforts fournis pour maintenir un système et l'insatisfaction de beaucoup d'utilisateurs constituent les principaux symptômes de ce qu'on appelle la *crise du logiciel*. Cette crise bien connue dans le monde informatique a été décrite et analysée à maintes reprises notamment dans [Con96], [Gib94], [Kro95], [Wie95].

Depuis toujours, la maintenance des logiciels est négligée dans le développement d'un nouveau système [Kem97]. Dans [Lie80], Lientz et Swanson affirment que la «*Maintenance of application software has often been treated as a neglected stepchild to new systems development. The former is the product of past romances; new systems represent current hopes and dreams.*» Actuellement, le constat est tout autre. Beaucoup d'organisations, employant des systèmes relativement anciens, hésitent à évoluer vers des applications plus modernes par crainte des coûts, des délais ou d'autres problèmes posés par la migration. L'existant est, somme toute, imparfait mais il fonctionne correctement, tandis que l'avenir est incertain et parfois voué à l'échec.

Comme on l'a vu auparavant, les problèmes de maintenance rencontrés par les systèmes d'information automatisés ne sont pas uniquement dus à des contraintes techniques. La complexité de la réalité organisationnelle d'un système semble être aussi la cause de nombreuses difficultés. Brooks [Bro86] constate que certaines propriétés inhérentes aux logiciels (elles sont au nombre de quatre) sont à l'origine des problèmes. Nous allons les reprendre brièvement dans la perspective de ce travail.

a) La complexité

Les logiciels peuvent être très complexes car il y a peu de composants semblables qui peuvent être réutilisés. Ils diffèrent des objets manufacturés (voiture, ordinateur, appareil électroménager,

...) où les composants identiques abondent. De plus, les logiciels collent souvent à une réalité complexe ce qui se traduit par une multitude de cas à traiter.

De cette complexité viennent des difficultés de communication entre les personnes concevant un programme (entraînant des dépassements de délais et de coûts), des problèmes pour traiter tous les cas possibles (engendrant un manque de fiabilité) et des difficultés d'extension des logiciels par de nouvelles fonctionnalités.

b) La conformité

Une grande part de la complexité des logiciels est arbitraire car elle est imposée par les institutions humaines ou les systèmes mis en place avec lesquels ils doivent être conformes. Par exemple, un logiciel sur la fiscalité est souvent complexe en raison de la législation régissant ce domaine. La complexité ne peut pas être seulement diminuée par une simplification des programmes.

c) Le changement

Les modifications sont comme une épée de Damoclès suspendue au-dessus des logiciels. Ils sont constamment sous la pression de modifications. Au contraire, les objets manufacturés sont rarement modifiés après leur fabrication et leur mise en vente (excepté les produits présentant un vice de fabrication). Les logiciels couramment utilisés doivent évoluer à cause des pressions exercées par les utilisateurs en vue de les améliorer ou des changements dans leur environnement technique qui les obligent à se conformer aux nouvelles exigences matérielles ou logicielles.

d) L'invisibilité

Malgré les progrès réalisés dans les structures des logiciels, elles restent très éloignées des schémas mentaux humains. Il est impossible de visualiser la structure d'un programme comme le plan d'une maison qui permet à l'architecte et à son client d'évaluer la dimension et l'agencement des pièces. Les représentations des structures logicielles actuelles ne permettent pas à l'esprit humain d'utiliser sa compréhension des représentations géométriques comme il le fait habituellement pour des cartes géographiques, des plans, des schémas électriques, ... La structures d'un programme se représente par le biais de plusieurs diagrammes (de flux, de données, des dépendances, ...) qui se superposent et s'entremêlent les uns avec les autres. Ces diagrammes sont trop complexes pour que l'esprit humain puisse les contrôler totalement. Ce manque de visibilité gêne le processus de conception et entrave la communication entre les personnes impliquées dans la maintenance d'un programme.

1.1.4 Constat

Depuis une trentaine d'années, les chercheurs réfléchissent et mettent au point des techniques, des méthodes et des outils pour supporter le concepteur dans toutes les étapes du cycle de vie d'un système d'information. Ils travaillent dans des domaines aussi variés que la conception, la maintenance, la documentation, la rétro-ingénierie[3] de systèmes, l'analyse des besoins ou la définition de méthodologies. Pourtant, tout le monde s'accorde pour dire que les résultats obtenus dans les disciplines de l'ingénierie des logiciels ont une faible pénétration dans le développement d'applications réelles. Les raisons de cet échec sont multiples et complexes. Bien que non exhaustive, la liste ci-dessous présente quelques raisons importantes de l'insuccès des recherches :

3. Pratique qui consiste à étudier un produit fini (comme un programme) pour connaître la manière dont celui-ci a été conçu (concept approfondi au point 1.2.4).

- Le processus de construction de la réalité propre à chaque individu (en particulier les concepteurs) est négligé dans la plupart des méthodologies de développement. Il n’est pas assez pris en compte comme support à la discussion et à la consolidation de modèles.
- Les méthodes existantes sont souvent lourdes et difficiles à mettre en œuvre. Dans les organisations, c’est la rentabilité qui prime souvent au détriment de la qualité.
- Les concepteurs ne sont pas suffisamment formés à la mise en œuvre et à l’utilisation d’outils complexes.
- Les outils et les approches traditionnels ne couvrent pas les aspects évolutionnistes des systèmes d’information automatisés.

Une des raisons primordiales du problème de la maintenance est probablement le manque de support au niveau des phases critiques dans le cycle de vie d’un système informatique. Confronté au problème de la maintenance, le concepteur est démuni tant du point de vue méthodologique que du point de vue technique. En effet, on ne dispose pas à l'heure actuelle de règles systématiques de traduction des modifications des besoins auxquels satisfait un système en modifications des composants techniques de l'application. On ne dispose même pas de recommandations méthodologiques garantissant des applications faciles à maintenir. Au niveau des outils, le développeur se voit offrir des AGL [4] qui ignorent pour la plupart les processus de maintenance et d'évolution. En ce qui concerne les bases de données, ces outils permettent de construire un schéma conceptuel, de le transformer de manière automatique en schéma logique et de générer les structures de la base de données. Le code généré doit souvent être remanié de manière significative pour devenir véritablement opérationnel et toute modification des spécifications entraînera la modification du schéma conceptuel, la transformation en un nouveau schéma logique et la production d'un nouveau code. Ce nouveau composant est sans lien formel avec la version précédente. La conversion des données et la modification des programmes est entièrement à la charge du développeur.

4. Atelier de Génie Logiciel ("CASE tool" en anglais).

1.2 Etat de l'art

L'objectif de cette section est de faire un rapide tour d'horizon concernant les recherches importantes accomplies dans le domaine de la maintenance de systèmes d'information automatisés. Le point 1.2.1 donne un bref aperçu des travaux réalisés dans le vaste domaine de la maintenance des logiciels. Les deux points suivants s'intéressent plus particulièrement aux travaux réalisés dans le cadre de la maintenance des bases de données en distinguant les bases de données traditionnelles (point 1.2.2) des bases orientées objets (point 1.2.3). Le dernier point (1.2.4) s'intéresse aux travaux réalisés dans un domaine étroitement lié au problème de la maintenance : la rétro-ingénierie.

1.2.1 Maintenance des logiciels

Depuis une trentaine d'années, la littérature, les conférences et les recherches abondent dans le domaine de la maintenance qui est un cadre de réflexion important dans l'ingénierie du logiciel. L'objectif est de donner un bref aperçu des recherches passées et présentes sans toutefois prétendre à une exhaustivité.

1.2.1.1 Techniques de conception de logiciels

Au début des années septante, Parnas [Par72] [Par79] affirme que les programmeurs doivent rendre leur logiciel plus facile à maintenir en prenant en compte les changements futurs qui pourraient survenir. Pour réaliser cela, il propose une technique d'encapsulation des informations pour séparer les éléments susceptibles de changer des éléments stables de l'application de manière à mieux localiser les modifications. Les chercheurs en maintenance ont toujours essayé de trouver des techniques de programmation permettant d'augmenter la flexibilité des programmes.

Plus récemment, Wiederhold [Wie95] s'intéresse à l'influence de l'architecture des systèmes (clients/serveurs) sur leur capacité à évoluer. Il introduit le concept de *médiateur* entre les clients et le serveur qui est un serveur d'information transformant les données du serveur en données nécessaires pour les clients. Les médiateurs augmentent la flexibilité de l'architecture en concentrant les informations dans des modules autonomes créant les objets informationnels des données sources ce qui réduit le coût de la maintenance.

1.2.1.2 Modèles et méthodes

Au-delà de techniques permettant d'assurer une meilleure capacité d'évolution des systèmes informatiques, des modèles et méthodes ont été mis au point pour améliorer le développement et la maintenance des systèmes. Un des modèles les plus connus et utilisés est sans conteste le modèle d'évolution des capacités CMM du SEI-CMU [5]. Ce projet s'intéresse entre autres à l'évaluation de la capacité d'évoluer de systèmes d'information existant sur base d'outils d'extraction d'informations du code source et d'une stratégie globale d'évolution [Bro96]. Le principal résultat des recherches du SEI est le guide CMM [Pau93] pour l'amélioration des pratiques en matière de développement et de maintenance de logiciels. Il est composé de pratiques clés exprimant les meilleures façons de travailler pour produire un logiciel de qualité dans le respect des délais et des budgets. Ces pratiques ont été identifiées suite à une vaste consultation de la communauté informatique à propos des succès ou des échecs de milliers de projets. Le modèle CMM est divisé en cinq niveaux de maturité (initial, reproductible, défini, maîtrisé et d'optimisation) comprenant des secteurs clés auxquels une organisation doit satisfaire pour considérer qu'elle a atteint le niveau de maturité correspondant. A mesure qu'elle progresse vers une maturité plus

5. CMM signifie "Capability Maturity Model". Ce modèle a été développé par le Software Engineering Institute de l'université Carnegie Mellon (SEI-CMU) de Pittsburgh.

élevée, l’organisation voit ses risques s’amenuiser alors que la qualité des logiciels, la satisfaction des utilisateurs, le respect des délais et des budgets augmentent. La communauté informatique utilise largement ce modèle pour évaluer la maturité de processus logiciel des entreprises et organismes, pour développer des plans d'amélioration ou comme livre de référence pour la mise en œuvre de pratiques plus matures.

Dans le même registre, un groupe de travail du ISO/CEI [6] a été mis sur pied pour tenter de normaliser un modèle de pratiques de developpement et de maintenance du logiciel ainsi qu'une méthode d'évaluation. Ce groupe vise une reconnaissance internationale de la norme résultant de leurs travaux. Ce fut au départ le projet SPICE [7] [Ema95], qui a, par la suite, cédé le pas au groupe de travail du ISO/CEI. Ce groupe a produit un ensemble d'exigences pour des modèles et des méthodes qui seraient déterminées conformes à la norme ISO/CEI. Cela a permis de faciliter le consensus international et de stimuler le développement de solutions qui s'adaptent à différents besoins tout en partageant un ensemble de règles communes. Le SEI fut un intervenant important du projet SPICE, sans doute pour influencer les exigences de la norme afin que son modèle (CMM) soit facile à rendre conforme à la future norme ISO.

1.2.1.3 Développement d’outils

Une partie des recherches se sont focalisées sur le développement d’outils pour aider l’ingénieur de maintenance. Une technique intéressante est celle de la fragmentation de programmes introduite par Weiser [Wei84]. Il s’agit d’une technique extrêmement puissante qui sélectionne dans un programme toutes les instructions qui contribuent à définir l’état d’une variable à un point déterminé d’un programme. Dans [Gal91], les auteurs appliquent la fragmentation de programmes au problème de la maintenance. Ils démontrent l’utilité des fragments de décomposition, qui capturent toutes les opérations effectuées sur une variable, pour délimiter les effets d’une modification de programme. Ils fournissent ainsi une technique pour déterminer les instructions et les variables qui sont impliquées dans une modification. La fragmentation de programmes est un composant majeur de l’approche proposée dans ce travail.

1.2.1.4 Mesure de la flexibilité

Certains chercheurs se sont concentrés sur la mesure de la complexité des programmes pour aider les concepteurs à évaluer la flexibilité de leurs applications. Basé sur l’idée que la capacité d’évoluer d’un programme est fonction de sa complexité, des métriques ont été développées pour mesurer certains aspects de la complexité des programmes. Ces métriques sont basées sur des mesures aussi différentes que le nombre d’opérations, d’opérandes, la fréquence des opérateurs ou la structure de décision d’un programme. Dans [Kaf87], les auteurs utilisent un spectre basé sur sept métriques existantes pour déterminer la complexité. Leurs expériences montrent une corrélation entre les valeurs fournies par les métriques pour des applications et l’idée que se font les ingénieurs de maintenance de la flexibilité de ces mêmes applications. La flexibilité d’un programme est liée à beaucoup de facteurs dont la complexité fait partie. Il est très difficile de dire, à l’heure actuelle, si les métriques sont utilisables pour prédire les coûts d’une maintenance.

1.2.1.5 Projets

Beaucoup de projets de recherche se sont préoccupés du problème de la maintenance dans des systèmes d'information automatisés. Nous avons choisi à titre d’exemple de présenter succinctement trois projets (REDO, MACS et NATURE) réalisés dans le cadre des programmes ESPRIT [8] de la Communauté Européenne ainsi qu’une liste de laboratoires internationaux travaillant dans le domaine.

6. Organisation internationale de normalisation / Commission électrotechnique internationale.
7. Acronyme de "Software Process Improvement and Capability dEtermination".
8. Acronyme de "European Strategic Programme for Research and development in Information Technology.".

a) REDO [9]

Références : [Bow91], [Bow93], [Sab92], [Vzu91].

Ce projet a étudié la maintenance, la validation et la documentation des logiciels avec comme objectif de développer des méthodes facilitant la maintenance, la restructuration entre des environnements de systèmes différents, d'implémenter un ensemble d'outils pour réaliser ces activités et de concevoir un cadre théorique pouvant influencer les comportements des concepteurs de logiciels. Il a identifié les méthodologies et les techniques permettant d'améliorer la maintenance des applications. Les techniques ont été implémentées sur des méthodes formelles et une conception orientée objets, l'utilisation des méthodes formelles devant permettre une meilleure compréhension du code existant. Le but du projet étant de rendre le code maintenable, cela impliquait une description compréhensible et utile des modules, fonctions et données en vue de corriger, améliorer et adapter les programmes.

Les outils développés permettaient d'extraire une spécification d'une section de code (COBOL ou FORTRAN). Les spécifications étaient représentées dans un langage intermédiaire dont une conception orientée objets dans le langage de spécification Z++ (une extension orientée objets de Z [10]) était extraite. Cela permettait d'encapsuler les spécifications des procédures agissant sur les données dans des classes en tant que méthodes, de simplifier les spécifications et de régénérer le code simplifié. L'aspect humain n'était pas ignoré puisque l'ingénieur de maintenance était impliqué dans un dialogue avec les outils pour la validation et la conversion.

b) MACS [11]

Références : [Des91], [Des92].

L'objectif du projet MACS consiste à définir et implémenter un prototype pour supporter la maintenance de logiciels. L'environnement MACS tente d'aider les programmeurs dans le diagnostic d'erreur, le débogage, la portabilité des programmes, l'amélioration des performances et l'évolution. Une base de connaissance et des techniques de système expert sont utilisées avec des techniques d'ingénierie logicielle. La conception de graphes est intégrée dans l'environnement MACS afin de rencontrer les besoins des ingénieurs de maintenance concernant leurs intentions et les décisions de conception. Ce projet inclut :

- le développement d'un système de représentation des connaissances pour l'enregistrement des graphes de conception et leur utilisation,
- le développement d'analyseur de programmes pour la construction de graphes de conception,
- l'implication des facteurs humains dans la conception d'une interface homme-machine,
- le développement d'un prototype d'assistance de la maintenance,
- l'intégration de représentation des connaissances et d'un système expert pour l'évolution logicielle,
- la validation par expérimentation d'applications existantes.

9. Projet réalisé entre 1989 et 1991 par le Programming Research Group dans le cadre du programme européen ESPRIT 2.
10. Développé à l'université d'Oxford, Z est une notation formelle de spécifications basée sur la théorie des ensembles et les prédicats de premier ordre. Il est utilisé dans l'industrie comme un des composants du processus de développement de logiciels.
11. MACS ("Maintenance Assistance Capability for Software") est un projet réalisé entre 1989 et 1993 dans le cadre du programme européen ESPRIT 2.

c) NATURE [12]

Références : [Rol93], [Jar94a], [Jar94b].

Le projet Nature se concentre sur l'ingénierie des besoins [13] : l'acquisition, la représentation et la gestion des besoins fonctionnels et non-fonctionnels. Face aux changements continuels des environnements des systèmes d'information, le projet NATURE définit un cadre basé sur l'idée que l'ingénierie des besoins est un processus continu pour établir les visions des différents intervenants dans un contexte complexe comme celui de la maintenance. Trois théories différentes sont développées :

- La théorie des domaines de besoins qui aide à déterminer les connaissances qui sont dignes d'intérêt et comment les organiser.
- La théorie des processus offre un méta-modèle des processus dans lequel un petit ensemble de blocs couvre un large éventail de stratégies de guidance de processus.
- La théorie des représentations des connaissances a pour but de définir quels domaines et quels processus doivent être appréhendés et comment les gérer. Cette théorie intègre des représentations formelles, semi-formelles et informelles dans le cadre du langage TELOS [14].

d) Laboratoires de recherche

De nombreux laboratoires de recherche travaillent ou ont travaillé dans le domaine de la maintenance des logiciels. La liste incomplète ci-dessous permet aux lecteurs de se faire une idée de l'intérêt accordé au problème de la maintenance :

- SWPM (Software Process Modeling) de l'université de Kaiserslautern - http://wwwagse.informatik.uni-kl.de/Projects/Current_Projects/SWPM/swpm.html;
- Institute for Information Technology du National Research Council - http://wwwsel.iit.nrc.ca;
- LASER de l'Université du Massachusetts - http://laser.cs.umass.edu/process.html;
- Software Engineering Laboratory de la NASA - http://sel.gsfc.nasa.gov/website/welcome.htm;
- Software Engineering and Testing Lab de l'université de l'Idaho - http://www.cs.uidaho.edu/~setl/setl.html;
- Software Engineering Institute de l'université de Carnegie-Mellon - http://www.sei.cmu.edu;
- Projet Prometheus du Software Engineering Research lab de l'université de Bari - http://serlab2.di.uniba.it/serlab/Prom.html;
- Informatics Process Group de l'université de Manchester - http://www.cs.man.ac.uk/ipg/index.html;
- Distributed Software Engineering de l'université de Londres - http://www-dse.doc.ic.ac.uk/index.html.

12. NATURE (acronyme de "Novel Approaches to Theories Underlying Requirements Engineering") est un projet ESPRIT 3 supporté par la Communauté Européenne de 1992 à 1995.
13. Le terme d'exigence est également utilisé.
14. TELOS est un langage de modélisation conceptuelle généralisant les formalismes de représentation des connaissances comme les diagrammes Entité/Association, les réseaux de sémantique et y intégrant des assertions prédictives et des informations temporelles.

1.2.2 Evolution de bases de données traditionnelles

Avant de faire un état de l'art des recherches sur l'évolution dans les bases de données traditionnelles (point 1.2.2.2) et de présenter quelques projets (point 1.2.2.3), il est opportun de rappeler ce qu'on entend par bases de données traditionnelles (point 1.2.2.1).

1.2.2.1 Modèles de données traditionnels

La fin des années soixante et le début des années septante ont été témoins de la naissance de trois grands modèles de données sur lesquels sont encore développés de nombreux Systèmes de Gestion de Bases de Données (SGBD). Il s'agit des modèles hiérarchiques, en réseaux et relationnels. Elmasri et Navathe [Elm94] définissent un modèle de données comme «*... a set of concepts that can be used to describe the structure of a database*». Il s'agit d'un type d'abstraction de données permettant l'indépendance des données et des programmes. Il est utilisé pour fournir la représentation des données sans les détails physiques concernant leur stockage. Des descriptions approfondies de ces modèles sont disponibles dans [Elm94], [Kor91] et [Kro92].

Outre ces trois modèles, on ne peut aborder les applications de bases de données traditionnelles sans parler du langage de programmation COBOL bien qu'il ne soit pas un modèle de données au sens propre du terme. Il est encore largement utilisé dans divers domaines comme Système de Gestion de Fichiers (SGF).

a) Modèle hiérarchique

Le modèle hiérarchique de données est apparu vers la fin des années soixante dans le SGBD Information Management System (IMS) pour les plates-formes IBM (International Business Machine). Il représente les données comme des structures d'arbres hiérarchiques. Il a été développé pour modéliser les nombreux types d'organisations hiérarchiques qui existent dans la réalité et qui aident l'homme à mieux comprendre le réel perçu. IMS d'IBM et System 2000 de SAS Institute sont deux des SGBD basés sur le modèle hiérarchique qui sont les plus connus à l'heure actuelle. Pour plus d'informations, le lecteur peut consulter [Gel89].

Pour décrire brièvement les structures de données d'un modèle hiérarchique, on peut dire qu'il emploie deux concepts principaux : les enregistrements et les relations parent-enfants. Un enregistrement est une collection de valeurs de champs qui donne des informations sur une instance d'une entité ou d'une relation. Les enregistrements de même type sont regroupés dans des types d'enregistrements qui ont un nom et dont la structure est définie par une collection de champs. Un type de relations parent-enfants est une relation un-à-plusieurs entre deux types d'enregistrements. Un schéma hiérarchique consiste en un certain nombre de types d'enregistrements et de types de relations parent-enfants.

b) Modèle en réseau

Au début des années septante, les structures et le langage du modèle en réseau ont été définis par le comité Database Task Group CODASYL [15] [DTG71]. Le modèle en réseau représente les données comme des types d'enregistrements et des relations un-à-plusieurs appelées type d'ensemble. Ce modèle est également connu sous le nom de CODASYL dont IDMS de Computer Associates et IDS2 de Bull sont les héritiers les plus connus. [Per81] donne des informations supplémentaires sur ce modèle.

Il y a deux structures de données de base dans le modèle en réseau : les enregistrements et les ensembles (ou "sets"). Les données sont stockées dans des enregistrements qui sont classés en types d'enregistrements ayant la même structure pour stocker le même type d'information. Un type d'enregistrement a un nom et des attributs qui ont un nom et un type. Il existe deux types complexes pour les attributs. Un attribut peut être de type vecteur (c'est-à-dire qu'il peut avoir

15. Acronyme de "COnference on DAta SYstems Language".

plusieurs valeurs dans un seul enregistrement) ou de type groupe répétitif (c'est-à-dire qu'un enregistrement peut avoir un ensemble de valeurs composites). Un type ensemble est une description d'une relation un-à-plusieurs entre deux types d'enregistrements. Le type d'enregistrements du côté un est appelé le "owner", celui du côté plusieurs est appelé le "member".

c) Modèle et algèbre relationnels

Le modèle relationnel a été introduit par Codd en 1970 [Cod70]. Il est basé sur une structure de données simple et uniforme : la relation. Il représente une base de données comme une collection de relations sous la forme de tables pouvant être stockées dans un fichier. Le modèle relationnel est bien ancré dans le monde des applications de bases de données. Il existe de nombreux SGBD relationnels commerciaux dont, parmi les plus importants, Oracle, DB2 d'IBM, SQL Server de Microsoft, Adaptive Server Enterprise de Sybase Incorporation et Informix Dynamic Server d'Informix Software.

Une relation est une table de valeurs. Chaque ligne de la table représente une instance de la table. Ces valeurs peuvent être interprétées comme des faits décrivant une entité ou une relation du mode réel. Les noms des colonnes spécifient l'interprétation des valeurs de chaque ligne. Toutes les valeurs d'une colonne sont du même type. Ce modèle est décrit de manière très précise dans le chapitre 3 au point 3.1.3 puisque ce travail se focalise sur la maintenance des bases de données relationnelles. [Dat00] donne un aperçu très complet du modèle et de l'algèbre relationnels.

d) Langage COBOL

COBOL [16] est né dans le courant de l'année 1959 lors d'une conférence au Pentagone à Washington à l'initiative d'une instance gouvernementale américaine. Il est le plus vieux langage de programmation encore en activité et largement utilisé. Son succès vient du fait qu'il est spécialement adapté à la résolution de problèmes de gestion commerciale. Il permet de décrire des structures de données variées et facilite la manipulation de fichiers volumineux. COBOL n'est pas un SGBD mais il possède tous les éléments nécessaires à la gestion de fichiers. Certains auteurs le désigne également comme un SGF.

Le modèle de données COBOL est centré sur la notion de fichier. Un fichier est constitué d'enregistrements d'un ou plusieurs types. Un type d'enregistrements est composé d'au moins un champ qui peut être élémentaire ou composé d'autres champs. Un champ peut également être un tableau ce qui signifie qu'il peut prendre plusieurs valeurs pour un même enregistrement. [Cla91], [Hut90] sont des livres de référence sur le langage de programmation COBOL et sa norme COBOL 85. Ils apportent des informations complémentaires sur le modèle des structures de données du langage.

1.2.2.2 Evolution de schémas

La recherche sur l'évolution du composant base de données d'un logiciel est plus récente et incontestablement moins abondante que celle sur la maintenance des logiciels. Le problème de l'évolution de bases de données a d'abord été analysé pour des structures de données standards. Au début des années quatre-vingts, Navathe [Nav80] souligne l'importance de l'analyse de schémas en développant une méthodologie d'analyse des structures de données pour aider à la restructuration de bases de données hiérarchiques et en réseau. Nous allons parcourir quelques thèmes importants liés à l'évolution de schémas en rapport avec l'objectif de ce travail. Les lecteurs intéressés par des informations supplémentaires trouveront dans [Rod92b] et [Rod95] une bibliographie commentée.

16. Acronyme de "COmmon Business Oriented Language".

a) Modification de schémas relationnels

La modification d'un schéma conforme au modèle relationnel a fait l'objet de plus de recherches que les autres modèles traditionnels (hiérarchique, réseaux ou COBOL). C'est vraisemblablement dû au fait qu'il se prête mieux aux modifications puisque la plupart des SGBD relationnels disposent d'instructions d'altération de structures de données et de conversion de données. La modification d'un schéma relationnel a été étudiée entre autres dans [Shn82], [Rod93], [Ewa96].

Shneiderman et Thomas [Shn82] analysent une série de transformations qui peuvent augmenter la flexibilité d'une base de données en facilitant le processus de conversion quand les besoins changent sur un schéma de données relationnel. Ils explorent la possibilité de développer un système de conversion de bases de données automatique pour le modèle relationnel en utilisant l'algèbre relationnel comme langage de manipulation de données.

Pour Ewald et Orlowska [Ewa96], les effets de l'évolution dans une base de données relationnelle peuvent être délimités en examinant les changements des contraintes nécessaires pour implémenter une modification des données. La partie d'un schéma relationnel affectée par l'addition ou le retrait d'une contrainte de dépendance fonctionnelle peut être déterminée. Ainsi, il n'est pas nécessaire de reconcevoir un schéma chaque fois qu'une modification doit être réalisée.

b) Modification des schémas conceptuels

L'impact des modifications conceptuelles sur les bases de données fait également partie des préoccupations des chercheurs du domaine. Il est maintenant reconnu que la conception de bases de données passe par la création d'un schéma conceptuel proche de la réalité des utilisateurs. Sur base de ce schéma, le concepteur dérive un ou plusieurs schémas conformes à une technologie. Dans ce contexte, les chercheurs se sont interrogés sur la propagation des modifications d'un schéma conceptuel vers les schémas physiques ou, formulé autrement, sur la traduction des modifications conceptuelles en modifications techniques.

La propagation des modifications conceptuelles sur les schémas relationnels est analysée entre autres dans [Mar87], [Cas93], [Rod93] et [vBo94]. Dans [Mar87], un ensemble de transformations conceptuelles sont proposées ainsi que les correspondances (appelées opérations de réorganisation de bases de données) entre ces transformations et les restructurations incrémentales et réversibles (utilisant une notation orientée Entité/Association) au niveau du schéma relationnel. Casanova et d'autres [Cas93] proposent une méthode de reconception qui accepte en entrée le schéma conceptuel, sa représentation relationnelle et une séquence de changements. Comme résultat, la méthode produit un nouveau schéma conceptuel et un plan de modifications permettant de créer une représentation relationnelle optimisée pour le nouveau schéma conceptuel et de restructurer la base de données relationnelles. Dans [Rod93], les auteurs proposent deux taxonomies des modifications relatives à un schéma : une applicable à un schéma conforme au modèle relationnel, l'autre à un schéma conceptuel Entité/Association. Ils donnent également une correspondance entre les modifications des deux classifications.

Dans [Liu94a], [Liu94b], les auteurs présentent une méthode basée sur la dérivation de schémas. Elle analyse les relations entre deux schémas (représentés sous la forme de diagrammes EVER - EVolutionary Entity/Relationship) avant et après modifications et dérive des schémas de bas niveau pouvant être mis en correspondance avec des modèles de bas niveau (hiérarchique, réseau, relationnel ou orienté objets). [Pro94] et [Pro97] décrivent des approches basées sur la modélisation et l'évolution de spécifications représentées par une hiérarchie de modèles. Leur modèle de l'évolution est centré sur la gestion des différentes versions de spécifications. Pour sa part, Wedemeijer [Wed99] décrit des changements de patrons sémantiques nécessaires pour adapter le schéma conceptuel aux modifications des besoins et des informations d'un domaine d'application. La préservation de la consistance d'un schéma est analysée dans [Ewa93]. L'article propose une solution procédurale et interactive qui permet d'ajouter ou de retirer une unité d'information dans un schéma conceptuel. van Bommel [vBo92] [vBo93] [vBo94] s'intéresse à l'optimisation de schémas de base de données basée sur la notion d'évolution par le biais d'algo-

rithmes génétiques. Il constate qu'à partir d'un schéma conceptuel, plusieurs schémas techniques peuvent être dérivés dont certains sont meilleurs que d'autres pour une implémentation efficace. Sur la base de ce constat, il développe un environnement pour l'évolution de structures de bases de données.

Dans la plupart des approches présentées, seules des règles de correspondances contraignantes entre le niveau conceptuel et le niveau technique sont prises en compte par une simplification du modèle conceptuel. De même, la propagation des modifications jusqu'aux instances et la modification des programmes est minimisée ou négligée.

c) Intégration de vues

Les vues [17] augmentent la flexibilité d'une base de données en autorisant la visualisation des données de différentes façons (par exemple, pour des utilisateurs distincts). Le problème de la modification de bases de données relationnelles au travers de vues est un domaine sur lequel plusieurs recherches se sont concentrées. Pour une bibliographie assez complète, le lecteur peut consulter [Lin96].

Les travaux dans le domaine s'attachent principalement aux modifications des vues. Dans [Ban81] et [Day82], les auteurs formalisent la traduction des modifications de vues dans le cadre du modèle relationnel et dérivent les conditions garantissant la consistance entre la vue et la base de données après la traduction des modifications. En plus, [Ban81] présente un traducteur implémentant les traductions des modifications de la base en modifications des vues. Harrison [Har95] présente une méthode pour calculer les modifications d'une vue à partir des modifications de la base de données et propose un algorithme de propagation des modifications pour calculer de manière incrémentale les modifications nécessaires à la vue sans forcer sa reconstruction. Dans [Lin96], le problème des modifications de vues est présenté dans un système de gestion de bases de données basé sur le modèle Entité/Association. Une théorie est développée dans le cadre de cette approche caractérisant les conditions à respecter pour qu'il existe une correspondance entre les modifications de vues et celles sur le schéma conceptuel.

Il n'existe pas à notre connaissance d'étude sur l'utilisation des vues comme une aide à la propagation des modifications dans une base de données relationnelle (comme cela existe dans les bases de données orientées objets : voir point 1.2.3.2). Dans ce contexte, l'utilisation des vues entraîne une indépendance permettant à certaines modifications d'être réalisées sur un schéma de spécifications sans affecter la base de données et les programmes d'applications. Ce concept est approfondi et exploité dans le chapitre 4.

d) Base de données temporelles

L'évolution de schémas est considérée comme une extension logique des travaux sur les bases de données temporelles. Beaucoup de concepts utilisés dans le domaine de l'évolution sont similaires à ceux associés aux bases de données temporelles.

Les modèles de données temporelles s'intéressent à la nature temporelle inhérente aux objets de la réalité et au moment de l'enregistrement des faits (décrivant la réalité) dans une base de données. Deux dimensions temporelles découlent de cette préoccupation : le *valid-time* qui est le temps du monde réel et le *transaction-time* qui est le temps du système [Jen94]. Outre les bases de données orientées valid-time ou transaction-time, une base de données peut être bitemporelle si elle a la capacité de gérer les deux dimensions. Certaines recherches proposent des langages de requêtes qui prennent en compte ces différentes notions de temps.

Lorsqu'il est important de retrouver l'historique de la structure d'une base de données, une troisième dimension temporelle est introduite. Il s'agit du *schema-time* qui indique le format des données grâce à la référence du schéma actif à un moment donné [Rod95]. Il fournit un mécanisme qui assure que les données sont conformes à une version spécifiée d'un schéma. Dans ce contexte, les modifications de schémas impliquent des opérations complexes qui permettent de gar-

17. Une vue est une relation construite par une requête et dérivée à partir d'autres vues ou de tables.

der les anciennes versions des données et des schémas. On se rapproche des mécanismes présentés dans le contrôle des versions des bases de données orientées objets (voir point 1.2.3.2).

Quelques recherches intéressantes réalisées dans le domaine des modèles de données temporelles sont décrites brièvement ci-dessous.

Ariav [Ari91] examine les implications de l'évolution des structures de données dans un modèle de données temporelles. Il discute entre autres des impacts des modifications sur les données existantes et des conditions à respecter pour modifier l'application correspondante.

McKenzie et Snodgrass [Mck90] proposent une extension de leurs travaux sur les bases de données temporelles et présentent une algèbre relationnelle pour la consultation et la modification des données avec un support pour l'évolution de schémas et les aspects temporels.

Roddick [Rod92a] présente SQL/SE, une extension du langage SQL, qui prend en compte l'évolution de schémas dans les SGBD relationnels. Il s'intéresse au support du *valid-time* et du *transaction-time*, à la problématique du recouvrement de toutes les données quel que soit le temps, aux problèmes des valeurs nulles et aux modifications du langage pour la présentation des résultats.

1.2.2.3 Projets

Contrairement à la maintenance des logiciels, les projets de recherche sur l'évolution des bases de données ne sont pas nombreux. Ce manque d'intérêt s'explique peut-être par le peu d'attrait suscité par les bases de données traditionnelles. Les projets de recherche préfèrent s'orienter vers des technologies plus récentes ou des domaines plus attrayants comme la conception, la migration, les systèmes distribués, ... Or les systèmes traditionnels (surtout relationnels) restent une réalité incontournable pour beaucoup d'organisations. Ils doivent évoluer pour faire face aux changements de leur environnement. Trois projets ont quand même attiré notre attention. Premièrement, le projet européen DAIDA s'intéresse à la maintenance des logiciels mais accorde une attention particulière à la partie base de données des applications. Deuxièmement, le projet M7 du laboratoire Matis travaille sur une approche évolutive de la conception des applications de bases de données. Troisièmement, le projet Varlet développe un environnement intégré pour la réingénierie de systèmes d'information. Finalement, les démarches des différents projets décrits sont comparées avec celle adoptée dans ce travail.

a) DAIDA [18]

Références : [Chu91b], [Jar92].

Le projet DAIDA a créé un environnement de développement de systèmes d'information selon une approche de l'ingénierie des connaissances. L'environnement se propose de capturer les structures et les contraintes de différentes perspectives (conceptuelle, physique des processus) dans une base de connaissances et de fournir une méthodologie et un ensemble d'outils pour utiliser la base de connaissances comme support du processus de développement. Le but est de construire un système expert avec lequel les systèmes d'information peuvent être créés et maintenus. L'architecture de l'environnement DAIDA adopte les concepts qui conviennent pour représenter les connaissances et les raisonnements sur les besoins, les conceptions et l'implémentation des bases de données, des programmes d'application et des interfaces constituant un système informatique. Il est basé sur trois langages : TELOS pour formaliser l'analyse des besoins, Taxis [19] pour la conception du système et DBPL [20] pour l'implémentation des programmes.

18. Acronyme de "Advanced Interactive Development of Data-Intensive Applications", ce projet a été mis en œuvre de 1986 à 1990 par un consortium universitaire et industriel dans le cadre du programme européen ESPRIT.
19. Taxis offre un modèle de données sémantique pour décrire les données, les transactions et les processus. Il possède les notions d'entités et d'associations avec des mécanismes d'agrégation, de généralisation et de classification.

b) M7

Référence : [Leo99].

Le projet M7 est une approche évolutive pour la conception de systèmes d'information développée par le laboratoire MATIS de l'université de Genève. Un outil de spécification et de conception qui accroît l'efficacité de la conception et la réalisation de systèmes évolutifs est actuellement développé dans le cadre du projet. Il offre un environnement permettant la saisie de spécifications, la génération en Oracle ainsi que l'évaluation de la cohérence des spécifications.

Dans [Leo99], l'auteur propose quelques recommandations intéressantes que les SGBD devraient satisfaire pour intégrer une approche évolutive comme celle définie dans le cadre du projet M7.

c) Varlet

Références : [Jah99a], [Jah99b].

Le projet de réingénierie [21] de bases de données Varlet a démarré en 1995 dans le groupe d'ingénierie logicielle de l'université de Paderborn (Allemagne). Il a pour objectif le développement d'un environnement intégré pour la réingénierie de systèmes d'information, d'outils d'aide au processus de réingénierie et la génération automatique d'un middleware [22] pour des systèmes d'information client/serveur orientés objets. Il utilise une approche basée sur l'analyse semi-interactive de systèmes traditionnels, l'analyse de code par détection de patrons, une représentation graphique des spécifications au niveau logique et conceptuel avec la correspondance et la restructuration de spécifications orientées objets.

L'approche Varlet se décompose en deux phases. Premièrement, l'analyse de schémas obtient à partir d'un schéma physique relationnel implémenté un schéma logique complet. Deuxièmement, les structures logiques sont transformées en un schéma conceptuel qui sera le point de départ pour des activités de modification ou de migration. Si le schéma conceptuel est modifié, il existe un mécanisme pour rétablir la consistance entre le schéma logique et le nouveau schéma conceptuel. Le schéma logique est automatiquement modifié pour refléter les modifications du schéma conceptuel.

d) Comparaison des différentes approches

Il est intéressant de comparer les trois projets qui proposent des approches très différentes du problème de la maintenance et de l'évolution. Les projet DAIDA et M7 s'intéressent au problème de la conception pour faciliter les modifications ultérieures alors que le projet Varlet s'attache à l'évolution d'applications de bases de données existantes.

L'approche proposée par le projet DAIDA a permis de bien formaliser le processus de développement de logiciel. La représentation formelle des connaissances améliore le processus de décision. L'adoption de trois niveaux de langage a permis de rassembler dans une base de connaissances des informations disparates venant de l'étude des besoins, de la conception et de l'implémentation. DAIDA prend en compte tous les aspects du processus de développement d'une application d'une base de données et répond ainsi à un reproche souvent émis à l'encontre des méthodologies existantes qui ne résolvent pas de manière satisfaisante des problèmes vitaux comme la maintenance.

Le projet M7 propose une approche évolutive de la conception qui, sur la base de recommandations, permet de concevoir des applications plus faciles à maintenir. Ce projet récent n'a pas

20. DBPL est un outil d'implémentation pour des applications centrées sur les données. Il est basé sur un modèle de données supportant l'accès prédicatif et le contrôle de grands ensembles de données. Il a le langage de programmation Modula-2 comme noyau algorithmique.
21. Démarche de restructuration d'un système en vue de l'améliorer.
22. Partie logicielle d'un système informatique en réseau qui regroupe les logiciels considérés du point de vue de l'interopérabilité des systèmes.

encore fourni de résultat probant permettant de l'évaluer. Mais la démarche est intéressante car elle tente d'anticiper les problèmes d'évolution au moment de la conception d'un système. Ce point de vue est très proche de celui développé dans le chapitre 6.

Le projet Varlet s'inscrit dans une démarche de réingénierie présentant beaucoup de similitudes avec celle présentée dans ce travail. Cette démarche s'attache à faire évoluer des systèmes existants. Pour ce faire, elle est décomposée en deux phases. Premièrement, des techniques de rétro-ingénierie sont utilisées pour redocumenter les bases de données. Un AGL permet de stocker les informations extraites dans des spécifications classées selon le modèle classique. Deuxièmement, des mécanismes spécifiques maintiennent la consistance entre les différentes spécifications suite à l'application de modifications pour assurer la conversion des bases de données. Malgré des ressemblances évidentes, il existe des différences essentielles entre notre démarche et celle du projet Varlet dont, notamment, les correspondances entre les différentes spécifications et le degré de liberté du concepteur dans le processus d'évolution. Grâce à un modèle conceptuel simplifié, l'approche Varlet définit des relations/liens entre les spécifications des niveaux logique et conceptuel pour implémenter les correspondances. Notre approche, quant à elle, se base sur un modèle conceptuel plus complet et implémente les correspondances sous la forme d'historique. Cette divergence s'explique par le fait que le processus de propagation des modifications du projet Varlet soit très automatisé alors que celui proposé dans ce travail laisse une marge de manoeuvre plus grande avec la possibilité pour le concepteur d'intervenir régulièrement.

1.2.3 Versions de bases de données orientées objets

Les bases de données traditionnelles stockent les données de manière à les rendre indépendantes des procédures. Les données sont ainsi accessibles par différentes applications de différentes façons. Au contraire, les bases de données orientées objets stockent des objets, c'est-à-dire des données et les méthodes qui les gèrent. Le point 1.2.3.1 présente les principales caractéristiques du modèle objet et le point 1.2.3.2 s'intéresse aux mécanismes de contrôle des versions qui font des bases de données orientées objets un cadre favorable à la résolution du problème de l'évolution.

1.2.3.1 Modèle orienté objets

Le paradigme d'objet est d'abord apparu dans les langages de programmation. Son histoire commence au milieu des années soixante avec le langage SIMULA. Par la suite, il a pris de l'importance par l'avènement dans le courant des années quatre-vingts des langages Smalltalk et C++. Ce paradigme est construit sur quelques concepts et constructions puissants pour la description des données :

- L'identité d'objet : chaque entité du monde réel est représentée par une structure indivisible appelée *objet*.
- La classification : chaque objet est une instance d'une classe d'objets dans laquelle toutes les instances ont les mêmes propriétés.
- L'encapsulation : à chaque classe peuvent être associées des opérations qui manipulent les objets de la classe.
- L'héritage : les classes prennent place dans une hiérarchie de manière à hériter des propriétés et des opérations des classes qui sont au-dessus.
- La surcharge : une classe peut redéfinir les opérations et les propriétés des classes dont elle hérite.

Le modèle orienté objets définit une base de données en termes d'objets, de leurs propriétés et de leurs opérations. Les objets ayant une même structure appartiennent à une classe. Les classes sont organisées en hiérarchie. Les opérations de chaque classe sont spécifiées en termes de procédures prédéfinies appelées méthodes.

La recherche universitaire dans le domaine des bases de données orientées objets a commencé dans les années quatre-vingts. Elle a donné naissance aux premiers SGBD orientés objets (SGBDOO) dont font partie O2, GemStone, ITASCA ou UniSQL. Les entreprises prirent ensuite le relais pour améliorer, créer et commercialiser des SGBD professionnels. Outre les logiciels déjà cités, on peut également mentionner ObjectStore (de Object Design Incorporation) et Objectivity/DB. En 1991, l'ODMG [23] est né pour faire progresser la standardisation des SGBDOO de manière à favoriser l'acceptation de cette technologie. Actuellement, la norme ODMG 3.0 [Cat00] est la dernière version du standard.

Les références [Adi93], [Mar93], [Boo94], [Cat97] et [Coo97] permettront aux lecteurs de compléter leurs connaissances sur le sujet.

1.2.3.2 Contrôle des versions

Le modèle orienté objets est reconnu comme un cadre favorable pour résoudre des problèmes d'évolution de structures de données. Il a permis de développer des solutions élégantes grâce au contrôle des versions de schémas et des versions d'instances [Alj95] [And91] [Ngu89] [Bel93]. En fait, ajouter un composant C au type d'une classe d'objet O1 consiste simplement à dériver une sous-classe O2 dont le type est celui de O1 plus le composant C. Les méthodes connaissant l'existence de C sont redéfinies alors pour O2.

Les constructeurs de SGBDOO ont bien compris que la gestion des versions est un besoin important dans la conception d'applications. Beaucoup de systèmes supportent la maintenance de versions d'objets. Toutefois, les mécanismes de gestion varient considérablement d'un système à l'autre et il n'existe pas d'accord sur la prise en charge de la propagation des modifications de structures au niveau des instances. Les principales approches pour le contrôle des versions sont :

- *Version de schéma* [Ban87] : toute modification relative à un schéma crée une nouvelle version du schéma complet correspondant à l'état de la base de données à un moment donné. Dans le cas de modifications mineures concernant une petite partie du schéma, le nombre de versions à gérer risque d'être élevé.
- *Version de classe* [Zdo86] : toute modification relative à une classe crée une nouvelle version de la classe et de ses sous-types. Une version de schéma est créée virtuellement en tenant compte des liens de dérivation entre les différentes versions des classes. Pour représenter l'état du schéma à un moment donné, il faut choisir une version pour chaque classe de l'état. Les modifications relatives à une classe racine engendrent une version avec toutes ses sous-classes.
- *Définition de vues* [Alj95] [And91] [Bel96] [Bel00] : l'évolution structurelle d'une base de données peut être simulée au travers de vues. Les modifications sont gérées comme des définitions de nouvelles vues. Plusieurs vues peuvent être générées d'une vue donnée, chacune correspondant à des modifications du schéma. Toutes les vues opèrent sur la même collection de données. Il y a deux tendances pour la définition de vues : soit elles appartiennent au schéma en tant que classe virtuelle, soit elles sont en dehors du schéma.

Pour terminer, nous allons examiner quelques recherches effectuées dans le cadre de la résolution du problème de l'évolution dans certains SGBDOO. [Ban87] et [Kim88] traitent de l'évolution de schémas dans le SGBDOO Orion. Ils définissent les contraintes et invariants qui assurent la consistance entre les modifications de schémas et les instances de la base et présentent une taxonomie des modifications permises. Dans [Ngu89], les auteurs passent en revue l'aide à l'évolution de schémas dans des systèmes existants (Cadb, Encore, Gemstone, Orion et Sherpa) et font une proposition pour supporter la propagation automatique des changements dans les SGBDOO. Cette proposition est basée sur la notion de classe d'objets tenant compte de la complétude des objets évoluant et un ensemble d'opérations de modifications de schémas. Le système Farandole 2 ([Alj95] et [And91]) gère les modifications de schémas en autorisant le

23. pour "Object Database Management Group".

développement de versions partielles de schémas (appelées aussi vues de schémas). Les auteurs définissent les modifications autorisées ainsi que les règles garantissant la cohérence après une modification. Le lecteur intéressé par d'autres références peut consulter [Ngu89] et [Rod92b].

Les systèmes orientés objets offrent un cadre favorable pour le développement de solutions intéressantes au problème de l'évolution de spécifications. Malheureusement, leur approche est de peu d'utilité avec les technologies traditionnelles et, particulièrement, les systèmes hérités [24].

1.2.4 Rétro-ingénierie

Beaucoup de systèmes d'information automatisés existants sont insuffisamment ou pas du tout documentés. Leur évolution implique en premier lieu de recouvrer des spécifications de conception, c'est-à-dire un historique possible du processus de conception avec les schémas de la base et d'autres documents (une description formelle des programmes par exemple). Quand aucune documentation n'est disponible, il est toujours possible de la recréer, du moins partiellement, grâce à un processus appelé rétro-ingénierie. La rétro-ingénierie consiste à reconstruire une spécification technique et fonctionnelle plausible à partir, principalement, des textes sources des programmes.

La rétro-ingénierie a souvent été analysée dans le contexte de la maintenance de logiciels. Par exemple, les projets REDO (point 1.2.1.5 a) et MACS (point 1.2.1.5 b) proposent des démarches de rétro-ingénierie comme support au processus de maintenance. Elle utilise généralement des techniques de compréhension de programmes. La compréhension de programmes est un domaine émergeant de l'ingénierie des logiciels. Rugaber [Rug95] la définit comme un processus d'acquisition de connaissances sur un programme existant généralement non documenté. L'amélioration des connaissances est un support pour des activités comme la correction d'erreurs, la réutilisation, l'amélioration ou, bien sûr, la rétro-ingénierie. De nombreuses approches ont été explorées pour analyser les programmes. Cela va de l'analyse de texte à l'analyse dynamique de l'exécution des programmes en passant par la fragmentation de programmes (point 1.2.1.3) et l'analyse des graphes de dépendances des variables.

Dans les applications traditionnelles où le composant base de données est central, les fichiers ou la base de données sont souvent partiellement documentés indépendamment des parties procédurales en utilisant un processus de rétro-ingénierie de base de données. Il ne s'agit là que d'une première étape du processus complet qui permet de travailler plus efficacement qu'en utilisant l'application entière comme source d'information. La rétro-ingénierie des programmes reste nécessaire pour parfaire les connaissances extraites de la base de données. La rétro-ingénierie d'une base de données donne comme résultats des spécifications logiques et conceptuelles plausibles pour la base. Les projets Phenix [Jor92] et Varlet (point 1.2.2.3 c) proposent des démarches et des environnements pour la rétro-ingénierie de bases de données.

L'étude approfondie de la rétro-ingénierie n'est pas du ressort de ce travail. Toutefois, nous ferons une brève incursion dans le domaine en vue de proposer une démarche d'évolution complète.

24. Il s'agit de systèmes informatiques issus d'une génération précédente et qui continuent d'être utilisés après avoir été adaptés à des systèmes plus contemporains. On parle aussi de systèmes patrimoniaux ou, en anglais, de "legacy systems".

1.3 Contribution

La maintenance des logiciels est une tâche vaste et complexe qu'il est illusoire d'étudier dans sa globalité. Ce travail se focalise sur la maintenance et l'évolution d'applications de bases de données relationnelles. Le terme application de bases de données désigne un système logiciel dont le composant données persistantes (ou base de données) est central. Pourquoi parle-t-on de maintenance et d'évolution ? La maintenance, abondamment commentée précédemment, s'applique aux programmes alors que l'évolution s'intéresse à la capacité d'une base de données à répondre aux changements du monde réel en permettant aux spécifications d'évoluer. Pour développer une méthodologie et des outils débouchant sur des réalisations concrètes, la démarche se base sur le modèle relationnel. Elle est donc applicable à des systèmes développés à l'aide de langages standards de troisième génération tels que COBOL/SQL ou C/SQL. Par la suite, on verra que cette limitation peut être levée moyennant des adaptations.

Au cours de leur cycle de vie, les applications considérées sont amenées à évoluer. Cette évolution provient de changements dans les besoins de l'entreprise ou de l'apparition de nouveaux besoins. Suite à cette évolution des besoins, le concepteur apporte des modifications techniques à l'application. Face au dénuement technique et méthodologique dont l'informaticien est victime, nous allons proposer une méthodologie pour maintenir un système informatisé ainsi que des outils pour soutenir le concepteur dans cette phase critique du cycle de vie d'une application.

L'objectif est d'analyser le phénomène de l'évolution du composant base de données des applications au travers d'une stratégie de propagation des modifications. Ce travail présente un modèle abstrait et développe un environnement d'assistance relatif à l'évolution des applications de bases de données, considérées comme l'intégration de trois composants : les structures de données, les données et les programmes. Dans ce contexte, nous limiterons le propos au composant données persistantes de l'application, c'est-à-dire à l'ensemble des fichiers, ou base de données, qui représentent essentiellement les objets remarquables du domaine d'application.

Le problème peut alors se formuler de la manière suivante :

> Suite à l'évolution des besoins des utilisateurs, l'analyste est amené à modifier les spécifications de la base de données à un certain niveau d'abstraction. Comment peut-on gérer le processus d'évolution de l'application complète suite à ces modification ?

1.4 Exemple intuitif

Cette section illustre la complexité du problème par un exemple simple. Le point 1.4.1 décrit les spécifications de l'exemple ainsi que les modifications apportées. Le problème de la conversion des structures de données et des instances ainsi que les difficultés posées par la maintenance des programmes sont examinés dans les points 1.4.2 et 1.4.3.

1.4.1 Description de la base de données

A titre d'illustration, considérons les deux schémas de la figure 1.5 représentant les structures d'une base de données relationnelle.

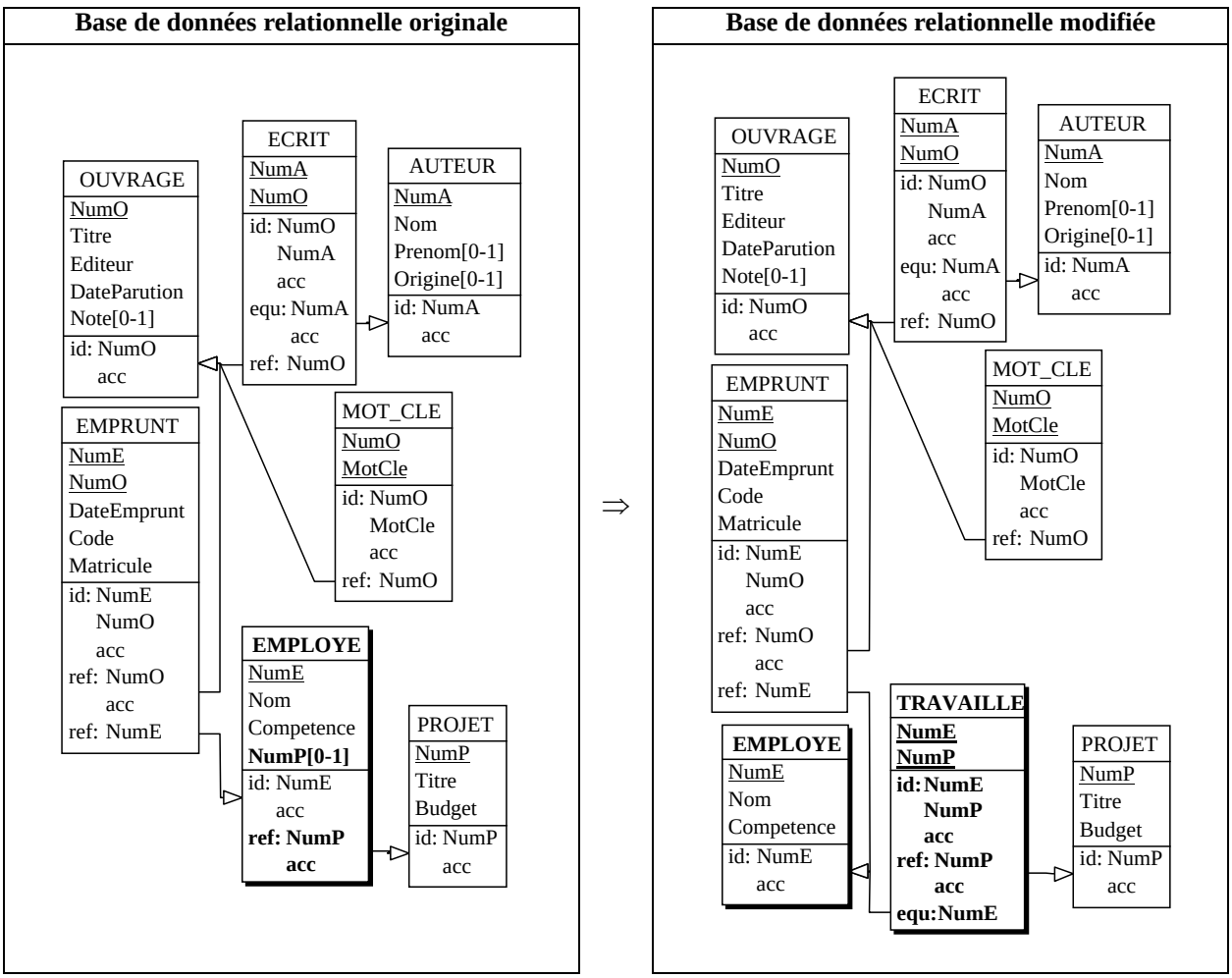


Figure 1.5 - Extension de la clé étrangère entre les tables *EMPLOYE* et *PROJET* en une table *TRAVAILLE* de manière à pouvoir représenter l'appartenance d'un employé à plusieurs projets.

Les représentations graphiques, utilisées dans la figure 1.5 et décrites en détail dans la section 3.1, peuvent être résumées comme suit :

- Les boîtes rectangulaires symbolisent les tables.
- Le premier tiroir d'une table contient son nom.
- Le deuxième tiroir contient la liste des colonnes de la table.

– Le dernier tiroir définit les contraintes de la table. Une contrainte *id* définit une clé primaire (les colonnes appartenant à une clé primaire sont soulignées), une contrainte *ref* définit une clé étrangère symbolisée par une flèche vers un identifiant, une contrainte *equ* définit une clé étrangère d'égalité dans laquelle l'identifiant doit toujours être référencé et une contrainte *acc* définit un index sur les colonnes de la contrainte.

Les spécifications de la base de données originale représentent une bibliothèque dans laquelle les ouvrages sont écrits (via la table *ECRIT*) par aucun, un ou plusieurs auteurs. Les ouvrages sont caractérisés par des mots-clés et sont empruntés (via la table *EMPRUNT*) par des employés. Un employé peut appartenir à un projet. La modification apportée au schéma original consiste à remplacer la clé étrangère entre les tables *EMPLOYE* et *PROJET* par une table *TRAVAILLE* reliée aux deux tables précitées. La sémantique du schéma est augmentée puisque le nouveau schéma permet de représenter le fait qu'un employé travaille dans plusieurs projets. Dans le nouveau schéma, la contrainte d'égalité de *TRAVAILLE* vers *EMPLOYE* impose qu'un employé travaille pour au moins un projet alors que, dans le schéma original, un employé ne travaillait pas nécessairement pour un projet. La table *TRAVAILLE* est la matérialisation d'une relation plusieurs-à-plusieurs entre les tables *EMPLOYE* et *PROJET*.

1.4.2 Evolution de la base de données

Il est possible de générer un script de conversion composé d'instructions SQL qui fait évoluer les structures et les instances de la base de manière à respecter les nouvelles spécifications. La figure 1.6 propose un tel script qui est décomposé en quatre parties :

1. Il faut d'abord vérifier que les modifications ne causent pas de pertes d'information ou une violation des contraintes dans la base. Le script donne une requête de vérification des contraintes. En cas de violation, la cohérence de la base de données doit être assurée avant sa modification. Si un employé ne travaille dans aucun projet, il violera la contrainte d'égalité définie sur la table *TRAVAILLE*. Plusieurs solutions sont envisageables :
   a) Soit la base de données est incorrecte (erreur d'encodage par exemple). Les employés qui ne travaillent pas dans un projet doivent être associés à un projet.
   b) Soit l'information est inconnue. Les instances de la table *EMPLOYE* violant la contrainte seront stockées dans une table temporaire en attendant leur correction. Cette solution n'est pas toujours réalisable. Si un employé est référencé par un emprunt, le transfert de cet employé dans une table temporaire est impossible sans la création d'une table temporaire pour les emprunts des employés violant la contrainte d'égalité. Comme on le constate, cette solution peut devenir disproportionnée si elle nécessite la mise à l'écart d'une partie plus ou moins grande des instances de la base de données.
   c) Soit la modification proposée n'est pas valide. Il faut autoriser qu'un employé ne travaille pour aucun projet. La modification est revue et un nouveau script de conversion est créé.
2. Le script crée ensuite les nouvelles structures induites par les modifications. La table *TRAVAILLE*, ses colonnes, ses clés étrangères et ses index sont créés.
3. Si des instances de la table *EMPLOYE* violent la contrainte du premier point, elles sont éventuellement transférées dans une table temporaire. Ensuite, les instances des *NumE* et *NumP* de la table *EMPLOYE* sont transférées dans la table *TRAVAILLE*.
4. Les anciennes structures sont détruites. La clé étrangère d'*EMPLOYE* vers *PROJET* ainsi que sa colonne *NumP* sont enlevées de la nouvelle base.

Il est toujours possible de créer des scripts de conversion des structures et des données d'une base relationnelle de manière à respecter les modifications apportées. Intuitivement et à condition de pouvoir garder une trace des modifications apportées sur un schéma de spécifications, une génération automatique des scripts de conversion des structures et des instances d'une base semble réalisable. L'intervention de l'ingénieur de bases de données est nécessaire pour

traiter les instances problématiques ou simplement pour décider de l'exécution du script sur la base du résultat de la requête de vérification du respect des contraintes.

1. Vérification de l'appartenance de chaque employé à un projet :

```
SELECT * FROM EMPLOYE WHERE NumP is NULL;
```

2. Création des nouvelles structures :

```
CREATE TABLE TRAVAILLE (NumP NUMERIC(5) NOT NULL,NumE NUMERIC(5) NOT NULL,
                        CONSTRAINT IDTRAVAILLE PRIMARY KEY(NumP,NumE));
ALTER TABLE TRAVAILLE ADD CONSTRAINT FKPROJET
                      FOREIGN KEY (NumP) REFERENCES PROJET(NumP);
ALTER TABLE TRAVAILLE ADD CONSTRAINT FKEMPLOYE
                      FOREIGN KEY (NumE) REFERENCES EMPLOYE(NumE);
ALTER TABLE EMPLOYE ADD CONSTRAINT EQTRAVAILLE
                    CHECK(EXISTS(SELECT * FROM TRAVAILLE
                                 WHERE TRAVAILLE.NumE = EMPLOYE.NumE));
CREATE UNIQUE INDEX IDXTRAVAILLE ON TRAVAILLE (NumP,NumE);
CREATE INDEX IDXPROJET ON TRAVAILLE (NumP);
```

3. Implémentation de la deuxième solution en cas de violation de contraintes et transfert des instances vers la nouvelle table :

```
CREATE TABLE EMPLOYE_TMP (NumE NUMERIC(5) NOT NULL,Nom CHAR(30) NOT NULL,
                          Competence CHAR(10) NOT NULL,NumP NUMERIC(5));
INSERT INTO EMPLOYE_TMP (SELECT * FROM EMPLOYE WHERE NumP is NULL);
DELETE FROM EMPLOYE WHERE NumP is NULL;
INSERT INTO TRAVAILLE(NumP,NumE) (SELECT Nump,NumE FROM EMPLOYE);
```

4. Destruction des anciennes structures :

```
ALTER TABLE EMPLOYE DROP CONSTRAINT FKPROJET;
ALTER TABLE EMPLOYE DROP NumP;
```

Figure 1.6 - Script de conversion (instructions SQL) des structures de données et des instances de la base.

1.4.3 Maintenance des programmes

La modification de la structure de la base de données nécessite, dans la majorité des cas, une propagation au niveau des programmes travaillant sur la base. L'impact des modifications au niveau opérationnel est souvent difficile à cerner et leur propagation demande une connaissance approfondie des programmes.

Considérons la section de pseudo-code de la figure 1.7 (basée sur la figure 1.5) qui illustre la complexité de la propagation. Elle exprime un traitement TrE appliqué à chaque employé ayant emprunté au moins un ouvrage et un traitement TrP appliqué au projet auquel appartiennent ces employés. Les sections TrP et TrE se présentent sous la forme de séquences d'instructions. TrE affiche le numéro et le nom des employés vérifiant la condition. TrP affiche le titre du projet de l'employé correspondant et met à jour son budget en fonction du nombre d'emprunts effectués par cette employé. La conversion consiste à transformer l'accès à l'unique projet de l'employé courant et la séquence TrP en une itération qui parcourt les projets de l'employé courant. La difficulté réside dans l'identification de la séquence TrP, dont la frontière avec TrE n'est pas toujours aisée à déterminer (c'est le cas des instructions d'affichage), d'autant plus que des instructions indépendantes du projet peuvent apparaître dans TrP. Idéalement, seules les instructions dépendant du curseur P doivent subsister dans le corps de la boucle mineure. Dans le programme modifié, les instructions d'affichage sont scindées. L'instruction d'affichage du numéro et du nom de l'employé fait partie de TrE et celles concernant les projets auxquels appartient l'employé sont dans TrP.

L'indéterminisme résultant de l'impossibilité d'identifier exactement les séquences d'instructions à modifier fait de la modification des programmes une tâche complexe, et a priori non automatisable, sauf dans des situations simples où les modifications sont mineures. En toute généralité, la modification des programmes est de la responsabilité du programmeur. Cependant, une pré-

paration du travail par une analyse du code permettrait de repérer et d'annoter les sections critiques des programmes en laissant la charge des corrections au programmeur.

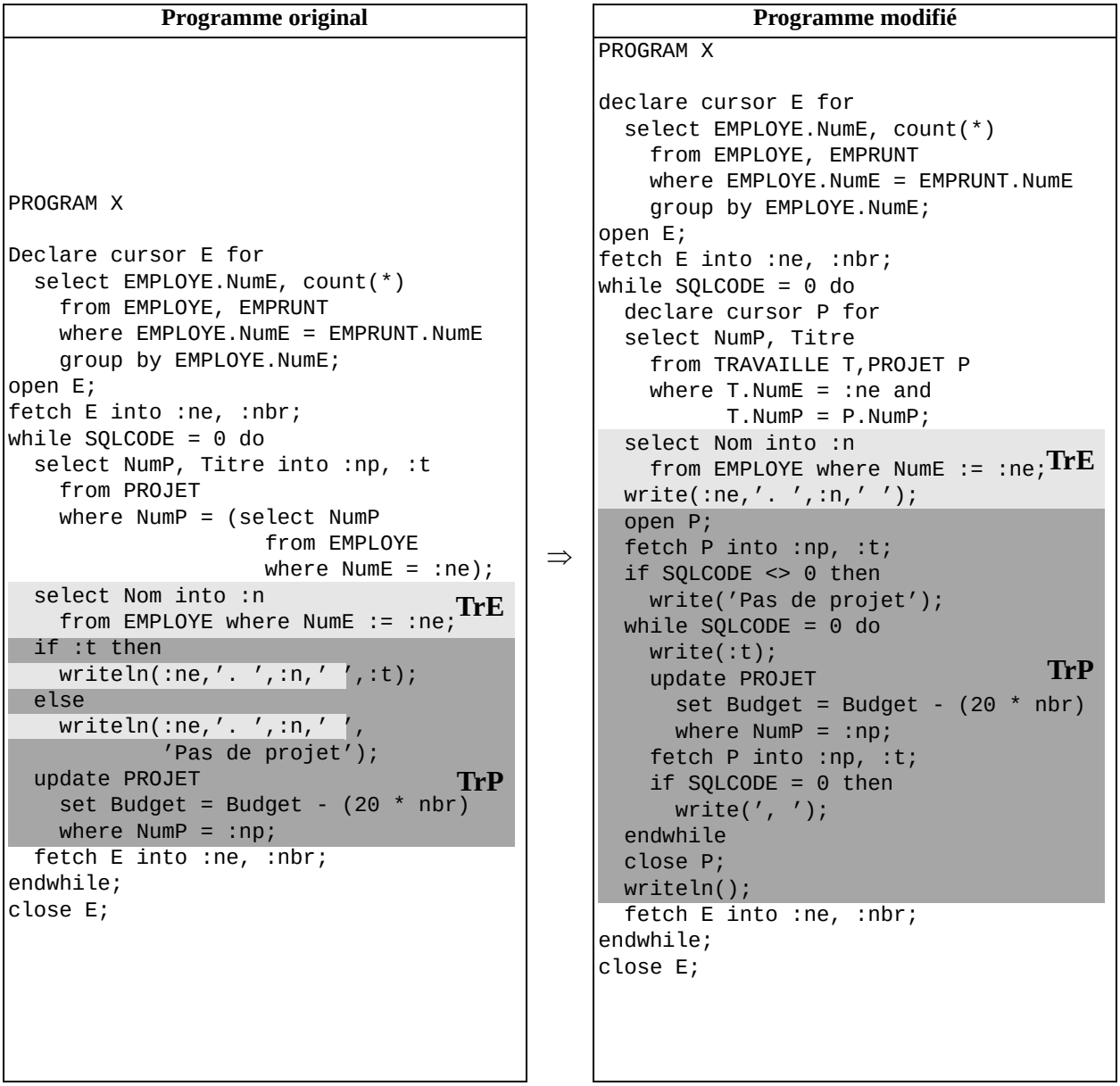


Figure 1.7 - Impact sur un programme de la modification de la figure 1.5.

1.4.4 Complexité du problème

La complexité du problème de la maintenance a été illustrée par un exemple simple. L'évolution des bases de données peut, à condition de disposer d'une trace précise des modifications opérées, être réalisée presque entièrement à partir de scripts de conversion créés automatiquement par un générateur. La modification des programmes est une tâche plus complexe qui semble difficilement automatisable excepté pour des modifications mineures. Toutefois, un support peut être fourni par un travail de repérage des zones de programmes à modifier.

Par le biais d'un environnement idéal pour les activités de maintenance, de stratégies, de techniques et d'outils, ce travail tente de répondre aux problèmes soulevés par l'exemple.

1.5 Contenu de la thèse

La thèse est structurée en huit chapitres hormis ce chapitre d'introduction.

Le chapitre 2 décrit le problème de la maintenance dans le monde de l'ingénierie du logiciel et montre les difficultés rencontrées par les organisations dans l'évolution de leur système d'information malgré les outils et les méthodes existantes. Ensuite, il définit le contexte dans lequel s'inscrit l'approche développée et précise la notion de besoins auxquels satisfait un système d'information ainsi que leurs modifications.

Le chapitre 3 présente les trois caractéristiques essentielles qu'un environnement destiné à assister le concepteur dans ses activités de maintenance d'une application de base de données doit respecter :

- L'environnement offre un modèle générique de représentation des spécifications quel que soit le niveau d'abstraction, les technologies ou les paradigmes de modélisation utilisés.
- Toutes les activités d'ingénierie des bases de données sont décrites de manière précise et rigoureuse grâce à une approche transformationnelle.
- Les relations entre les spécifications de différents niveaux d'abstraction, ainsi qu'entre les différentes versions successives de celles-ci, sont formalisées en utilisant le concept d'historique des opérations.

Dans le chapitre 4, la mise en place d'une classification de toutes les modifications possibles sur des spécifications est la première étape à réaliser avant de construire une méthodologie et des outils pour l'évolution d'une application de bases de données relationnelles. Ce travail, bien que fastidieux, constitue le point de départ sans lequel aucun résultat probant n'aurait pu être mis en valeur. La typologie des modifications est décomposée selon un enrichissement progressif du modèle de représentation des spécifications de manière à ne pas être dépasser par un trop grand nombre de modifications.

Le chapitre 5 présente trois stratégies de propagation correspondant chacune à des problèmes de modification de spécifications à un niveau d'abstraction donné. Il analyse également l'hypothèse selon laquelle il faut reconstruire les spécifications d'une application grâce à des techniques de rétro-ingénierie avant de songer à faire évoluer le système. L'étude se limite au contrôle de la propagation de modifications vers les autres niveaux et, notamment, le niveau opérationnel composé des structures de données, des données et des programmes d'application.

Eviter ou, du moins, atténuer les problèmes de maintenance en minimisant l'impact des modifications futures des spécifications sur les applications par des choix de représentation adéquats constitue le principal objectif du chapitre 6. Il tente d'extraire de l'étude réalisée des conseils pour aider les informaticiens dans leur tâche de conception en rendant les applications plus flexibles et résistantes aux modifications.

Un des buts de ce travail est de donner des outils pour aider l'ingénieur dans son activité de maintenance. Le chapitre 7 décrit un atelier (DB-Main) ainsi que les fonctions nécessaires pour mettre en œuvre la méthodologie développée. Il propose une implémentation des stratégies dans l'environnement DB-Main et des outils spécifiques pour faciliter la tâche de l'ingénieur.

Une étude de cas, reprenant les concepts, les méthodes et les outils développés, est analysée dans le chapitre 8. Elle permet de visualiser la mise en œuvre d'une stratégie d'évolution et d'apprécier l'efficacité des outils développés.

Le chapitre 9 résume la contribution de la thèse à la résolution du problème de la maintenance de systèmes d'information automatisés, centrés sur des bases de données. Il explore également de nouvelles voies de recherche dans lesquelles l'environnement, la méthode et les outils proposés pourraient être utilisés moyennant certaines adaptations.

«Les hommes sont guidés, non par des arguments, mais par des modèles».

F. Papillon

CHAPITRE 2 MODÈLE DE L'ÉVOLUTION

Ce chapitre traite du problème de l'évolution et du contexte dans lequel s'inscrit le modèle de l'évolution développé. Il est divisé en trois parties.

Premièrement, il décrit le problème dans le monde de l'ingénierie du logiciel et montre que, malgré les supports disponibles (modèles, méthodes et outils), les organisations connaissent de graves difficultés pour concevoir et surtout faire évoluer leur système d'information automatisé.

Deuxièmement, il définit le contexte dans lequel s'inscrit l'approche proposée. Pour ce faire, une démarche de conception ainsi que l'existant nécessaire avant la résolution d'un problème d'évolution sont présentés. Un repositionnement du problème est ensuite opéré vers le contexte considéré. Il nous aide à mettre en évidence les exigences qu'un outil supportant une démarche d'évolution et de maintenance doit respecter.

Finalement, la notion de besoins auxquels satisfait un système d'information est précisée ainsi que celle de modification des besoins qui est à l'origine des problèmes de maintenance de ce système.

2.1 Position du problème

2.1.1 Crise du logiciel

Depuis une trentaine d'années, des modèles et des méthodes ont été développés pour décrire, analyser et concevoir des bases de données. Plus tard sont apparus des ateliers de génie logiciel qui devaient aider le concepteur dans sa démarche de conception. A l'heure actuelle, des modèles tels les modèles Entité/Association, NIAM, UML, ..., des méthodes de conception de bases de données (comme Merise, OMT, ...) et des ateliers (Tramis, SDesigner, AMC designer, Rational Rose, Win Design, ...) sont largement répandus et disponibles dans la communauté informatique internationale par le biais de livres, de publications ou de logiciels.

De là à penser que les applications de bases de données sont conçues de manière systématique, qu'elles sont documentées, bien structurées, cohérentes et faciles à maintenir, il n'y a qu'un pas. Malheureusement, la réalité est toute autre. Les modèles et méthodes sont connus du public mais rarement mis en pratique. Les ateliers de conception vendus aux entreprises sont trop souvent inadéquats et mal utilisés (voir les points 1.1.3 et 1.1.4).

Les raisons de la sous-utilisation des supports mis à la disposition des concepteurs d'application sont nombreuses. Citons, par exemple, le manque de moyens de certaines entreprises pour acquérir des logiciels coûteux, le manque de formation des concepteurs en ce qui concerne les méthodes et les modèles, la compétitivité qui oblige les entreprises à construire des applications en peu de temps sans aucun souci pour la maintenance future. Le manque de support lors des phases critiques du cycle de vie d'un logiciel (comme la maintenance et l'évolution) est également une cause importante de cet échec.

2.1.2 Modélisation incomplète de la conception

Dans le processus de conception d'un nouveau système d'information, les supports (outils et méthodes) se basent sur un modèle incomplet du cycle de vie du système dans lequel des tâches critiques comme l'intégration de spécifications, la dérivation de vues sont très souvent absentes. Des étapes importantes du processus comme la conception logique, physique ou l'optimisation sont sous-estimées. A titre d'exemple, le concepteur dispose d'ateliers permettant de construire un schéma conceptuel du système, de le transformer de manière automatique en un schéma relationnel et de générer le code SQL correspondant. Ce code doit être fortement remanié pour tenir compte de contraintes techniques et pour améliorer les performances. En conséquence, les outils génèrent des prototypes permettant de valider le schéma conceptuel mais sans correspondance avec le code optimisé pour un environnement spécifique. En cas de restructuration, le schéma conceptuel étant sans correspondance directe avec l'application, l'outil n'apporte aucun soutien pour la maintenance de la base. Dans le cas de grosses applications, faire évoluer un système peut se révéler extrêmement coûteux. Alors que si toutes les phases de la démarche étaient correctement documentées, la maintenance serait mieux supportée et, dans certains cas, automatisable.

On constate également que les supports (méthodes et ateliers) négligent complètement des activités qui peuvent apparaître après la conception du système. Ils n'offrent aucun support pour des tâches comme la migration de données, la redocumentation, la conversion, la maintenance ou l'évolution.

2.1.3 Modification, évolution ou version de schémas ?

Un schéma de spécifications de données n'est pas statique ce qui engendre le besoin de mécanismes pour l'évolution. La communauté scientifique reconnaît l'existence de trois approches dif-

férentes pour satisfaire cette exigence. Les définitions proposées s'accordent avec la majorité des recherches effectuées dans le domaine [And91] [Ari91] [Jen94] [Rod95] :

1. La *modification de schémas* consiste à modifier de manière irréversible les spécifications sans se soucier des données et de l'état existant avant le processus de modification.
2. L'*évolution de schémas* autorise la modification des spécifications indépendamment des données. Les changements sont ensuite propagés sur les instances de la base pour correspondre au nouveau schéma tout en conservant les données existantes. L'implémentation du nouveau schéma détruit le précédent sans possibilité de retrouver les anciennes spécifications.
3. La technique des *versions de schémas* [1] permet de conserver l'état du schéma avant sa modification. Elle nécessite des mécanismes de stockage et d'accès à tous les états (spécifications et données) passés et présents. Pour plus d'informations sur les différentes mises en œuvre du contrôle des versions, le lecteur peut consulter [And91].

La modification de schémas est une approche radicale du problème de l'évolution qui satisfait rarement aux besoins des organisations qui désirent améliorer leur système informatique en respectant l'existant, notamment les instances de la base de données. Cette technique est parfois employée dans le cas très particulier où la base est vide.

Dans le contexte des systèmes relationnels traditionnels, l'analyse proposée se base sur l'évolution de schémas (point 2), c'est-à-dire un scénario fréquent selon lequel tous les composants de l'application sont remplacés par une nouvelle version, et que l'application nouvelle se substitue complètement à l'ancienne. Sauf procédure de sauvegarde des anciens composants, il n'est plus possible d'accéder à la version précédente de l'application, et en particulier aux données qui n'auraient pas été reprises dans la nouvelle version. Concrètement, l'ajout d'une nouvelle propriété à une classe d'objets conceptuelle se traduira, dans une base relationnelle, par la modification de celle-ci via une requête du type `alter table <table name> add <column spec>`.

Autrement dit, on s'écarte des architectures avancées (essentiellement les systèmes orientés objets), dans lesquelles la modification du schéma de la base de données peut se traduire par l'ajout d'une nouvelle version de celui-ci avec conservation de l'ancien schéma et de l'accès à ses données, mémorisées ou recalculées ("schema/data versioning").

1. Le terme "schema versioning" s'utilise en anglais.

2.2 Contexte de travail

2.2.1 Modélisation du processus de conception

Comme cela a déjà été dit dans le chapitre 1 (point 1.1.2.1), on considère l'analyse du phénomène de l'évolution dans le cadre de la modélisation classique qui envisage la conception d'une base de données comme une activité complexe décomposable en processus plus élémentaires. Ces processus vont de la collecte des informations concernant les besoins des utilisateurs jusqu'à la génération de code. La conception de bases de données est un processus qui transforme l'expression des besoins des utilisateurs en une collection de schémas et de programmes qui respectent des critères de correction, d'efficacité, d'opérationalité, ... La figure 2.1 présente la démarche de conception adoptée qui se base sur le cycle de vie d'une base de données représenté à la figure 1.3 (page 7).

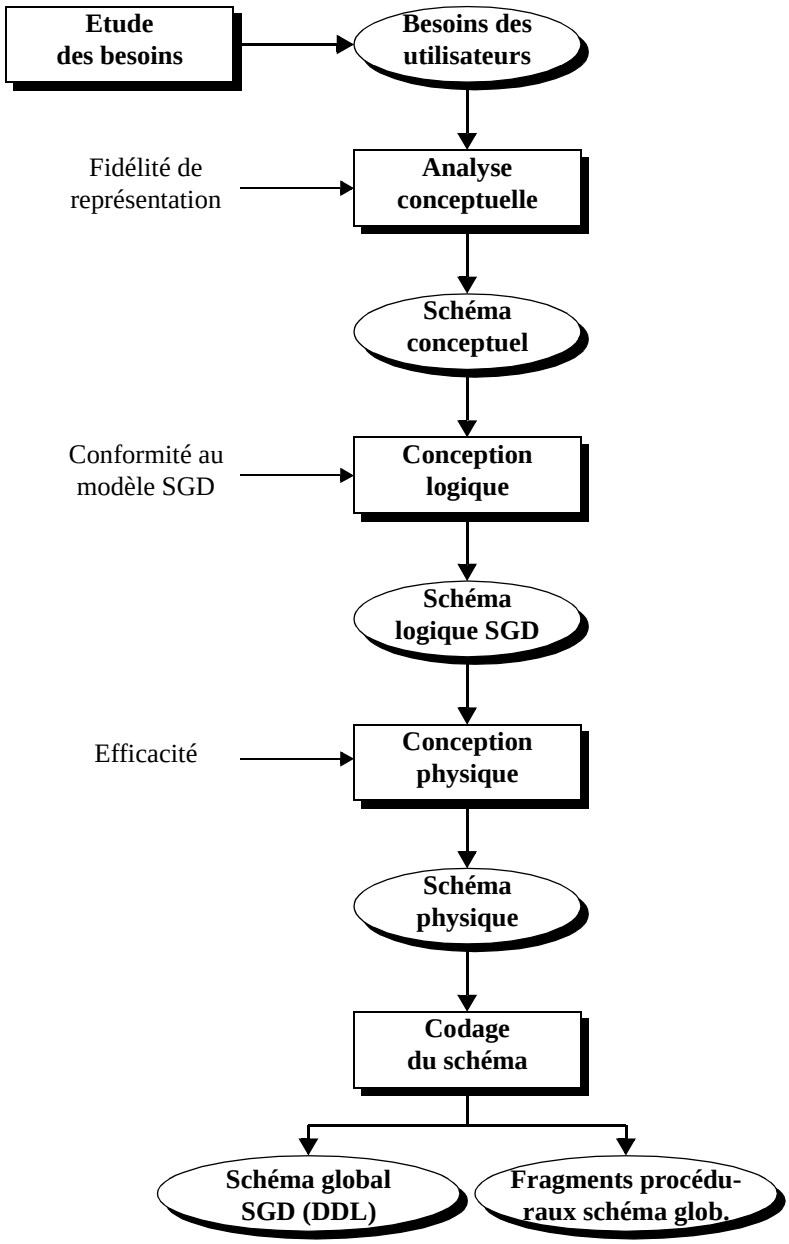


Figure 2.1 - Une stratégie standard pour la conception de bases de données.

La démarche de conception doit permettre la définition du schéma global nécessaire aux besoins des utilisateurs. Elle se décompose en :

- L'étude des besoins qui délimite le *domaine d'application*, détermine les services que le système doit offrir et les spécifications des caractéristiques d'utilisation.
- L'analyse conceptuelle qui donne une spécification formelle (langage précis et rigoureux), complète (exprimant tous les besoins), minimale (rien que les besoins), non ambiguë (n'ayant pas plusieurs interprétations), abstraite (indépendante de tout modèle technique), lisible (image fidèle de la réalité perçue pas les utilisateurs) et non technique des informations. Cette spécification est matérialisée dans un schéma conceptuel qui est un modèle abstrait pas directement implémentable dans un système informatique.
- La conception logique qui élabore les structures de données en fonction du modèle logique choisi (le schéma logique). Ce modèle logique est propre à une famille de SGD [2].
- La conception physique qui spécifie les caractéristiques techniques d'implémentation de la base de données (le schéma physique).
- Le codage qui produit le code exécutable selon le SGD choisi.

On pourrait se demander pourquoi ne pas construire un schéma physique directement implémentable sur la base de l'étude des besoins en sautant plusieurs étapes du processus. Il existe plusieurs arguments plaidant en faveur de cette décomposition. Citons les trois principaux :

1. Dans le cas de domaines d'applications complexes [3], un raisonnement indépendant des outils de gestion de données (matérialisé dans un schéma conceptuel) s'avère indispensable. Le modèle conceptuel (s'il est bien choisi) permet de décrire plus naturellement les concepts du domaine, sans s'attacher aux détails techniques, selon une représentation graphique attrayante. Le schéma conceptuel est plus clair et plus facile à interpréter que les schémas logique et physique.
2. Le modèle conceptuel indépendant de toute technologie peut conduire à des implémentations dans des familles d'outils différentes (SQL, COBOL, CODASYL, SGBD orientés objets, ...).
3. Le caractère de lisibilité du schéma conceptuel permet de l'utiliser dans le dialogue avec les utilisateurs du système afin de parvenir à une solution acceptable.

La démarche choisie (avec toutefois quelques variantes) est décrite et utilisée par nombre d'auteurs et, donc, familière aux concepteurs. Les lecteurs intéressés trouveront des démarches similaires dans [Bat92], [Bod89], [Con96], [Elm94] et [One94].

2.2.2 Existant

Selon la démarche adoptée, trois niveaux d'abstraction correspondant chacun à des spécifications intégrant une famille de critères de conception sont définis. Il s'agit des niveaux conceptuel, logique et physique. Selon la figure 2.2, le schéma conceptuel répond aux besoins fonctionnels et organisationnels R1, le schéma logique aux besoins organisationnels et techniques R2 et le schéma physique aux besoins techniques R3. La partie opérationnelle contient la base de données (données et structures de données) et les programmes.

Selon cette hypothèse de travail, nous disposons donc des spécifications aux différents niveaux d'abstraction. La modification des besoins se traduisant, suivant des règles ignorées dans le cadre de ce travail, en modification des spécifications à un certain niveau, le problème consiste alors à rendre cohérentes toutes les spécifications, et donc à propager les modifications vers les autres niveaux d'abstraction, tant en amont qu'en aval.

2. Système de Gestion de Données : terme générique recouvrant ceux de SGBD et SGF.
3. La complexité d'un domaine d'application est souvent déterminée par la quantité de structures représentées. Il n'est pas rare que certains domaines contiennent des centaines de milliers de types d'informations différentes.

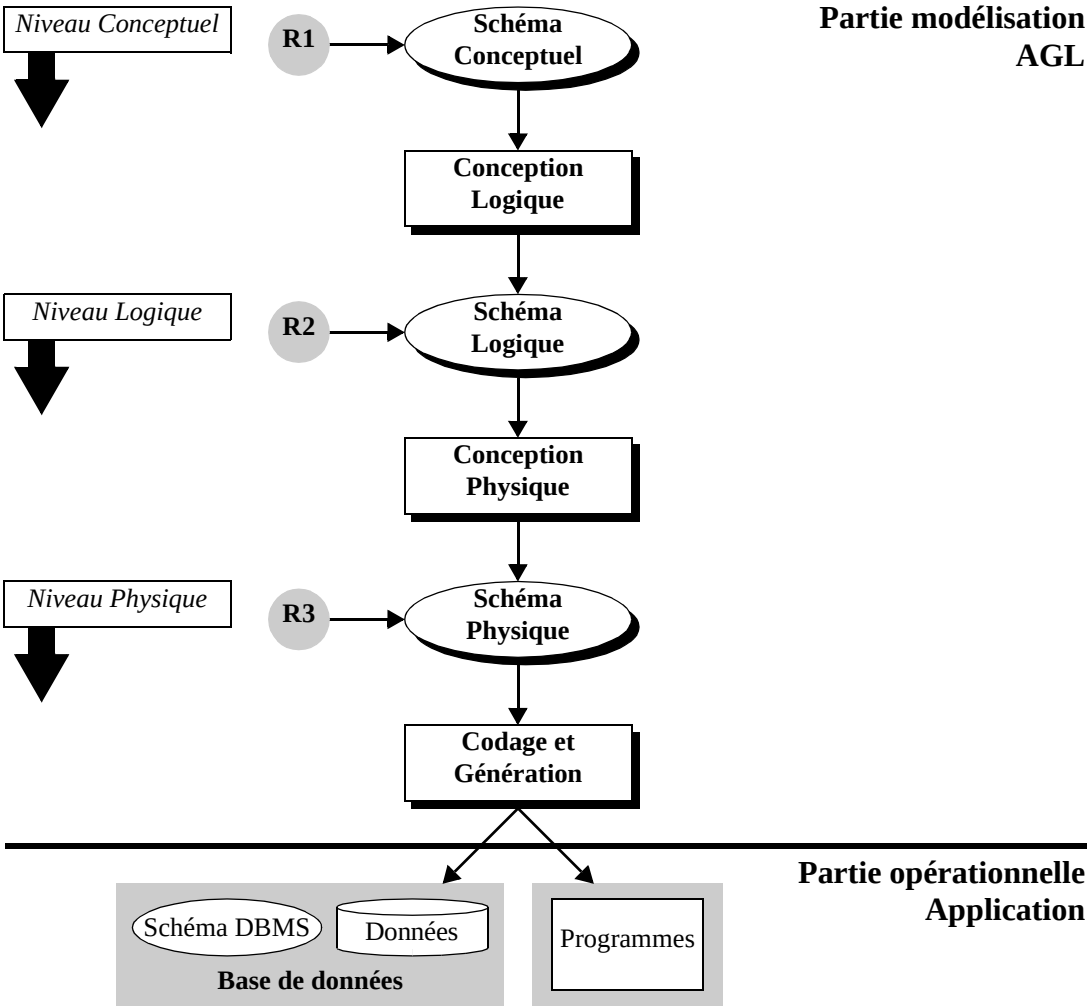


Figure 2.2 - Modélisation classique définie selon trois niveaux d'abstraction.

2.2.3 Repositionnement du problème

Dans le contexte présenté, le problème de l'évolution peut être affiné et décomposé en quatre catégories de problèmes. Considérons une base de données qui satisfait les besoins R0 (figure 2.3). Cette base comprend des spécifications S0 et une instance de la base D0. S0 est composée d'un schéma conceptuel SC0, un schéma logique SL0 et un schéma physique SP0. Un ensemble de programmes P0 a été construit sur la base de données. Ils travaillent sur les données D0 à travers les spécifications SP0.

Si les besoins R0 changent en R1, il faut traduire les changements en modification des spécifications de S0 (SC0, SL0 et SP0) qui produit les nouvelles spécifications S1 (SC1, SL1 et SP1). Les instances D0 ne sont plus valides et doivent être converties en instances D1. Les programmes P0 doivent être réécrits en partie (P1 contient les nouveaux programmes) pour être conformes aux nouvelles instances D1.

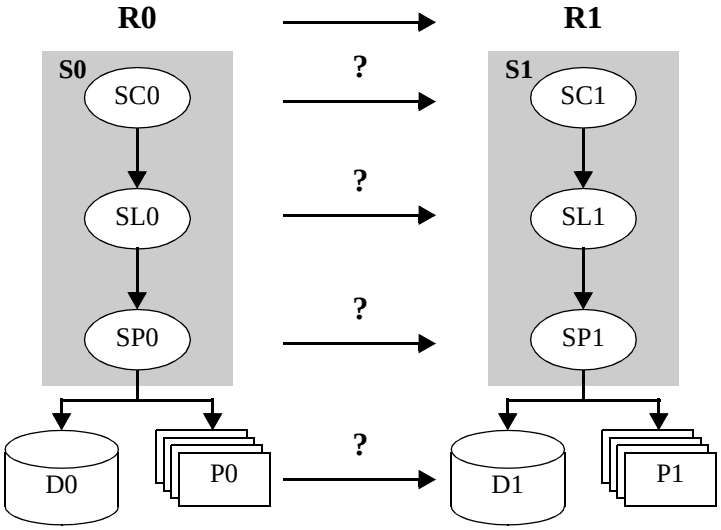


Figure 2.3 - Représentation du problème de l'évolution d'une base de données.

De cette représentation, quatre catégories de situations se dégagent [4] :

1. Les spécifications S0 sont disponibles et correctes. La modification des besoins est traduite par des changements dans le schéma conceptuel SC0. SC1 est le nouveau schéma conceptuel qui vérifie R1. Le problème est de propager les modifications de SC0 en aval vers les schémas logique SL0 et physique SP0, les données D0 et le programme P0 pour produire SL1, SP1, D1 et P1.
2. Les spécifications S0 sont disponibles. Mais, pour diverses raisons (contraintes de temps, faiblesse des outils, changements de besoins techniques), les changements dans les besoins R0 sont directement traduits en modifications relatives au schéma logique SL0 pour que SL1 satisfasse les besoins R1. Le problème est de propager les modifications de SL0 en amont vers le schéma conceptuel SC0 pour obtenir le nouveau schéma conceptuel SC1 qui est la conceptualisation exacte du schéma logique SL1. La propagation en aval vers le schéma physique SP0, les instances D0 et les programmes P0 pour obtenir SP1, D1 et P1 relève de la première catégorie de situations.
3. Les spécifications S0 sont toujours disponibles. Les changements des besoins (passage de R0 à R1) sont traduits par des modifications du schéma physique SP0 qui devient SP1 pour des raisons variées (modifications des besoins liées au SGBD ou à des concepts techniques uniquement présents dans le schéma physique, amélioration des performances, ...). Les modifications du schéma physique doivent être propagées en amont vers les schémas logique et conceptuel pour obtenir les nouveaux schémas SL1 et SC1. D0 et P0 doivent être changés pour satisfaire les nouveaux besoins R1. On obtient D1 et P1.
4. Dans de nombreux cas, les seules informations disponibles sont le code DDL [5], les données de la base et les programmes d'applications. Les spécifications S0 sont manquantes. Le quatrième type de problème concerne la reconstruction des spécifications avant de faire évoluer le système (ce qui nous ramène à un des trois types de situations décrits ci-dessus).

Comme on le voit, les trois premières catégories de problèmes concernent les modifications d'un niveau d'abstraction. Une fois que les modifications du niveau sont faites, il faut propager les modifications (en amont et/ou en aval) pour que les spécifications S1, les données D1 et les programmes P1 satisfassent les nouveaux besoins R1. La quatrième catégorie pose le problème des spécifications incomplètes ou simplement absentes. Le chapitre 5 tentera d'apporter des solutions à ces catégories de problèmes sous la forme de stratégies de propagation.

4. Pour rappel, le travail ne concerne pas le problème de la modification de S1 suite à la modification de R0.
5. pour "Data Definition Language". Il s'agit des scripts permettant de créer les structures de la base de données.

Une autre catégorie de problèmes pourrait être envisagée à partir du constat suivant : certains systèmes sont plus faciles à faire évoluer à cause de certains choix de conception et de style de programmation. Le problème de la maintenance devrait être envisagé plus tôt, c'est-à-dire dès la conception d'un nouveau système. Dans ce contexte, il serait intéressant de proposer des techniques de conception et des raisonnements pour construire des bases de données et des programmes plus robustes face aux modifications des besoins ou, du moins, plus facile à faire évoluer. Le chapitre 6 proposera une série de recommandations méthodologiques qui peuvent aider le concepteur à atteindre cet objectif.

2.2.4 Exigences pour un environnement de maintenance

Sur la base des cas de figure mis en évidence dans le point 2.2.3, certaines exigences peuvent être mises en avant sur un environnement qui aiderait le concepteur dans sa tâche d'évolution et de maintenance d'un système d'information :

1. L'environnement doit permettre de représenter les spécifications à n'importe quel niveau d'abstraction. Il doit aussi être suffisamment complet pour décrire toutes les spécifications.
2. L'environnement doit être indépendant d'un SGBD. Il doit être capable de représenter les spécifications des différents modèles tels que les modèles relationnels, orientés objets ou basés sur des fichiers standards. Dans le cas contraire, il est limité à un type de SGBD.
3. Les relations entre les spécifications doivent être formalisées. Elles doivent être analysées et manipulées de manière à permettre le raisonnement et la dérivation de nouvelles spécifications. Par exemple, les relations entre deux niveaux d'abstraction doivent être correctement formalisées pour décrire comment des spécifications sont dérivées d'autres spécifications afin de permettre une propagation quasi-automatique des modifications.
4. L'environnement doit supporter toutes les activités d'ingénierie. Si une activité n'est pas décrite dans l'environnement, elle échappe à tout contrôle et ne peut plus être manipulée.
5. L'environnement doit être indépendant de toute méthodologie. Il n'y a pas de méthodologie standard acceptée par tous les acteurs du monde informatique et certaines activités ne sont pas formalisées.

Dans le chapitre 7, nous présenterons l'AGL DB-Main, conçu de manière à respecter les exigences décrites ci-dessus. Les fonctions de cet atelier permettent de fournir au concepteur une aide pour les activités complexes d'ingénierie comme la maintenance et l'évolution.

2.3 Besoins par niveau d'abstraction

2.3.1 Introduction

Les applications de bases de données sont amenées à évoluer au cours de leur cycle de vie. Comme nous l'avons déjà dit, cette évolution provient de changements dans les besoins de l'entreprise ou de l'apparition de nouveaux besoins. Suite à cette évolution des besoins, le concepteur apporte des modifications techniques à l'application. La démarche de conception proposée est basée sur trois types de besoins.

Premièrement, les systèmes d'information satisfont les besoins exprimés par les organisations. Dans [Bod89], les auteurs affirment que «*Les organisations conçoivent, réalisent et utilisent des systèmes d'information pour satisfaire les besoins en information engendrés par leurs comportements organisationnels*». Le cadre de travail des entreprises définit un certain nombre de besoins appelés *besoins organisationnels*.

Deuxièmement, le processus d'analyse des besoins va dégager un certain nombre d'exigences des utilisateurs du système en termes de fonctions exécutées. Il s'agit de *besoins fonctionnels* qui représentent les exigences du système en termes de fonctionnalité.

Troisièmement, le système développé est tributaire de contraintes matérielles (dépendant du système informatique), logicielles (dépendant du SGBD mis en œuvre) et techniques concernant la performance. Il doit donc satisfaire un certain nombre de *besoins techniques*.

Il existe d'autres classifications des besoins. [Chu91a], [Som92], [Hai94] et [Kro95] distinguent les besoins fonctionnels définissant les fonctionnalités des composants du domaine d'application et les besoins non-fonctionnels comme la sécurité, la convivialité, le coût, les performances et d'autres contraintes globales. Cette classification est proche de celle présentée si on considère que les besoins non-fonctionnels regroupent les besoins techniques et organisationnels. [Kro92] et [Elm94] parlent aussi des besoins du modèle de données et des besoins fonctionnels. Kano [Kan84] distingue dans son modèle trois types de besoins influençant la satisfaction des utilisateurs : les besoins attendus qui sont les critères de base (souvent implicites) d'un produit, les besoins normaux qui sont les désirs exprimés par les utilisateurs et les besoins attractifs qui ne sont pas attendus mais qui procurent un niveau de satisfaction élevé (voir aussi [Mac96]).

La traçabilité des besoins est nécessaire lorsqu'on décrit les besoins respectés par chaque niveau d'abstraction. Il faut conserver les relations entre les spécifications de tous les niveaux et les exigences qui les ont générées ou influencées. De cette façon, la modification d'un besoin est directement répercutée vers les spécifications influencées grâce aux relations existantes. Ces relations facilitent la propagation des modifications des besoins ce qui simplifie la tâche du concepteur qui ne doit plus localiser les spécifications touchées par des modifications. La recherche dans le domaine de l'ingénierie des exigences et de la traçabilité des besoins est riche. Le lecteur intéressé peut consulter, par exemple, [Got94], [Ram97] ainsi que le projet NATURE décrit au point 1.2.1.5 (page 13). Pourtant, dans la pratique, on constate que les besoins sont souvent informels ou inexistants, ce qui empêche toute tracabilité. Ce constat ne fait que renforcer l'idée d'une modélisation incomplète du processus de conception d'une application rendant encore plus difficile des tâches cruciales comme la maintenance.

Le but de ce travail n'est pas d'analyser la traçabilité des besoins ou leurs modifications. Il s'agit simplement de décrire un principe, communément admis par chacun, qui affirme que la modification des besoins auxquels satisfait un système d'information engendre des modifications des spécifications de la base de données.

Dans cette section, nous allons voir comment les besoins satisfaits par chaque niveau d'abstraction sont spécifiques et vitaux pour produire un système d'information intégrant les fonctionnalités nécessaires aux utilisateurs en respectant certaines contraintes techniques. Cette introduction sur les besoins et leur influence sur les spécifications met en évidence la nécessité

de définir plusieurs stratégies de propagation des modifications car, quel que soit le niveau d'abstraction, tous les besoins sont susceptibles d'évoluer.

2.3.2 Niveau conceptuel

Un schéma conceptuel est le résultat de l'analyse conceptuelle qui est basée sur une analyse des besoins. L'objectif est de produire une spécification des informations qui correspondent aux besoins exprimés explicitement ou implicitement par les utilisateurs. Ces besoins sont fonctionnels ou organisationnels et sont tirés de sources aussi différentes les unes des autres que des rapports d'interview, des formulaires, des saisies d'écran, des bordereaux, des textes techniques, bref toute source d'information sur l'organisation et la façon dont le travail y est réalisé. A titre d'exemple, citons :

- Besoins fonctionnels :
  - représentation d'un domaine d'activités,
  - interprétation des informations stockées dans une entreprise,
  - gestion des départements d'une société.
- Besoins organisationnels :
  - possibilité de réagir plus rapidement au mouvement du marché,
  - amélioration de la communication interservices dans une organisation,
  - satisfaction et motivation du personnel.

Dans la démarche choisie, l'analyse conceptuelle formalise sous la forme de schémas Entité/Association [6] les concepts sous-jacents aux diverses sources d'information concernant les besoins des utilisateurs. La traduction des besoins en structures formelles est le point essentiel et crucial de la phase d'analyse conceptuelle. Bien qu'il dépasse le cadre de ce travail, le chapitre 6 y reviendra brièvement pour la mise en place de recommandations méthodologiques lors de la phase de conception d'une application de bases de données.

2.3.3 Niveau logique

L'objectif de la conception logique est de produire un schéma logique équivalent au schéma conceptuel en ce qui concerne le contenu informationnel et conforme au modèle logique. Pour obtenir un tel schéma, il faut utiliser des transformations réversibles qui ne modifient pas la sémantique du schéma et qui transforment des spécifications du niveau conceptuel en structures et contraintes directement exprimables dans le modèle.

Le schéma logique répond à des besoins techniques et organisationnels. A titre d'exemple, voici quelques critères classés par type de besoins :

- Besoins techniques :
  - respect de standards méthodologiques de conception, de développement et de programmation,
  - utilisation d'atelier de génie logiciel,
  - extensibilité,
  - conformité au modèle utilisé par le SGBD.
- Besoins organisationnels :
  - délai de développement,
  - coût du développement,

6. Egalement représenté par le sigle E/A.

- coût de la maintenance,
- ergonomie,
- facilité d'exploitation,
- sécurité, confidentialité.

Le schéma logique dépend des contraintes techniques définies par le modèle. Dans ce contexte, un changement de SGBD peut introduire un nouveau modèle au sein de l'organisation et nécessiter un nouveau schéma logique. On est alors dans une problématique de migration de données. En effet, les anciennes données qui vérifiaient les contraintes des schémas logiques et physiques doivent migrer dans une autre base satisfaisant les contraintes d'un nouveau modèle. La migration de données ne pourrait-elle pas être appréhendée comme un cas particulier du problème de la maintenance et de l'évolution d'une application de bases de données ? Dans la conclusion, le point 9.4 tentera d'apporter un début de réponse à cette question. La méthode définie dans ce document ainsi que les outils développés pourraient, moyennant certaines adaptations, servir de support au problème de la migration de données.

2.3.4 Niveau physique

La conception physique a pour objectif de produire un schéma de structures de données techniques qui est équivalent au schéma logique, conforme au modèle physique du SGBD et efficace vis-à-vis des traitements. Le schéma physique est obtenu par l'enrichissement du schéma logique. Il ne spécifie que les contraintes et constructions techniques offertes par le SGBD. En plus, il doit offrir des performances satisfaisantes par rapport à son utilisation. Par exemple, il répond à des besoins techniques tels que :

- les performances en temps d'exécution (pour certaines requêtes ou certains programmes),
- les performances en espace occupé (sur les disques, en mémoire centrale),
- les paramètres de stockage,
- la taille, la gestion ou le verrouillage des tampons, ...

Les critères de performance sont un des points sensibles de la conception physique. Une application gourmande en temps et en espace a peu de chance de s'imposer auprès des utilisateurs. Pour répondre à ces exigences, une analyse des besoins des traitements est nécessaire. Elle permet de déterminer les plans d'accès aux données, de collecter et de calculer les statistiques et les fréquences. Sur la base de ces informations (les accès nécessaires et les quantifications), il sera possible de produire un schéma physique optimisé. [Hai86] et [Hai99] approfondissent l'optimisation des schémas physiques.

«Plus l'édifice doit monter haut, plus les fondations doivent être solides».

R. Plus, s.j.

CHAPITRE 3 CADRE MÉTHODOLOGIQUE

Comme on l'a décrit au chapitre 2, on considère l'analyse du phénomène de l'évolution dans le cadre de la modélisation classique qui envisage la conception d'une base de données comme une activité complexe décomposable en processus plus élémentaires. Selon cette hypothèse de travail, des spécifications aux différents niveaux d'abstraction sont disponibles. La modification des besoins se traduisant en modification des spécifications à un certain niveau, le problème consiste alors à rendre cohérentes toutes les spécifications, et donc à propager les modifications vers les autres niveaux d'abstraction, tant en amont qu'en aval.

Dans ce contexte, un environnement destiné à assister le concepteur dans ses activités de maintenance et d'évolution d'une application de bases de données et satisfaisant les exigences présentées au point 2.2.4 doit respecter trois caractéristiques essentielles. Premièrement, l'environnement doit offrir un modèle générique de représentation des spécifications quel que soit le niveau d'abstraction, les technologies ou les paradigmes de modélisation utilisés. C'est la propriété de généricité. Deuxièmement, l'environnement doit pouvoir décrire de manière précise et rigoureuse toutes les activités d'ingénierie des bases de données. Toute opération exécutée sur une spécification, telle que la dérivation ou la modification, doit pouvoir être décrite de manière formelle. On parle d'une propriété de formalité. Troisièmement, les relations entre les spécifications de différents niveaux, ainsi qu'entre les différentes versions successives de celles-ci, doivent être formalisées. Elles doivent pouvoir être analysées et manipulées de manière à fournir les informations nécessaires à la propagation des modifications. Il s'agit de la propriété de traçabilité.

Les trois sections de ce chapitre présentent les concepts de base sur lesquels la méthodologie a été élaborée, c'est-à-dire, un modèle générique permettant de représenter des spécifications multiniveaux et multiparadigmes (section 3.1), une approche transformationnelle de l'ingénierie de bases de données, y compris de la rétro-ingénierie (section 3.2), le concept d'historique des opérations effectuées sur des spécifications (section 3.3).

3.1 Modèles génériques de représentation

3.1.1 Introduction

Le modèle utilisé dans l'approche proposée présente des caractéristiques de généricité selon deux dimensions. Il permet :

- de représenter des spécifications aux différents niveaux d'abstraction, et notamment des spécifications conceptuelles, logiques et physiques;
- de couvrir, à chaque niveau, les principaux paradigmes de modélisation et technologies tels que les modèles Entité/Association, UML, NIAM, à Objets, relationnels, CODASYL, IMS, fichiers, etc.

Il est basé sur un modèle Entité/Objet-Association [1] étendu dont les objets sont présentés au point 3.1.2 et qui permet de définir les différents modèles propres à la démarche méthodologique de conception (point 3.1.3).

3.1.2 Présentation des objets du modèle

Le modèle de spécification est basé sur le modèle Entité/Association standard avec quelques extensions et contient sept objets génériques (schéma, type d'entités, type d'associations, collection, attribut, groupe et contrainte) définissant des structures de données. Chaque type d'objet est présenté en détail ci-dessous.

3.1.2.1 Schéma

Un schéma est une description complète ou partielle d'une structure de données. Il est constitué de types d'entités, de type d'associations et/ou de collections. Un schéma est représenté graphiquement par une ellipse épaisse (figure 3.1).

Figure 3.1 - Représentation graphique d'un schéma contenant les spécifications d'un système de gestion des commandes.

3.1.2.2 Type d'entités

Un type d'entités représente une classe d'entités concrète ou abstraite du monde réel, une unité d'information qui peut être perçue et manipulée comme un tout, à n'importe quel niveau du processus de conception. Il peut être utilisé pour modéliser des objets conceptuels ou des objets techniques (tables, enregistrements, segments, ...).

Dans un modèle orienté objets, on utilise le terme de *classe d'objets*. Les classes d'objets ont des méthodes et apparaissent dans des hiérarchies.

Un type d'entités peut être un sous-type d'un ou plusieurs autres types d'entités qui sont appelés les surtypes (figure 3.2). Cela signifie que chaque entité du sous-type est une entité des surtypes. Une relation surtype/sous-type est interprétée comme "*chaque sous-type est un surtype*", en utilisant la traduction anglaise (is a), on parle de relations *IS-A*. L'ensemble des sous-types d'un type d'entités est déclaré *total* si chaque entité du surtype appartient au moins à un sous-type sinon il est partiel. L'ensemble est qualifié de *disjoint* si une entité d'un sous-type ne peut

1. Le modèle Entité/Association a été proposé pour la première fois par Chen [Che76].

appartenir à un autre sous-type sinon il y a recouvrement. Un ensemble de sous-types disjoint et total forme une *partition*.

Un type d'entités peut avoir des attributs, peut jouer des rôles dans des types d'associations, peut appartenir à des collections et peut avoir des contraintes (définies sur des groupes).

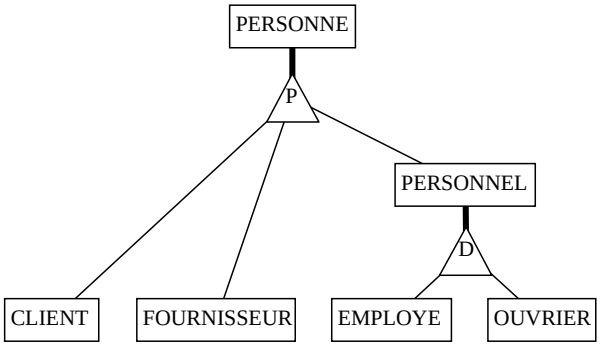


Figure 3.2 - Une hiérarchie de types d'entités : *CLIENT*, *FOURNISSEUR* et *PERSONNEL* sont des sous-types de *PERSONNE*. *PERSONNEL* est surtype de *EMPLOYE* et *OUVRIER*.

3.1.2.3 Type d'associations (et rôle)

Un type d'associations représente une classe d'associations entre entités, une agrégation significative d'au moins deux types d'entités (figure 3.3). Chaque position dans l'agrégation est appelé un rôle qui est joué par un ou plusieurs types d'entités. Un type d'associations avec deux rôles est dit *binaire*, tandis qu'un type d'associations avec n (supérieur à 2) rôles est généralement appelé *n-aire* [2]. Un type d'associations avec deux rôles joués par le même type d'entités est dit *cyclique*.

Ce concept peut être utilisé pour modéliser un type d'associations conceptuel ou des constructions techniques (comme par exemple les "set types" CODASYL, les relations parent/enfant IMS, les chemins TOTAL, ...).

En règle générale, un rôle est joué par un seul type d'entités. Si plusieurs types d'entités jouent le rôle, on parle de rôle *multi-TE*. Chaque rôle est caractérisé par ses cardinalités [i-j], signifiant qu'une entité de ce type doit apparaître dans i à j associations par le biais du rôle concerné. On a comme contrainte sur les cardinalités : $0 \leq i \leq j \leq N$ et $j > 0$ (N représentant l'infini).

Un type d'associations binaire entre les types d'entités E1 et E2 avec les cardinalités [i1-j1] pour E1 et [i2-j2] pour E2 est appelé :

- un-à-un si j1 = 1 et j2 = 1,
- un-à-plusieurs de E1 vers E2 si j1 > 1 et j2 = 1,
- plusieurs-à-un de E1 vers E2 si j1 = 1 et j2 > 1,
- plusieurs-à-plusieurs si j1 > 1 et j2 > 1,
- facultatif pour E1 si i1 = 0,
- obligatoire pour E1 si i1 > 0.

Un type d'associations peut avoir des attributs et des contraintes (par le biais de ces groupes). Un type d'associations qui a des attributs ou qui est n-aire (n>2) est appelé type d'associations *complexe*. Un type d'associations un-à-un ou un-à-plusieurs sans attribut est appelé type d'associations *fonctionnel*.

2. n est le degré du type d'associations. Il spécifie le nombre de rôles.

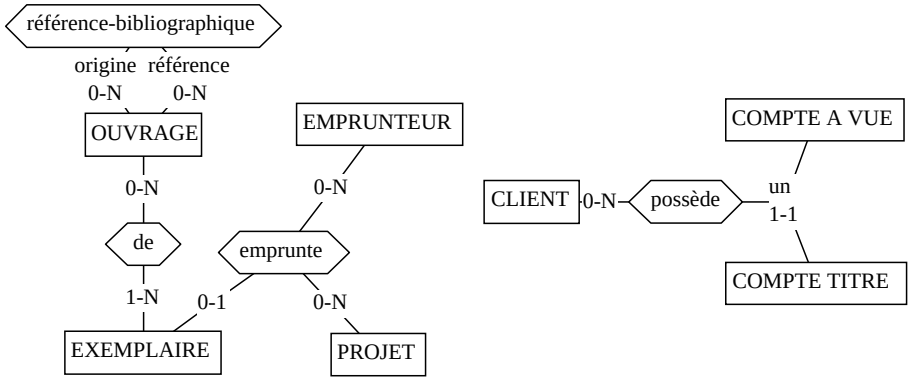


Figure 3.3 - Exemples de types d'associations. *référence-bibliographique*, *de* et *possède* sont binaires. *emprunte* est ternaire. Le rôle *un* de *possède* est multi-TE et *référence-bibliographique* est un type d'associations cyclique.

3.1.2.4 Collection

Une collection est un ensemble d'entités (figure 3.4). Une collection peut contenir des entités de différents types et des entités d'un type donné peuvent appartenir à différentes collections. Ce concept est souvent utilisé dans les schémas physiques pour représenter des espaces de stockage comme des fichiers COBOL, des "dbspaces", des "tablespaces", ...

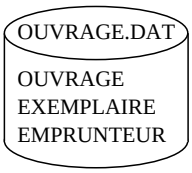


Figure 3.4 - *OUVRAGE.DAT* est une collection dans laquelle les entités *EMPRUNTEUR*, *EXEMPLAIRE* et *OUVRAGE* sont stockées.

3.1.2.5 Attribut

Un attribut représente une propriété commune à toutes les entités (ou associations) d'un type donné (figure 3.5). Un attribut *simple* a un domaine de valeurs défini par un type élémentaire (numérique, caractère, booléen, date ou réel) et une longueur (valeur entière de 1 à N). On parle d'attribut *atomique*.

Un attribut peut aussi avoir des attributs comme composants. Dans ce cas, l'attribut est *décomposable*. Le parent d'un attribut est soit un type d'entités, soit un type d'associations, soit un attribut décomposable. Un attribut d'un type d'entités ou d'un type d'associations est de niveau 1. Par contre, un attribut d'un attribut décomposable de niveau i est de niveau i+1.

Chaque attribut est caractérisé par une cardinalité [i-j] qui est une contrainte indiquant que chaque parent a de i à j valeurs de cet attribut. Les cardinalités doivent respecter les deux règles suivantes : $0 \leq i \leq j \leq N$ et $j > 0$. La cardinalité par défaut est [1-1] et n'est pas représentée graphiquement. Un attribut avec la cardinalité [i-j] est appelé :

- monovalué si j = 1,
- multivalué si j > 1,
- facultatif si i = 0,
- obligatoire si i > 0.

Un attribut multivalué représente un ensemble de valeurs, c'est-à-dire une collection non structurée de valeurs distinctes. En fait, il existe six catégories de collections de valeurs :

- Ensemble (représenté graphiquement par le mot anglais *set*) : une collection non structurée de valeurs distinctes (c'est la catégorie définie par défaut).
- Amas (*bag*) : une collection non structurée de valeurs pas nécessairement distinctes.
- Liste unique (*u-list*) : une collection séquencée de valeurs distinctes.
- Liste (*list*) : une collection séquencée de valeurs pas nécessairement distinctes.
- Tableau unique (*u-array*) : une séquence de cellules pouvant contenir des valeurs. Les valeurs sont distinctes.
- Tableau (*array*) : une séquence de cellules pouvant contenir des valeurs. Les valeurs ne sont pas nécessairement distinctes.

Ces catégories peuvent être classées selon deux dimensions : l'unicité et la structure (tableau 3.1).

	Non structuré	**Séquence**	**Tableau**
Unique	ensemble (set)	liste unique (u-list)	tableau unique (u-array)
Pas unique	amas (bag)	liste (list)	tableau (array)

Tableau 3.1 - Les six catégories de collections de valeurs classées selon l'unicité et la structure.

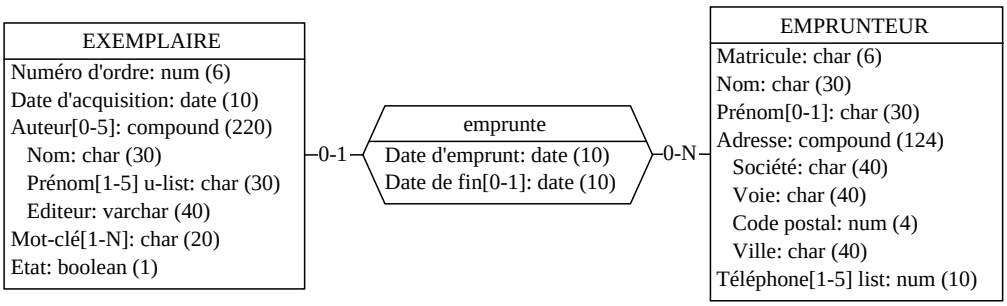


Figure 3.5 - Dans *EMPRUNTEUR*, *Nom* est obligatoire, *Prénom* est facultatif, *Adresse* est décomposable, *Matricule* est atomique, *Téléphone* est multivalué de type liste. Dans *EXEMPLAIRE*, *Prénom* est multivalué de type liste unique et *Mot-clé* de type ensemble.

3.1.2.6 Groupe

Un groupe est une collection d'attributs, de rôles et/ou d'autres groupes. Il représente une construction attachée à un parent (un type d'entités, un type d'associations ou un attribut décomposable multivalué) qui est utilisée pour modéliser des contraintes telles que des identifiants, des clés étrangères, des clés d'accès ou des contraintes particulières.

Un groupe d'un type d'entités peut contenir ses attributs (quel que soit leur niveau), ses rôles éloignés (rôles du type d'associations dans lequel le type d'entités joue un rôle, ce dernier étant exclu si le type d'associations n'est pas récursif), ses autres groupes et/ou des attributs et des rôles hérités de ses surtypes (direct ou indirect). Un groupe d'un type d'associations peut contenir ses attributs (quel que soit leur niveau), ses rôles et/ou ses autres groupes. Un groupe d'un attribut décomposable multivalué peut contenir ses sous-attributs (quel que soit leur niveau).

Un groupe peut avoir une contrainte de cardinalité [i-j] qui indique le nombre d'instances du parent du groupe qui peuvent partager les mêmes valeurs pour ce groupe. Ce concept est essentiellement utile pour les contraintes référentielles (point 3.1.2.7) car il permet de préserver

la cardinalité du rôle transformé dans la transformation d'un type d'associations en attributs de référence (3.2.3.2 b).

On peut assigner une ou plusieurs fonctions à un groupe. Elles sont décrites ci-dessous.

a) Identifiant primaire

Les composants du groupe forment le principal identifiant du parent. Cette fonction signifie que chaque instance du parent est identifiée par les valeurs des composants du groupe. Elle apparaît avec le symbole *id* (figure 3.6). Un attribut multivalué décomposable A peut être identifié par un groupe composé de ses attributs (quel que soit le niveau). Cet identifiant d'attribut signifie que, pour chaque instance du parent de A, deux instances de A ne peuvent partager les mêmes valeurs pour l'identifiant.

Un identifiant primaire d'un type d'entités E est composé de :

- soit un ou plusieurs attributs monovalués obligatoires de E (ou des surtypes de E);
- soit un seul attribut multivalué obligatoire de E (ou des surtypes de E), on parle d'*identifiant multivalué* (deux instances du parent ne peuvent pas partager les mêmes valeurs pour cet attribut);
- soit deux ou plusieurs rôles éloignés de E (ou des surtypes de E);
- soit un ou plusieurs rôles éloignés de E (ou des surtypes de E) + un ou plusieurs attributs monovalués obligatoires de E (ou des surtypes de E).

Un identifiant primaire d'un type d'associations R est composé de :

- soit un ou plusieurs attributs monovalués obligatoires de R,
- soit deux ou plusieurs rôles de R,
- soit un ou plusieurs rôles de R + un ou plusieurs attributs monovalués obligatoires de R.

Un identifiant primaire d'un type d'attribut décomposable multivalué A est composé de :

- un ou plusieurs attributs monovalués obligatoires de A.

Si le parent d'un groupe est un type d'entités ou un type d'associations et si tous les composants sont des attributs, alors les composants sont soulignés dans les graphiques. Un parent peut avoir au maximum un identifiant primaire.

b) Identifiant secondaire

Les composants du groupe forment un identifiant secondaire du parent qui apparaît graphiquement comme *id'*. Un parent peut avoir un nombre quelconque d'identifiants secondaires. Les règles sont identiques à celles de l'identifiant primaire excepté que les composants du groupe peuvent être facultatifs (figure 3.6).

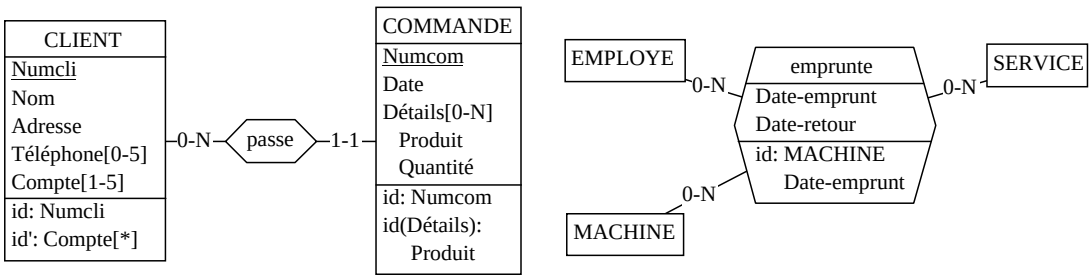


Figure 3.6 - *Numcli* est un identifiant primaire de *CLIENT*, *Compte* un identifiant secondaire multivalué. *Produit* est un identifiant de l'attribut décomposable multivalué *Détails*. Le type d'associations *emprunte* est identifié par une machine et la date de son emprunt.

c) Contrainte de coexistence

Les composants du groupe doivent être simultanément présents ou absents pour les instances du parent. La coexistence est représentée graphiquement par le symbole *coex* (figure 3.7). Le parent du groupe est soit un type d'entités, soit un type d'associations et les composants sont tous facultatifs.

d) Contrainte d'exclusivité

Parmi les composants du groupe, au maximum un doit être présent pour chaque instance du parent. Le groupe apparaît graphiquement avec le symbole *excl*. Le parent du groupe est soit un type d'entités, soit un type d'associations et les composants sont tous facultatifs.

e) Contrainte au-moins-un

Parmi les composants du groupe, au moins un doit être présent pour chaque instance du parent. Le groupe apparaît graphiquement avec le symbole *at-lst-1* (figure 3.7). Le parent du groupe est soit un type d'entités, soit un type d'associations et les composants sont tous facultatifs.

f) Contrainte exactement-un

Parmi les composants du groupe, un et un seul doit être présent pour chaque instance du parent (exclusivité et au-moins-un). Le groupe apparaît graphiquement avec le symbole *exact-1*. Le parent du groupe est soit un type d'entités, soit un type d'associations et les composants sont tous facultatifs.

g) Clé d'accès

Les composants du groupe forment un mécanisme d'accès aux instances du parent (généralement à un type d'entités interprété comme une table, type de record ou un segment à un niveau logique ou physique). La clé d'accès est une abstraction de constructions comme des index, chemins d'accès ou des B-Tree. Elle est représentée graphiquement par *acc* (figure 3.7). Le parent du groupe est soit un type d'entités, soit un type d'associations.

h) Contrainte définie par l'utilisateur

N'importe quelle contrainte n'apparaissant pas précédemment peut être définie en langage naturel par un utilisateur qui lui donne un nom significatif. Par exemple, une contrainte *au-plus-2* signifie pas plus de deux composants avec une valeur par instance du parent ou une contrainte *moins-que* vérifie que la valeur du premier composant est inférieure à la valeur du second pour toutes les instances du parent. Le parent du groupe est soit un type d'entités, soit un type d'associations.

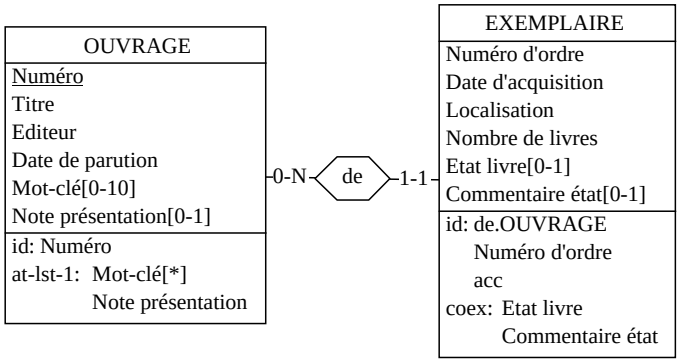


Figure 3.7 - L'identifiant primaire de *EXEMPLAIRE* est en plus une clé d'accès. *Etat livre* et *Commentaire état* sont simultanément présents ou absents. Un des attributs *Mot-clé* et *Note présentation* doit avoir une valeur pour chaque instance de *OUVRAGE*.

3.1.2.7 Contrainte inter-groupes

Indépendamment de leurs fonctions, deux groupes avec des composants compatibles (même nombre, type, longueur) peuvent être reliés par une relation exprimant une contrainte d'intégrité entre groupes.

Les contraintes suivantes sont autorisées :

- contrainte de référence (ou référentielle) : le premier groupe est une clé étrangère et le second est l'identifiant (primaire ou secondaire) référencé. La clé étrangère apparaît avec le symbole *ref* (figure 3.8).
- contrainte d'égalité : le premier groupe est une clé étrangère et le second est l'identifiant (primaire ou secondaire) référencé. En plus, une contrainte d'inclusion est définie du second groupe vers le premier. La clé étrangère apparaît avec le symbole *equ* (figure 3.8).

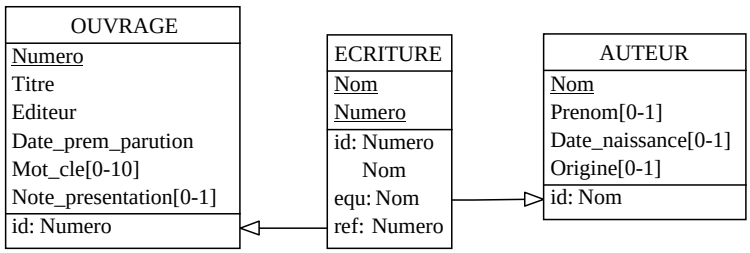


Figure 3.8 - *ECRITURE.Numero* forme un groupe de référence de l'identifiant d'*OUVRAGE*. *ECRITURE.Nom* référence l'identifiant d'*AUTEUR*. La contrainte d'égalité impose qu'un auteur est référencé par une instance d'*ECRITURE* au minimum.

3.1.3 Définition des sous-modèles

Le modèle Entité/Objet-Association générique permet de définir, par spécialisation, les différents modèles propres à notre démarche méthodologique de conception. Chacun de ces modèles est défini comme un sous-modèle du modèle générique. Un sous-modèle M est obtenu par spécialisation [3] du modèle générique, c'est-à-dire par une restriction du modèle générique selon laquelle

3. Le terme de *spécialisation* peut apparaître comme impropre au sens traditionnel qui lui est généralement assigné. Etant donné un modèle G et un modèle M, déclaré spécialisation de G, tout schéma conforme à M est également conforme à G.

on sélectionne les objets pertinents et on précise les lois définissant les assemblages licites dans M, ainsi que par un renommage des objets selon la nomenclature propre à M, éventuellement accompagné de conventions graphiques appropriées pour la représentation des objets. L'intérêt de procéder par spécialisation est évident :

1. tout raisonnement, toute technique et tout outil développés pour le modèle générique sont d'application pour tout sous-modèle M,
2. le passage d'une spécification exprimée dans un modèle M vers une spécification équivalente dans le modèle N est un processus interne au seul modèle générique; ainsi, si on dispose de n modèles distincts, la conversion d'un modèle vers un autre nécessite un seul jeu de règles, plutôt que n(n-1) jeux différents si ces modèles étaient définis indépendamment les uns des autres.

Les trois points suivants présentent des spécialisations du modèle générique en trois sous-modèles classiques (correspondant chacun à un niveau d'abstraction) qui apparaissent lors de l'élaboration d'une base de données relationnelle.

3.1.3.1 Modèle Entité / Association du niveau conceptuel

Les schémas du niveau conceptuel sont exprimés dans un sous-modèle qui est une variante du modèle Entité/Association. Le modèle Entité/Association permet de décrire les concepts d'un domaine d'application comme des ensembles d'entités, dotées de propriétés et en relation les unes avec les autres par le biais d'associations. Nous avons choisi ce modèle car il est un des plus populaire à l'heure actuelle. On aurait également pu choisir le modèle UML [4] proposant un ensemble de notations pour représenter divers aspects d'une application. Il est divisé en sept sous-ensembles : les modèles des classes, des états, des cas d'utilisation, d'interaction, d'activité, de composants et de déploiement [Mul97a]. Actuellement, le *modèle des classes*, proche du modèle Entité/Association, est proposé comme alternative aux approches plus classiques pour la description des classes d'objets de la base de données. Malheureusement, ce modèle est peu adapté à l'expression de schémas conceptuels. Par exemple, il ne propose aucune contrainte d'intégrité telles que les identifiants, les contraintes inter-groupes (les contraintes s'expriment sous la forme de texte libre annotant un schéma, excepté l'exclusion d'associations) et il ne fournit aucune information sur la nature et la structure des attributs (le type de l'attribut n'est pas spécifié). Le manque de formalité d'UML altère la mise en œuvre de processus sur des spécifications, qu'il s'agisse d'opérations simples (transformer un objet, ...) ou plus complexes (rejouer un ensemble de transformations, ...). Par exemple, chaque concepteur aurait à définir ses propres contraintes ce qui entraînerait une prolifération des processeurs travaillant sur les contraintes des objets. Si on recherche une certaine automatisation de l'activité d'évolution d'une base de données, il faut choisir un modèle plus formel. Dans ces conditions, le modèle Entité/Association est mieux adapté que le modèle des classes d'UML.

Le tableau 3.2 présente les concepts du modèle générique appartenant au modèle spécifié ainsi que la terminologie utilisée.

Modèle générique	Modèle conceptuel (E/A)	Terminologie propre
type d'objets	oui	type d'entités
relation IS-A	oui	sous-type / surtype
type d'associations	oui	type d'associations
attribut monovalué/atomique	oui	attribut monovalué/atomique
attribut multivalué	oui	attribut multivalué
attribut décomposable	oui	attribut décomposable
attribut obligatoire	oui	attribut obligatoire

Tableau 3.2 - Définition d'un modèle conceptuel par spécialisation du modèle générique.

4. pour "Unified Modeling Language" ([Uml99], [Har98]).

Modèle générique	Modèle conceptuel (E/A)	Terminologie propre
attribut facultatif	oui	attribut facultatif
identifiant primaire	oui	identifiant primaire
identifiant secondaire	oui	identifiant secondaire
attributs de référence	-	-
clé d'accès	-	-
collection d'objets	-	-

Tableau 3.2 - Définition d'un modèle conceptuel par spécialisation du modèle générique.

La figure 3.9 présente un exemple de schéma conceptuel exprimé dans le sous-modèle Entité/ Association. *PERSONNE, CLIENT*, *FOURNISSEUR, COMMANDE* et *ARTICLE* sont des types d'entités. *CLIENT* et *FOURNISSEUR* sont des sous-types de *PERSONNE* (qui est leur surtype). Des contraintes de totalité et de disjonction (P pour partition) ont été définies sur ces sous-types. Les attributs *NumPers, Nom*, *Adresse* et *Téléphone* caractérisent le type d'entités *PERSONNE* (et donc aussi CLIENT et FOURNISSEUR). *Adresse* est un attribut décomposable et *Téléphone* un attribut multivalué (de type *set*). Les attributs *Numéro*, *Rue* et *Localité* sont les composants de l'attribut *Adresse*. *Adresse.Numéro* est facultatif (de cardinalité [0-1]). *passe, référence* et *offre* sont des types d'associations binaires (deux rôles). *référence* possède un attribut. *COMMANDE* joue deux rôles, l'un dans *passe*, l'autre dans *référence*. Chaque rôle est caractérisé par ses cardinalités minimum et maximum. *CLIENT* est identifié par l'attribut *NumCli*.

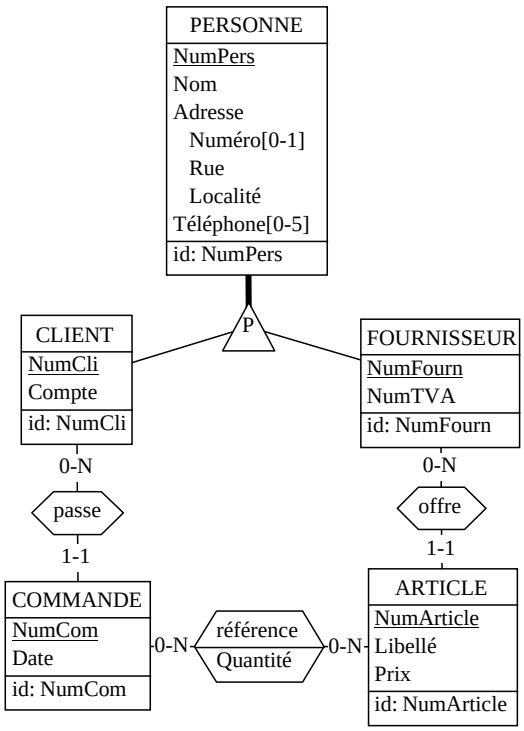


Figure 3.9 - Vue graphique d'un schéma conceptuel typique.

3.1.3.2 Modèle relationnel du niveau logique

Les schémas du niveau logique sont exprimés dans un sous-modèle qui est un modèle logique relationnel. La démarche se base sur le modèle relationnel pour dégager une typologie des modifications (chapitre 4) et permettre ainsi de concevoir des stratégies d'évolution et des outils de conversion opérationnels (chapitre 5).

Le choix du modèle relationnel s'impose essentiellement par sa popularité. Ce choix n'enlève rien à la démarche qui pourrait être mise en œuvre avec d'autres technologies moyennant une typologie adaptée des modifications et des outils de conversion spécifiques pour le niveau opérationnel.

Le modèle relationnel choisi n'intègre pas les concepts orientés objets apparus récemment dans le modèle *relationnel objet*. La norme SQL:1999 [Eis99] définit la notion de types structurés dont les principales caractéristiques sont :

- la définition de colonnes pouvant être des collections de valeurs (type "array") ou des types structurés;
- la description de comportements dans des méthodes, des fonctions ou des procédures;
- la possibilité de participer à des hiérarchies en temps que sous-types ou surtypes.

Malgré les avantages indéniables du modèle relationnel objet, la majorité des bases de données sont implémentées dans le modèle relationnel traditionnel ce qui explique la raison de notre choix. Comme cela a déjà été précisé précédemment, la démarche proposée pourrait très bien être adaptée au modèle relationnel objet.

Le tableau 3.3 présente les concepts du modèle générique appartenant au modèle spécifié ainsi que la terminologie utilisée. Bien que relevant du niveau physique, les clés d'accès et espaces de stockage ont été ajoutés à titre d'information.

Modèle générique	Modèle logique (relationnel)	Terminologie propre
type d'objets	oui	table
relation IS-A	-	-
type d'associations	-	-
attribut monovalué/atomique	oui	colonne
attribut multivalué	-	-
attribut composé	-	-
attribut obligatoire	oui	colonne *not null*
attribut facultatif	oui	colonne *null*
identifiant primaire	oui	clé primaire
identifiant secondaire	oui	clé candidate
attribut de référence	oui	clé étrangère
clé d'accès	(oui : niveau physique)	(index)
collection d'objets	(oui : niveau physique)	(espace de stockage (dbspace))

Tableau 3.3 - Définition du modèle logique relationnel par spécialisation du modèle générique.

La figure 3.10 présente un exemple de schéma logique exprimé dans le sous-modèle logique relationnel ainsi que son interprétation. Ce schéma logique est la transformation relationnelle du schéma conceptuel de la figure 3.9. *PERSONNE*, *CLIENT*, *FOURNISSEUR*, *TELEPHONE*, *COMMANDE, ...* sont des tables. *NumPers, NumCli, Nom* sont des colonnes de *PERSONNE*. *Nom* est obligatoire et *Adr_Numéro* est facultatif. *PERSONNE* possède une clé primaire *NumPers* et deux clés secondaires *NumCli* et *NumFourn*. *COMMANDE.NumCli* est une *clé étrangère* (*ref*) vers *CLIENT* (via sa clé primaire). *PERSONNE.NumCli* est une clé primaire et une clé étrangère vers *CLIENT*. En outre, toute valeur de *CLIENT.NumCli* est aussi une valeur non nulle de *PERSONNE.NumCli*. Ces deux contraintes d'inclusion inverses forment une *contrainte d'égalité* (*equ*). *PERSONNE* est soumise à une contrainte d'existence *exactement-un* indiquant que, pour toute ligne de cette table, une et une seule des colonnes *NumFourn* et *NumCli* possède une valeur non nulle. Les noms des tables et des attributs sont rendus conformes à la syntaxe SQL (les accents et les espaces sont enlevés par exemple).

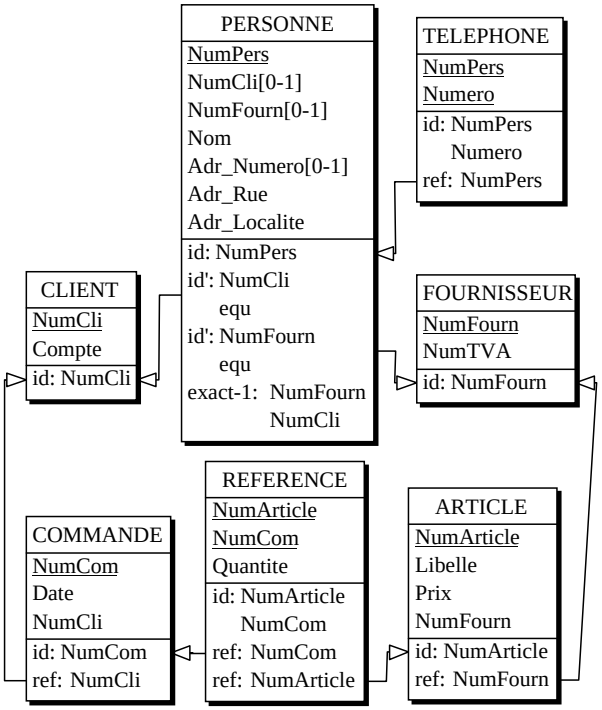


Figure 3.10 - Vue graphique d'un schéma logique relationnel.

3.1.3.3 Modèle relationnel du niveau physique

Les schémas du niveau physique sont exprimés dans un sous-modèle qui est un modèle physique relationnel. Le tableau présentant les concepts du modèle générique appartenant au modèle spécifié ainsi que la terminologie utilisée correspond à celui du sous-modèle logique (tableau 3.3) avec les clés d'accès et les collections en plus.

La figure 3.10 présente un exemple de schéma physique exprimé dans le sous-modèle physique relationnel ainsi que son interprétation. Ce schéma est le même que le schéma logique de la figure 3.11 avec les concepts physiques d'index et d'espace de stockage en plus. Les noms des tables et des attributs sont rendus conformes à la syntaxe SQL du SGBD (par exemple, ils perdent leurs accents). Le nom *Date* (mot réservé dans certains systèmes) est remplacé par *DATECOM*. Des index (acc) sont définis sur les colonnes *NUMPERS* de *PERSONNE*, *NUMCLI* de *PERSONNE*, *NUMFOURN* de *ARTICLE*, *{NUMARTICLE*, *NUMCOM}* de *REFERENCE*, etc. La collection *SPC_FOU* définit un espace de stockage destiné à contenir les lignes des tables *ARTICLE* et *FOURNISSEUR*.

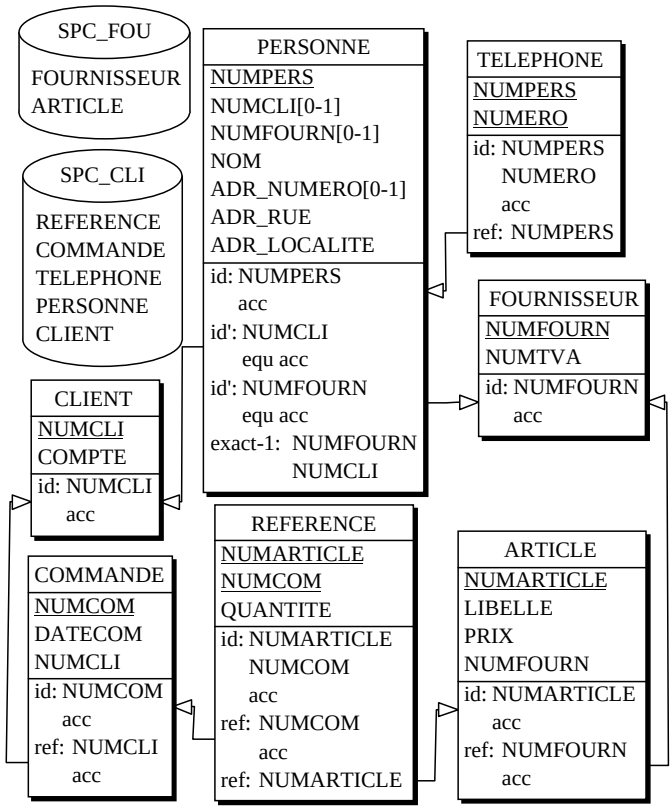


Figure 3.11 - Vue graphique d'un schéma physique relationnel.

3.2 Description formelle des opérations : approche transformationnelle

Depuis longtemps dans l'ingénierie du logiciel, la conception d'une application peut être modélisée comme la transformation systématique de spécifications formelles en programmes efficaces et opérants [Bal81] [Fic85].

De même dans le domaine des bases de données, tout processus d'ingénierie (conception, intégration, normalisation, ...) peut être défini comme une suite de modifications de structures de données [Bat92]. Une transformation est généralement considérée comme un opérateur qui remplace une structure de données par une autre ayant certaines relations sémantiques avec la structure initiale. Des transformations ont été proposées dans beaucoup d'activités liées à l'ingénierie de bases de données : la normalisation, la traduction logique, l'intégration, la rétro-ingénierie, l'optimisation de schémas, ... [Nav80] [Kob86] [Koz87] [Ros87]

Par exemple, la conception logique sera modélisée comme une chaîne de transformations du schéma conceptuel tel que le résultat final soit conforme au modèle relationnel et satisfasse à des critères de performance. De même, la construction d'un schéma conceptuel, la normalisation d'un schéma conceptuel, l'optimisation d'un schéma logique, la génération de code SQL ou la rétro-ingénierie de fichiers COBOL peuvent être perçues comme des séquences de transformations. L'ajout d'un type d'entités, la modification du nom d'un attribut ou le changement d'un type d'associations en clés étrangères sont des transformations élémentaires qu'on peut combiner de manière à former des processus plus complexes. Cette section présente brièvement la notion de transformation.

La figure 3.12 propose un exemple d'une transformation souvent utilisée en conception logique et qui remplace un type d'associations binaire plusieurs-à-plusieurs (*détail*) par un type d'entités (*DETAIL*) et deux types d'associations un-à-plusieurs (*dc* et *dp*).

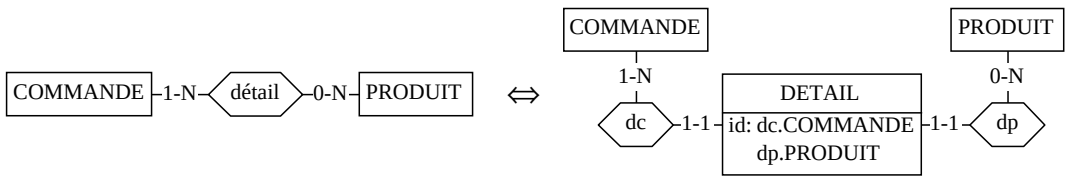


Figure 3.12 - Un exemple de transformation d'un type d'associations en un type d'entités.

3.2.1 Définition d'une transformation

Une transformation est un opérateur T qui substitue à un ensemble (éventuellement vide) d'objets C d'un schéma un autre ensemble d'objets (éventuellement vide) C'. Cet opérateur est défini par les préconditions minimales que C doit satisfaire avant l'opération et les postconditions maximales que C' vérifie après transformation. Si T est une transformation, C et C' des structures de données, dont l'une peut être vide, on peut écrire : C' = T(C). Spécifier une transformation requiert non seulement la description des relations inter-structures mais aussi inter-instances. La définition d'une transformation doit aussi inclure la fonction t de transformation des instances de C en instances de C' (figure 3.13).

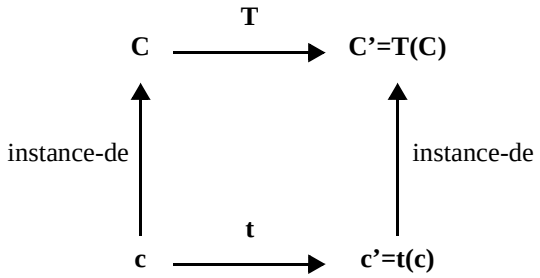


Figure 3.13 - Schéma d'une transformation.

T est le mapping structurel et représente la syntaxe de la transformation. Tandis que t est le mapping d'instances qui évoque la sémantique de la transformation. Une transformation Σ est complètement définie par ces deux mappings : Σ = <T,t>.

Une façon de définir le mapping T consiste en une paire de prédicats (approche prédicative) <P,Q> où P représente les préconditions minimales (établissant le type de structure C pouvant être un candidat à la transformation) et Q les postconditions maximales (établissant la forme générale de la structure C'). On en tire une nouvelle notation : Σ = <P,Q,t>. P et Q sont des prédicats du second ordre qui affirment l'existence et les propriétés des constructions C et C' et qui identifient chacun de leurs composants. Pour décrire P et Q, nous utiliserons une notation purement prédicative dont la syntaxe est présentée dans l'annexe A.1. Le mapping des instances peut être exprimé à travers un langage de requête (détaillé à l'annexe A.1), tel une extension de l'algèbre relationnel.

3.2.2 Sémantique d'une transformation

Lorsque les deux constructions (C et C') sont équivalentes, c'est-à-dire qu'elles expriment la même sémantique ou décrivent le même univers du discours, on dit que la transformation T préserve la sémantique. Deux autres situations extrêmes sont envisageables. Si C est vide, la transformation consiste à ajouter C' au schéma. Elle augmente la sémantique du schéma. Dans le cas contraire, lorsque que C' est vide, la transformation détruit C et diminue la sémantique du schéma. Il existe donc trois catégories de transformations. La catégorie T^+ augmente la sémantique (par exemple ajouter un type d'entités), T^- diminue la sémantique (par exemple supprimer un attribut) et $T^=$ préserve la sémantique du schéma (par exemple remplacer un type d'associations par un type d'entités).

Les transformations de $T^=$ sont appelées *réversibles*. Une transformation <T,t> est réversible s'il existe une transformation <T',t'> telle que : T'(T(C)) = C et t'(t(c)) = c pour toute instance c de C. Si <T',t'> est également réversible, on dit que ces transformations sont *symétriquement réversibles*.

Bien que souvent citée, la notion de *sémantique d'un schéma* n'a pas reçu de définition généralement acceptée. Une première approche consiste à comparer les univers du discours (les objets du monde réel et leurs relations) décrits par deux schémas. C'est un concept intéressant mais difficile à mettre en œuvre à cause de la difficulté de construire des preuves. Une deuxième approche consiste à examiner les instances peuplant les schémas (comparaison en termes d'informations stockées). Dans ce cadre, la sémantique du schéma S1 inclut celle du schéma S2 si le domaine d'application représenté par S2 est une partie de celui que représente S1. Par conséquent, ajouter un type d'objets à un schéma en augmente la sémantique tandis que l'ajout d'une contrainte la diminue. C'est cette deuxième approche qui a été choisie pour ce travail. En l'absence de consensus sur le sujet et parce que le concept n'est pas critique pour la suite, nous nous en tiendrons à cette définition intuitive.

Les transformations $T^{=}$ sont principalement utilisées dans la production de schémas logiques et physiques, tandis que les transformations T^{+} et T^{-} forment la base du processus d'évolution des spécifications. Parmi la masse des transformations existantes, citons par exemple, la désagrégation d'un attribut composé, l'agrégation d'attributs, la transformation d'attributs multivalués en listes d'attributs monovalués, la fusion et l'éclatement de types d'entités, etc.

Une transformation T remplace une structure C par une autre structure C' dans un schéma S. On obtient ainsi un nouveau schéma S'. L'effet de T peut être analysé en termes de structures en considérant les fonctions C^{-}, C^{+}, $C^{=}$, C et C' définies comme suit :

$C^{-}(T) = S-S'$	retourne les objets de S disparaissant dans S';
$C^{+}(T) = S'-S$	retourne les nouveaux objets apparaissant dans S';
$C^{=}(T)$	retourne les objets de S concernés par T et préservés dans S';
$C(T) = C^{=}(T) \cup C^{-}(T)$	retourne les objets de S concernés par T;
$C'(T) = C^{=}(T) \cup C^{+}(T)$	retourne les objets de S' concernés par T.

3.2.3 Présentation des transformations

Il existe un très grand nombre de transformations de schéma. Construire un tel catalogue serait fastidieux. Dans cette section, nous allons décrire les principales transformations utilisées dans les processus d'ingénierie de bases de données et, plus spécialement, les transformations de catégorie $T^{=}$ qui forment la base du processus de production de schémas logiques et physiques à partir de schémas conceptuels ainsi que le processus inverse (rétro-ingénierie). Les transformations à la base du processus d'évolution sont analysées dans la typologie du chapitre 4.

Seule la transformation d'un type d'entités en un type d'associations est présentée complètement avec ses préconditions (P), ses postconditions (Q) et la transformation des instances (I). Pour les autres transformations, seuls une brève description (D) avec un exemple, les signatures (T et T^{-1}) et quelques domaines d'application (A) sont proposés. Toutes ces transformations sont approfondies de manière plus formelle dans l'annexe B.

Deux précisions sont nécessaires avant de décrire les transformations :

1. Les préconditions et les postconditions définies sur les transformations dépendent du modèle décrit dans la section 3.1. Les spécifications doivent satisfaire les contraintes du modèle avant et après chaque transformation.
2. Les transformations sont sans intelligence. C'est le concepteur qui choisit la meilleure transformation en fonction de ses besoins. Si une transformation n'est pas réalisée car les préconditions ne sont pas vérifiées, le concepteur peut appliquer une ou plusieurs autres transformations pour atteindre son objectif ou modifier les spécifications pour satisfaire les préconditions.

3.2.3.1 Transformations de types d'entités

a) Transformation d'un type d'entités en un type d'associations

D Un type d'entités *EMPRUNT* peut être transformé en un type d'associations *emprunt* à condition qu'il ait un identifiant et qu'il soit lié à au moins deux autres types d'entités (*EXEMPLAIRE* et *EMPRUNTEUR*) par des types d'associations binaires un-à-plusieurs (figure 3.14).

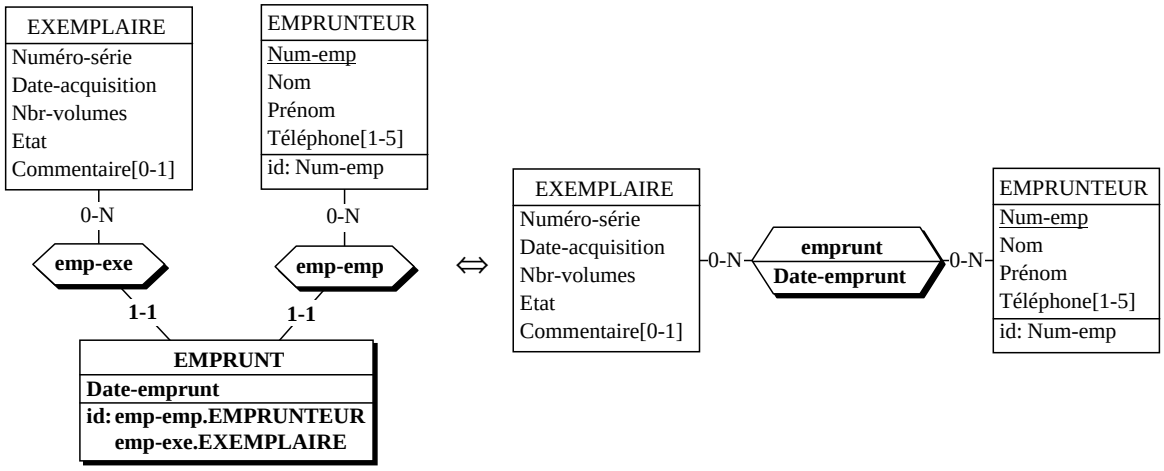


Figure 3.14 - Transformation du type d'entités *EMPRUNT* en un type d'associations.

S emprunt ← TE-en-TA(EMPRUNT)

P
- *EMPRUNT* est un type d'entités sans sous-type ni surtype.
- Tous les rôles joués par *EMPRUNT* sont mono-TE et de cardinalité [1-1].
- Les types d'associations *emp-exe* et *emp-emp* sont binaires, sans attribut et leurs rôles sont mono-TE.
- EMPRUNT a un identifiant (composé de *emp-emp.EMPRUNTEUR* et *emp-exe.EXEMPLAIRE*).
- *EMPRUNT* ne contient pas de groupe avec une contrainte référentielle ou d'identifiant référencé par une contrainte référentielle.

Q
- *emprunt* est un type d'associations.
- Les groupes et les attributs de *EMPRUNT* sont transférés dans *emprunt*.
- Les rôles joués par *EXEMPLAIRE* dans *emp-exe* et *EMPRUNTEUR* dans *emp-emp* sont transférés dans *emprunt*.
- L'identifiant de *emprunt* composé de l'ensemble de ces rôles est supprimé car il s'agit d'un identifiant implicite d'un type d'associations.
- Le type d'entités *EMPRUNT* est supprimé.
- Les types d'associations *emp-exe* et *emp-emp* sont supprimés.

I Pour chaque instance de *EMPRUNT*, on crée une instance de *emprunt* qui est reliée au mêmes instances de *EXEMPLAIRE* et *EMPRUNTEUR* et qui a la même valeur pour l'attribut *Date-emprunt*.

A Normalisation conceptuelle : transformation d'un type d'entités ressemblant à un type d'associations.

Intégration de schémas : équivalence de structures à intégrer.

T^{-1} (EMPRUNT,{emp-exe,emp-emp}) ← TA-en-TE(emprunt) : transformation d'un type d'associations en un type d'entités (point 3.2.3.2 a).

b) Transformation d'un type d'entités en un attribut

D Un type d'entités *TELEPHONE* est transformé en un attribut *Téléphone* de *EMPRUNTEUR* à condition qu'il joue un et un seul rôle (figure 3.15).

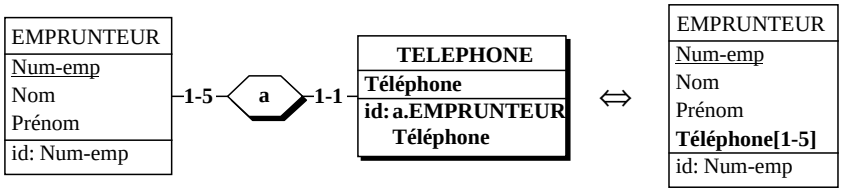


Figure 3.15 - Transformation du type d'entités *TELEPHONE* en un attribut *Telephone*.

S (EMPRUNTEUR,Téléphone) ← TE-en-Att(TELEPHONE,a)

A Normalisation conceptuelle : transformation d'un type d'entités ressemblant à un attribut.
Intégration de schémas : équivalence de structures à intégrer.

T^{-1} (TELEPHONE,a) ← Att-en-TE(EMPRUNTEUR,Téléphone) : transformation d'un attribut en un type d'entités (point 3.2.3.3 a).

c) Eclatement d'un type d'entités

D Certains composants (attributs et rôles) du type d'entités *EMPRUNTEUR* sont extraits et transférés dans un nouveau type d'entités *INFO-EMP* (figure 3.16).

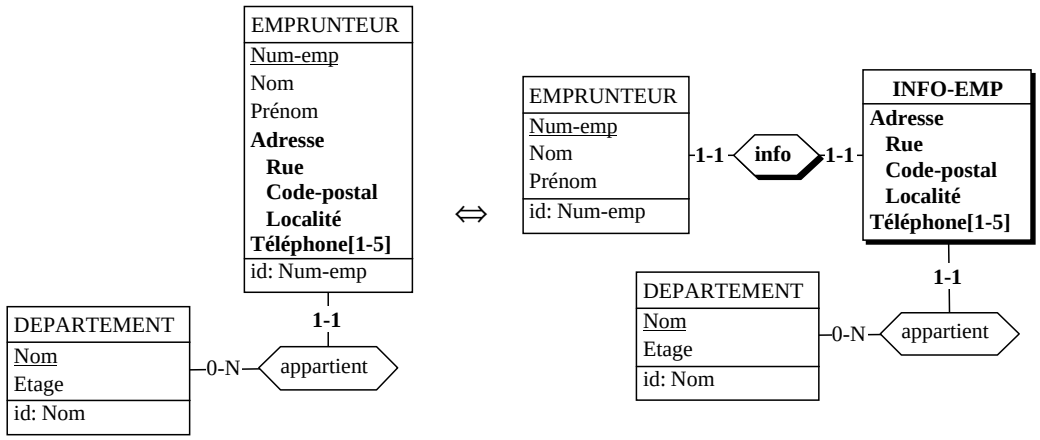


Figure 3.16 - Eclatement du type d'entités *EMPRUNTEUR*.

S (INFO-EMP,info) ← eclater-TE(EMPRUNTEUR, {Adresse, Téléphone, appartient.EMPRUNTEUR})

A Conception logique (optimisation) : amélioration des performances d'un schéma logique.
Intégration de schémas : équivalence de structures à intégrer.

T^{-1} EMPRUNTEUR ← fusionner-TE(INFO-EMP,info) : fusion de deux types d'entités (point d).

d) Fusion de deux types d'entités

D Les composants (attributs et rôles) du type d'entités *INFO-EMP* sont intégrés dans le type d'entités *EMPRUNTEUR* (voir la figure 3.16). Ces deux types d'entités sont reliés par un type d'associations un-à-un.

S EMPRUNTEUR ← fusionner-TE(INFO-EMP,info)

A Conception logique (optimisation) : amélioration des performances d'un schéma logique.
Intégration de schémas : équivalence de structures à intégrer.

T^{-1} (INFO-EMP,info) ← eclater-TE(EMPRUNTEUR,{Adresse,Téléphone,appartient.EMPRUNTEUR}) : éclatement d'un type d'entités (point c).

3.2.3.2 Transformations de types d'associations

a) Transformation d'un type d'associations en un type d'entités

D Un type d'associations *emprunt* est transformé en un type d'entités *EMPRUNT* et deux types d'associations (*emp-exe* et *emp-emp*) qui relient *EMPRUNT* aux types d'entités jouant un rôle dans *emprunt* (voir la figure 3.14).

S (EMPRUNT,{emp-exe,emp-emp}) ← TA-en-TE(emprunt)

A Normalisation conceptuelle : promotion d'un type d'associations au rang de type d'entités.

Conception logique : transformation d'un type d'associations complexe en un type d'entités.

Intégration de schémas : équivalence de structures à intégrer.

T^{-1} emprunt ← TE-en-TA(EMPRUNT) : transformation d'un type d'entités en un type d'associations (point 3.2.3.1 a).

b) Transformation d'un type d'associations en une clé étrangère

D Un type d'associations binaire fonctionnel *de* est transformé en une clé étrangère dans le type d'entités *EXEMPLAIRE* (figure 3.17).

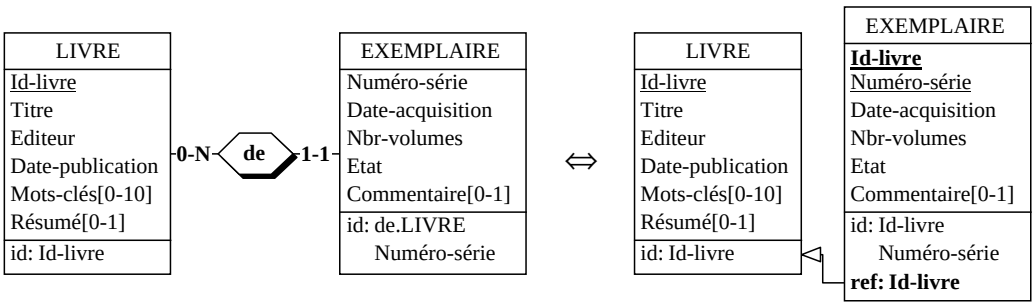


Figure 3.17 - Transformation du type d'associations *de* en une clé étrangère.

S (EXEMPLAIRE,{Id-livre}) ← TA-en-FK(de)

A Conception logique relationnelle ou COBOL (fichiers standards) : transformation principale pour produire des schémas conformes au modèle relationnel ou COBOL.

T^{-1} de ← FK-en-TA(EXEMPLAIRE,{Id-livre}) : transformation d'une clé étrangère en un type d'associations (point 3.2.3.3 b).

3.2.3.3 Transformations d'attributs

a) Transformation d'un attribut en un type d'entités

D Un attribut *Mot-clé* est transformé en un type d'entités *MOT-CLE* indépendant (figure 3.18). Il y a deux variantes selon que chaque instance de *MOT-CLE* représente une valeur distincte de l'attribut (représentation par valeur) ou une instance de l'attribut (représentation par instance).

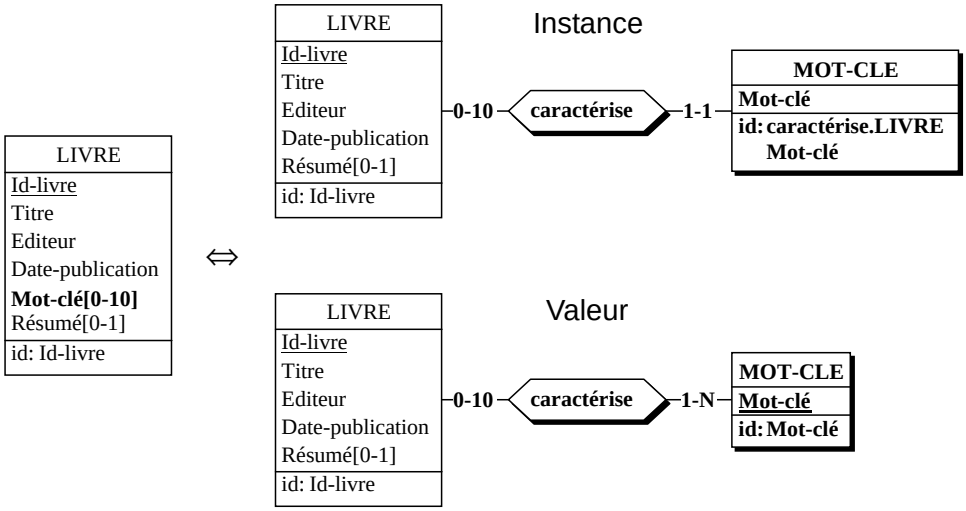


Figure 3.18 - Transformation de l'attribut multivalué *Mot-clé* en un type d'entités.

S (MOT-CLE,caractérise) ← Att-en-TE-inst(LIVRE,Mot-clé)

(MOT-CLE,caractérise) ← Att-en-TE-val(LIVRE,Mot-clé)

A Conception logique : élimination d'attributs multivalués ou décomposables.

Normalisation conceptuelle : promotion d'un attribut au rang de type d'entités.

Rétro-ingénierie COBOL : élimination d'une clé étrangère contenant un attribut décomposable.

T^{-1} (LIVRE,Mot-clé) ← TE-en-Att(MOT-CLE,caractérise) : transformation d'un type d'entités en un attribut (point 3.2.3.1 b).

b) Transformation d'une clé étrangère en un type d'associations

D La clé étrangère (*Id-livre*) de *EXEMPLAIRE* est transformée en un type d'associations binaire fonctionnel *de* (voir la figure 3.17).

S de ← FK-en-TA(EXEMPLAIRE,{Id-livre})

A Rétro-ingénierie (conceptualisation) : détraduction d'une clé étrangère d'un schéma logique relationnel ou COBOL.

T^{-1} (EXEMPLAIRE,{Id-livre}) ← TA-en-FK(de) : transformation d'un type d'associations en une clé étrangère (point 3.2.3.2 b).

c) Concaténation d'un attribut multivalué

D Un attribut multivalué *Mot-clé* est remplacé par un attribut monovalué *Mots-clés* dont chaque valeur est la concaténation des valeurs de l'attribut *Mot-clé* d'origine (figure 3.19).

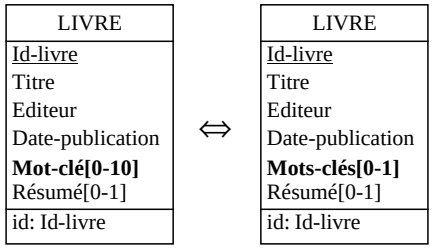


Figure 3.19 - Transformation par concaténation de l'attribut multivalué *Mot-clé*.

S (LIVRE,Mots-clés) ← Attmult-en-Attmono(LIVRE,Mot-clé)

A Conception logique : élimination d'attributs multivalués non conformes au modèle relationnel.

Intégration de schémas : équivalence de structures à intégrer.

T^{-1} (LIVRE,Mot-clé) ← Attmono-en-Attmult(LIVRE,Mots-clés) : transformation d'un attribut monovalué en un attribut multivalué (point d).

d) Transformation d'un attribut monovalué en un attribut multivalué

D Un attribut monovalué *Mots-clés* est remplacé par un attribut multivalué *Mot-clé* par déconcaténation de chaque valeur de *Mots-clés* dans les valeurs de l'attribut *Mot-clé* (voir la figure 3.19).

S (LIVRE,Mot-clé) ← Attmono-en-Attmult(LIVRE,Mots-clés)

A Rétro-ingénierie (conceptualisation, désoptimisation) : transformation d'un attribut monovalué long en un attribut multivalué.

T^{-1} (LIVRE,Mots-clés) ← Attmult-en-Attmono(LIVRE,Mot-clé) : concaténation d'un attribut multivalué (point c).

e) Transformation d'un attribut multivalué en une série d'attributs

D L'attribut multivalué *Téléphone* est remplacé par une série d'attributs monovalués : *Téléphone1*, *Téléphone2* et *Téléphone3* (figure 3.20). Cette transformation permet de remplacer des attributs multivalués par des attributs monovalués.

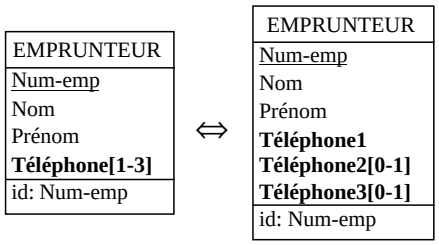


Figure 3.20 - Transformation de l'attribut multivalué *Téléphone* en une série d'attributs.

S (EMPRUNTEUR,{Téléphone1,Téléphone2,Téléphone3}) ← Attmult-en-Serie-Att(EMPRUNTEUR,Téléphone)

A Conception logique : élimination d'attributs multivalués non conformes au modèle relationnel.

T^{-1} (EMPRUNTEUR,Téléphone) ← Serie-Att-en-Attmult(EMPRUNTEUR, {Téléphone1, Téléphone2, Téléphone3}) : transformation d'une série d'attributs en un attribut multivalué (point f).

f) Transformation d'une série d'attributs en un attribut multivalué

D La série d'attributs (*Téléphone1*, *Téléphone2* et *Téléphone3*) est remplacée par un attribut multivalué *Téléphone* (voir la figure 3.20).

S (EMPRUNTEUR,Téléphone) ← Serie-Att-en-Attmult(EMPRUNTEUR, {Téléphone1, Téléphone2, Téléphone3})

A Rétro-ingénierie (conceptualisation, désoptimisation) : transformation d'une série d'attributs monovalués (avec les mêmes types, longueurs et des noms presque identiques) en un attribut multivalué.

T^{-1} (EMPRUNTEUR,{Téléphone1,Téléphone2,Téléphone3}) ← Attmult-en-Serie-Att(EMPRUNTEUR,Téléphone) : transformation d'un attribut multivalué en une série d'attributs (point e).

g) Désagrégation d'un attribut décomposable

D L'attribut décomposable *Localisation* est remplacé par ses composants (figure 3.21).

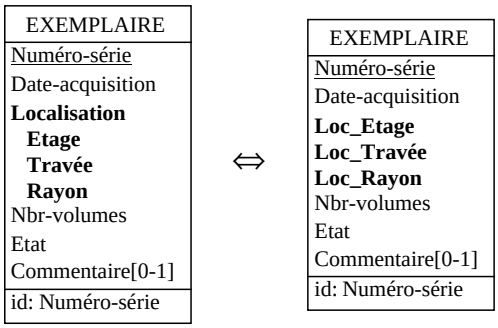


Figure 3.21 - Transformation par désagrégation de l'attribut décomposable *Localisation*.

S (EXEMPLAIRE,{Etage,Travée,Rayon}) ← desagreger-Att(EXEMPLAIRE,Localisation)

A Conception logique : élimination d'une structure non conforme au modèle relationnel.

Rétro-ingénierie COBOL (conceptualisation) : détraduction d'un attribut monovalué décomposable appartenant à une clé étrangère.

T^{-1} (EXEMPLAIRE,Localisation) ← agreger-Att(EXEMPLAIRE,{Etage,Travée,Rayon}) : agrégation d'une liste d'attributs en un attribut décomposable (point h).

h) Agrégation d'une liste d'attributs en un attribut décomposable

D La liste d'attributs (*Loc_Etage*, *Loc_Travée*, *Loc_Rayon*) est groupée dans un attribut décomposable *Localisation* (voir la figure 3.21). Les attributs doivent avoir le même parent.

S (EXEMPLAIRE,Localisation) ← agreger-Att(EXEMPLAIRE,{Etage,Travée,Rayon})

A Rétro-ingénierie (conceptualisation) : transformation d'une liste d'attributs (préfixes communs, types et longueurs différents) en un attribut décomposable.

T^{-1} (EXEMPLAIRE,{Etage,Travée,Rayon}) ← desagreger-Att(EXEMPLAIRE,Localisation) : désagrégation d'un attribut décomposable (point g).

3.2.3.4 Transformations des rôles multi-TE

a) Transformation d'un rôle multi-TE en types d'associations

D Le rôle *compte* joué par les types d'entités (*COMPTE-COURANT* et *COMPTE-EPARGNE*) est remplacé par deux types d'associations (*possède1* et *possède2*) reliés aux types d'entités jouant le rôle transformé (figure 3.22).

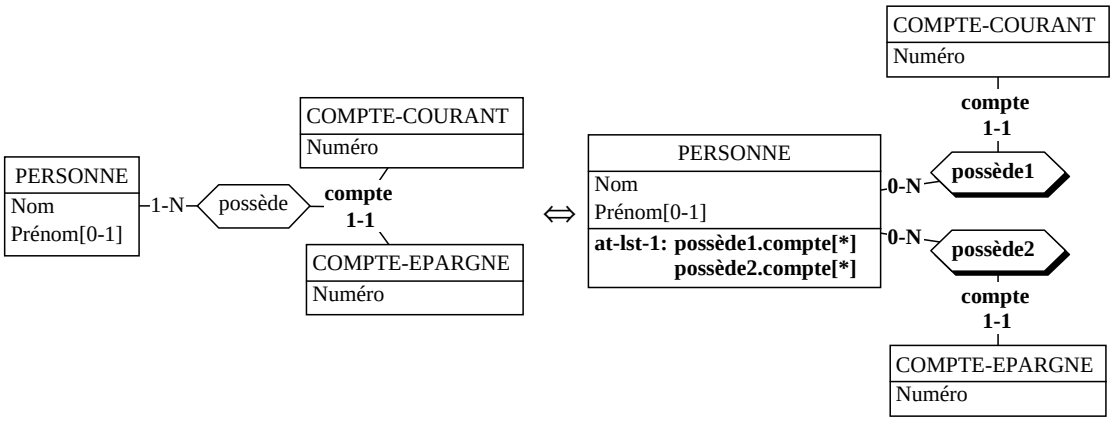


Figure 3.22 - Transformation du rôle multi-TE *compte* en deux types d'associations.

S (possède1,possède2) ← Rolemult-en-TA(possède,possède.compte)

A Conception logique : élimination d'une structure conceptuelle complexe.

T^{-1} (possède,possède.compte) ← TA-en-Rolemult(possède1,possède2) : transformation de type d'associations en un rôle multi-TE (point b).

b) Transformation de types d'associations en un rôle multi-TE

D Des types d'associations binaires liés à un type d'entités commun (*PERSONNE*) sont remplacés par un type d'associations binaire lié à *PERSONNE* et aux autres types d'entités (*COMPTE-COURANT* et *COMPTE-EPARGNE*) impliqués dans les types d'associations transformés par le biais d'un rôle multi-TE (voir la figure 3.22).

S (possède,possède.compte) ← TA-en-Rolemult(possède1,possède2)

A Normalisation conceptuelle : élimination d'une contrainte d'un type d'entités (au-moins-un par exemple) portant sur plusieurs rôles.

T^{-1} (possède1,possède2) ← Rolemult-en-TA(possède,possède.compte) : transformation d'un rôle multi-TE en types d'associations (point a).

3.2.3.5 Transformations de relations IS-A

a) Transformation de types d'associations en relations IS-A

D Le type d'entités *PERSONNE* commun à un ensemble de types d'associations un-à-un (*pa* et *pb*) est transformé en un surtype des types d'entités (*AUTEUR* et *EMPRUNTEUR*). Les types d'associations (*pa* et *pb*) sont transformés en relations IS-A (figure 3.23).

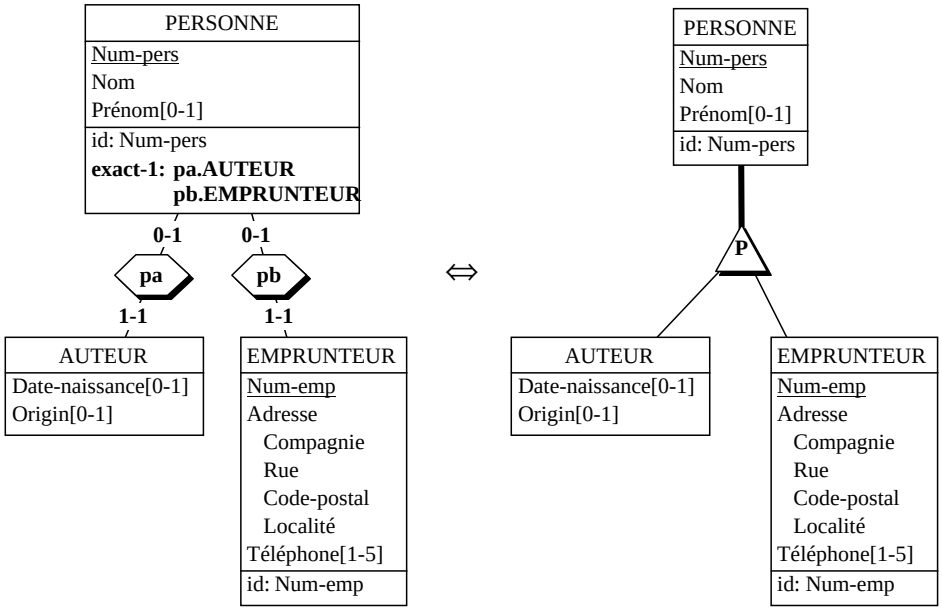


Figure 3.23 - Transformation des types d'associations *pa* et *pb* en relations IS-A.

S (PERSONNE,{AUTEUR,EMPRUNTEUR}) ← TA-en-ISA(PERSONNE,{pa,pb})

A Conception logique : élimination d'une structure conceptuelle non conforme aux modèles traditionnels (relationnel, hiérarchique, en réseau ou fichiers standards).

Intégrations de schémas : équivalence de structures à intégrer.

T^{-1} (PERSONNE,{pa,pb}) ← ISA-en-TA(PERSONNE,{AUTEUR,EMPRUNTEUR}) : transformation de relations IS-A en types d'associations (point b).

b) Transformation de relations IS-A en types d'associations

D Les relations IS-A entre le surtype *PERSONNE* et ses sous-types (*AUTEUR* et *EMPRUNTEUR*) sont transformées en types d'associations un-à-un (voir la figure 3.23).

S (PERSONNE,{pa,pb}) ← ISA-en-TA(PERSONNE,{AUTEUR,EMPRUNTEUR})

A Normalisation conceptuelle : remplacement de types d'associations simulant des relations de sous-typage par des relations IS-A.

Intégrations de schémas : équivalence de structures à intégrer.

T^{-1} (PERSONNE,{AUTEUR,EMPRUNTEUR}) ← TA-en-ISA(PERSONNE,{pa,pb}) : transformation de types d'associations en relations IS-A (point a).

3.3 Historique des modifications

3.3.1 Introduction

Le contrôle des processus de conception est reconnu comme un besoin primordial pour améliorer la qualité du développement et de la maintenance des systèmes d'information. Il consiste à analyser les activités de conception et à en déduire des informations importantes sur les produits, les processus et les acteurs impliqués. Concrètement, ce contrôle est basé sur la trace des activités, ce qu'on appelle également l'historique de la conception. On définit l'*historique* d'un processus complexe comme la trace de toutes les opérations qui ont été exécutées pour le réaliser [Hai96b]. Cette définition est valable pour l'exécution de programmes par exemple (cela permet de comprendre ce qu'un programme fait) mais elle est également applicable à des domaines où l'activité humaine est très présente comme l'ingénierie de systèmes d'information. La conception d'un système d'information est l'historique des processus par lesquels le système est construit ainsi que les produits qui en résultent. Cet historique devrait toujours être disponible car il forme la base de la maintenance et de l'évolution d'un système existant.

Cette section est divisée en trois parties. Le point 3.3.2 souligne le rôle primordial joué par les historiques dans le cadre de l'évolution. Pour modéliser un processus d'ingénierie, il faut définir un modèle des processus capable de décrire plus ou moins finement les méthodologies établies par les développeurs, c'est-à-dire leur façon de travailler. Le point 3.3.3 propose donc un modèle des processus de conception. La notion d'historique est approfondie dans le point 3.3.4.

3.3.2 Rôle des historiques dans le processus d'évolution

Une interprétation immédiate d'un historique est l'explication du comment le système a été développé d'une certaine façon, en d'autres mots, son aspect documentation. Un autre usage consiste à rejouer un historique. Par exemple, considérons qu'on dispose de l'historique de conception d'une base de données. Cet historique est la trace de toutes les activités de conception. Il comprend toutes les activités exécutées et tous les schémas construits lors de celles-ci.

Supposons qu'on modifie le schéma conceptuel en ajoutant un attribut à un type d'entités existant. Idéalement, il faudrait reconstruire les schémas logique et physique à partir du nouveau schéma conceptuel. La modification apportée étant mineure, il suffirait de rejouer l'historique pour produire de manière automatique les nouvelles versions des schémas logique et physique à condition que l'historique ait été mémorisé dans un atelier de génie logiciel. Pour des modifications plus profondes, il est clair que rejouer l'historique est insuffisant si la modification engendre de nouveaux choix de conception. Néanmoins, rejouer de manière partielle l'historique soulagerait fortement le travail du concepteur.

Comme nous l'avons vu au point 3.2, tout processus d'ingénierie est modélisable comme une séquence de transformations de spécifications. Dans le contexte de l'évolution défini au chapitre 2, où les modifications apportées à un niveau d'abstraction doivent être propagées vers les autres niveaux, garder une trace des transformations est une nécessité si le concepteur veut éviter de reformuler les séquences de transformations lors de chaque modification. Sur base de ces constatations, il apparaît clairement que le concept d'historique joue un rôle important dans l'évolution d'un système d'information [Big89] [Hai94].

3.3.3 Méthodologie

Un historique n'est pas une séquence arbitraire d'opérations. Il obéit à une organisation formalisée du travail appelée méthodologie. Une *méthodologie* spécifie les produits et les processus qui apparaissent lors d'une activité d'ingénierie. Décrire une méthodologie fait partie de la modélisation de processus, une discipline qui s'occupe de la compréhension, de la représentation et du

support logiciel des activités d'ingénierie incluant le développement des structures de données dans les applications dont le composant central est la base de données. De telles activités peuvent être modélisées comme un ensemble de produits (documents, programmes, schémas, types d'entités, ...) et un ensemble de processus d'ingénierie (primitifs ou non) qui transforment des produits pour satisfaire des besoins spécifiques.

Le modèle des processus de conception développé dans cette section dérive de propositions faites par [Pot88] et [Rol93] et étendues aux activités d'ingénierie de bases de données. Il décrit aussi bien des méthodologies de conception que des activités diverses comme l'intégration ou la normalisation de schémas. [Hai96b], [Rol97], [Rol99a] et [Rol00] apportent des précisions supplémentaires sur le modèle dont voici les concepts remarquables.

- Un *produit* est une spécification identifiée dans une activité spécifique. Un schéma conceptuel, un type d'entités, une table, un rapport d'évaluation sont des produits. Des produits similaires appartiennent à un *type de produits* tel que l'ensemble des schémas conceptuels, les types d'entités, ...
- Un *processus* est un ensemble d'opérations qui transforme un produit dans un autre produit. La transformation d'un schéma conceptuel en un schéma logique relationnel est un processus. Des processus similaires font partie d'un *type de processus*. Il y a deux catégories de processus. Un *processus d'ingénierie* veille à ce que ses produits résultats satisfassent des besoins spécifiques. La normalisation, la conception logique sont des exemples de processus d'ingénierie. Un *processus primitif* est une opération atomique déterministe [5]. La création d'un type d'entités, la transformation d'un type d'associations en type d'entités sont des processus primitifs.
- Une *stratégie* est la spécification du déroulement d'un processus. Elle est soit déterministe auquel cas elle est réduite à un algorithme souvent implémenté comme un processus primitif, soit non déterministe lorsque le concepteur décide de la réalisation des instances du processus.
- Une *hypothèse* est une caractéristique essentielle d'un processus qui influence le déroulement d'une stratégie. Lorsque le concepteur essaie une nouvelle hypothèse, il accomplit une autre instance d'un processus en générant une nouvelle instance d'un type de produit. Il convient alors de choisir le meilleur produit en regard des exigences à satisfaire.
- L'*historique* est la trace du déroulement de la stratégie d'un processus. On définit l'historique d'un processus P comme l'ensemble de tous les processus et produits qui contribuent à l'accomplissement de P.

Le point 7.1.4 donnera une description plus approfondie du modèle des processus dans le cadre de son implémentation dans l'approche DB-Main. La figure 3.24 a) présente sous la forme d'un graphe un exemple d'historique de conception de la base de données de gestion des commandes dont les spécifications sont décrites dans le point 3.1.3. Dans cette figure, les rectangles représentent les processus, les ellipses représentent les produits et les polygones (marqués *Selection*) symbolisent la sélection d'un produit (celui du côté de la double flèche) parmi plusieurs produits engendrés par des hypothèses différentes (rectangles en pointillés).

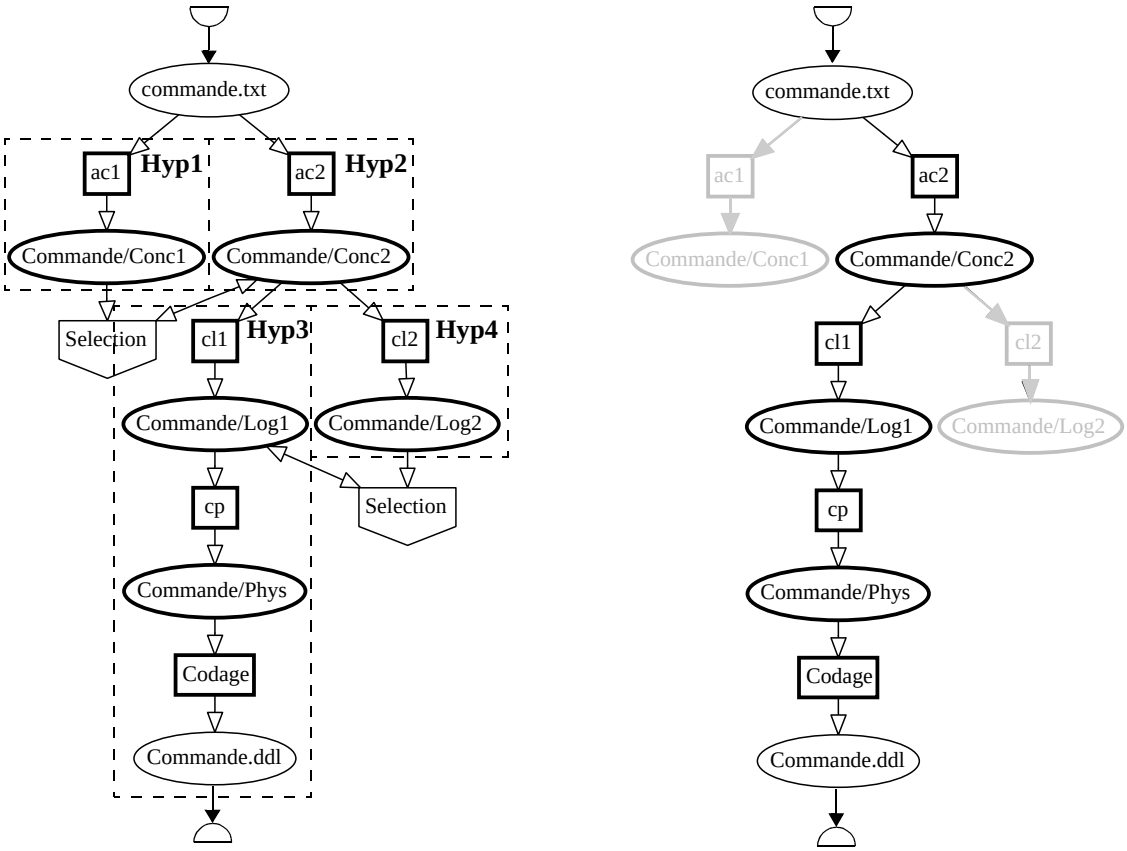
3.3.4 Historique

3.3.4.1 Niveau d'agrégation

Un historique peut être matérialisé par une séquence d'opérations. Cependant, puisque le modèle autorise les hypothèses, cette séquence peut se transformer en *graphe* complexe. Un historique peut être simplifié en associant chaque processus de sélection avec la branche réussie, c'est-à-dire la branche développée à partir du produit sélectionné. Par conséquent, l'historique simplifié apparaît comme un *arbre*. Plus concrètement, la figure 3.24 b) montre l'arbre de

5. C'est-à-dire une opération non décomposable dont le résultat est prévu à l'avance.

l'historique de la figure 3.24 a) dans laquelle les processus de sélection ont été enlevés. Si les branches correspondant à des hypothèses non retenues ou des produits écartés sont élaguées, on obtient un *historique linéaire*. Un historique linéaire décrit la manière dont les produits finaux peuvent être dérivés. Rejouer cet historique sur les produits sources produit les mêmes produits résultats. Les produits et les processus non grisés de la figure 3.24 b) forment l'historique linéaire de la conception.



a) Graphe

b) Arbre et historique linéaire (produits et processus non grisés)

Figure 3.24 - Un historique de conception présenté sous la forme d'un graphe, d'un arbre et d'un historique linéaire.

Un historique peut être présenté avec différents niveaux de détails. La figure 3.24 a) décrit une instance d'un processus d'ingénierie sans détailler les sous-processus qu'il contient. Par contre, le processus de conception logique peut être détaillé en donnant toutes les transformations qui le composent. On distingue deux niveaux d'agrégation pour un historique :

- Les *historiques structurés* apparaissent comme des arbres où chaque nœud représente un processus ou un produit (figure 3.24 b).
- Les *historiques plats* montrent les processus primitifs. Ils sont indépendants de toute méthodologie et peuvent être enregistrés sous une forme assez agréable et simple à comprendre. Dans ce cas, il s'agit d'une simple séquence de transformations qui peut être matérialisée sous la forme d'un journal.

3.3.4.2 Définition

Dans le cadre de ce travail, on s'intéresse à la trace des transformations qui produisent une spécification S_j à partir d'une spécification S_i et qu'on appelle *historique* [6] (H_{ij}) du processus de trans-

6. Dans la suite du travail, le terme d'historique est abusivement utilisé pour désigner un historique plat.

formation. H_{ij} étant lui-même une macro-transformation, on utilise la notation fonctionnelle $S_j = H_{ij}(S_i)$ avec $H_{ij} = <T_1\ T_2\ ...\ T_n>$. H_{ij} exprime une séquence de transformations. De manière plus pragmatique, la notation T_i ($1 \leq i \leq n$) peut être remplacé par la signature de la transformation correspondante. Par exemple, l'historique H_{CL} représente l'historique de conception logique transformant le schéma conceptuel de la figure 3.9 en un schéma logique (figure 3.10) :

```
H_CL = <
  T1: (PERSONNE,{pers_cli,pers_four})<-ISA-en-TA(PERSONNE,{CLIENT,FOURNISSEUR})
  T2: (REFERENCE,{com_ref,art_ref})<-TA-en-TE(référence)
  T3: (TELEPHONE,a)<-Att-en-TE-inst(PERSONNE,Téléphone)
  T4: (PERSONNE,{Adr_Numero,Adr_Rue,Adr_Localite})<-desagreger(PERSONNE,Adresse)
  T5: (PERSONNE,{NumCli})<-TA-en-FK(pers_cli)
  T6: (PERSONNE,{NumFourn})<-TA-en-FK(pers_four)
  T7: (TELEPHONE,{NumPers})<-TA-en-FK(a)
  T8: (COMMANDE,{NumCli})<-TA-en-FK(passe)
  T9: (ARTICLE,{NumFourn})<-TA-en-FK(offre)
  T10:(REFERENCE,{NumCom})<-TA-en-FK(com_ref)
  T11:(REFERENCE,{NumArticle})<- TA-en-FK(art_ref)
>
```

3.3.4.3 Propriétés

Les propriétés définies ci-dessous sont employées dans les chapitres 5 et 7 pour la simplification ou la réorganisation des historiques de conception et d'évolution.

a) Indépendance

Si on considère l'historique $H_{ij} = <... T_i ... T_j ...>$ dans lequel il y a entre autres deux transformations T_i et T_j, on peut se demander si l'exécution de T_j dépend du résultat de l'exécution de T_i ou si ces exécutions sont indépendantes. La relation d'ordre *avant(T_i,T_j)* établissant que la transformation T_i doit être exécutée avant T_j est définie en termes de structures comme :

$$avant(T_i,T_j) \Leftrightarrow C^{=}(T_i) \cap C^{-}(T_j) \neq \varnothing \vee C^{+}(T_i) \cap C(T_j) \neq \varnothing$$

T_j doit suivre T_i si elle détruit des objets préservés par T_i ou si elle utilise des objets créés par T_i. Par conséquent, T_i et T_j sont *indépendants* si on a :

$$\neg avant(T_i,T_j) \wedge \neg avant(T_j,T_i)$$

Dans l'historique H_{CL} du point 3.3.4.2, on obtient : $avant(T_3,T_7) \neq \varnothing$ car $C^{+}(T_3) \cap C(T_7)$ = {TELEPHONE,a} puisque $C^{+}(T_3)$ = {TELEPHONE,a} et $C(T_7)$ = {TELEPHONE,a}. Par conséquent, T_7 est dépendant de T_3.

b) Equivalence

Deux historiques sont équivalents par rapport à un produit P si on a :

$$H_i(P) = H_j(P)$$

H_i est équivalent à H_j si H_j est construit à partir de H_i en intervertissant des transformations indépendantes. Mais ce n'est pas la seule condition. En effet, on peut imaginer que l'historique H_j contienne la même séquence de transformations que l'historique H_i avec deux transformations supplémentaires (l'une qui crée un objet et l'autre qui le détruit). H_i est bien équivalent à H_j alors qu'ils ne contiennent pas les mêmes transformations.

A partir de l'historique H_{CL}, l'historique H_{CL1} peut être construit en intervertissant les transformations T_3 et T_9 qui sont indépendantes.

c) Minimalité

Un historique enregistre le résultat des décisions d'un concepteur. Elles sont parfois mauvaises ce qui peut engendrer des retours en arrière, des essais et des erreurs. Il a une structure complexe dans laquelle plusieurs branches matérialisent l'exploration de différentes hypothèses. Une telle structure peut être simplifiée en éliminant les branches inutiles.

Dans le cas des historiques plats, on dit qu'un historique est *minimal* si :

$\forall H' \subset H : H'(p) \neq H(p)$

C'est-à-dire qu'il n'existe pas de sous-ensemble des transformations de H équivalents à H.

3.3.4.4 Opérations

Pour exploiter les historiques, nous avons défini deux opérations intéressantes qui permettent de rejouer et d'inverser un historique. Elles sont largement utilisées dans la suite de ce travail.

a) Action de rejouer un historique

Rejouer un historique est une opération cruciale dans le contexte de l'évolution. La notation fonctionnelle $S_j = H_{ij}(S_i)$ indique que H_{ij} est *rejoué* sur le schéma S_i en exécutant les transformations enregistrées dans H_{ij} sur S_i. Les transformations de H_{ij} sont appliquées une après l'autre sur le schéma S_i qui devient le schéma S_j.

Un historique est parfois rejoué sur un schéma modifié. Les modifications ont une influence sur le déroulement de cette opération. Supposons que $H_{ij} = <... T_i ...>$ et $C(T_i) = \{O_1, ..., O_n\}$. Lorsqu'on exécute T_i sur S_i, trois cas sont possibles :

- Les objets de $C(T_i)$ n'ont pas été modifiés ou détruits dans S_i. T_i est exécutée normalement sur S_i.
- Un ou plusieurs objets de $C(T_i)$ ont été modifiés ou détruits dans S_i et T_i ne peut pas être exécutée sur S_i. Cela signifie que, pour tout T_k appartenant à H_{ij} et vérifiant la propriété $avant(T_i,T_k) \neq \emptyset$, T_k n'est pas exécutée sur S_i. Supposons, dans le schéma de la figure 3.9, que l'attribut *Téléphone* du type d'entités *CLIENT* est supprimé. Lorsqu'on rejoue H_{CL}, les transformations T_3 et T_7 (dépendante de T_3) ne sont pas exécutées.
- Un ou plusieurs objets de $C(T_i)$ ont été modifiés ou détruits dans S_i et T_i peut être exécutée sur S_i. Supposons, dans le schéma de la figure 3.9, que la cardinalité maximum de l'attribut *Téléphone* du type d'entités *CLIENT* passe de 5 à 10. Lorsqu'on rejoue H_{CL}, T_3 et T_7 sont rejouées normalement car la modification n'influence pas l'exécution de T_3.

b) Inversion d'un historique

Il est également possible de dériver de H_{ij} la fonction inverse H_{ij}^{-1} telle que :

$S_i = H_{ij}^{-1}(S_j)$ avec $H_{ij} = <T_1 ... T_i ... T_n>$ et $H_{ij}^{-1} = <T_n^{-1} ... T_i^{-1} ... T_1^{-1}>$

H_{ij}^{-1} peut être obtenue en substituant à chaque transformation d'origine son inverse, puis en inversant l'ordre des opérations. Dans l'exemple, H_{CL}^{-1} est l'historique inversé de H_{CL} :

```
H_CL^-1 = <
  art_ref<-FK-en-TA(REFERENCE,{NumArticle})
  com_ref<-FK-en-TA(REFERENCE,{NumCom})
  offre<-FK-en-TA(ARTICLE,{NumFourn})
  passe<-FK-en-TA(COMMANDE,{NumCli})
  a<-FK-en-TA(TELEPHONE,{NumPers})
  pers_four<-FK-en-TA(PERSONNE,{NumFourn})
  pers_cli<-FK-en-TA(PERSONNE,{NumCli})
```

```
  (PERSONNE,Adresse)<-agreger-Att(PERSONNE,{Adr_Numero,Adr_Rue,Adr_Localite})
  (PERSONNE,Téléphone)<-TE-en-Att(TELEPHONE,a)
  référence<-TE-en-TA(REFERENCE)
  (PERSONNE,{CLIENT,FOURNISSEUR})<-TA-en-ISA(PERSONNE,{pers_cli,pers_four})
>
```

Les historiques plats étant des séquences de transformations, ils peuvent être inversés aisément car ils respectent les trois règles suivantes :

1. *Exhaustivité* : toute transformation est enregistrée dans l'historique de manière précise et complète pour permettre son inversion, ce qui est essentiel pour les transformations qui ne conservent pas la sémantique. Dans la figure 3.25, l'enregistrement de la suppression du type d'entités *E* nécessite l'enregistrement de la suppression de ses attributs, son groupe et le rôle qu'il joue dans *R*. L'historique de l'enregistrement contient les signatures suivantes :

```
()<-supprimer-Id(E,{A1})
()<-supprimer-Att(E,A1)
()<-supprimer-Att(E,A2)
()<-supprimer-Role(R,R.E)
()<-supprimer-TE(E)
```

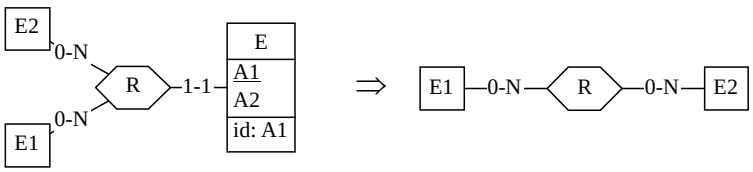


Figure 3.25 - Suppression du type d'entités *E* ainsi que ses attributs *A1* et *A2*, son identifiant et le rôle *R.E* qu'il joue dans *R*.

2. *Normalisation* : l'historique est monotone (pas de retours en arrière) et linéaire (pas de branchements multiples). Les historiques plats sont indépendants de toute méthodologie. A ce titre, le concepteur utilise des opérations primitives sans possibilité de développer plusieurs hypothèses basées sur des choix spécifiques de transformation ou revenir en arrière pour opérer des changements de transformation.
3. *Non-concurrence* : un historique est attaché à un schéma qui ne peut être modifié que par une seule personne à la fois. Cela empêche de développer en parallèle plusieurs versions différentes d'un même schéma de spécifications et, de cette manière, multiplier les historiques.

«Labor omnia vincit improbus.»

Virgile, Géorgiques, I, 145-146

CHAPITRE 4

TYPOLOGIE DES MODIFICATIONS DES SPÉCIFICATIONS

La première chose qu'il est important de mettre en place avant de construire une méthodologie et des outils pour l'évolution d'une application de base de données relationnelles, c'est une classification de toutes les modifications possibles sur des spécifications (basées sur le modèle de représentations décrit dans la section 3.1). On utilise le terme de typologie pour désigner cette classification car elle permet de dégager différents types de modifications en fonction du type d'objet modifié et du niveau d'abstraction considéré. Cette typologie constitue la base de la méthodologie de l'évolution (chapitre 5), des recommandations méthodologiques pour le concepteur (chapitre 6) et des outils d'aide à l'évolution (chapitre 7).

Vu le nombre de modifications possibles ainsi que leur complexité, seules certaines d'entre elles parmi les plus représentatives sont étudiées dans ce chapitre. L'étude complète se trouve dans l'annexe C. Cette annexe doit être appréhendée comme un répertoire dans lequel le concepteur peut puiser toutes les informations nécessaires concernant des modifications précises.

La section 4.1 présente la décomposition adoptée pour la typologie ainsi que la façon de présenter chaque modification. Les sections 4.2, 4.3 et 4.4 sont consacrées à l'analyse détaillée de certaines modifications.

4.1 Typologie des modifications

4.1.1 Introduction

Pour rappel, la conception est envisagée dans le cadre de la modélisation classique qui distingue trois niveaux d'abstraction, chaque niveau répondant à des besoins spécifiques. Les changements intervenant dans ces besoins engendrent des modifications dans le niveau correspondant. Vu le cadre méthodologique de ce travail, il semble plus approprié d'établir la classification par niveau d'abstraction. Pour chaque niveau, les modifications sont répertoriées par types d'objets et chaque modification est analysée en terme d'influence qu'elle peut avoir sur le système d'information.

Certains auteurs utilisent d'autres critères pour classer les modifications. Shneiderman [Shn82] met en place une classification basée sur l'influence qu'une modification peut avoir sur les structures de données, les données et les programmes. Roddick [Rod93] classe les modifications par types d'objets sur lesquels elles interviennent.

Le point 4.1.2 explique la décomposition adoptée pour la typologie et le point 4.1.3 décrit l'organisation par niveaux d'abstraction de l'analyse des modifications.

4.1.2 Décomposition de la typologie

La typologie [1] des modifications est composée de tous les types de modifications possibles pouvant être apportées à des spécifications du niveau conceptuel, logique ou physique. Vu la richesse du modèle de représentations des spécifications (chapitre 3), beaucoup de modifications sont envisageables. Certaines modifications, au niveau conceptuel, engendrent une série de modifications dans les niveaux inférieurs car les objets modifiés du niveau conceptuel n'ont pas de correspondant unique dans les autres niveaux. Par exemple, la modification d'un attribut multivalué au niveau conceptuel engendre plusieurs modifications relatives à une table (s'il a été transformé en un type d'entités) au niveau logique. Pour simplifier l'analyse et gérer la quantité et la complexité des modifications, le modèle générique est d'abord restreint puis enrichi de manière à scinder la typologie en trois parties (figure 4.1). La découpe adoptée est purement technique. Elle ne correspond pas à des modèles spécifiques de la littérature et n'est pas influencée par les stratégies de propagation proposées au chapitre 5. Cette découpe a été proposée pour la première fois par [Wag95]. Détaillons les sous-modèles issus de la découpe.

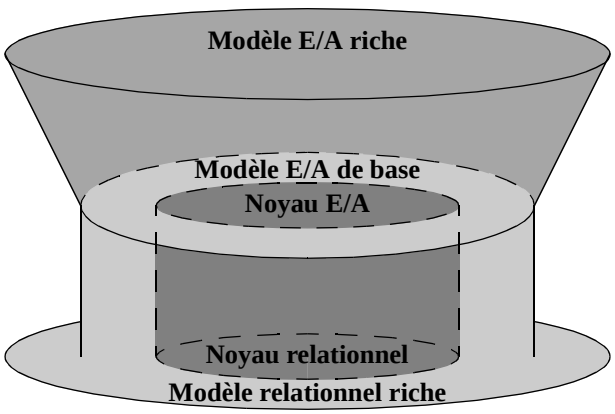


Figure 4.1 - Décomposition de l'analyse typologique des modifications par restriction et enrichissement du modèle générique.

1. Typologie : «... Etude des traits caractéristiques dans un ensemble de données, en vue d'y déterminer des types, des systèmes.» [Lar93]

4.1.2.1 Le noyau

Le modèle est d'abord restreint pour obtenir un noyau de base. Un sous-modèle relationnel restreint, appelé *noyau relationnel*, est défini. On construit un *noyau E/A* dans lequel les concepts sont directement et entièrement traduisibles en concepts du noyau relationnel. Cela signifie que chaque concept d'un des deux noyaux a une contrepartie dans l'autre. A cause de la relation bijective entre les deux modèles, les modifications réalisées sur les schémas du noyau E/A sont facilement traduisibles dans les schémas du noyau relationnel. La section 4.2 et l'annexe C.2 étudient les impacts des modifications sur ces concepts à tous les niveaux d'abstraction (conceptuel, logique et physique) ainsi qu'au niveau opérationnel (structures de données, instances et programmes). Le tableau 4.1 détaille les concepts autorisés. La lecture en ligne assure la correspondance entre les concepts des sous-modèles. Dans le noyau E/A, les types d'associations sont fonctionnels, non complexes, et la cardinalité de leurs rôles est déterminée de manière à éviter les contraintes référentielles d'égalité qui nécessitent l'utilisation de clauses complexes (CHECK, TRIGGER) ou de procédures (STORED PROCEDURE) au niveau opérationnel. Les concepts physiques (index et espaces de stockage) n'ont évidemment pas de correspondant dans le noyau E/A.

Noyau E/A	→	**Noyau relationnel**
Niveau Conceptuel	**Niveau logique**	**Niveau Physique**
Types d'entités	Tables	
Attributs monovalués et atomiques, facultatifs ou obligatoires	Colonnes *null* ou *not null*	
Identifiants (contenant des attributs ou des rôles)	Clés primaires	
Types d'associations binaires, sans attribut et du type (1-1/0-N, 1-1/0-1, 0-1/0-N, 0-1/0-1)	Clés étrangères	
		Index
		Espace de stockage

Tableau 4.1 - Description des concepts autorisés dans les noyaux de base.

4.1.2.2 Le modèle de base

Le noyau défini précédemment est un modèle pauvre. Il est enrichi pour obtenir le *modèle E/A de base*. Lorsque les concepts de ce sous-modèle sont traduits en concepts relationnels, on constate que le noyau relationnel est insuffisant. Il faut donc définir un sous-modèle relationnel enrichi, appelé *modèle relationnel riche*. Les concepts du modèle E/A de base ont une traduction unique et directe dans le modèle relationnel riche. Chaque schéma exprimé dans le modèle E/A de base est entièrement traduisible dans le modèle relationnel riche, mais le contraire n'est pas nécessairement vrai. Par exemple, certaines clauses CHECK n'ont pas d'équivalent dans le modèle E/A. L'enrichissement des noyaux nous oblige à revoir les modifications établies pour le noyau et implique de nouvelles modifications. Les sections 4.3 et C.3 analysent les impacts de ces modifications à tous les niveaux de la conception. Les deux sous-modèles enrichis sont décrits dans le tableau 4.2. Les nouvelles contraintes sont traduites par des clauses CHECK, TRIGGER ou des procédures au niveau opérationnel.

Modèle E/A de base	→	**Modèle relationnel riche**
Niveau Conceptuel		**Niveau logique**
Noyau E/A		Noyau relationnel
Types d'associations binaires, sans attribut et du type (0-1/1-N, 1-1/1-N, 1-1/1-1)		Clés étrangères et contraintes d'égalité
Contraintes d'exclusion, au-moins-un, coexistence, exactement-un		Contraintes d'exclusion, au-moins-un, coexistence, exactement-un

Tableau 4.2 - Description des concepts autorisés dans les sous-modèles E/A de base et relationnel riche.

4.1.2.3 Le modèle riche

Tous les schémas de bases de données ne peuvent pas s'exprimer dans le modèle E/A de base conceptuellement assez pauvre. On introduit donc le *modèle E/A riche* qui utilise tous les concepts du modèle. L'étude consistera à analyser comment les modifications sur le modèle E/A riche sont propagées vers le modèle E/A de base. Chaque modification du modèle E/A riche doit avoir une contrepartie dans le modèle E/A de base. Cette contrepartie est composée d'une ou plusieurs modifications. Par exemple, si un attribut décomposable devient facultatif, dans le schéma conforme au modèle E/A de base, les sous-attributs deviennent facultatifs et la contrainte de coexistence définie sur ceux-ci est supprimée. Le tableau 4.3 présente les concepts du modèle E/A riche. Vu la complexité des correspondances avec le modèle E/A de base, elles sont analysées dans la section 4.4 et l'annexe C.4 avec les nouvelles modifications engendrées par l'enrichissement du modèle conceptuel.

Modèle E/A riche → Modèle E/A de base
Modèle E/A de base
Attributs décomposables et multivalués
Types d'associations complexes
Identifiants d'attributs et de types d'associations
Rôles multi-TE
Relations IS-A

Tableau 4.3 - Description des concepts autorisés dans le modèle E/A riche.

4.1.3 Etude des modifications

Dans la typologie, chaque modification est décrite pour les niveaux conceptuel, logique et physique par les caractéristiques suivantes :

- une brève description (avec une représentation graphique générique),
- une signature,
- la catégorie sémantique à laquelle elle appartient en termes de préservation de l'information (voir point 3.2.2),
- les préconditions à respecter avant la modification,
- les postconditions après la modification,
- une description des modifications des instances,
- un exemple.

Au niveau opérationnel (données et programmes), chaque modification est analysée en termes de dépendance/indépendance par rapport aux données ou aux programmes. Elles sont décrites par les propriétés suivantes :

- au niveau des données et structures de données :
  - (in)dépendance des données par rapport à la modification (point 4.1.3.1),
  - script de conversion (en SQL-92) des données et structures de données (point 4.1.3.2),
  - un exemple.
- au niveau des programmes :
  - (in)dépendance des programmes par rapport à la modification (point 4.1.3.3),
  - modification du code dans les programmes (point 4.1.3.4).

Shneiderman et Thomas [Shn82] proposaient déjà une classification similaire basée sur l'indépendance ou la dépendance des données et des programmes ainsi que sur la préservation de sémantique des modifications.

4.1.3.1 (In)dépendance par rapport aux données

Une transformation est *indépendante des données* si, dans le nouveau schéma des structures de données, les instances vérifient toujours les contraintes. Par exemple, lorsqu'une colonne est retirée d'un identifiant primaire, il faut s'assurer que les autres colonnes assurent l'unicité des lignes de la table. Cette transformation est dépendante des données. Si on constate une inconsistance de données dans les nouvelles structures, l'administrateur de la base doit prendre une décision quant à l'exécution du processus de conversion. Pour la conversion des données, nous donnerons les scripts (en SQL-92) de conversion des données ainsi que ceux de vérification des contraintes, en cas de dépendance des données par rapport à une modification. Les problèmes rencontrés par l'administrateur de la base en cas de dépendance des données sont abordés dans les stratégies de propagation du chapitre 5.

4.1.3.2 Scripts SQL-92

Les SGBDR [2] présentent les données sous la forme de table et proposent un langage de requêtes (instructions, opérations exécutées par le SGBD) SQL [3]. Apparu en 1973, ce langage a été adopté comme langage standard [4] dans les SGBDR. Quatre normes ont déjà vu le jour : SQL-86, SQL-89, SQL-92 (aussi appelée SQL2) et la dernière en date dénommée SQL:1999 (dont les grandes nouveautés sont présentées dans [Eis99]). Le langage SQL (à travers les SGBDR) est présent sur tous les types de machines et de systèmes d'exploitation. Parmi les grands systèmes disponibles sur le marché, on peut citer Oracle de Oracle Corporation, DB2 d'IBM, Informix Dynamic Server d'Informix Software, Adaptive Server Enterprise de Sybase Incorporation et SQL Server de Microsoft. Il existe aussi de nombreux petits systèmes qui ont l'avantage d'être moins onéreux et plus faciles à mettre en œuvre (comme InterBase de Borland ou MySQL de MySQL AB).

Les scripts sont un ensemble de requêtes SQL. Une requête, qui est la description d'une instruction exécutée par le SGBDR, peut être soit interactive, soit contenue dans un programme. Si la requête est interactive, le résultat apparaît directement à l'écran. Une requête peut aussi être envoyée par un programme (écrit en COBOL, C++, Pascal, Java ou un langage de programmation propre au SGBDR). Dans ce cas, le résultat de la requête est stocké dans des variables du programme. Lorsque c'est possible, les scripts sont donnés pour être exécutés de manière interactive. Dans certains cas où le résultat d'une requête doit être utilisé par une autre requête, le script se présente sous la forme d'un programme écrit dans le langage PL/SQL d'Oracle.

Les éditeurs de SGBDR ne suivent pas toujours les normes. La plupart de SGBDR se basent sur la norme SQL-92 en n'hésitant pas à modifier la syntaxe ou à ajouter de nouvelles instructions. Peu de systèmes intègrent déjà les spécifications de la norme SQL:1999 [Eis99]. Dans un souci de généricité, la syntaxe SQL-92 a été adoptée pour les scripts de conversion. L'annexe A.2 propose cette syntaxe sous la forme d'une grammaire BNF [5]. Vu les lacunes de la norme pour des composants physiques comme les index et les espaces de stockage ou des composants procéduraux comme les déclencheurs (ou triggers) et les procédures stockées (stored procedures), la syntaxe présentée sera pour les concepts manquants celle d'un ou plusieurs SGBDR particuliers. Pour montrer les différences entre la norme SQL et le langage SQL des éditeurs, SQL-92

2. Système de gestion de bases de données relationnelles.
3. "Structured Query Language".
4. Les organismes internationaux de normalisation ANSI et ISO s'occupent, depuis 1986, date de la parution de la première norme SQL-86, de la définition et de mise en place de la norme SQL.
5. "Backus-Naur Form" (aussi "Backus-Normal Form") : technique standard de définition de syntaxes de langages informatiques développée par John Backus et Peter Naur.

est comparé à Oracle 8 [6] dans certains scripts. Les livres de références qui ont aidé à la construction des scripts sont :

- pour la norme SQL-92 : [Cel97], [Dat94], [Del95], [Lyn90], [Mar94], [Mel93], [SQL92], [vdL93], annexe A.2;
- pour Oracle 8 d'Oracle Corporation : [Abd95], [Loc97], [Ora97], [Owe96];
- pour DB2 5.2 d'IBM : [Cha96], [Mul97b], [DB298], [Cha98];
- pour SQL server 2000 de Microsoft : [Ser99];
- pour Informix Dynamic Server.2000 9.2 d'Informix Software : [Inf99a], [Inf99b];
- pour Adaptive Server Enterprise 12.0 de Sybase Incorporation : [Syb00].

4.1.3.3 (In)dépendance par rapport aux programmes

Une transformation est *indépendante des programmes* si elle est autorisée sans aucune modification de structures des programmes. On distingue trois niveaux de dépendance des programmes.

Premièrement, on parle d'*indépendance totale* lorsqu'un programme ne nécessite aucune correction pour continuer à fonctionner. C'est le cas lorsqu'on ajoute un index à une table. Cela améliore seulement les performances du programme qui s'exécute de manière identique.

Deuxièmement, l'*indépendance structurelle* indique que les programmes moyennant quelques petites adaptations sont toujours fonctionnels. Ils ne nécessitent pas de changements de structures. Le changement de nom d'une table ne nécessite pas de modification profonde des programmes mais seulement quelques remplacements de nom. On entend par modification de la structure d'un programme, un changement dans la séquence de ses instructions ainsi qu'un changement dans la structure d'une requête SQL.

Troisièmement, la *dépendance structurelle* indique que la modification des structures de la base de données entraîne des modifications profondes au niveau de la structure des programmes. L'ajout d'une colonne à une table nécessite la révision des requêtes de sélection et de modification ainsi que l'ajout de nouvelles instructions.

La distinction entre indépendance et dépendance structurelle n'est pas toujours aisée. Par exemple, augmenter le domaine d'une colonne (passer du type caractère de longueur 10 au type caractère de longueur 15) permet-il de considérer les programmes dépendants ou indépendants structurellement de la modification ? Cette modification va, par exemple, entraîner des changements dans l'interface où les champs (de saisie ou d'affichage de la colonne) seront augmentés. On considère donc que les programmes sont structurellement dépendants de la modification.

4.1.3.4 Modification du code

En ce qui concerne la modification des programmes, le principe adopté consiste à localiser les instructions ou les requêtes à modifier dans l'application. Le but est de donner une indication des parties de programmes où des modifications pourraient intervenir, à charge pour les programmeurs de réaliser les modifications nécessaires. Par exemple, la modification du nom d'une colonne entraîne des modifications des requêtes sur la table contenant cette colonne mais aussi, éventuellement, dans les dialogues avec l'utilisateur où le nouveau nom doit peut-être apparaître. Pour être plus précis dans la localisation, les requêtes ont été classées en deux types : les requêtes de sélection (SELECT ...) et les requêtes de modification (UPDATE, DELETE, INSERT). Pour chaque modification, on essaiera d'identifier quels types de requête sont impliqués.

Cette rubrique va permettre également de préciser l'élément sur lequel la recherche va s'effectuer. Dans l'exemple ci-dessus (modification du nom d'une colonne), les requêtes de sélection

6. A l'heure actuelle, Oracle est le logiciel le plus utilisé sur le marché des SGBDR.

de la forme "`SELECT ..., Ancien_nom_colonne, ... FROM ...`" ou "`SELECT * FROM ...`" sont susceptibles d'être corrigées. Donc, la recherche devra s'effectuer sur le nom de la colonne ainsi que le nom de sa table. De plus, il faudra suivre les variables contenant des instances de la colonne modifiée pour repérer les éléments de l'interface devant être changés.

Finalement, il peut être utile de préciser le type de correction à mettre en œuvre. Dire s'il s'agit simplement d'un remplacement de nom ou d'une modification profonde de la structure du code.

Comme on le verra dans le chapitre 5, la modification du code est loin d'être triviale, et donc, difficilement automatisable. Le but de cette rubrique est de donner une idée des conséquences de la modification et de l'ampleur de la correction à effectuer.

4.2 Noyau E/A → Noyau relationnel

Après une introduction sur les concepts des sous-modèles ainsi que les modifications associées (point 4.2.1), certaines modifications sont développées pour illustrer les mécanismes de l'analyse et les solutions proposées. Il s'agit de la création d'un type d'entités (point 4.2.2), du renommage d'un type d'entités (point 4.2.3), de la restriction du domaine d'un attribut (point 4.2.4), de la création d'un type d'associations (point 4.2.5), de la création d'un index (point 4.2.6) et de l'ajout d'une table dans un espace de stockage (point 4.2.7).

4.2.1 Introduction

Nous allons définir les contraintes que chaque spécification doit respecter pour le niveau auquel elle appartient (point 4.2.1.1). Les correspondances entre les concepts des deux modèles sont rappelées au point 4.2.1.2. Les deux tableaux récapitulatifs du point 4.2.1.3 résument les modifications analysées en précisant leur référence dans l'annexe C (consacrée à l'étude complète des modifications).

4.2.1.1 Propriétés des spécifications

Chaque spécification appartenant à un niveau d'abstraction doit respecter certaines contraintes avant et après chaque modification. Ci-dessous, les contraintes de chaque niveau sont détaillées. Les contraintes concernant le niveau opérationnel explicitent des choix de traduction de certains concepts.

- Niveau conceptuel :
  - un type d'entités a au moins un attribut;
  - les attributs sont monovalués et atomiques;
  - les seules contraintes autorisées sont les identifiants;
  - les types d'associations sont sans attribut et binaires du type 0-1/0-N, 1-1/0-N, 1-1/0-1, 0-1/0-1.
- Niveau logique :
  - une table a au moins un attribut;
  - les types d'associations sont interdits;
  - les clés étrangères et primaires sont autorisées;
- Niveau physique :
  - les index sont autorisés;
  - toute clé primaire est un index.
  - les espaces de stockage sont autorisés;
  - une table ne peut être stockée dans des espaces de stockage différents.
- Niveau opérationnel : les clés primaires sont traduites par des clauses PRIMARY KEY, les clés candidates par des clauses UNIQUE et les clés étrangères par des clauses FOREIGN KEY.

4.2.1.2 Correspondance entre les deux modèles

Le tableau 4.4 résume les correspondances entre les concepts des deux modèles.

Noyau Entité/Association	Noyau relationnel
type d'entités	table
type d'associations	clé étrangère
attribut monovalué/atomique	colonne
attribut obligatoire	colonne *not null*
attribut facultatif	colonne *null*
identifiant primaire	clé primaire
identifiant secondaire	clé candidate

Tableau 4.4 - Correspondance directe entre le noyau E/A et le noyau relationnel.

4.2.1.3 Tableau récapitulatif des modifications

Les tableaux 4.5 et 4.6 présentent les modifications classées par niveau d'abstraction avec la référence de la page. Le tableau se lit ligne par ligne de manière à suivre une modification à travers tous les niveaux du processus de conception. Pour les tables, les colonnes, les clés primaires, candidates et étrangères, l'analyse des modifications dans le niveau physique est absente car elle n'apporte pas d'informations nouvelles par rapport à celle du niveau logique. Seules les modifications relatives aux index et espaces de stockage sont développées dans le niveau physique.

Niveau conceptuel (C.2.2)	Niveau logique (C.2.3)	Données et programmes (C.2.5)
Types d'entités (C.2.2.1)	**Table (C.2.3.1)**	**Table (C.2.5.1)**
Création (p. 38)	Création (p. 53)	Création (p. 74)
Destruction (p. 38)	Destruction (p. 54)	Destruction (p. 75)
Renommage (p. 39)	Renommage (p. 54)	Renommage (p. 75)
Attributs (C.2.2.2)	**Colonnes (C.2.3.2)**	**Colonnes (C.2.5.2)**
Création (p. 39)	Création (p. 55)	Création (p. 76)
Destruction (p. 40)	Destruction (p. 55)	Destruction (p. 77)
Renommage (p. 40)	Renommage (p. 56)	Renommage (p. 78)
Changement d'une contrainte obligatoire en facultative (p. 41)	Changement d'une contrainte NOT NULL en NULL (p. 56)	Destruction d'une contrainte NOT NULL (p. 78)
Changement d'une contrainte facultative en obligatoire (p. 42)	Changement d'une contrainte NULL en NOT NULL (p. 57)	Création d'une contrainte NOT NULL (p. 79)
Extension de domaine (p. 42)	Extension de domaine (p. 57)	Extension de domaine (p. 80)
Restriction de domaine (p. 43)	Restriction de domaine (p. 58)	Restriction de domaine (p. 81)
Changement de type (p. 44)	Changement de type (p. 59)	Changement de type (p. 82)
Type d'associations (C.2.2.3)	**Clés étrangères (C.2.3.3)**	**Clés étrangères (C.2.5.3)**
Création (p. 44)	Création (p. 59)	Création (p. 82)
Destruction (p. 45)	Destruction (p. 60)	Destruction (p. 83)
Renommage (p. 45)	Renommage (p. 61)	Renommage (p. 84)
Augmentation de la cardinalité maximum d'un rôle (p. 46)	Destruction de la clé primaire définie sur les colonnes de référence (p. 61)	Destruction de la clé primaire définie sur les colonnes de référence (p. 84)
Diminution de la cardinalité maximum d'un rôle (p. 47)	Création d'une clé primaire sur les colonnes de référence (p. 62)	Création d'une clé primaire sur les colonnes de référence (p. 84)
Augmentation de la cardinalité minimum d'un rôle (p. 47)	Changement des contraintes NULL définies sur les colonnes de référence en NOT NULL (p. 63)	Création de contraintes NOT NULL sur les colonnes de référence (p. 85)

Tableau 4.5 - Tableau récapitulatif de modifications conceptuelles sur le noyau E/A et logiques sur le noyau relationnel.

Niveau conceptuel (C.2.2)	Niveau logique (C.2.3)	Données et programmes (C.2.5)
Diminution de la cardinalité minimum d'un rôle (p. 48)	Changement des contraintes NOT NULL définies sur les colonnes de référence en NULL (p. 64)	Destruction des contraintes NOT NULL définies sur les colonnes de référence (p. 86)
Identifiants primaires et secondaires (C.2.2.4)	**Clés primaires et candidates (C.2.3.4)**	**Clés primaires et candidates (C.2.5.4)**
Création (p. 49)	Création (p. 65)	Création (p. 86)
Destruction (p. 50)	Destruction (p. 65)	Destruction (p. 87)
Renommage (p. 50)	Renommage (p. 66)	Renommage (p. 88)
Changement d'un identifiant primaire en secondaire et inversement (p. 51)	Changement d'une clé primaire en candidate et inversement (p. 66)	Changement d'une clé primaire en candidate et inversement (p. 88)
Ajout d'un composant (p. 51)	Ajout d'un composant (p. 66)	Ajout d'un composant (p. 89)
Retrait d'un composant (p. 52)	Retrait d'un composant (p. 67)	Retrait d'un composant (p. 90)

Tableau 4.5 - Tableau récapitulatif de modifications conceptuelles sur le noyau E/A et logiques sur le noyau relationnel.

Niveau physique (C.2.4)	Données et programmes (C.2.5)
Index (C.2.4.1)	**Index (C.2.5.5)**
Création (p. 68)	Création (p. 90)
Destruction (p. 69)	Destruction (p. 91)
Renommage (p. 69)	Renommage (p. 91)
Ajout d'un composant (p. 70)	Ajout d'un composant (p. 91)
Retrait d'un composant (p. 71)	Retrait d'un composant (p. 92)
Espaces de stockage (C.2.4.2)	**Espaces de stockage (C.2.5.6)**
Création (p. 71)	Création (p. 92)
Destruction (p. 72)	Destruction (p. 93)
Renommage (p. 72)	Renommage (p. 94)
Ajout d'une table (p. 73)	Ajout d'une table (p. 95)
Retrait d'une table (p. 73)	Retrait d'une table (p. 95)

Tableau 4.6 - Tableau récapitulatif de modifications physiques sur le noyau relationnel.

4.2.2 Création d'un type d'entités / d'une table

4.2.2.1 Niveau conceptuel

Il s'agit de créer dans un schéma conceptuel un nouveau type d'entités. Cette modification est de catégorie T^+ car elle augmente la sémantique du schéma et sa signature est : E ← creer-TE(S,n).

La précondition est :

- Le nom du nouveau type d'entités est unique dans le schéma conceptuel.

La postcondition est :

- Un type d'entités avec le nom unique n est créé dans le schéma S.

Au niveau des instances, le type d'entités est vide.

La création d'un type d'entités est souvent accompagnée de la création de ses attributs ainsi que de types d'associations et de rôles pour la relier à d'autres types d'entités.

Dans la figure 4.2, le type d'entités *FOURNISSEUR* est créé avec un numéro (*Nfour)*, un nom (*Nom*), une adresse (*Adresse*), un numéro de téléphone (*Telephone*) et un jour de livraison (*Jour*).

Figure 4.2 - Création du type d'entités *FOURNISSEUR* et de ses attributs.

4.2.2.2 Niveau logique

La nouvelle table est créée dans le schéma logique. Cette modification est de catégorie T^+ et sa signature est : T ← creer-TE(S,n).

La précondition est :

- Le nom de la nouvelle table est unique dans le schéma logique.

Les postconditions sont :

- Une table de nom n est créée dans S.
- La table possède au moins une colonne (contrainte liée au modèle relationnel).

Au niveau des instances, le table est vide.

La création d'une table est accompagnée de la création d'au moins une colonne ainsi que de contraintes éventuelles (identifiantes, référentielles, ...).

Dans l'exemple, la table *FOURNISSEUR* est créée avec les colonnes *Nfour*, *Nom*, *Adresse*, *Telephone* et *Jour*.

4.2.2.3 Données et programmes

a) Données

La création d'une table n'influence pas les instances existantes de la base de données, elle est indépendante des programmes.

Le script de création d'une table est :

```
CREATE TABLE t (c1 datatype NOT NULL,...,constraint ...);
```

Certains SGBDR considèrent que, par défaut, une colonne est NOT NULL (obligatoire) et, donc, exigent l'utilisation de la contrainte NULL pour les colonnes facultatives. En règle générale, dans la plupart des SGBDR, une colonne est déclarée NULL par défaut. Le tableau annexe C.1 (page 36) présente la contrainte définie par défaut pour certains SGBDR.

La création d'une table s'accompagne de la création des colonnes, de clés étrangères, des clés primaires ou candidates, ... La norme SQL ainsi que les SGBDR imposent que la table créée possède au moins une colonne.

Dans l'exemple, on crée une table *FOURNISSEUR* avec les colonnes *Nfourn*, *Nom*, *Adresse*, *Telephone* et *Jour* à l'aide du script :

```
CREATE TABLE fournisseur (nfourn NUMERIC(5) NOT NULL,nom CHAR(30) NOT NULL,
                          adresse CHAR(100) NOT NULL,telephone CHAR(10),
                          jour CHAR(8) NOT NULL);
```

b) Programmes

La création de la table n'affecte pas les programmes. Toutefois, pour prendre en compte la nouvelle table, il faut créer de nouvelles fonctions de gestion (création, modification, suppression, affichage, ...). Ces modifications sont à la charge du programmeur.

4.2.3 Renommage d'un type d'entités / d'une table

4.2.3.1 Niveau conceptuel

Un type d'entités est renommé dans un schéma conceptuel. Cette modification est de catégorie $T^{=}$ car elle ne modifie pas la sémantique du schéma. Sa signature est : E ← modifier-TE(E,n').

La précondition est :

- Le nouveau nom n' est unique parmi les types d'entités du schéma conceptuel.

La postcondition est :

- Le type d'entités est renommé n'.

Au niveau des instances, le renommage n'a pas d'impact sur les données du type d'entités.

Dans la figure 4.3, le type d'entités *FOURNISSEUR* est renommé *GROSSISTE*.

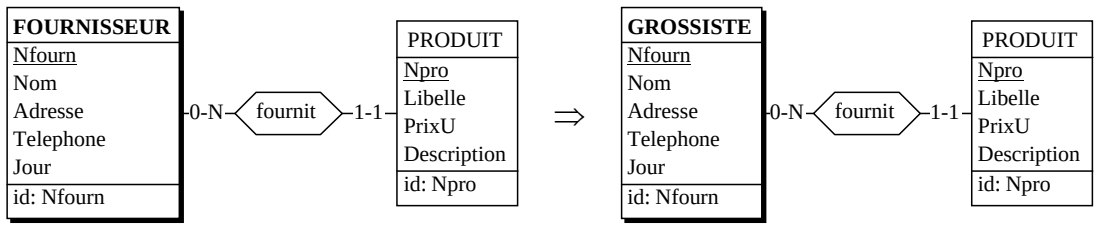


Figure 4.3 - Le type d'entités *FOURNISSEUR* est renommé *GROSSISTE*.

4.2.3.2 Niveau logique

La table du schéma logique est renommée. Cette modification est de catégorie $T^{=}$ et sa signature est : T ← modifier-TE(T,n').

La précondition est :

- Le nouveau nom de la table est unique dans le schéma logique.

La postcondition est :

- La table est renommée n'.

Les instances de la table restent inchangées.

Dans la figure 4.4, la table *FOURNISSEUR* est renommée *GROSSISTE*.

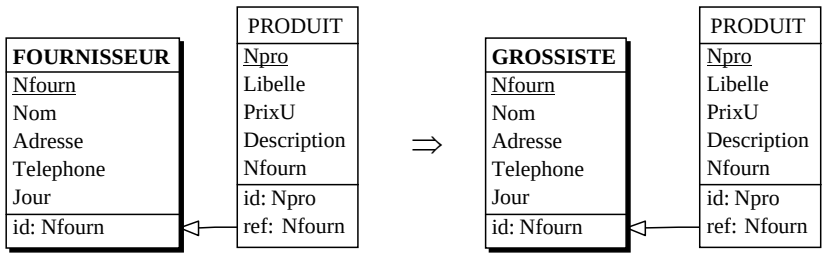


Figure 4.4 - La table *FOURNISSEUR* est renommée *GROSSISTE*.

4.2.3.3 Données et programmes

a) Données

Les instances de la table modifiée sont indépendantes du renommage.

Certains SGBDR disposent d'une commande pour renommer une table (voir tableau annexe C.1 page 36). La renommage de la table se résume donc à une instruction SQL, toutes les contraintes et les instances de la table sont conservées. En Oracle, la commande de renommage est :

```
RENAME t TO t’;
```

Dans la norme SQL-92, cette commande n'existe pas. Une première solution consiste à créer une nouvelle table avec les mêmes colonnes et contraintes que la table initiale. La clause CASCADE autorise la destruction d’une table référencée par des clés étrangères. Il n’est pas nécessaire de détruire ces clés avant de détruire la table. Le script de renommage est :

```
-- Créer t’ identique à t.
CREATE TABLE t’ (c1 datatype [NOT NULL],...,constraint ...,...) [IN spc];
-- Recréer les index sur t’.
[CREATE INDEX idx_table ON t’ (c1,...);
 ...;]
-- Transférer les instances de t vers t’.
INSERT INTO t’ SELECT * FROM t;
-- Créer les clés étrangères référençant t’.
[ALTER TABLE tref ADD CONSTRAINT fk_t’
   FOREIGN KEY (colref,...) REFERENCES t’ (colid,...);
 ...;]
-- Détruire t.
DROP TABLE t CASCADE;
```

Pour le schéma de la figure 4.4, le script implémentant la première technique est :

```
CREATE TABLE grossiste (nfourn NUMERIC(5) NOT NULL,nom CHAR(30) NOT NULL,
                        adresse CHAR(100) NOT NULL,telephone CHAR(10) NOT NULL,
                        jour CHAR(8) NOT NULL,
                        CONSTRAINT id_gross PRIMARY KEY (nfourn));
INSERT INTO grossiste SELECT * FROM fournisseur;
ALTER TABLE produit ADD CONSTRAINT fk_grossiste
  FOREIGN KEY (nfourn) REFERENCES grossiste (nfourn);
DROP TABLE fournisseur CASCADE;
```

Une deuxième solution se base sur la technique des vues[7]. Il faut créer une vue modifiable (nommée comme le nouveau nom de la table) qui contient toutes les colonnes de la table. L’instruction de création d’une telle vue est :

```
CREATE VIEW t’ AS SELECT * FROM t;
```

Par vue modifiable, on sous-entend une vue dans laquelle des données peuvent être ajoutées (par l’instruction INSERT) ou retirées (par l’instruction DELETE). Il est préférable que la vue soit modifiable de façon à assurer une cohérence entre le schéma des spécifications et les instructions SQL. En effet, si la vue n’était pas modifiable, il faudrait, pour insérer ou détruire une ligne, utiliser l’ancien nom de la table. Pour être modifiable, l’instruction de création de la vue doit respecter les contraintes suivantes dans la norme SQL-92[8] :

– L’instruction SELECT ne contient pas les mots-clés DISTINCT, JOIN, UNION, INTERSECT, EXCEPT, GROUP BY ou HAVING.

7. Les lecteurs intéressés par plus d’informations sur les vues peuvent consulter [Dat94] et [Dat00].
8. Dans la réalité, chaque constructeur de SGBDR définit ses propres règles.

- La clause FROM du SELECT porte sur une seule table.
- Chaque champ de la clause SELECT est soit une référence à une colonne de la table concernée par la clause FROM, soit une référence à l'ensemble des colonnes (représentée par le caractère générique '*'). Les mots-clés COUNT, AVG, SUM, ... sont proscrits. Les SGBDR récents sont moins stricts pour les champs de la clause SELECT.
- Les requêtes imbriquées sont interdites (pas de clause SELECT dans la clause WHERE).

Une troisième solution peut être l'utilisation de synonymes. Certains SGBDR disposent d'une instruction permettant de créer des synonymes pour une table (voir le tableau annexe C.1 page 36), la table n'étant pas réellement renommée. Il est normalement possible d'effectuer toutes les opérations sur la table en utilisant son synonyme. Pour Oracle 8, le script de création d'un synonyme est :

```
CREATE SYNONYM t' for t;
```

Le tableau 4.7 présente les avantages et inconvénients de chaque solution. Au niveau de la complexité et de la performance, l'utilisation de vues ou de synonymes sont des techniques simples qui utilisent peu de ressources. Par contre, la création d'une nouvelle table est une technique délicate à implémenter. Les colonnes, les clés de la table et les clés étrangères qui la référencent doivent être recréées. Dans le cas de tables contenant beaucoup de données, le transfert de données consomme des ressources importantes. Au niveau de la flexibilité, les techniques des vues et des synonymes posent des problèmes car les modifications (ajout, suppression de colonnes ou de contraintes, destruction de la table) ne peuvent se faire que sur la table. Il y a une incohérence entre le schéma de spécification physique et le script de modification qui ne peut être résolue qu'en conservant l'ancien nom de la table pour effectuer les modifications. Dans le cas de la vue, le problème est plus complexe puisque, si on ajoute une colonne à une table, elle ne fait pas partie de la vue. La vue doit être détruire et recréée pour tenir compte de la nouvelle colonne. Les problèmes de modification des vues ont été souvent abordés dans la littérature (voir point 1.2.2.2 c).

Lorsque le système utilisé ne possède pas d'instruction spécifique pour renommer une table, le choix d'une des trois techniques analysées repose sur les épaules de l'ingénieur de la maintenance qui doit évaluer la meilleure solution en fonction des ressources disponibles et des modifications futures.

	Table	**Vue**	**Synonyme**
Complexité et consommation de ressources	+	-	-
Modification de la table (ALTER et DROP)	Oui	Non	Non
Prise en compte des modifications de la table	Oui	Non	Oui
Modification des données de la table (INSERT, UPDATE et DELETE)	Oui	Oui	Oui

Tableau 4.7 - Comparaison des quatre solutions techniques pour le renommage d'une table.

b) Programmes

Il convient de distinguer les trois solutions proposées précédemment.

Si une nouvelle table est créée, les programmes d'application ne dépendent pas structurellement de la modification. La recherche du nom original de la table et son remplacement par son nouveau nom dans le code est la seule modification nécessaire pour assurer le bon fonctionnement des programmes.

Si les techniques de création de vues ou de synonymes sont utilisées, les programmes sont totalement indépendants de la modification de renommage puisque la table renommée existe toujours avec son ancien nom.

Pour assurer la cohésion des applications par rapport aux structures de données, il faut peut-être changer certains noms de champs, de titres ou de commentaires dans les interfaces et les dialogues quelle que soit la technique utilisée.

4.2.4 Restriction du domaine d'un attribut / d'une colonne

4.2.4.1 Niveau conceptuel

Un attribut appartenant à un type d'entités a son domaine de valeurs restreint dans un schéma conceptuel. Cette modification est de catégorie T^- car elle modifie la sémantique du schéma en réduisant le nombre de valeurs possibles pour l'attribut modifié. Sa signature est : A ← modifier-Att(E,A,l').

La précondition est :

- L'attribut est de type caractère, numérique, réel.

La postcondition est :

- La nouvelle longueur du domaine de l'attribut modifié est inférieure à l'ancienne tout en restant supérieure à zéro.

Au niveau des instances, les valeurs de l'attribut modifié doivent faire partie des valeurs possibles du nouveau domaine. Dans le cas contraire, il existe un problème d'intégrité des données. L'analyse de la modification au niveau logique donne des pistes de solutions.

Dans la figure 4.5, le domaine de l'attribut *Adresse* du type d'entités *CLIENT* passe de 60 à 50 caractères.

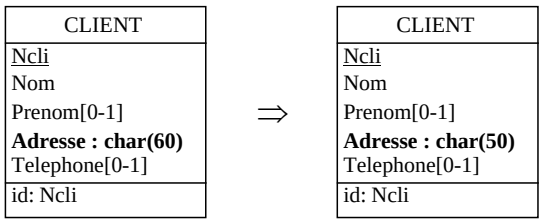


Figure 4.5 - Restriction du domaine de l'attribut *Adresse* du type d'entités *CLIENT*.

4.2.4.2 Niveau logique

Le domaine de valeurs de la colonne du schéma logique est réduit. Cette modification est de catégorie T^- et sa signature est : C ← modifier-Att(T,C,l').

La précondition est :

- La colonne est de type numérique, caractère ou réel.
- La colonne n'appartient pas à une clé étrangère. Si c'est le cas, la modification est refusée car elle risque d'altérer l'intégrité des données.

La postcondition est :

- La nouveau domaine de valeurs de la colonne modifiée a une longueur inférieure à l'ancien et supérieure à zéro.

Les valeurs des instances de la table pour la colonne modifiée doivent faire partie des valeurs possibles du nouveau domaine. Lorsqu'on réduit le domaine de valeurs, il se peut que certaines valeurs présentes dans la table ne respectent plus la nouvelle contrainte de domaine. Si la colonne modifiée appartient à un identifiant référencé par des clés étrangères, la modification doit être propagée dans les colonnes appartenant à ces clés. Cette propagation peut être en

cascade si, à leur tour, les colonnes de référence sont référencées. Dans la figure 4.6, la restriction du domaine de C1 entraîne la restriction des domaines de CT et CT1.

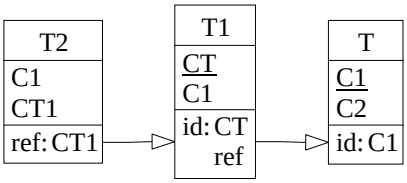


Figure 4.6 - Propagation des modifications via les clés étrangères.

Dans la figure 4.5, le domaine de la colonne *Adresse* de la table *CLIENT* passe de 60 caractères à 50.

4.2.4.3 Données et programmes

a) Données

Les données de la table contenant la colonne modifiée sont dépendantes de la restriction du domaine. Les valeurs des instances de cette colonne doivent être comprises dans le nouveau domaine de valeurs. Pour voir si les instances dépendent de la modification, une requête, vérifiant si la longueur des instances ne dépasse pas la longueur autorisée par le nouveau domaine, est exécutée :

```
SELECT * FROM t WHERE LENGTH(c) > l';
```

Si cette requête ne donne pas de résultat, l'ingénieur peut passer à la modification de la colonne. Dans le cas contraire, la violation de la nouvelle contrainte de domaine est prouvée. Trois solutions sont envisageables :

1. La modification est, soit refusée, soit reportée après la correction des instances problématiques.
2. La modification est appliquée en tronquant les valeurs trop grandes. Il faut utiliser des opérateurs de troncature et effectuer la même opération sur les colonnes qui référencent la colonne modifiée.

   ```
   UPDATE t SET c = SUBSTRING(c,1,l') WHERE LENGTH(c) > l';
   ```

   Pour les caractères, la norme SQL-92 définit la fonction SUBSTRING mais chaque SGBDR a ses fonctions propres de traitement de chaînes de caractères.
3. La modification est réalisée et les instances illicites de la table sont transférées dans une table temporaire.

   ```
   -- Créer la table contenant les lignes illicites.
   CREATE TABLE violation_t (c datatype [NOT NULL], ...);
   -- Transférer les lignes concernées dans violation_t.
   INSERT INTO TABLE violation_t SELECT * FROM t WHERE LENGTH(c) > l';
   -- Enlever les lignes illicites de t.
   DELETE FROM t WHERE LENGTH(c) > l';
   ```

Suivant les contraintes auxquelles appartient la colonne modifiée, les solutions disponibles varient. Trois cas de figure ont été relevés :

1. Si la colonne modifiée n'appartient à aucune clé primaire ou candidate, les trois solutions présentées sont réalisables.

2. Si la colonne appartient à une clé primaire ou candidate non référencée par des clés étrangères, la troncature n'est pas autorisée car elle risque d'entraîner la violation de la contrainte d'unicité.
3. Si la colonne appartient à une contrainte identifiante référencée par une clé étrangère, la solution du transfert des instances illicites, dans une table temporaire, est également abandonnée. En effet, elle nécessite l'extraction des instances des tables qui référencent les instances illicites et, ainsi de suite, si ces instances sont, à leur tour, référencées. La propagation des modifications complique fortement la sauvegarde des instances problématiques. La troncature n'est pas autorisée pour la même raison qu'au deuxième point.

La meilleure solution, pour l'ingénieur de la maintenance, est de corriger les instances problématiques avant de restreindre le domaine de la colonne modifiée. Elles sont souvent dues à des erreurs d'encodage ou révélatrices d'anomalies dans les données. Dans le cas contraire, il vaut mieux abandonner la modification.

Dans la norme SQL-92, il n'existe pas d'instruction de modification d'une colonne. La seule solution possible consiste à travailler avec une colonne temporaire. Le script de modification est :

```
-- Créer une colonne temporaire ctmp.
ALTER TABLE t ADD ctmp new_datatype [DEFAULT value NOT NULL];
-- Transférer les données de c vers ctmp.
UPDATE t SET ctmp = c;
-- Détruire c.
ALTER TABLE t DROP c CASCADE;
-- Recréer c avec son nouveau type.
ALTER TABLE t ADD c new_datatype [DEFAULT value NOT NULL];
-- Transférer les données de ctmp vers c.
UPDATE t SET c = ctmp;
-- Détruire ctmp.
ALTER TABLE DROP ctmp;
-- Recréer les identifiants sur c.
[ALTER TABLE t ADD CONSTRAINT id_t <PRIMARY KEY | UNIQUE> (c,...);]
-- Recréer les index sur c.
[CREATE INDEX idx ON t (c,...);
 ...;]
-- Propager la modification aux colonnes qui référencent l'identifiant dont c fait
-- partie. Un script équivalent à celui ci-dessus est appliqué autant de fois qu'il
-- y a de colonnes de référence à modifier.
-- Recréer les clés étrangères référençant c (et d'autres colonnes éventuellement).
[ALTER TABLE tref ADD CONSTRAINT fk_t
   FOREIGN KEY (cref,...) REFERENCES t (c,...);
 ...;]
```

Dans la figure 4.5, le script de restriction du domaine de la colonne *Adresse* de la table *CLIENT* est :

```
ALTER TABLE client ADD adresse_tmp CHAR(50) DEFAULT ' ' NOT NULL;
UPDATE client SET adresse_tmp = adresse;
ALTER TABLE client DROP adresse CASCADE;
ALTER TABLE client ADD adresse CHAR(50) DEFAULT ' ' NOT NULL;
UPDATE client SET adresse = adresse_tmp;
ALTER TABLE DROP adresse_tmp;
```

Certains SGBDR disposent d'une commande spécifique pour modifier une colonne (voir tableau annexe C.1 page 36). La restriction d'un domaine de valeurs se résume donc à une instruction SQL, toutes les contraintes et les instances de la table sont conservées. Cette commande de modification de colonne exige que la colonne soit vide dans le cas d'une restriction de domaine. Cette exigence est parfaitement compréhensible puisqu'elle évite de s'embarrasser des problèmes évoqués ci-dessus à propos des instances violant la nouvelle contrainte de domaine. Elle

permet de simplifier le script en évitant de devoir détruire puis recréer la colonne modifiée. Le script devient :

```
-- Créer une colonne temporaire ctmp.
ALTER TABLE t ADD ctmp new_datatype [DEFAULT value NOT NULL];
-- Transférer les données de c vers ctmp.
UPDATE t SET ctmp = c;
-- Vider les instances de la colonne c.
UPDATE t SET c = NULL;
-- Modifier le domaine de valeurs de c (syntaxe Oracle 8).
ALTER TABLE t MODIFY c new_datatype;
-- Transférer les données de ctmp vers c.
UPDATE t SET c = ctmp;
-- Détruire ctmp.
ALTER TABLE DROP ctmp;
-- Propager la modification aux colonnes qui référencent l'identifiant
-- dont c fait partie.
...
```

b) Programmes

Les programmes sont structurellement indépendants de la modification. Ils fonctionnent correctement mais, pour les nouvelles instances qui seront introduites, il faut modifier le format des variables et des champs des interfaces susceptibles de contenir les instances de la colonne modifiée. En partant de l'exemple de la figure 4.5, supposons qu'une boîte de dialogue saisisse toutes les informations concernant un nouveau client. Si le champ contenant l'adresse du client autorise un maximum de 60 caractères, cela engendrera des problèmes lorsque le nouveau client sera introduit dans la base de données. Une recherche sur le nom de la colonne dont on a restreint le domaine ainsi que les variables qui en dépendent doit être effectuée de manière à corriger ces variables ainsi que la longueur des champs de saisie correspondant.

4.2.5 Création d'un type d'associations / d'une clé étrangère

4.2.5.1 Niveau conceptuel

Un nouveau type d'associations ainsi que ses rôles sont créés dans un schéma conceptuel. Cette modification est de catégorie T^+ car elle augmente les informations représentables par les spécifications. Ses signatures sont : R $\leftarrow$ creer-TA(S,n) $\wedge$ RO1 $\leftarrow$ creer-Role(R,nro1,{E1},[i1-j1]) $\wedge$ RO2 $\leftarrow$ creer-Role(R,nro2,{E2},[i2-j2]).

La précondition est :

- Le nom du nouveau type d'associations est unique dans l'ensemble des types d'associations du schéma conceptuel.

Les postconditions sont :

- Le nouveau type d'associations est créé avec le nom n.
- Il est de type un-à-plusieurs ou un-à-un. Les cardinalités de ses rôles sont : [0-1]/[0-1], [0-1]/[1-1], [0-1]/[0-N] ou [1-1]/[0-N].

Au niveau des instances, si un des rôles du type d'associations créé a une cardinalité minimum de 1, cela signifie qu'une instance du type d'entités jouant ce rôle doit être obligatoirement reliée à une instance du nouveau type d'associations. La création du type d'associations entraîne, dans certains cas, la création d'instances pour ce type d'associations.

Dans la figure 4.7, le type d'associations *fournit* (0-N/1-1) est créé entre les types d'entités *FOURNISSEUR* et *PRODUIT*.

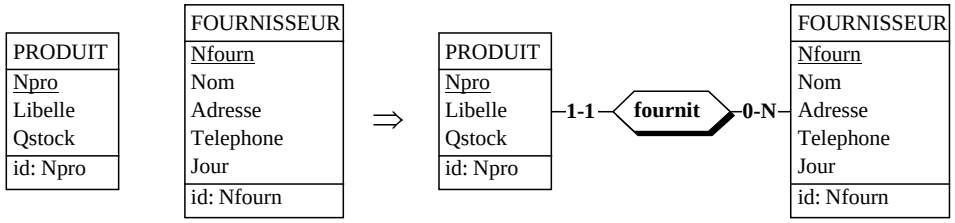


Figure 4.7 - Création du type d'associations *fournit* entre les types d'entités *PRODUIT* et *FOURNISSEUR*.

4.2.5.2 Niveau logique

Une ou plusieurs colonnes de référence ainsi qu'une clé étrangère sont créées dans le schéma logique. Cette modification est de catégorie T^+ et sa signature est : C1 ← creer-Att(T,n1,[i-1]) ∧ ... ∧ Ck ← creer-Att(T,nk,[i-1]) ∧ G ← creer-FK(T,ng,{C1,...,Ck},{Id1,...,Idk}).

Les préconditions sont :

- La table référencée doit contenir au moins un identifiant.
- Les noms des nouvelles colonnes de référence ainsi que le nom de la nouvelle clé étrangère sont uniques dans la table.

Les postconditions sont :

- On crée dans la table de référence autant de colonnes que dans la clé primaire (ou candidate) de la table référencée. Ces colonnes ont le même type que les colonnes de l'identifiant correspondant.
- Une clé étrangère est créée sur les nouvelles colonnes, elle référence la clé primaire (ou candidate) de la table référencée.
- En fonction des cardinalités des rôles du type d'associations que représente la clé étrangère, les cardinalités des colonnes et leurs contraintes sont différentes :
  - –si les rôles ont les cardinalités 0-1/0-N, alors les colonnes de référence sont NULL;
  - –si les rôles ont les cardinalités 0-1/0-1, alors les colonnes de référence sont NULL et forment une clé candidate de la table de référence;
  - –si les rôles ont les cardinalités 1-1/0-N, alors les colonnes de référence sont NOT NULL;
  - –si les rôles ont les cardinalités 1-1/0-1, alors les colonnes de référence sont NOT NULL et forment une clé primaire (ou candidate si la table de référence possède déjà une clé primaire).

Les colonnes de référence NOT NULL doivent avoir des valeurs pour chaque instance de la table de référence. Ces valeurs doivent correspondre avec celles des instances des colonnes identifiantes de la table référencée. Si les colonnes de référence sont facultatives, il suffit de les créer avec la valeur nulle par défaut.

Dans la figure 4.8, une colonne NOT NULL *Nfourn* est créée dans la table *PRODUIT*. Elle référence la clé primaire *Nfourn* de la table *FOURNISSEUR* par le biais d'une clé étrangère.

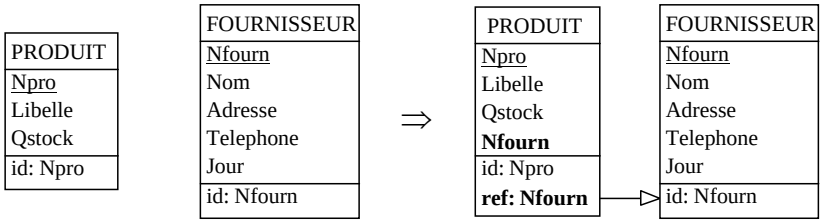


Figure 4.8 - Création de la colonne de référence *Nfourn* dans la table *PRODUIT* et de la clé étrangère qui référence la clé primaire de la table *FOURNISSEUR*.

4.2.5.3 Données et programmes

a) Données

Si la table de référence contient des instances, les données de la base sont dépendantes de la modification. Si les colonnes de référence sont obligatoires, elles doivent avoir des valeurs correctes correspondant à des valeurs d'instances de l'identifiant référencé. Si, en plus, elles forment un identifiant, ces valeurs doivent être distinctes. Le script de création est décomposé en trois phases.

La première phase crée les colonnes de référence en leur donnant une valeur par défaut si elles sont déclarées NOT NULL ou la valeur NULL si elles sont facultatives.

La deuxième phase s'occupe du remplissage des colonnes obligatoires. Pour certains SGBDR, la valeur NULL est considérée comme une valeur à part entière. A ce titre, elle ne peut figurer qu'une seule fois dans un identifiant. Dans le cas où les colonnes NULL sont des clés primaires ou candidates, on rencontre le même problème puisqu'une seule instance de la table avec des valeurs NULL pour les colonnes de référence peut exister, les autres violant la contrainte d'unicité. Cette tâche incombe à l'ingénieur de maintenance qui doit veiller à introduire des valeurs correctes respectant les contraintes.

Une fois les données introduites, la troisième phase consiste à créer les clés étrangères, primaires et candidates. Le script se présente comme suit :

```
-- Première phase : création des colonnes de référence.
ALTER TABLE t ADD c1 datatype <DEFAULT value NOT NULL | NULL>;
...;
ALTER TABLE t ADD ck datatype [DEFAULT value NOT NULL];
-- Deuxième phase : remplissage éventuel de valeurs pour les colonnes créées.
-- Troisième phase : création des contraintes.
ALTER TABLE t ADD CONSTRAINT fk_t_id
  FOREIGN KEY (c1,...,ck) REFERENCES t_id (cid1,...,cidk);
[ALTER TABLE t ADD CONSTRAINT
   id_t <PRIMARY KEY | uniq_t UNIQUE> (c1,...,ck);]
[CREATE INDEX idx ON table (c1,...,ck);]
```

D'après la figure 4.8, on crée dans la table *PRODUIT* une colonne obligatoire *Nfourn* qui référence la clé primaire *Nfourn* de la table *FOURNISSEUR*. Le script de modification de la table est :

```
ALTER TABLE produit ADD nfourn NUMERIC(5) DEFAULT 0 NOT NULL;
-- + Remplissage de valeurs dans la colonne nfourn
ALTER TABLE produit ADD CONSTRAINT fk_fournisseur
  FOREIGN KEY (nfourn) REFERENCES fournisseur (nfourn);
```

b) Programmes

Les programmes dépendent structurellement de la modification. Dans certaines instructions de manipulation de données (destruction, sélection, insertion ou modification), il faut tenir compte des colonnes de référence. Par exemple, une instruction du type `INSERT INTO T VALUES (...)` pose des problèmes puisque le nombre de paramètres de VALUES ne correspond plus au nombre de colonnes de la table. De même, la déclaration d'un curseur du type `DECLARE x CURSOR FOR SELECT * FROM T ...` pose des problèmes lorsqu'on analyse le curseur. Les interfaces (écrans, boîtes de saisies, ...) doivent être revues pour insérer des fonctions de gestion de la clé étrangère. Dans l'exemple, il est impossible de créer dans les programmes un nouveau produit sans le relier à un fournisseur. L'interface de création d'un produit doit être adaptée pour tenir compte de la modification.

A priori, il faut rechercher dans tous les programmes le nom de la table dans laquelle on a créé la clé étrangère ainsi que les variables dans lesquelles sont stockées des instances ou partie d'instances de la table. Dans l'exemple, une requête donnant la liste des produits disponibles n'est pas susceptible d'être modifiée. Par contre, une requête qui insère un nouveau produit dans la table doit être modifiée pour prendre en compte le lien entre un produit et son fournisseur.

4.2.6 Création d'un index

4.2.6.1 Niveau physique

Dans un schéma logique, un index est créé sur une ou plusieurs colonnes d'une table. Cette modification est de catégorie T^+ et sa signature est : I ← creer-Index(T,n,{C1,...,Ck}).

La précondition est :

- Le nom de l'index est unique parmi tous les index du schéma physique.

Les postconditions sont :

- Un index contenant des colonnes de la table est créé avec le nom choisi.
- Les colonnes appartenant à l'index ne forment pas un préfixe d'un autre index. Sinon il est inutile.

Cette modification n'a aucune influence sur les instances du schéma.

Dans la figure 4.9, un index sur les colonnes *Nom* et *Prenom* a été créé dans la table *CLIENT*.

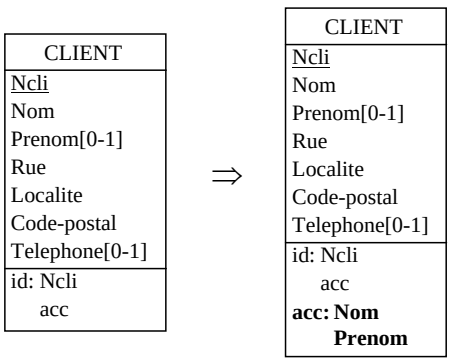


Figure 4.9 - Création d'un index sur les colonnes *Nom* et *Prenom* de la table *CLIENT*.

4.2.6.2 Données et programmes

a) Données

Les instances de la table sur laquelle un index est créé sont indépendantes de la modification. La norme SQL-92 ne parle pas du concept d'index. Les SGBDR disposent d'une commande pour les créer (voir le tableau annexe C.1 page 36). Le script de création d'un index (syntaxe Oracle 8) est :

```
CREATE [UNIQUE] INDEX idx ON t (c1, ..., cn);
```

La clause UNIQUE n'est pas nécessaire si on a déjà créé une clé primaire ou candidate sur les mêmes colonnes. Cette clause vérifie simplement si les entrées dans l'index sont uniques. Certains SGBDR (dont Oracle 8) ajoutent systématiquement un index sur toutes les colonnes d'une table pour lesquelles on a défini une clé primaire ou candidate. Par conséquent, il devient inutile de créer des index pour ces colonnes.

D'après la figure 4.9, le script de création d'un index sur les colonnes *NOM* et *PRENOM* de la table *CLIENT* est :

```
CREATE INDEX idx_client ON client (nom, prenom);
```

b) Programmes

En règle générale, les modifications relatives aux index ne nécessitent pas de modifications au niveau des programmes d'application. Les index permettent d'accélérer certaines requêtes et le moteur SQL du SGBD les utilise pour améliorer ces temps de réponses. Dans ce contexte, certaines requêtes (recherche sur la table et les colonnes de l'index), dans les programmes, peuvent être revues suite à la création d'un nouvel index de manière à utiliser au mieux l'amélioration des temps d'accès.

4.2.7 Ajout d'une table dans un espace de stockage

4.2.7.1 Niveau physique

Dans un schéma physique, une table est ajoutée [9] dans un espace de stockage. Cette modification est de catégorie T^+ et sa signature est : (C,{E1,...,En,E}) ← modifier-Coll(C,{E}).

La précondition est :

- La table n'appartient pas à l'espace de stockage C.

La postcondition est :

- La table est transférée dans l'espace de stockage C.

Cette modification n'a aucune influence sur les instances du schéma.

Dans la figure 4.10, la table *FOURNISSEUR* est transférée dans l'espace de stockage *DBSPC_PROD*.

9. En générale, une table est transférée d'un espace de stockage dans un autre. Dans la typologie, un transfert est analysé comme deux modifications distinctes : un retrait d'une table d'un espace de stockage et un ajout de cette table dans un autre espace. Au niveau opérationnel, les deux modifications sont traitées simultanément pour optimiser au maximum le script de conversion.

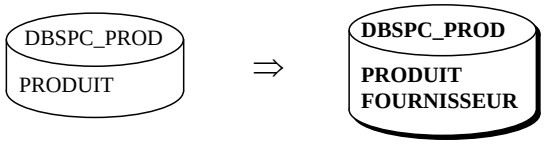


Figure 4.10 - Ajout de la table *FOURNISSEUR* dans l'espace de stockage *DBSPC_PROD*.

4.2.7.2 Données et programmes

a) Données

Les instances de la table sont indépendantes de la modification. La norme SQL-92 ne donnant aucune information sur la notion d'espace de stockage, nous utiliserons la syntaxe d'Oracle (voir le tableau annexe C.1 page 36 pour d'autres syntaxes). L'ajout d'une table, dans un espace de stockage, doit être réalisé à la création de la table ce qui pose des problèmes si elle est déjà créée et non vide. Dans ce cas, la seule solution envisageable consiste à utiliser une table temporaire pour stocker les instances et à recréer la table dans l'espace de stockage. Le script correspondant est :

```
-- Créer t_tmp avec la structure de t.
CREATE TABLE t_tmp (c1 datatype [NOT NULL], ...);
-- Insérer les données de t dans t_tmp.
INSERT INTO t_tmp SELECT * FROM t;
-- Supprimer t.
DROP TABLE t CASCADE;
-- Recréer t avec ses colonnes et ses contraintes en l'ajoutant à l'espace
-- de stockage.
CREATE TABLE t (c1 datatype [NOT NULL],...) TABLESPACE tablespace;
-- + création des clés primaires, candidates et étrangères de la table.
-- Insérer les données de t_tmp dans t.
INSERT INTO t SELECT * FROM t_tmp;
-- Recréer les clés étrangères qui référencent t.
[ALTER TABLE t_ref ADD CONSTRAINT fk_t
   FOREIGN KEY (cref,...) REFERENCES table (cid,...);]
-- Recréer les index de t.
[CREATE INDEX idx ON t (c1,...);
 ...;]
-- Détruire la table temporaire t_tmp.
DROP TABLE t_tmp;
```

Les modifications mettant en jeu des espaces de stockage sont des opérations qui peuvent être très lourdes pour de grandes quantités de données. Il faut les utiliser avec beaucoup de prudence. La meilleure solution reste la création d'une table directement dans son espace de stockage.

Sur la base de l'exemple de la figure 4.10 où la table *FOURNISSEUR* est ajoutée à l'espace de stockage *DBSPC_PROD*, le script de modification est :

```
CREATE TABLE fournisseur_tmp (nfourn NUMERIC(5) NOT NULL,
  nom CHAR(30) NOT NULL,adresse CHAR(100) NOT NULL,telephone CHAR(10),
  jour CHAR(8) NOT NULL);
INSERT INTO fournisseur_tmp SELECT * FROM fournisseur;
DROP TABLE fournisseur CASCADE CONSTRAINT;
CREATE TABLE fournisseur (nfourn NUMERIC(5) NOT NULL,
  nom CHAR(30) NOT NULL,adresse CHAR(100) NOT NULL,telephone CHAR(10),
  jour CHAR(8) NOT NULL, PRIMARY KEY (nfourn), TABLESPACE dbspc_prod);
INSERT INTO fournisseur SELECT * FROM fournisseur_tmp;
CREATE INDEX fkpl ON fournisseur(nfourn);
```

```
ALTER TABLE produit ADD CONSTRAINT fkpf
  FOREIGN KEY (nfourn) REFERENCES fournisseur;
DROP TABLE fournisseur_tmp;
```

b) Programmes

Les programmes sont totalement indépendants des modifications relatives aux espaces de stockage. Bien que l'emplacement de stockage d’une table peut influencer les performances de certaines requêtes de mise à jour dans les programmes (notamment en fonction du support de stockage : CD, disques rapides ou lents, bandes, ...), aucune modification n'est nécessaire pour assurer le bon fonctionnement des applications.

4.3 Modèle E/A de base → Modèle relationnel riche

Après une introduction sur les concepts des modèles et les modifications associées (point 4.3.1), deux modifications sont analysées pour illustrer les mécanismes et les solutions proposées. Il s'agit de l'augmentation de la cardinalité minimum d'un rôle (point 4.3.2) et de l'ajout d'un composant à une contrainte (point 4.3.3).

4.3.1 Introduction

Avant d'analyser quelques modifications relatives aux concepts du modèle E/A de base et du modèle relationnel riche, il convient de définir les contraintes que chaque spécification doit respecter pour le niveau auquel elle appartient (point 4.3.1.1). Le tableau du point 4.3.1.2 récapitule les modifications analysées dans l'annexe C.3.

4.3.1.1 Propriétés des spécifications

Chaque spécification appartenant à un niveau d'abstraction doit respecter certaines propriétés avant et après chaque modification. Les contraintes données au point 4.2.1.1 (noyau E/A → modèle relationnel de base) sont toujours valables et complétées par les nouvelles contraintes :

- Niveau conceptuel :
  - les contraintes d'exclusion, au-moins-un, coexistence et exactement-un sont autorisées;
  - tous les types d'associations fonctionnels sont autorisés, en particulier les cardinalités 0-1/1-N, 1-1/1-N, 1-1/1-1.
- Niveau logique :
  - les clés étrangères associées à des contraintes d'égalité sont autorisées;
  - les contraintes d'exclusion, au-moins-un, coexistence et exactement-un sont autorisées.
- Niveau physique : pas de nouvelles contraintes.
- Niveau opérationnel : les clauses CHECK et TRIGGER ainsi que les procédures (STORED PROCEDURE) sont utilisées pour traduire les nouvelles contraintes.

4.3.1.2 Tableau récapitulatif des modifications

Le tableau 4.8 présente les modifications classées par niveau d'abstraction avec leur référence dans l'annexe C.3. Le tableau se lit ligne par ligne de manière à suivre une modification à travers tous les niveaux du processus de conception.

Niveau conceptuel (C.3.2)	Niveau logique (C.3.3)	Données et programmes (C.3.4)
Attributs (C.3.2.1)	**Colonnes (C.3.3.1)**	**Colonnes (C.3.4.1)**
Changement d'une contrainte facultative en obligatoire (p. 96)	Changement d'une contrainte NULL en NOT NULL (p. 105)	Création d'une contrainte NOT NULL (p. 113)
Types d'associations (C.3.2.2)	**Clés étrangères (C.3.3.2)**	**Clés étrangères (C.3.4.2)**
Création (p. 96)	Création (p. 105)	Création (p. 113)
Destruction (p. 97)	Destruction (p. 106)	Destruction (p. 114)
Renommage (p. 98)	Renommage (p. 107)	Renommage (p. 115)
Diminution de la cardinalité minimum d'un rôle (p. 98)	Suppression d'une contrainte d'égalité sur une clé étrangère (p. 108)	Destruction d'une contrainte d'égalité (p. 115)

Tableau 4.8 - Tableau récapitulatif de modifications conceptuelles sur le modèle E/A de base et logiques sur le modèle relationnel riche.

Niveau conceptuel (C.3.2)	Niveau logique (C.3.3)	Données et programmes (C.3.4)
Augmentation de la cardinalité minimum d'un rôle (p. 99)	Ajout d'une contrainte d'égalité sur une clé étrangère (p. 108)	Création d'une contrainte d'égalité (p. 116)
Contraintes (C.3.2.3)	**Contraintes (C.3.3.3)**	**Contraintes (C.3.4.3)**
Création (p. 100)	Création (p. 109)	Création (p. 117)
Destruction (p. 101)	Destruction (p. 110)	Destruction (p. 118)
Renommage (p. 101)	Renommage (p. 110)	Renommage (p. 118)
Changement de type (p. 102)	Changement de type (p. 111)	Changement de type (p. 119)
Ajout d'un composant (p. 103)	Ajout d'un composant (p. 111)	Ajout d'un composant (p. 119)
Retrait d'un composant (p. 104)	Retrait d'un composant (p. 112)	Retrait d'un composant (p. 120)

Tableau 4.8 - Tableau récapitulatif de modifications conceptuelles sur le modèle E/A de base et logiques sur le modèle relationnel riche.

4.3.2 Augmentation de la cardinalité minimum d'un rôle / ajout d'une contrainte d'égalité sur une clé étrangère

4.3.2.1 Niveau conceptuel

La cardinalité d'un rôle passe de 0 à 1 dans un schéma conceptuel. Cette modification est de catégorie T^- car l'ajout d'une contrainte obligatoire sur un rôle diminue la quantité d'informations représentables par les spécifications puisque les instances du type d'entités doivent être reliées à au moins une instance du type d'associations. Sa signature est : RO ← modifier-Role(R,RO,[1-j]).

La précondition est :

- R est un type d'associations fonctionnel dont les cardinalités des rôles sont [1-1]/[0-1], [1-1]/[0-N] ou [0-1]/[0-N].

La postcondition est :

- Les cardinalités de R deviennent : [1-1]/[1-1], [1-1]/[1-N] ou [0-1]/[1-N].

Cette modification impose que chaque instance du type d'entités soit reliée à au moins une instance de R.

Dans la figure 4.11, le type d'associations *fournit* a les cardinalités 1-1/0-N pour ses rôles. Un fournisseur peut ne fournir aucun produit alors qu'un produit est fourni par un et un seul fournisseur. L'augmentation de la cardinalité minimum du rôle, joué par *FOURNISSEUR* dans *fournit*, impose qu'un fournisseur fournisse au moins un produit pour apparaître dans la base de données.

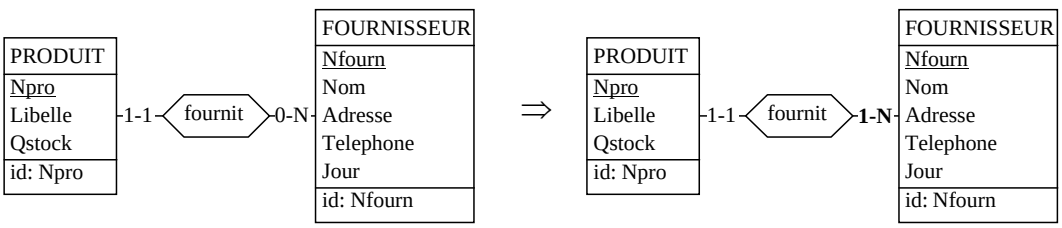


Figure 4.11 - Augmentation de la cardinalité minimum du rôle joué par *FOURNISSEUR* dans *fournit*.

4.3.2.2 Niveau logique

Dans le schéma logique, une contrainte d'égalité est ajoutée sur une clé étrangère. Cette modification est de catégorie T^- et sa signature est : G ← modifier-Groupe(T,G,egal).

La précondition est :

- Le groupe G est une clé étrangère.

La postcondition est :

- G devient une clé étrangère associée à une contrainte d'égalité.

La modification influence les instances du schéma car elle impose que chaque instance de la table cible de la contrainte soit référencée par au moins une instance de la table source contenant la clé étrangère.

Dans la figure 4.12, une contrainte d'égalité est ajoutée à la clé étrangère entre *PRODUIT* et *FOURNISSEUR*. Un instance de *FOURNISSEUR* est toujours référencée par au moins une instance de *PRODUIT*.

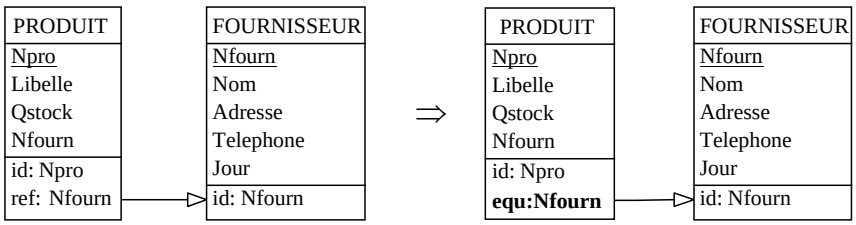


Figure 4.12 - Ajout d'une contrainte d'égalité à la clé étrangère entre *FOURNISSEUR* et *PRODUIT*.

4.3.2.3 Données et programmes

a) Données

Les instances de la table sont dépendantes de la modification. Toutes les instances de la table cible doivent être référencées par au moins une instance de la table source afin de respecter la contrainte d'égalité. Dans la figure 4.13, cette contrainte est vérifiée par la requête :

```
SELECT * FROM t WHERE id1,...,idk NOT IN (SELECT tid1,...,tidk FROM t1);
```

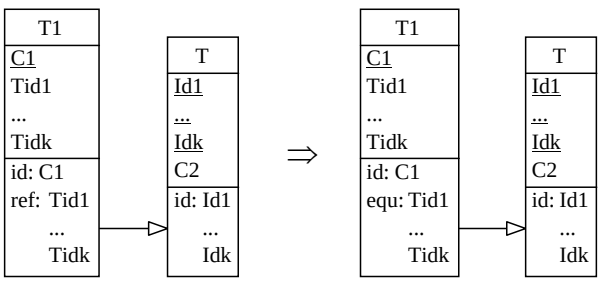


Figure 4.13 - La contrainte d'égalité est ajoutée entre *T1* et *T*.

Si la requête donne un résultat, la contrainte d'égalité est violée. Dans ce cas, deux solutions sont envisageables :

1. La base contient des instances erronées. La modification est, soit reportée jusqu'à la correction des instances illicites, soit abandonnée car elle est non pertinente.

2. La modification est réalisée mais les instances problématiques sont transférées dans une table temporaire. Cette solution est difficilement réalisable si la table contenant les instances illicites est référencée par une autre clé étrangère (figure 4.14 a) ou si elle contient une clé étrangère avec une contrainte d'égalité (figure 4.14 b). Dans ce cas, il devient impossible d'extraire les instances problématiques sans toucher à la cohérence de la base de données. Le transfert des instances illicites de T dans une table temporaire pose des problèmes pour les instances de T2 qui référencent les instances illicites de T dans la figure 4.14 a) et pour les instances de T2 référencées par les instances illicites de T dans la figure 4.14 b). Dans ces deux cas, le transfert dans une table temporaire n'est pas autorisé.
   Le script correspondant à cette solution est :

```
-- Créer la table contenant les lignes illicites
CREATE TABLE violation_t (c datatype [NOT NULL], ...);
-- Transférer les lignes concernées dans violation_t
INSERT INTO TABLE violation_t SELECT * FROM t WHERE id1,...,idk
                                                NOT IN (SELECT tid1,...,tidk FROM t1);
-- Enlever les lignes illicites de t
DELETE FROM t WHERE id1,...,idk NOT IN (SELECT tid1,...,tidk FROM t1);
```

a) T est référencée par une clé étrangère de T2.

b) T contient une clé étrangère avec une contrainte d'égalité vers T2.

Figure 4.14 - Illustration du problème posé par la solution du transfert des instances illicites dans une table temporaire.

Pour ajouter la contrainte d'égalité, trois techniques sont possibles :

1. Soit on ajoute une clause CHECK vérifiant la contrainte d'égalité dans la table référencée. Certains SGBDR n'acceptent pas de requêtes vers une autre table dans un CHECK (voir tableau annexe C.1 page 36).

```
ALTER TABLE t ADD CONSTRAINT chk_equ
  CHECK(EXISTS(SELECT * FROM t1 WHERE (t1.tid1 = t.id1 AND ... AND
                                       t1.tidk = t.idk)));
```

2. Soit on ajoute la contrainte sous la forme d'un TRIGGER déclenché à chaque insertion ou modification d'instance dans la table référencée (syntaxe Oracle 8) :

```
CREATE TRIGGER trig_equ
  BEFORE INSERT OR UPDATE OF id1,...,idk ON t
  REFERENCING NEW AS NEW OLD AS OLD
  FOR EACH ROW
  DECLARE
    x NUMBER;
    violated_constraint EXCEPTION;
  BEGIN
    SELECT COUNT(*) INTO x FROM t1
                    WHERE (tid1 = :new.id1 AND ... AND tidk = :new.idk);
    IF x = 0 THEN RAISE violated_constraint; END IF;
  END;
```

3. Soit on ajoute le code de vérification de la contrainte d'égalité dans une procédure stockée (syntaxe Oracle 8).

```
CREATE OR REPLACE PROCEDURE verify_constraint_equ IS
  DECLARE
     x NUMBER;
     violated_constraint EXCEPTION;
  BEGIN
    SELECT COUNT(*) INTO x FROM t
                   WHERE (id1,...,idk) NOT IN (SELECT tid1,...,tidk FROM t1);
    IF x = 0 THEN RAISE violated_constraint; END IF;
  END verify_constraint_equ;
```

Dans l'exemple, une contrainte d'égalité est créée entre *PRODUIT* vers *FOURNISSEUR* par le script :

```
ALTER TABLE fournisseur ADD CONSTRAINT chk_produit
  CHECK(EXISTS(SELECT * FROM produit WHERE produit.num = nfourn));
```

La création d'une contrainte d'égalité risque de mettre les programmes dans une situation autobloquante lors de l'insertion d'une instance dans la table cible. Dans l'exemple, un nouveau produit référençant un nouveau fournisseur ne peut être inséré dans la base tant que le nouveau fournisseur n'y figure pas et inversement. L'implémentation de la contrainte par un CHECK ou un TRIGGER provoque ce type de situation. Pour résoudre le problème, il faut utiliser les mécanismes de mise en veille des vérifications des contraintes pendant la création des instances et réactiver la vérification après les insertions (notion de transaction). La technique d'implémentation des STORED PROCEDURE offre plus de souplesse puisqu'il suffit de l'appeler lorsque les instances sont introduites pour vérifier la contrainte d'égalité.

b) Programmes

Les programmes dépendent structurellement de la modification. Les interfaces et les dialogues gérant le lien entre les instances de la table de référence et la table référencée doivent intégrer la nouvelle contrainte. Les instructions de modification, de destruction et d'insertion concernant les colonnes de la clé étrangère et celles de la clé primaire ou candidate référencée sont susceptibles d'être modifiées.

Une recherche, sur les instructions INSERT, DELETE et UPDATE concernant les colonnes de référence et celles référencées ainsi que les variables qui en dépendent, est nécessaire pour localiser les sections de code à modifier.

4.3.3 Ajout d'un composant à une contrainte

4.3.3.1 Niveau conceptuel

Un composant est ajouté à une contrainte de coexistence, exclusive, au-moins-un ou exactement-un dans un schéma conceptuel. Cette modification est de catégorie T^- car l'ajout d'un composant à une contrainte diminue la quantité d'informations exprimables par les spécifications. Sa signature est : (G,{A1,...,An,A}) ← modifier-Groupe(E,G,{A}).

Les préconditions sont :

- Le groupe G est une contrainte de coexistence, exclusive, au-moins-un ou exactement-un dont les composants sont facultatifs.
- Le composant (attribut ou rôle) facultatif A n'appartient pas à G.

La postcondition est :

- Le composant A appartient à G.

La modification impose que les instances du groupe vérifient la nouvelle contrainte :

– Coexistence : pour chaque instance, la valeur de A est nulle (respectivement non nulle) lorsque les valeurs des autres composants de la contrainte sont nulles (respectivement non nulles).

– Exclusivité : pour chaque instance, la valeur de A est nulle si une des valeurs des autres composants de la contrainte est non nulle.

– Au-moins-un : aucune vérification n'est nécessaire.

– Exactement-un : pour chaque instance, la valeur de A est nulle.

Dans la figure 4.15, l'attribut *Rue* est ajouté au groupe de coexistence de *REPRESENTANT* contenant déjà les attributs *Localite* et *Codepostal*. Pour une instance de *REPRESENTANT*, il faut que les valeurs des attributs *Rue*, *Localite* et *Codepostal* soient simultanément présentes ou absentes.

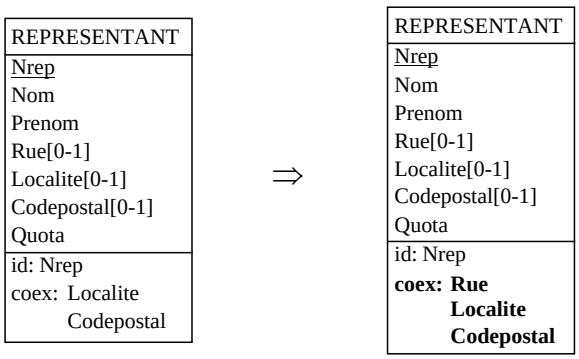


Figure 4.15 - Ajout de l'attribut *Rue* au groupe de coexistence de *REPRESENTANT*.

4.3.3.2 Niveau logique

Dans le schéma logique, un composant est ajouté à une contrainte de coexistence, exclusive, au-moins-un ou exactement-un. Cette modification est de catégorie T^- et sa signature est : (G,{C1,...,Cn,C}) ← modifier-Groupe(T,G,{C}).

Les préconditions sont :

- Le groupe G est une contrainte de coexistence, exclusive, au-moins-un ou exactement-un dont les colonnes sont facultatives.
- La colonne facultative C n'appartient pas à G.

La postcondition est :

- La colonne C appartient à G.

Comme au niveau conceptuel (point 4.3.3.1), la modification impose que les instances du groupe vérifient la nouvelle contrainte.

Dans l'exemple de la figure 4.15, la colonne *Rue* est ajoutée au groupe de coexistence de la table *REPRESENTANT* contenant les colonnes *Localite* et *Codepostal*.

4.3.3.3 Données et programmes

a) Données

Les données de la base sont dépendantes de la modification car l'ajout d'une colonne supplémentaire à une contrainte revient à définir une nouvelle contrainte sur une table. Il faut donc véri-

fier si les instances de la table sont cohérentes avec la nouvelle contrainte. Comme cela a été expliqué au niveau conceptuel (point 4.3.3.1), seul l'ajout d'une colonne à une contrainte au-moins-un ne nécessite pas de vérification. Pour la figure 4.16, les scripts de vérification sont :

```
-- Contrainte de coexistence
SELECT * FROM t WHERE (c3 IS NULL OR c4 IS NULL) AND
                      (c3 IS NOT NULL OR c4 IS NOT NULL);
-- Contrainte d'exclusion
SELECT * FROM t WHERE (c2 IS NOT NULL OR c3 IS NOT NULL) AND c4 IS NOT NULL;
-- Contrainte exactement-un
SELECT * FROM t WHERE c4 IS NOT NULL;
```

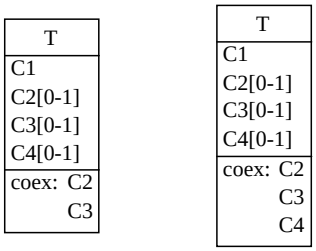


Figure 4.16 - Ajout de la colonne *C4* au groupe de coexistence de *T*.

Si la requête donne un résultat, la contrainte est violée. Soit la modification est reportée jusqu'à la correction des instances problématiques, soit ces instances sont transférées dans une table temporaire à condition qu'elles ne référencent pas des instances d'une autre table contenant une contrainte d'égalité ou qu'elles ne soient pas référencées par des instances d'une autre table. Ces situations ont déjà été analysées dans le point 4.3.2.3 (ajout d'une contrainte d'égalité).

```
-- Créer une table temporaire de même structure que la table d'origine.
CREATE TABLE t_tmp (c1 datatype [DEFAULT value NOT NULL], ...);
-- Insérer dans t_tmp les instances de t1 qui ne respectent pas
-- la contrainte de coexistence.
INSERT INTO t_tmp
  SELECT * FROM t WHERE (c3 IS NULL OR c4 IS NULL) AND
                        (c3 IS NOT NULL OR c4 IS NOT NULL);
-- Supprimer dans t les instances qui ne respectent pas la contrainte.
DELETE FROM t WHERE (c3 IS NULL OR c4 IS NULL) AND
                    (c3 IS NOT NULL OR c4 IS NOT NULL);
```

Finalement, il faut détruire la contrainte et la recréer avec tous ces composants.

```
-- Détruire la contrainte
ALTER TABLE t DROP CONSTRAINT chk_cst;
-- Créer la contrainte de coexistence :
ALTER TABLE t ADD CONSTRAINT chk_coex
  CHECK((c2 IS NOT NULL AND c3 IS NOT NULL AND c4 IS NOT NULL) OR
        (c2 IS NULL AND c3 IS NULL AND c4 IS NULL));
-- Créer la contrainte d'exclusivité :
ALTER TABLE t ADD CONSTRAINT chk_excl
  CHECK((c2 IS NOT NULL AND c2 IS NULL AND c4 IS NULL) OR
        (c3 IS NOT NULL AND c2 IS NULL AND c4 IS NULL) OR
        (c4 IS NOT NULL AND c2 IS NULL AND c3 IS NULL) OR
        (c2 IS NULL AND c3 IS NULL AND c4 IS NULL));
-- Créer la contrainte au-moins-un :
ALTER TABLE t ADD CONSTRAINT chk_atlst1
  CHECK(c2 IS NOT NULL OR c3 IS NOT NULL OR c4 IS NOT NULL);
-- Créer la contrainte exactement-un :
ALTER TABLE t ADD CONSTRAINT chk_exact1
```

```
  CHECK((c2 IS NOT NULL AND c3 IS NULL AND c4 IS NULL) OR
        (c3 IS NOT NULL AND c2 IS NULL AND c4 IS NULL)) OR
        (c4 IS NOT NULL AND c2 IS NULL AND c3 IS NULL));
```

Dans l'exemple de la figure 4.15, la colonne *Rue* est ajoutée à la contrainte de coexistence de la table *REPRESENTANT* portant déjà sur les colonnes *Codepostal* et *Localite*.

```
ALTER TABLE representant DROP CONSTRAINT chk_representant;
ALTER TABLE representant ADD CONSTRAINT chk_representant
  CHECK((localite IS NOT NULL AND codepostal IS NOT NULL AND rue IS NOT NULL) OR
        (localite IS NULL AND codepostal IS NULL AND rue IS NULL));
```

a) Programmes

Les programmes dépendent structurellement de la modification. Au niveau des requêtes de sélections, il n'y a rien à changer. Par contre, les interfaces ou les dialogues gérant la saisie des composants de la contrainte doivent intégrer la nouvelle contrainte.

Une recherche est nécessaire, sur les instructions de modification, de destruction et d'insertion portant sur les colonnes appartenant à la contrainte ainsi que sur les variables qui en dépendent, dans le but de localiser les sections de code à modifier.

4.4 Modèle E/A riche → Modèle E/A de base

Après une introduction sur les concepts des modèles ainsi que les modifications associées (point 4.4.1), deux modifications sont développées pour illustrer les mécanismes de l'analyse et les solutions proposées. Il s'agit du passage d'une instanciation à une transformation par instance (point 4.4.2) et de la transformation d'un type d'associations fonctionnel en un type d'associations complexe (point 4.4.3).

4.4.1 Introduction

Nous allons définir les contraintes que chaque spécification du niveau conceptuel doit respecter (point 4.4.1.1). La façon d'appréhender les modifications, différente des deux sections précédentes, est décrite dans le point 4.4.1.2. Le tableau du point 4.4.1.3 récapitule les modifications analysées dans l'annexe C.4.

4.4.1.1 Propriétés des spécifications

Les spécifications doivent respecter certaines propriétés avant et après chaque modification. Ci-dessous, les contraintes du niveau conceptuel sont détaillées. Les contraintes données aux points 4.2.1.1 (noyau E/A → noyau relationnel) et 4.3.1.1 (modèle E/A de base → modèle relationnel riche) sont toujours valables. Elles sont complétées par les contraintes :

- Niveau conceptuel :
  - les attributs décomposables et multivalués sont autorisés;
  - les types d'associations complexes (avec attributs, n-aires ou plusieurs-à-plusieurs) sont autorisés;
  - les identifiants d'attributs décomposables et multivalués ainsi que les identifiants de types d'associations sont autorisés;
  - les rôles multi-TE sont autorisés;
  - les relations IS-A sont autorisées.
- Niveau logique : pas de nouvelles contraintes.
- Niveau physique : pas de nouvelles contraintes.

4.4.1.2 Etude des modifications

Pour chaque type de modifications, nous allons voir comment on peut passer du modèle E/A riche au modèle E/A de base. Ce passage s'accomplira à l'aide des transformations décrites au point 3.2.3.

Chaque structure complexe peut se rapporter à une ou plusieurs structures simples analysées dans les sections précédentes. Le travail consiste à relever ces structures simples et à mentionner leur référence.

Dans certains cas, une modification apportée à un schéma conceptuel peut nécessiter plusieurs transformations. Par exemple, si on ajoute un attribut multivalué à un type d'associations fonctionnel, deux structures complexes sont introduites : un type d'associations avec attribut et un attribut multivalué. Au niveau du modèle E/A de base, la modification nécessite l'application de deux transformations (d'un type d'associations complexe en un type d'entités et d'un attribut en un type d'entités).

L'analyse des modifications au niveau du modèle E/A riche va être décomposée en cinq parties : les modifications relatives aux attributs, aux types d'entités, aux types d'associations, aux rôles multi-TE et aux relations IS-A.

Pour chaque partie, nous allons déterminer les modifications qui apportent, enlèvent ou modifient des structures complexes. Pour chacune de ces modifications, nous allons décrire la modification au niveau des modèles E/A riche et de base. Les modifications au niveau du modèle E/A de base sont basées sur les transformations qui sont applicables sur la structure complexe. Il est évident que l'application de ces transformations doit respecter certaines conditions (voir point 3.2.3). Finalement, lorsque cela s'avèrera nécessaire (changements de transformation), on augmentera les scripts de conversion de données vus dans les sections précédentes.

4.4.1.3 Tableau récapitulatif des modifications

Le tableau 4.9 présente les modifications classées par niveau d'abstraction avec la référence de la page correspondant dans l'annexe C.4. Le tableau se lit ligne par ligne de manière à suivre une modification à travers tous les niveaux du processus de conception.

Niveau conceptuel		**Données et programmes (C.4.4)**
Modèle E/A riche (C.4.2)	**Modèle E/A de base (C.4.3)**	
Attributs décomposables ou multi-valués (C.4.2.1)	**Attributs décomposables ou multi-valués (C.4.3.1)**	
Création (p. 121)	Création (p. 153)	
Destruction (p. 122)	Destruction (p. 155)	
Renommage (p. 122)	Renommage (p. 155)	
Extension de domaine (p. 123)	Extension de domaine (p. 155)	
Restriction de domaine (p. 124)	Restriction de domaine (p. 156)	
Changement de type (p. 124)	Changement de type (p. 156)	
Augmentation de la cardinalité minimum (p. 125)	Augmentation de la cardinalité minimum (p. 156)	
Diminution de la cardinalité minimum (p. 126)	Diminution de la cardinalité minimum (p. 157)	
Augmentation de la cardinalité maximum (p. 127)	Augmentation de la cardinalité maximum (p. 157)	
Diminution de la cardinalité maximum (p. 128)	Diminution de la cardinalité maximum (p. 157)	
Changement de la nature d'un attribut (p. 129)	Changement de transformation (p. 157)	Changement de transformation sur les colonnes (p. 198)
Types d'associations complexes (C.4.2.2)	**Types d'associations complexes (C.4.3.2)**	
Création (p. 139)	Création (p. 190)	
Destruction (p. 140)	Destruction (p. 190)	
Renommage (p. 141)	Renommage (p. 190)	
Création d'un rôle (p. 141)	Création d'un rôle (p. 191)	
Destruction d'un rôle (p. 142)	Destruction d'un rôle (p. 191)	
Renommage d'un rôle (p. 143)	Renommage d'un rôle (p. 191)	
Modification des cardinalités d'un rôle (p. 144)	Modification des cardinalités d'un rôle (p. 191)	
Changement de la nature d'un type d'associations (p. 144)	Changement de transformation (p. 191)	Changement de transformation sur les clés étrangères (p. 222)
Rôles multi-TE (C.4.2.3)	**Rôles multi-TE (C.4.3.3)**	
Création (p. 147)	Création (p. 194)	
Destruction (p. 147)	Destruction (p. 194)	
Renommage (p. 148)	Renommage (p. 194)	
Ajout d'un type d'entités (p. 148)	Ajout d'un type d'entités (p. 195)	

Tableau 4.9 - Tableau récapitulatif de modifications conceptuelles sur le modèle E/A riche et de base ainsi que celles sur le niveau opérationnel.

Niveau conceptuel		Données et programmes (C.4.4)
Modèle E/A riche (C.4.2)	**Modèle E/A de base (C.4.3)**	
Retrait d'un type d'entités (p. 149)	Retrait d'un type d'entités (p. 195)	
Changement de la nature d'un rôle (p. 150)	Changement de transformation (p. 195)	Changement de transformation sur les colonnes de référence (p. 224)
Relations IS-A (C.4.2.4)	**Relations IS-A (C.4.3.4)**	
Création (p. 151)	Création (p. 197)	
Destruction (p. 152)	Destruction (p. 197)	

Tableau 4.9 - Tableau récapitulatif de modifications conceptuelles sur le modèle E/A riche et de base ainsi que celles sur le niveau opérationnel.

4.4.2 Passage d'une transformation d'un attribut en une série d'attributs monovalués à une transformation en type d'entités par instance

4.4.2.1 Niveau conceptuel

a) Modèle E/A riche

Dans le modèle E/A riche, on considère qu'il existe un attribut multivalué (figure 4.17). Dans le modèle E/A de base, cette structure est transformée pour respecter les conditions du modèle.

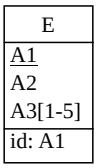


Figure 4.17 - Le type d'entités *E* possède un attribut multivalué *A3*.

b) Modèle E/A de base

Supposons que la modification consiste à changer de transformation pour un attribut multivalué. Auparavant, un attribut multivalué était transformé en une série d'attributs monovalués. La modification consiste à le transformer en un type d'entités (représentation par instance). Une telle modification est utilisée, par exemple, pour augmenter la flexibilité des structures. Le concepteur choisit la représentation en type d'entités qui permet d'augmenter la cardinalité maximum de l'attribut multivalué sans devoir propager la modification au niveau des données et des programmes.

La modification (figure 4.18) a la signature : (EA3,R) ← Serie-Att-en-TE-inst(E,{A31,A32,A33,A34,A35}). Elle est de catégorie $T^{=}$ car elle utilise une transformation à sémantique constante.

Les préconditions sont :

- Les attributs A31, ..., A35 sont monovalués, de mêmes types et longueurs, de cardinalités maximum 1.
- Les attributs A31, ..., A35 appartiennent au même parent (E).

Les postconditions sont :

- Création d'un type d'entités EA3
- Création d'un type d'associations R reliant E et EA3. le rôle joué par E est de cardinalité [1-5] où la cardinalité minimum (respectivement maximum) est la somme des cardinalités

minimum (respectivement maximum) des attributs A31, A32, A33, A34 et A35. Le rôle joué par EA3 est de cardinalité [1-1].

- Création d'un attribut A3 dans EA3 dont le type et la longueur sont identiques à A31. Sa cardinalité est [1-1].
- Création d'un identifiant primaire dans EA3 contenant l'attribut A3 et le rôle R.E.
- Suppression des attributs A31, A32, A33, A34 et A35 de E.

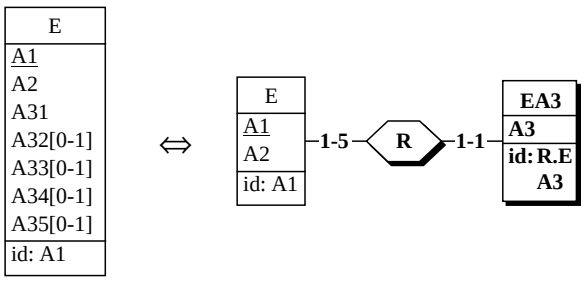


Figure 4.18 - Changement de transformation pour un attribut multivalué. La transformation en une série d'attributs monovalués est remplacée par la transformation en type d'entités (représentation par instance).

Pour chaque instance de E et pour chaque valeur non nulle des attributs (A31, A32, A33, A34 et A35), une instance de EA3 et une instance de R sont créées.

4.4.2.2 Niveau logique

Au niveau logique, les modifications implémentant les modifications du niveau conceptuel ont déjà été analysées dans la section 4.2 et l'annexe C.2. Il s'agit de la création d'une table et de ses colonnes, de la création d'une clé étrangère entre les tables E et EA3 ainsi que des suppressions des colonnes A31, A32, A33, A34 et A35. La figure 4.19 présente au niveau logique les modifications de la figure 4.18.

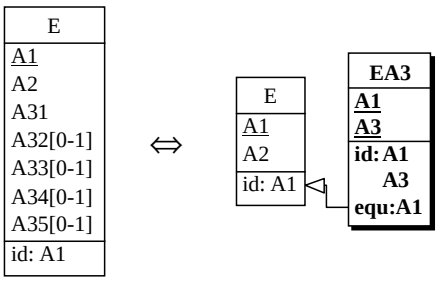


Figure 4.19 - Les colonnes A31, A32, A33, A34 et A35 sont représentées par une table EA3 liée par une clé étrangère à la table E.

Pour chaque instance de la table E, une instance de EA3 est créée pour chaque valeur non nulle des colonnes A31, A32, A33, A34 et A35 dans laquelle la colonne EA3.A3 prend la valeur de la colonne considérée dans E et EA3.A1 prend la valeur de E.A1.

4.4.2.3 Données et programmes

a) Données

Les données de la base sont indépendantes de la modification. Pour propager les modifications vers les données et structures de données, les colonnes A31, A32, A33, A34 et A35 sont transformées en une table. Comme il s'agit d'une transformation à sémantique constante, aucune vérification de contrainte n'est nécessaire. Le script de conversion des structures et des données est :

```
-- Création de EA3 avec ses colonnes et ses contraintes (annexe C.2.5.1 a).
CREATE TABLE EA3 (A1 datatype NOT NULL,A3 datatype NOT NULL,
                  CONSTRAINT id_er PRIMARY KEY (A1,A3));
ALTER TABLE EA3 ADD CONSTRAINT fk_E FOREIGN KEY (A1) REFERENCES E (A1);
ALTER TABLE E ADD CONSTRAINT chk_equ
  CHECK(EXISTS(SELECT * FROM EA3 WHERE (EA3.A1 = E.A1)));
-- Transfert des instances de E.A31, E.A32, E.A33, E.A34 et E.A35 dans EA3.
INSERT INTO EA3 (A1,A3) SELECT A1,A31 from E;
INSERT INTO EA3 (A1,A3) SELECT A1,A32 from E WHERE A31 IS NOT NULL;
INSERT INTO EA3 (A1,A3) SELECT A1,A33 from E WHERE A31 IS NOT NULL;
INSERT INTO EA3 (A1,A3) SELECT A1,A34 from E WHERE A31 IS NOT NULL;
INSERT INTO EA3 (A1,A3) SELECT A1,A35 from E WHERE A31 IS NOT NULL;
-- Destruction des colonnes E.A31, E.A32, E.A33, E.A34 et E.A35 (annexe C.2.5.2 b).
ALTER TABLE E DROP A31 CASCADE;
ALTER TABLE E DROP A32 CASCADE;
ALTER TABLE E DROP A33 CASCADE;
ALTER TABLE E DROP A34 CASCADE;
ALTER TABLE E DROP A35 CASCADE;
```

b) Programmes

Les programmes sont structurellement dépendants de la modification. La transformation de colonnes en une table nécessite la modification des requêtes accédant aux colonnes transformées. Il faut une jointure supplémentaire pour accéder aux valeurs de la colonne A3 de EA3. Au niveau des interfaces et des dialogues, il faut gérer la nouvelle clé étrangère et la table EA3 pour toutes les modifications ou consultations d'informations relatives à la table E.

Une recherche sur la table E, les colonnes A31, A32, A33, A34 et A35 ainsi que sur toutes les variables qui en dépendent permet de localiser les sections de codes suceptibles d'être modifiées.

4.4.3 Modification d'un type d'associations fonctionnel en type d'associations complexe

4.4.3.1 Niveau conceptuel

a) Modèle E/A riche

Un type d'associations fonctionnel (binaire, un-à-plusieurs ou un-à-un, sans attribut) peut devenir un type d'associations complexe soit par la création d'attributs (figure 4.20 a), soit par la création de rôles (figure 4.20 b) ou soit par l'augmentation d'une cardinalité maximum d'un de ses rôles (figure 4.20 c). Cette modification est de catégorie T^+ car, dans les trois cas, la sémantique du schéma augmente. Sa signature correspond aux signatures des modifications altérant le type d'associations : C1 ← creer-Att(R,nc1), R.E3 ← creer-Role(R,nre3,{E3},[i-j]) ou R.E2 ← modifier-Role(R,R.E2,[0-N]).

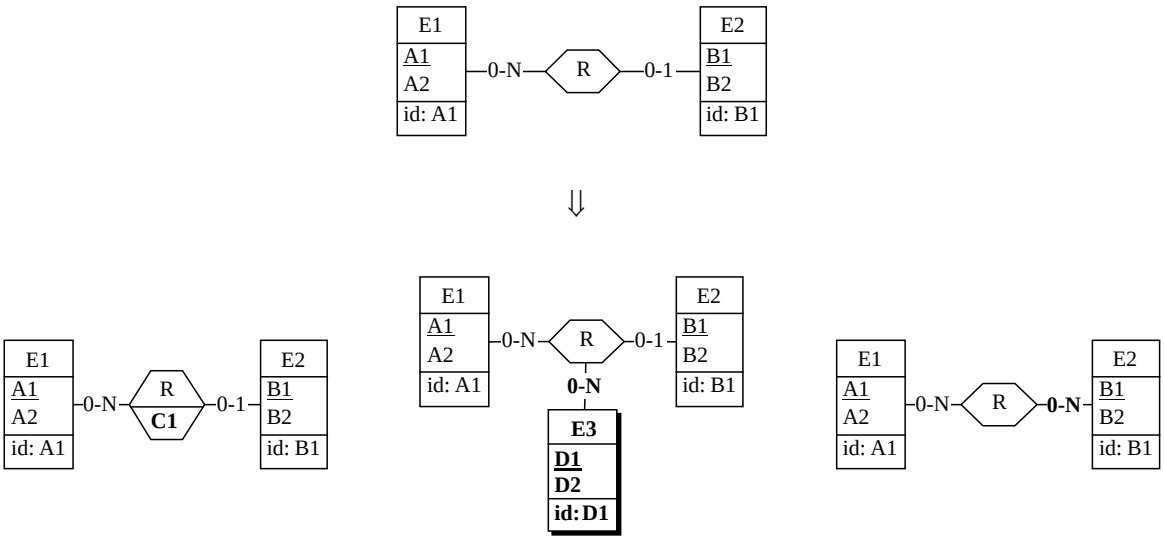
La précondition est :

- R est un type d'associations fonctionnel sans attribut dont les cardinalités des rôles sont 0-1/0-1, 0-1/1-1, 0-1/0-N, 0-1/1-N, 1-1/1-1, 1-1/0-N ou 1-1/1-N.

La postcondition est :

- R devient un type d'associations complexe soit parce qu'il possède un nouvel attribut, soit parce qu'il est plus que binaire ou soit parce qu'il devient plusieurs-à-plusieurs.

Les instances de la base sont dépendantes de la modification pour la création d'attributs obligatoires ou de rôles. Dans la cas de la création d'attributs obligatoires dans le type d'associations, ils doivent avoir une valeur pour chaque instance du type d'associations. Les attributs facultatifs sont créés avec la valeur nulle par défaut. Chaque instance du type d'associations doit également posséder des valeurs pour les nouveaux rôles créés. Dans le cas de l'augmentation de la cardinalité maximum d'un rôle, la cohérence de la base est toujours vérifiée puisque le type d'entités jouant ce rôle vérifie la contrainte de cardinalité minimum.



a) Création de l'attribut C1 dans le type d'associations R. **b) Création du rôle R.E3 entre R et E3.** **c) Augmentation de la cardinalité maximum du rôle R.E2.**

Figure 4.20 - Un type d'associations fonctionnel devient complexe.

Dans la figure 4.21, le type d'associations *fournit* reçoit un attribut *Date* et les cardinalités du rôle joué par *PRODUIT* dans *fournit* deviennent 1-N. Les instances de *fournit* sont indépendantes de la modification de la cardinalité du rôle. Par contre, le nouvel attribut obligatoire *Date* doit avoir une valeur pour chaque instance de *R*.

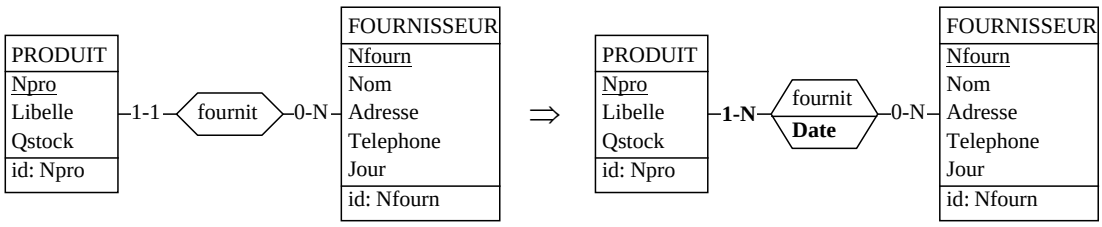


Figure 4.21 - Création de l'attribut *Date* dans le type d'associations *fournit* et augmentation de la cardinalité maximum du rôle joué par *PRODUIT* dans *fournit*.

b) Modèle E/A de base

L'objectif est de retraduire les modifications du point précédent en modifications déjà analysées dans les sections précédentes. Premièrement, la modification originale qui consiste à modifier un type d'associations fonctionnel en un type d'associations complexe, engendre la transformation du type d'associations fonctionnel en un type d'entités (partie gauche de la figure 4.22). Cette transformation, décrite formellement dans le chapitre 3, est de catégorie $T^{=}$ car elle conserve la sémantique du schéma. Sa signature est : ER ← TA-en-TE(R).

Deuxièmement, les modifications décrites dans le modèle E/A riche sont traduites en modifications relatives au modèle E/A de base. La création d'un attribut d'un type d'associations (figure 4.20 a) devient la création d'un attribut du type d'entités ER (figure 4.22 a) qui est une modification de catégorie T^{+} dont la signature est : C1 ← creer-Att(ER,nc1). La création d'un rôle d'un type d'associations (figure 4.20 b) devient la création d'un type d'associations entre deux types d'entités (figure 4.22 b) qui est une modification de catégorie T^{+} dont la signature est : R3 ← creer-TA(S,nr3). L'augmentation de la cardinalité maximum d'un rôle d'un type d'associations complexe (figure 4.20 c) devient l'augmentation de la cardinalité maximum d'un rôle d'un type d'associations fonctionnel (figure 4.22 c) qui est une modification de catégorie T^{+} dont la signature est : R2.E2 ← modifier-Role(R2,R2.E2,[0-N]). Par conséquent, les modifications complexes du modèle E/A riche sont ramenées à des modifications déjà analysées dans le modèle E/A de base.

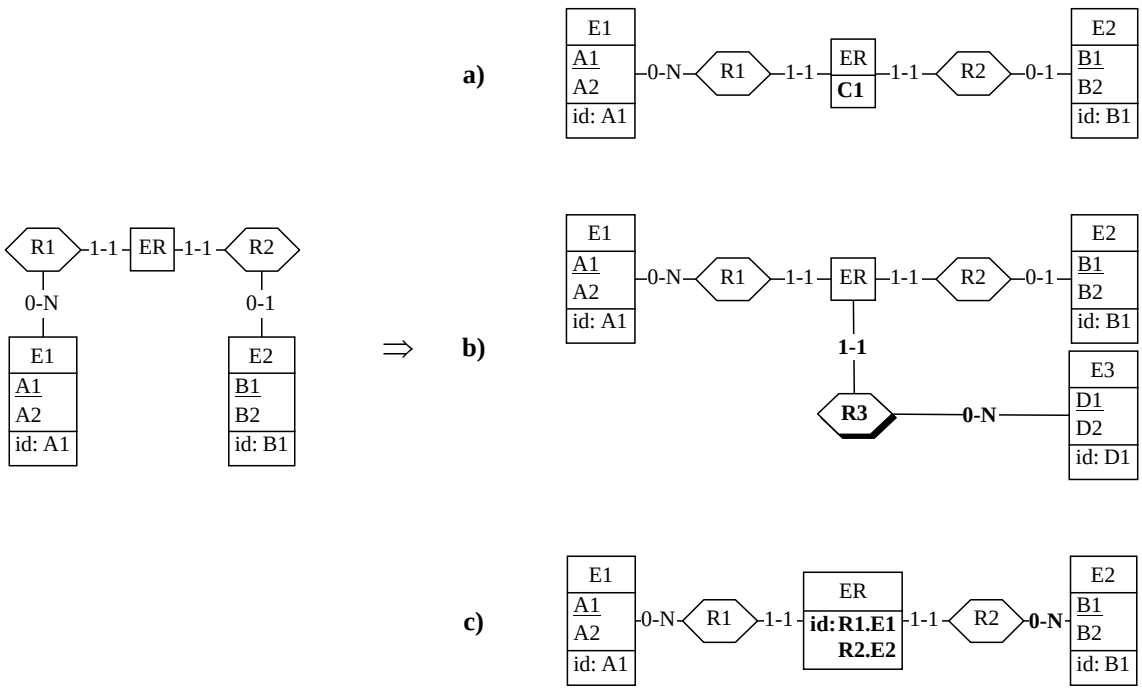


Figure 4.22 - Modification d'un type d'entités par la création de l'attribut *C1* dans *ER* (a), par la création du type d'associations fonctionnel *R3* entre *ER* et *E3* (b) ou par l'augmentation de la cardinalité maximum du rôle *R2.E2* (c).

Dans la première étape, les instances de R sont tranférées dans celles de ER sans perte d'information (transformation préservant la sémantique). Dans la deuxième étape, des valeurs doivent être attribuées à l'attribut obligatoire C1 et des instances du nouveau type d'associations R3 sont créées pour chaque instance de ER.

Dans la figure 4.23, le type d'associations *fournit* est transformé en type d'entités *FOURNITURE* qui possède un attribut *Date* et les cardinalités du rôle joué par *PRODUIT* dans *est-fournit* sont 1-

N. Pour chaque instance de *fournit*, des instances de *FOURNITURE*, de *pf* et de *ff* sont créées. Le nouvel attribut obligatoire *Date* doit avoir une valeur pour chaque instance de *FOURNITURE*.

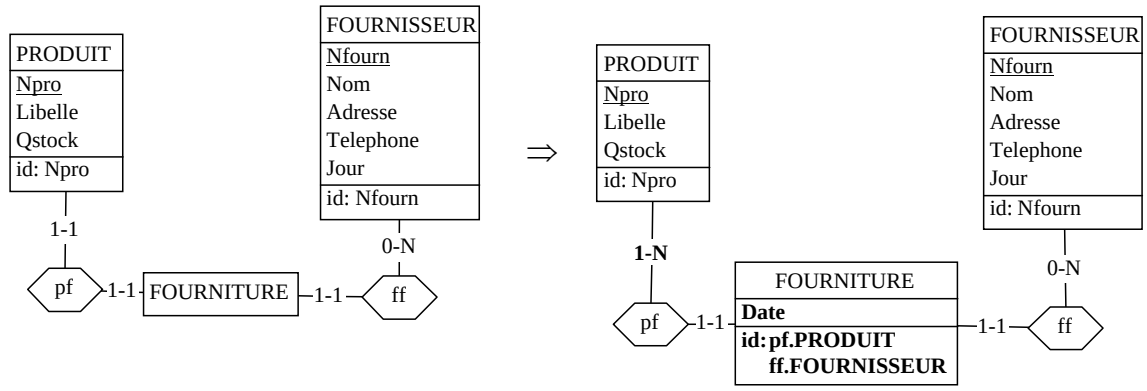
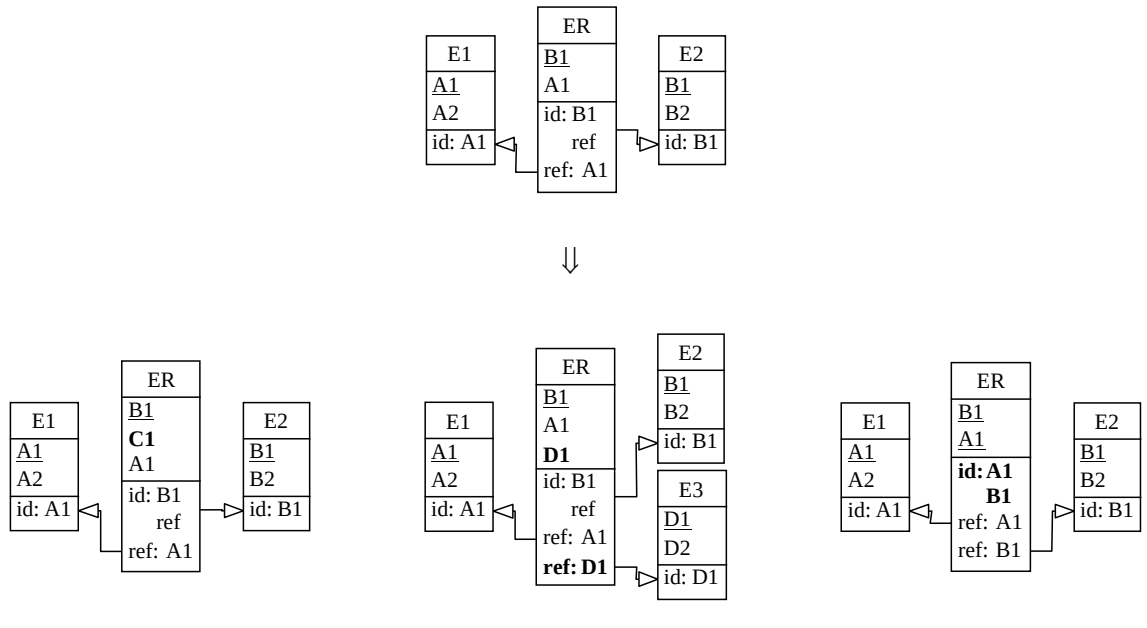


Figure 4.23 - Transformation du type d'associations *fournit* en type d'entités *FOURNITURE* avec la modification de la cardinalité de *pf.PRODUIT* et la création de l'attribut *Date*.

4.4.3.2 Niveau logique

Puisque les modifications du modèle E/A riche sont traduites en modifications du modèle E/A de base, elles ont déjà été analysées pour le niveau logique dans la section 4.2 et l'annexe C.2. La figure 4.24 présente au niveau logique les modifications de la figure 4.22.



a) Création de la colonne C1 dans ER.

b) Création d'une clé étrangère vers la table E3.

c) Ajout de la colonne A1 à la clé primaire de ER.

Figure 4.24 - Création d'une colonne, d'une clé étrangère et ajout d'un composant à une clé primaire.

Dans la figure 4.25, le type d'entités *FOURNITURE* reçoit une colonne NOT NULL supplémentaire *Date* et la colonne *Nfourn* est ajoutée à la clé primaire de *FOURNITURE*. La nouvelle colonne *Date* doit avoir une valeur pour chaque instance de *FOURNITURE*. Les instances de *FOURNITURE* vérifient la nouvelle contrainte clé primaire.

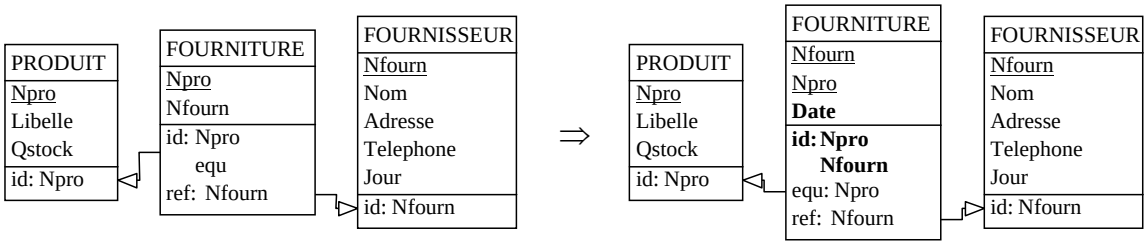


Figure 4.25 - Création de la colonne *Date* dans *FOURNITURE* et ajout de la colonne *Nfourn* dans la clé primaire de *FOURNITURE.*

4.4.3.3 Données et programmes

a) Données

Pour propager les modifications vers les données et structures de données, nous allons procéder en deux temps comme dans les niveaux conceptuel et logique.

Premièrement, la clé étrangère est transformée en une table. Comme il s'agit d'une transformation à sémantique constante, aucune vérification préalable de contrainte n'est nécessaire. Le script de conversion est :

```
-- Création de ER avec ses colonnes et ses contraintes (annexe C.2.5.1 a).
CREATE TABLE ER (B1 datatype NOT NULL,A1 datatype NOT NULL,
                 CONSTRAINT id_er PRIMARY KEY (B1));
ALTER TABLE ER ADD CONSTRAINT fk_E1 FOREIGN KEY (A1) REFERENCES E1 (A1);
ALTER TABLE ER ADD CONSTRAINT fk_E2 FOREIGN KEY (B1) REFERENCES E2 (B1);
-- Transfert des instances de E2.A1 et E2.B1 dans ER
INSERT INTO ER (B1,A1) SELECT B1,A1 from E2;
-- Destruction de la colonne E2.A1 et de sa clé étrangère (annexe C.2.5.3 b).
ALTER TABLE E2 DROP A1 CASCADE;
```

Deuxièmement, les modifications par rapport au schéma original sont introduites. Il s'agit de la création de la colonne C1 (figure 4.24 a), de la création de la colonne de référence D1 ainsi que la clé étrangère qui y correspond (figure 4.24 b) et de l'ajout de la colonne A1 à la clé primaire de ER (figure 4.24 c). Toutes ces modifications ont déjà été étudiées dans le modèle relationnel de base. Nous en donnerons simplement le script de conversion :

```
-- Création de la colonne C1 dans la table ER (annexe C.2.5.2 a).
ALTER TABLE ER ADD C1 datatype DEFAULT value;
-- Création de la colonne de référence D1 et de sa clé étrangère (annexe C.2.5.3 a).
ALTER TABLE ER ADD D1 datatype DEFAULT value;
+ Remplissage de valeurs pour la colonne créée
ALTER TABLE ER ADD CONSTRAINT fk_e3 FOREIGN KEY (D1) REFERENCES E3 (D1);
-- Ajout de la colonne A1 à la clé primaire de ER (annexe C.2.5.4 e).
ALTER TABLE ER DROP CONSTRAINT id_er;
ALTER TABLE ER ADD CONSTRAINT id_er PRIMARY KEY (A1,B1);
```

Pour optimiser le script, il est possible de créer la nouvelle clé primaire de la table ER au moment de la création de cette table. Ainsi, on supprime les deux dernières instructions du script qui ajoute la colonne A1 à la clé primaire de ER.

Dans l'exemple, la clé étrangère de *PRODUIT* vers *FOURNISSEUR* est transformée en une table *FOURNITURE*. La colonne *Date* est créée dans cette nouvelle table dont la clé primaire est composée des colonnes *Nfourn* et *Npro*.

```
-- Création de la table FOURNITURE avec ses colonnes et ses contraintes.
```

```
CREATE TABLE FOURNITURE (Nfourn integer(6) NOT NULL,Npro integer(6) NOT NULL,
                         CONSTRAINT id_fourniture PRIMARY KEY (Npro,Nfourn));
ALTER TABLE FOURNITURE ADD CONSTRAINT fk_produit
  FOREIGN KEY (Npro) REFERENCES PRODUIT (Npro);
ALTER TABLE FOURNITURE ADD CONSTRAINT fk_fournisseur
  FOREIGN KEY (Nfourn) REFERENCES PRODUIT (Nfourn);
-- Transfert des instances de PRODUIT.Nfourn et PRODUIT.Npro dans FOURNITURE.
INSERT INTO FOURNITURE (Nfourn,Npro) SELECT Nfourn,Npro from PRODUIT;
-- Destruction de la colonne de référence PRODUIT.Nfourn et de sa clé étrangère.
ALTER TABLE PRODUIT DROP Nfourn CASCADE;
-- Création de la colonne Date dans la table FOURNITURE.
ALTER TABLE FOURNITURE ADD Date CHAR(12) DEFAULT value NOT NULL;
```

b) Programmes

Les programmes sont structurellement dépendants des modifications. La transformation d'une clé étrangère en table nécessite la modification des requêtes imbriquées et des jointures entre E1 et E2. Au niveau des interfaces et des dialogues, la gestion du lien entre les deux tables doit également être remaniée. Pour les modifications apportées à la nouvelle table créée, nous renvoyons le lecteur à l'annexe C.2.5 pour des informations complémentaires.

Une recherche sur la table et les colonnes de référence de la clé étrangère transformée ainsi que sur toutes les variables qui en dépendent permet de localiser les sections de codes susceptibles d'être modifiées.

«C'est pas le tout d'avoir des bagages,
encore faut-il savoir où les poser.»

Coluche

CHAPITRE 5

PROPAGATION DES MODIFICATIONS DES SPÉCIFICATIONS

Le problème de la propagation est d'autant plus complexe qu'il peut y avoir plusieurs contextes d'évolution. En effet, idéalement, une application de base de données s'inscrit dans un contexte où les trois niveaux sont présents et documentés. Cette hypothèse est généralement irréaliste. Certains niveaux peuvent être absents ou incomplets et, dans les cas les plus graves, seuls les programmes et les données sont disponibles. Pour pallier à ces lacunes, il faut d'abord reconstruire la documentation d'une application grâce à des techniques de rétro-ingénierie.

Dans ce chapitre, le problème de la reconstruction des spécifications est brièvement présenté (section 5.2). Ensuite, trois stratégies de référence répondant chacune à des problèmes de modifications des spécifications à un niveau d'abstraction seront analysées (sections 5.3 à 5.5). L'étude de ces stratégies est limitée au contrôle de la propagation des modifications vers les autres niveaux. Ces trois stratégies sont basées sur l'hypothèse selon laquelle les spécifications des différents niveaux de la modélisation sont disponibles.

Les sections 5.6 et 5.7 approfondiront la propagation des modifications au niveau opérationnel (c'est-à-dire la conversion des données et structures de données et la modification des programmes).

5.1 Introduction

Dans le cadre de la modélisation classique définie selon trois niveaux d'abstraction, le problème de l'évolution a été décomposé en quatre catégories de problèmes (point 2.2.3). Si les besoins changent, il faut traduire ces changements en modifications des spécifications. Au niveau opérationnel, les structures de données doivent être restructurées pour coller aux nouvelles spécifications, les données doivent être converties et les programmes doivent être réécrits en partie pour être conformes aux nouvelles données. Des quatre catégories de problèmes répertoriées, quatre stratégies se dégagent :

1. La modification des besoins est traduite par des changements dans le schéma du niveau conceptuel. Le problème consiste à propager les modifications en aval vers les schémas logique et physique, les données et les programmes.
2. Les changements dans les besoins sont directement traduits en modifications relatives au schéma du niveau logique. Le problème est la propagation des modifications en amont pour reconstruire les spécifications du niveau conceptuel. La propagation en aval vers le schéma physique, les instances et les programmes relève de la première stratégie.
3. Les changements des besoins sont traduits par des modifications du schéma du niveau physique pour des raisons variées. Les modifications du schéma physique doivent être propagées en amont vers les schémas logique et conceptuel. Les données et les programmes doivent être changés pour être conformes aux nouvelles spécifications.
4. Dans de nombreux cas, les seules informations disponibles sont les scripts de création des structures de la base, les données et les programmes d'applications. Les spécifications des différents niveaux d'abstraction sont manquantes. La quatrième stratégie concerne la reconstruction des spécifications avant de faire évoluer le système (de manière à retomber sur une des trois premières stratégies).

Les trois premières stratégies (sections 5.3, 5.4 et 5.5) concernent les modifications des spécifications d'un niveau d'abstraction. Elles supposent que toutes les spécifications sont disponibles quel que soit le niveau considéré. La quatrième stratégie (section 5.2) reconstruit les spécifications incomplètes ou simplement absentes avant de commencer le processus d'évolution.

Avant d'analyser les stratégies déjà présentées dans [Hic98] et [Hic99], il convient de préciser l'approche disciplinée qui a été choisie. Elle se base sur deux hypothèses fondamentales dans le but de simplifier le problème déjà suffisamment complexe de la propagation des modifications :

1. Si le concepteur décide de modifier les spécifications de sa base, il doit choisir le niveau d'intervention adéquat en fonction des modifications à réaliser. Pour ce faire, nous avons déterminé, dans les stratégies de propagation, les modifications applicables à chaque niveau d'abstraction. Cette limitation n'est pas arbitraire, elle tient compte des changements de besoins à l'origine du problème d'évolution. On exclut, par exemple, la façon de procéder des ingénieurs de maintenance qui décident d'appliquer toutes leurs modifications au niveau physique sans souci de les propager aux niveaux supérieurs. Bien que contraignante, cette hypothèse de travail permet de garder une documentation cohérente du processus de conception. Il est clair que le choix des transformations applicables à chaque niveau d'abstraction peut être revu en fonction des besoins propres à chaque organisation.
2. On adopte également un processus d'évolution linéaire sans hypothèse en n'excluant pas pour autant le développement de plusieurs branches. Dans la réalité, lorsque le concepteur modifie un schéma, il teste différentes hypothèses, procède par essais et erreurs. Au niveau de l'historique, ce comportement se matérialise dans le développement de plusieurs branches. Comme on l'a vu au point 3.3.4.1, il est possible d'extraire un historique linéaire en ne gardant que les branches correspondant à des hypothèses retenues. Pour simplifier la démarche, les stratégies de propagation travaillent sur des historiques linéaires.

5.2 Reconstruction des spécifications

La rétro-ingénierie d'une application est un processus d'analyse de la version opérationnelle de l'application qui vise à en reconstruire les spécifications. Ce processus est d'autant plus complexe que l'application est ancienne, mal structurée et non documentée. Actuellement, on constate que beaucoup d'applications ne sont pas documentées et, par conséquent, que les spécifications qui sont à la base de la démarche d'évolution présentée dans ce chapitre sont manquantes. Bien qu'une étude approfondie de la rétro-ingénierie sorte du cadre de ce travail, il est important de présenter une démarche de rétro-ingénierie qui s'inscrit dans le cadre méthodologique défini au chapitre 3. La démarche proposée est décomposée en deux phases principales : l'extraction des structures de données (point 5.2.1) et la conceptualisation (point 5.2.2). Une décomposition similaire est proposée dans [Sab92]. La figure 5.1 décrit la démarche de rétro-ingénierie. Les schémas logique et physique sont construits lors de la phase d'extraction des structures de données. Le schéma conceptuel est le résultat obtenu lors de la phase de conceptualisation. Le point 5.2.3 recentre la démarche générique de rétro-ingénierie de manière à l'intégrer dans le cadre méthodologique de l'analyse du problème de l'évolution des bases de données. Une analyse plus approfondie de la démarche est présentée dans [Hai93] et [Hai97].

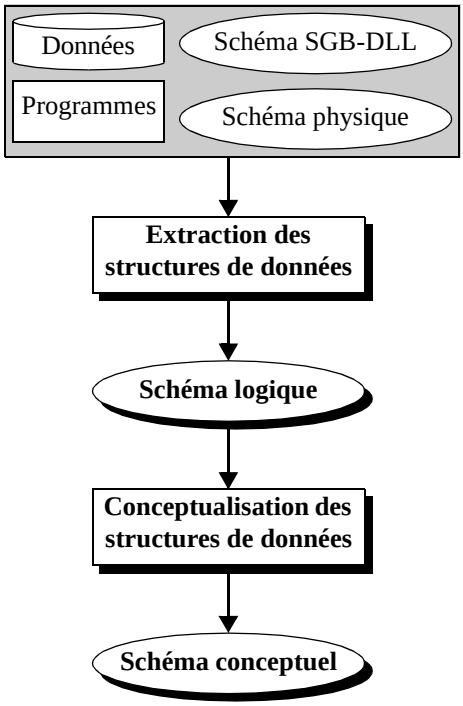


Figure 5.1 - Méthode générique de rétro-ingénierie de bases de données.

5.2.1 Extraction des structures de données

Cette première phase a pour objet la reconstruction du schéma logique complet selon le modèle du SGD, y compris toutes les contraintes et les structures de données non explicitement déclarées. Certains systèmes (comme les SGBDR) offrent, sous la forme de code déclaratif, une description du schéma global des données (communément appelée code DDL). Bien que des informations essentielles manquent dans ce schéma, il constitue un excellent point de départ qui devra être enrichi et affiné par une analyse plus approfondie d'autres composants de l'application (vues, code procédural, contenu de la base de données, écrans de saisie, fragments de documentation, exécution des programmes, ...).

Les processus principaux de l'extraction des structures de données (figure 5.2) sont :

- L'analyse de code DDL :
  L'analyse des instructions de déclaration des structures de données produit les schémas physiques bruts. Ces instructions peuvent aussi bien être dans des programmes d'applications que dans des dictionnaires de données (tels que les catalogues système dans les SGBDR).
- L'intégration physique :
  Après l'analyse du code DDL, il est plus que probable que l'analyste dispose de plusieurs schémas extraits (par exemple, le schéma des tables et celui des vues dans un SGBDR). Les spécifications des schémas physiques bruts sont intégrées dans le schéma physique augmenté.
- Le raffinement du schéma physique :
  Le raffinement de schéma est une tâche complexe qui consiste à extraire les structures et contraintes dites *implicites*, c'est-à-dire celles qui n'ont pas été traduites explicitement dans le code déclaratif, mais qui ont été exprimées sous une forme procédurale (procédures, scripts, écrans, "triggers", "stored procedures", ...) ou qui ont été ignorées. Seule une analyse méticuleuse de toutes les sources d'information disponibles permet de retrouver les spécifications. Le schéma physique complet est obtenu lors de cette tâche.
- Le nettoyage du schéma physique :
  Les constructions techniques comme les index, les espaces de stockage ne sont plus nécessaires et sont supprimées pour obtenir le schéma logique.

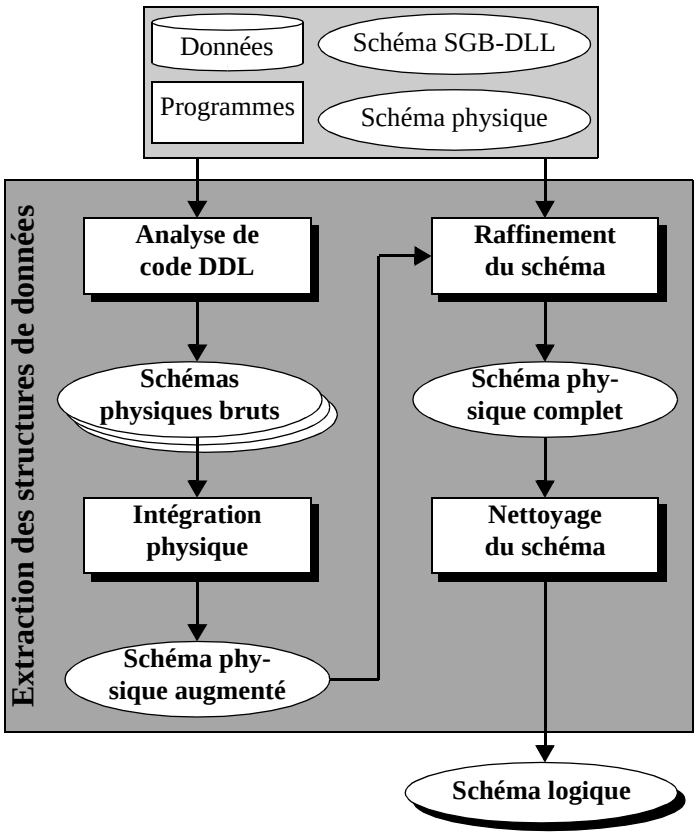


Figure 5.2 - Architecture de la phase d'extraction des structures de données.

5.2.2 Conceptualisation des structures de données

La deuxième phase de la démarche s'occupe de l'interprétation conceptuelle du schéma logique. Cela consiste à détecter et transformer les structures non conceptuelles comme la redondance, l'optimisation et les structures dépendant du SGD.

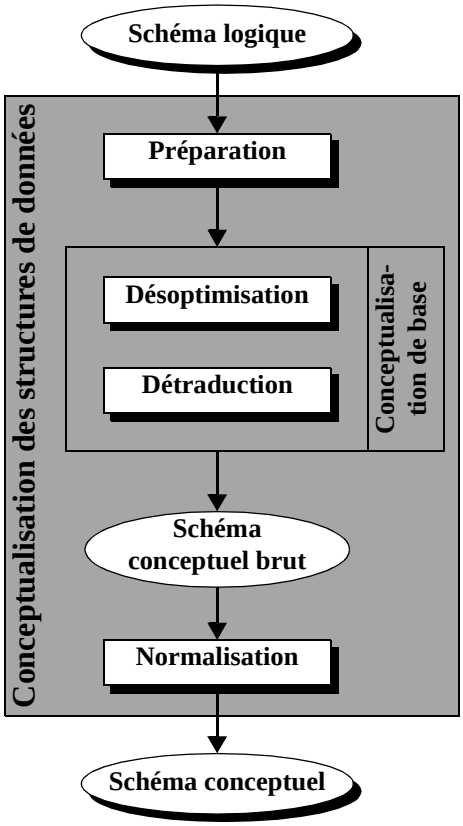


Figure 5.3 - Architecture de la phase de conceptualisation des structures de données.

Les principales étapes de la conceptualisation des structures de données (figure 5.3) sont :

- La préparation :
  Le schéma logique obtenu dans la phase d'extraction peut contenir des constructions ne contenant aucune sémantique. Il s'agit soit de structures de données obsolètes laissées dans la base de données par les différents programmeurs qui ont participé aux différents développements (c'est le cas de tables sans instances ou des instances non modifiées depuis longtemps par exemple), soit de structures de données techniques introduites pour faciliter la programmation mais sans intérêt pour le domaine d'application modélisé (des compteurs, le numéro du dernier utilisateur, ...). Cette étape apporte également des modifications cosmétiques sur le schéma comme l'amélioration des noms pour les rendre plus significatifs (suppression des préfixes par exemple) ou sa réorganisation graphique pour faciliter la lecture.
- La conceptualisation de base :
  Le but de cette étape est d'éliminer ou transformer toutes les stuctures de données non conceptuelles. Elle est décomposée en deux processus :
  - La détraduction :
    Le schéma logique est la traduction technique de structures conceptuelles. Le travail de l'analyste consiste à identifier ces traductions et à les transformer en structures conceptuelles équivalentes.
  - La désoptimisation :
    Le schéma a été restructuré et enrichi par des structures dans un souci d'optimisation (redondances structurelles par exemple). Ces structures doivent être détectées et éliminées par transformation.
- La normalisation conceptuelle :
  Le schéma doit être restructuré pour avoir les qualités habituellement requises par un schéma

conceptuel telles que l'expressivité, la simplicité, la minimalité, la lisibilité, la généricité, l'extensibilité et, parfois, le respect d'un standard méthodologique propre à une organisation.

5.2.3 Rétro-ingénierie dans le cadre de l'évolution

La reconstruction des schémas physique et logique fait partie de la première phase du processus de rétro-ingénierie (extraction des structures de données). Lors de cette phase, le schéma physique (désigné par SP0 dans la suite du chapitre) est obtenu après la réalisation des processus d'analyse du code DDL, d'intégration physique et de raffinement du schéma. Le concepteur enregistre dans l'historique $H_{CP0'}$ l'ensemble des transformations CP0' (correspondant au processus de nettoyage du schéma) effectuées sur SP0 pour obtenir le schéma logique (appelé SL0).

La reconstruction du schéma conceptuel (appelé SC0) à partir de SL0 fait partie de la deuxième phase de la démarche (conceptualisation ou interprétation des structures de données). L'historique $H_{CL0'}$ contient l'enregistrement de l'ensemble des transformations CL0' effectuées sur SL0 pour obtenir SC0 pendant cette phase. Ces transformations correspondent aux processus de préparation, détraduction, désoptimisation et normalisation du schéma logique SL0.

Pour compléter la démarche, les historiques $H_{CP0'}$ et $H_{CL0'}$ sont inversés pour donner les processus de conception physique (CP0) et logique (CL0). Ainsi, une documentation complète des spécifications a été recréée (figure 5.4). Les schémas de chaque niveau d'abstraction et les historiques de conception sont disponibles pour appliquer les stratégies de propagation développées dans les sections suivantes.

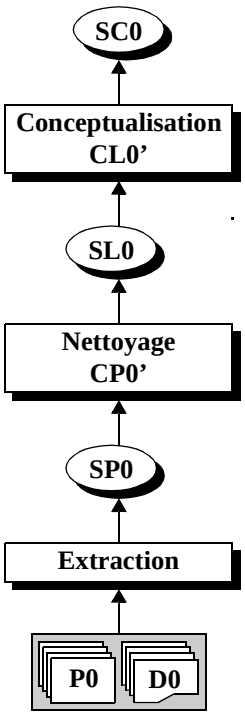


Figure 5.4 - Redocumentation des spécifications : schémas et processus.

5.3 Première stratégie : Modifications des spécifications conceptuelles

Dans ce premier scénario, les modifications proviennent de changements dans les besoins (R1) auxquels satisfont les spécifications du niveau conceptuel. Le problème est celui de la propagation de ces modifications aux couches inférieures (logique et physique). On suppose que les spécifications de tous les niveaux de modélisation existent : le schéma conceptuel SC0, le schéma logique SL0, le schéma physique SP0, la base de données D0 (schéma + données) et les programmes P0 de l'application (figure 5.5).

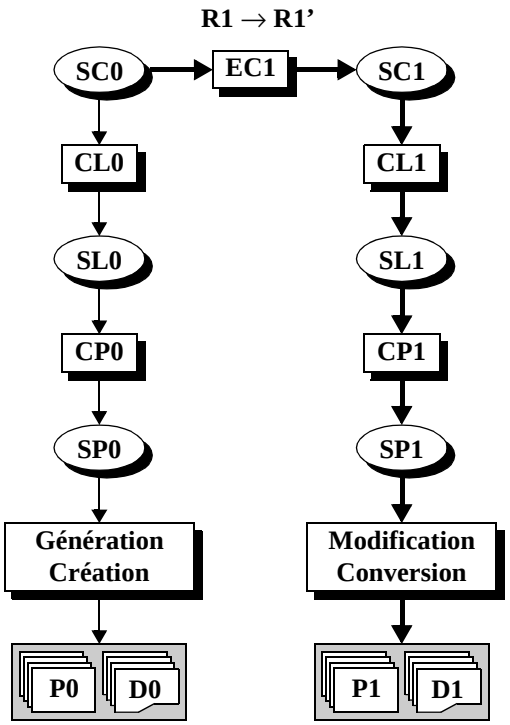


Figure 5.5 - Propagation vers l'aval des modifications déduites de R1 → R1'.

Les transformations appliquées sur SC0 pour obtenir SL0 (conforme au modèle relationnel) et sur SL0 pour obtenir SP0 sont désignées par <CL0,cl0> [1] (conception logique) et <CP0,cp0> (conception physique). On a : SL0 = CL0(SC0) et SP0 = CP0(SL0). CL0 est une composition de transformations simples (elles seront détaillées dans le point 5.3.2). CL0 est enregistré dans l'historique H_{CL0} ce qui permet de garder une trace des transformations appliquées. On peut aussi écrire que SL0 = H_{CL0}(SC0). De même, CP0 est une composition de transformations simples (elles seront détaillées dans le point 5.3.4). CP0 est enregistré dans l'historique H_{CP0} ce qui permet de garder une trace des transformations appliquées. On a : SP0 = H_{CP0}(SL0). Cette situation est traitée en quatre étapes :

- la modification du schéma conceptuel SC0 suite à l'évolution des besoins (R1 → R1'),
- la reconstruction du nouveau schéma logique SL1 en fonction des modifications conceptuelles,
- la reconstruction du schéma physique SP1 proche de SP0 et intégrant les modifications,
- la reconstruction de la base de données et des applications.

1. cl0 désigne les transformations appliquées sur les instances de SC0 pour obtenir les instances de SL0. Pour simplifier le discours, les instances sont consciemment oubliées dans la suite de l'exposé.

Pour rappel, l'analyse des modifications des besoins R1 auxquels satisfaisait le schéma conceptuel SC0 ainsi que leur traduction en modifications de SC0 ne font pas partie de ce travail. Par conséquent, l'étape préliminaire de modification des besoins n'est pas prise en compte dans les trois stratégies présentées.

5.3.1 Etape 1 : Modification du schéma conceptuel de la base de données

On suppose que les besoins auxquels SC0 satisfait évoluent de R1 vers R1'. Le changement est traduit par l'analyste en modifications du schéma SC0 qui devient SC1. Les transformations <EC1,ec1> appliquées pour obtenir SC1 à partir de SC0 sont enregistrées dans l'historique H_{EC1}. On a donc la formulation transformationnelle : SC1 = H_{EC1}(SC0).

Le tableau 5.1 présente une classification des principales modifications par type d'objets et degré de conservation de la sémantique. Une typologie complète des modifications aux niveaux conceptuel, logique et physique est proposée dans le chapitre 4 et l'annexe C.

Type d'objets	**Sémantique**		
	T^+	**T^-**	**$T^=$**
Type d'entités	Ajout	Suppression	Modification du nom Transformation en attribut Transformation en TA Eclatement / Fusion
Type d'associations	Ajout	Suppression	Modification du nom Transformation en TE
Rôle	Ajout Augmentation card. max. Diminution card. min. Ajout type d'entités	Suppression Diminution card. max. Augmentation card. min. Retrait type d'entités	Modification du nom
Relation IS-A	Ajout Changement de type	Suppression Changement de type	
Attribut	Ajout Augmentation card. max. Diminution card. min. Extension domaine Changement de type	Suppression Diminution card. max. Augmentation card. min. Restriction domaine Changement de type	Modification du nom Transformation en TE
Identifiant (primaire, secondaire)	Suppression Ajout composant	Ajout Retrait composant	Modification du nom Changement de type (P/S)
Contraintes (coexistence, exclusivité, au moins un, exactement un)	Suppression Ajout composant Changement de type	Ajout Retrait composant Changement de type	Modification du nom

Tableau 5.1 - Classification des modifications possibles au niveau conceptuel.

5.3.2 Etape 2 : Propagation des modifications en aval vers le niveau logique

Cette étape consiste d'abord à transformer le schéma SC1 en un schéma logique relationnel SL1 qui soit le plus proche possible de l'ancienne version, mais qui intègre les modifications conceptuelles. Pour réaliser cela, on applique la transformation de conception logique CL0 sur SC1 en rejouant l'historique H_{CL0} sur SC1. Cet historique contient la trace des transformations nécessaires pour transformer le schéma conceptuel SC0 en un schéma SL0 conforme au modèle relationnel. Le tableau 5.2 présente une classification des principales transformations de conception logique (de type $T^=$) par type d'objets.

Type d'objets	$T^{=}$
Type d'entités / table	Modification du nom Eclatement / Fusion
Type d'associations / clé étrangère	Transformation en type d'entités + transformation en clé étrangère Transformation en clé étrangère
Hiérarchie (IS-A)	Transformation en type d'associations + transformation en clé étrangère Héritage ascendant Héritage descendant
Attribut / colonne	Modification du nom Transformation en type d'entités par instance + transformation en clé étrangère Transformation en type d'entités par valeur + transformation en clé étrangère Désagrégation d'un attribut décomposable Instantiation d'un attribut multivalué en attributs monovalués Concaténation d'un attribut multivalué
Identifiant	Modification du nom
Contraintes (coex., excl., au moins un, exactement un)	Modification du nom

Tableau 5.2 - Classification des transformations utilisées en conception logique.

Dans le schéma SC1, des structures ont été modifiées, ajoutées, supprimées ou simplement conservées par rapport au schéma SC0. Chaque situation va être analysée en détail :

1. *Un objet de SC0 est conservé sans modification dans SC1*. Deux cas sont possibles :
   - Les transformations le concernant dans H_{CL0} sont appliquées telles quelles sur SC1.
     Exemple : un type d'associations non modifié est transformé en clé étrangère.
   - L'analyste décide d'appliquer d'autres transformations concernant l'objet que celles dans H_{CL0}. Dans ce cas, il s'agit d'une modification des spécifications logiques (voir point 5.4.1).
2. *Un objet de SC0 est modifié dans SC1*. Quatre cas sont possibles :
   - Les transformations de H_{CL0} le concernant restent appropriées et sont appliquées.
     Exemple : un attribut multivalué dont la cardinalité maximum passe de 5 à 10 sera toujours transformé en type d'entités.
   - Les transformations de H_{CL0} le concernant restent appropriées, mais l'analyste décide d'appliquer d'autres transformations (traitement sous la responsabilité de l'analyste).
     Exemple : un attribut multivalué dont la cardinalité maximum passe de 5 à 10 et, qui était transformé en une série d'attributs monovalués, sera transformé en type d'entités.
   - Les transformations de H_{CL0} sont inappropriées et l'objet modifié doit être traité de manière spécifique (traitement sous la responsabilité de l'analyste).
     Exemple : un type d'associations qui devient plusieurs-à-plusieurs ne peut plus être transformé en clé étrangère, il doit d'abord être transformé en type d'entités.
     La modification du nom de l'objet est un problème à part entière. Les objets étant désignés par leur nom (il est unique dans le contexte de l'objet) dans un historique, la modification du nom d'un objet empêche la réapplication des transformations le concernant. L'objet renommé doit être traité de manière spécifique.
   - Aucune transformation ne concernait l'objet dans H_{CL0} mais la modification entraîne la nécessité de le transformer dans SC1 (traitement sous la responsabilité de l'analyste).
     Exemple : un attribut monovalué qui devient multivalué doit être transformé dans SC1 alors qu'aucune transformation ne le concernait dans H_{CL0}.
3. *Un objet de SC0 a été supprimé dans SC1* :
   - Les éventuelles transformations de H_{CL0} le concernant sont exécutées sans effet.
4. *Un objet absent dans SC0 est introduit dans SC1* :

– Les transformations de H_{CL0} restent sans effet; l'objet devant être, si nécessaire, traité de manière spécifique (sous la responsabilité de l'analyste).

L'application à SC1 de l'historique ancien H_{CL0}, auquel s'ajoute et s'enlève le traitement de certains objets (modifiés, conservés, ajoutés ou supprimés), représente le nouveau processus de conception logique CL1 (SL1=CL1(SC1)). Il y correspond un nouvel historique H_{CL1}.

5.3.3 Etape 3 : Propagation des modifications en aval vers le niveau physique

Le schéma SL1 doit être transformé en un schéma physique relationnel SP1 proche de l'ancienne version (SP0) et intégrant les modifications. On procède comme dans l'étape 2 en rejouant l'historique H_{CP0} sur SL1. Cet historique contient les transformations concernant les structures physiques relationnelles (index et espaces de stockage) introduites au niveau physique. Le tableau ci-dessous présente une classification des principales transformations (de type T^+, T^- et $T^=$).

Type d'objets	**Sémantique**		
	T^+	**T^-**	**$T^=$**
Index	Ajout Ajout colonne	Suppression Retrait colonne	Modification du nom
Espace de stockage	Ajout Ajout table	Suppression Retrait table	Modification du nom

Tableau 5.3 - Classification des transformations utilisées en conception physique.

Les modifications des structures dans le schéma SL1 peuvent engendrer quatre situations de base identiques à l'étape 2 lorsqu'on rejoue l'historique H_{CP0}. Il en résulte un historique H_{CP1} contenant les transformations des objets non modifiés de SL1 ainsi que celles des nouveaux objets; de cet historique sont soustraites les transformations sans effet.

5.3.4 Etape 4 : Propagation des modifications vers les données et les traitements

Il reste à convertir la base de données (D0) - structures et données - et les programmes (P0) qui doivent se conformer aux nouvelles spécifications. La conversion des données (D0) est une tâche presque déterministe [2], qui peut donc être automatisée par la génération de convertisseurs comme on le verra dans la section 5.6. L'analyse de l'historique H_{EC1} permet de repérer les modifications opérées au niveau conceptuel et l'analyse différentielle de H_{CL0} et H_{CL1}, H_{CP0} et H_{CP1} nous fournit les informations nécessaires pour en dériver les structures physiques.

Rejouer successivement l'historique H_{CL1} puis H_{CP1} sur SC1 donne le schéma physique SP1. On a : SP1 = CP1(CL1(SC1)) respectivement SP0 = CP0(CL0(SC0)). Pour simplifier l'exposé, nous appellerons C1 (C0) la concaténation des historiques CL1 et CP1 (CL0 et CP0). On obtient : SP1 = C1(SC1) et SP0 = C0(SC0).

En fonction du type des modifications présentes dans EC1, trois comportements distincts se présentent :

1. Dans le cas de la création d'un nouvel objet, l'analyse de C1 donne les nouvelles structures physiques à créer en fonction des transformations éventuelles, appliquées successivement sur l'objet créé.

2. Certaines instances peuvent contredire les nouvelles spécifications ce qui nécessite souvent une intervention humaine.

2. Dans le cas d'une suppression, l'analyse de C0 fournit les anciennes structures physiques à détruire en fonction des transformations éventuelles qui étaient appliquées sur l'objet détruit.
3. Le cas de la modification est plus complexe. Trois situations différentes peuvent se présenter :
   - L'objet modifié n'est pas à l'origine de transformations dans C0 et C1. L'objet est la structure à modifier.
   - L'objet modifié est à l'origine de transformations identiques dans C0 et C1. L'analyse de C0 ou C1 donne les structures physiques à modifier.
   - L'objet modifié est l'origine de transformations différentes dans C0 et C1. Premièrement, les nouvelles structures sont créées sur la base de l'analyse de C1. Ensuite, les instances sont transférées des anciennes structures vers les nouvelles. Et finalement, les anciennes structures sont détruites d'après le résultat de l'analyse de C0.

Sur la base de l'analyse des historiques H_{EC1}, H_{C0} et H_{C1}, on déduit les transformations des structures de données et les transformations des instances de SP0 (c'est-à-dire D0) en instances de SP1 (D1). Ces transformations sont traduites sous la forme de convertisseurs, constitués de scripts SQL dans les cas simples, et de programmes dans les situations plus complexes. On obtient ainsi la nouvelle version de la base de données par application d'une chaîne de transformations sur l'ancienne base.
Modifier les programmes est une tâche beaucoup plus complexe, et pour l'instant non automatisable, sauf dans des situations simples où les modifications sont mineures. En toute généralité, la modification des programmes est de la responsabilité du programmeur. Cependant, il est possible de préparer ce travail par une analyse du code qui permet de repérer et d'annoter les sections critiques des programmes. La section 5.7 aborde de manière détaillée le problème de la localisation des lignes de code susceptibles d'être modifiées.

5.4 Deuxième stratégie : Modifications des spécifications logiques

Selon un deuxième scénario, l'analyste apporte des modifications résultant du changement des besoins techniques ou organisationnels au niveau logique (figure 5.6). Le problème est celui de la reconstitution d'un historique de conception logique correct et de la propagation des modifications, en aval, vers la couche physique, pour rendre les modifications opérationnelles. Cette situation est traitée en quatre étapes :

- modification du schéma logique de la base de données suite à l'évolution des besoins (R2 $\rightarrow$ R2'),
- reconstruction du processus de conception logique CL1,
- propagation des modifications en aval vers le niveau physique,
- propagation des modifications en aval vers les données et les programmes.

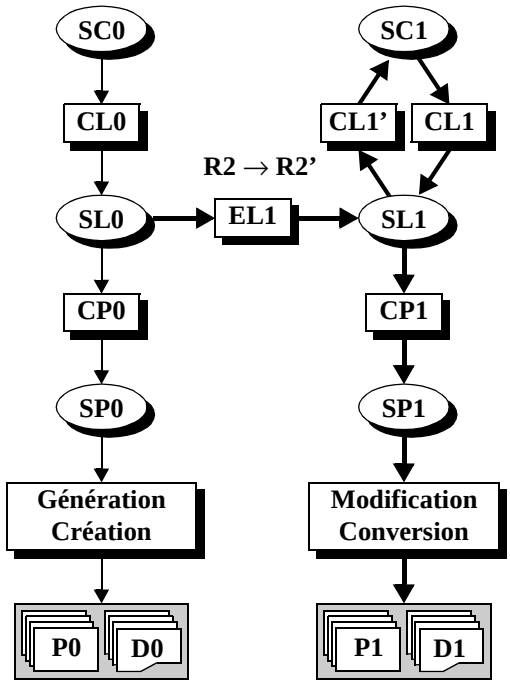


Figure 5.6 - Propagation vers l'aval des modifications déduites de R2 $\rightarrow$ R2' et reconstruction de CL1.

5.4.1 Etape 1 : Modification du schéma logique de la base de données

Cette étape consiste à modifier le schéma SL0 pour obtenir le schéma SL1. Les transformations sont enregistrées dans l'historique H_{EL1}. Il faut propager les modifications conduisant à SL1 en aval, vers SP1, D1 et P1, mais aussi reconstruire l'historique de conception logique CL1 pour obtenir SL1 à partir de SC1 et disposer ainsi d'une documentation complète du processus de conception. Une table des principales modifications au niveau logique [3] est présentée dans le tableau 5.4 pour un sous-modèle relationnel.

3. Il existe d'autres transformations permettant de prendre en compte des critères organisationnels ou techniques. Seules les plus fréquentes sont mentionnées.

Type d'objets	Sémantique		
	T^+	T^-	$T^=$
Table			Modification du nom Eclatement / Fusion
Colonne	Ajout (colonne technique)	Suppression (colonne technique)	Modification du nom
Identifiant	Ajout (identifiant technique)	Supression (identifiant technique)	Modification du nom
Clé étrangère			Modification du nom
Contrainte			Modification du nom

Tableau 5.4 - Classification des modifications possibles au niveau logique.

Une autre façon de modifier le schéma logique consiste à représenter par une autre structure logique un objet conceptuel non modifié. Par exemple, un attribut multivalué représenté par une série de colonnes dans SL0 est transformé en une table dans SL1. Il s'agit bien de modifications du niveau logique. Toutefois, pour les mettre en œuvre, il faudrait, d'abord, remonter vers le niveau conceptuel en appliquant la transformation inverse et, ensuite, redescendre vers le niveau logique en appliquant la nouvelle transformation de conception logique.

Afin de simplifier le processus, on propose plutôt d'utiliser la première stratégie pour ce type de modifications. L'historique d'évolution conceptuel est vide, SC1 est identique à SC0 (la première étape est inutile). Lorsque le concepteur rejoue H_{CL0} sur SC1 (deuxième étape), il n'exécute pas les transformations de H_{CL0} sur l'objet qui doit être transformé différemment. Il enregistre à la fin de H_{CL1} la nouvelle transformation appliquée sur l'objet. La troisième étape reste inchangée. La quatrième étape se base sur C0 et C1 pour modifier les données et les programmes. C0 contient les anciennes structures physiques à détruire et C1 les nouvelles structures à créer.

5.4.2 Etape 2 : Reconstruction du processus de conception logique CL1

La propagation est basée sur les mêmes principes que ceux développés dans la première stratégie. On suppose qu'on dispose de l'historique $H_{CL0'}$. Si ce n'est pas le cas, on l'obtient par inversion de H_{CL0}. SC1 est obtenu à partir de SL1 en deux phases : application de $H_{CL0'}$, puis conceptualisation des objets de SL1 qui ont été créés ou modifiés par rapport à SL0. Pour cette dernière phase qui est pilotée par l'analyste, nous avons répertorié quatre cas différents :

1. *Un objet de SL0 est conservé sans modification dans SL1* : les transformations le concernant dans H_{CL0} sont appliquées telles quelles sur SL1.
   Exemple : une clé étrangère non modifiée est transformée en type d'associations.
2. *Un objet de SL0 est modifié dans SL1*. Trois situations sont possibles :
   - Les transformations de $H_{CL0'}$ sont inappropriées et l'objet modifié doit être traité de manière spécifique (traitement sous la responsabilité de l'analyste). Il s'agit essentiellement des modifications relatives aux noms qui empêche la réapplication des transformations concernant l'objet dans l'historique. L'objet renommé doit être traité de manière spécifique.
   - Aucune transformation ne concerne l'objet dans $H_{CL0'}$ mais la modification entraîne la nécessité de le transformer dans SC1 (traitement sous la responsabilité de l'analyste).
     Exemple : une table est éclatée en deux tables qui sont reliées par une clé étrangère. Les deux tables doivent être fusionnées en un seul type d'entités dans SC1.
   - Les transformations de $H_{CL0'}$ le concernant restent appropriées et sont appliquées.
     Exemple : si l'éclatement de deux tables est modifié (transfert d'autres colonnes), la transformation de fusion enregistrée dans $H_{CL0'}$ est toujours d'application.

3. *Un objet de SC0 a été supprimé dans SC1* : les éventuelles transformations de $H_{CL0'}$ le concernant sont appliquées sans effet.
   Exemple : la fusion de deux tables.
4. *Un objet absent dans SC0 est introduit dans SC1* : les transformations de H_{CL0} restent sans effet, l'objet devant être, si nécessaire, traité de manière spécifique (sous la responsabilité de l'analyste).
   Exemple : l'éclatement d'une table.

L'application à SL1 de l'historique ancien $H_{CL0'}$ auquel s'ajoute le traitement de certains objets modifiés, conservés ou ajoutés et duquel s'enlève le traitement de certains objets modifiés, conservés ou supprimés représente le nouveau processus de conceptualisation CL1' (SC1=CL1'(SL1)) dont l'inversion donne le nouveau processus de conception logique CL1. Il y correspond un nouvel historique $H_{CL1'}$ dont l'inversion fournit H_{CL1}. On a reconstruit une documentation plausible de l'historique de conception logique permettant par la suite de réappliquer les stratégies pour d'autres modifications de spécifications. Le tableau 5.5 présente les principales transformations utilisées pour la conceptualisation d'un schéma logique.

Type d'objets	Transformations
Table / Type d'entités	Modification du nom Transformation en type d'associations Transformation en attribut Eclatement / Fusion
Table / relation IS-A	Héritage descendant ou ascendant
Clé étrangère / Type d'associations	Transformation en type d'associations Transformation en type d'associations + transformation en relations IS-A.
Colonne / Attribut	Modification du nom Agrégation d'un attribut décomposable Transformation d'une liste d'attributs monovalués en un attribut multivalué Instantiation d'un attribut multivalué Destruction des colonnes techniques
Identifiant	Modification du nom Destruction des identifiants techniques
Contraintes (coexistence, exclusivité, au-moins-un, exactement-un)	Modification du nom

Tableau 5.5 - Classification des transformations utilisées pour la conceptualisation.

5.4.3 Etape 3 : Propagation des modifications en aval vers le niveau physique

Cette étape est similaire à la troisième étape de la première stratégie (point 5.3.3).

5.4.4 Etape 4 : Propagation des modifications en aval vers les données et les programmes

Cette étape consiste à propager les modifications vers la partie opérationnelle. Cette propagation a déjà été analysée dans le premier scénario. On peut toutefois préciser que, pour générer les convertisseurs, les historiques H_{EL1}, H_{CP0} et H_{CP1} sont analysés à la place des historiques H_{EC1}, H_{C0} et H_{C1} dans la première stratégie.

5.5 Troisième stratégie : Modifications des spécifications physiques

La troisième stratégie consiste à modifier le schéma physique suite à des modifications des besoins techniques R3 (figure 5.7). Les modifications doivent être propagées en aval vers les données et les programmes. L'historique de conception physique doit également être mis à jour. Cette situation est traitée en trois étapes :

- modification du schéma physique de la base de données suite à l'évolution des besoins (R3 → R3'),
- reconstruction de l'historique de conception logique CP1,
- propagation des modifications vers les données et les traitements.

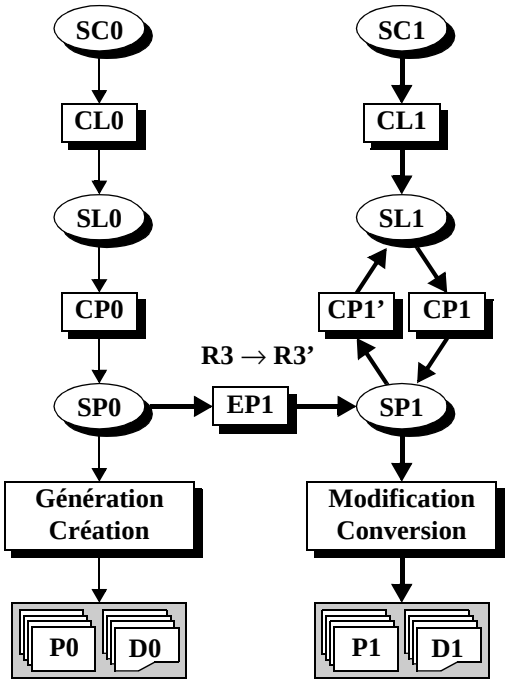


Figure 5.7 - Propagation vers l'aval des modifications déduites de R3 → R3' et reconstruction de CP1.

5.5.1 Etape 1 : Modification du schéma physique de la base de données

Suite à des changements de besoins techniques, le schéma SP0 est transformé en un schéma SP1. Les transformations sont enregistrées dans l'historique H_{EP1}. Il faut propager les modifications conduisant à SL1 en aval, vers D1 et P1, mais aussi reconstruire l'historique de conception physique CL1 pour obtenir SP1 à partir de SL1. La table 5.6 présente les principales modifications au niveau physique pour un sous-modèle relationnel.

Type d'objets	T^+	T^-	$T^=$
Index	Création Ajout colonne	Suppression Retrait colonne	Modification du nom
Espace de stockage	Création Ajout table	Suppression Retrait table	Modification du nom

Tableau 5.6 - Classification des principales modifications au niveau physique.

5.5.2 Etape 2 : Reconstruction de l'historique de conception logique CP1

On inverse l'historique CP0 pour obtenir CP0'. SL1 est obtenu à partir de SP1 en deux phases : application de CP0', puis transformation des nouveaux objets de SP1, cette dernière phase étant pilotée par l'analyste. L'historique de ces deux phases constitue CP1', dont l'inversion fournit CP1. Notons que la conception physique travaille uniquement sur des index et des espaces de stockage. Les transformations des nouveaux objets de SP1 sont exclusivement des suppressions de ces structures.

Ces structures techniques étant ignorées dans la conception logique, il n'est pas nécessaire de reconstruire l'historique CL1 et le schéma conceptuel SC1 car ils équivalent à CL0 et SC0.

On suppose qu'on dispose de l'historique $H_{CP0'}$. Si ce n'est pas le cas, on l'obtient par inversion de H_{CP0}. SL1 est obtenu à partir de SP1 en deux phases : l'application de $H_{CP0'}$, puis les transformations des objets de SP1 qui ont été créés ou modifiés par rapport à SP0. La conception physique travaille uniquement sur des index et des espaces de stockage. Les transformations des nouveaux objets de SP1 sont exclusivement les suppressions de ces structures. Suivant les modifications apportées au niveau physique, nous avons répertorié quatre situations différentes :

1. *Un objet de SP0 est conservé sans modification dans SP1* : les transformations le concernant dans H_{CP0} sont appliquées telles quelles sur SP1.
2. *Un objet de SP0 est modifié dans SP1*. Deux situations sont possibles :
   - Les transformations de $H_{CP0'}$ sont inappropriées et l'objet modifié doit être traité de manière spécifique (traitement sous la responsabilité de l'analyste). Il s'agit essentiellement des modifications relatives aux noms qui empêchent la réapplication des transformations concernant l'objet dans l'historique.
   - Les transformations de $H_{CP0'}$ le concernant restent appropriées et sont appliquées.
     Exemple : si on ajoute une colonne à un index, la suppression enregistrée dans $H_{CP0'}$ est toujours d'application.
3. *Un objet de SP0 a été supprimé dans SP1* : les éventuelles transformations de $H_{CP0'}$ le concernant sont exécutées sans effet.
   Exemple : la suppression d'un index.
4. *Un objet absent dans SP0 est introduit dans SP1* : les transformations de H_{CP0} restent sans effet, l'objet devant être traité de manière spécifique (sous la responsabilité de l'analyste).
   Exemple : la création d'un index.

L'application à SP1 de l'historique ancien $H_{CP0'}$ auquel s'ajoute le traitement de certains objets modifiés, conservés ou ajoutés et duquel s'enlève le traitement de certains objets modifiés, conservés ou supprimés représente le nouveau processus de nettoyage CP1' (SL1=CP1'(SP1)) dont l'inversion donne le nouveau processus de conception logique CP1. Il y correspond un nouvel historique $H_{CP1'}$ dont l'inversion fournit H_{CP1}. On a donc de la sorte reconstruit une documentation plausible de l'historique de conception physique. Le tableau 5.7 présente les principales transformations utilisées pour le nettoyage d'un schéma physique.

Type d'objets	Transformations
Index	Suppression
Espace de stockage	Suppression

Tableau 5.7 - Classification des transformations utilisées pour le nettoyage physique.

5.5.3 Etape 3 : Propagation des modifications vers les données et traitements

La propagation des modifications vers les données et les programmes a été analysée dans le premier scénario. Toutefois, pour générer les convertisseurs, l'analyse de l'historique H_{EP1} est suffisante puisque EP1 contient les modifications relatives au modèle physique qui est conforme aux structures de données de l'application.

5.6 Conversion des données et structures de données

5.6.1 Introduction

La conversion des données consiste à générer les scripts (code DDL) qui permettront de modifier les structures de données (et leurs instances) de l'application, de manière à satisfaire les modifications apportées aux niveaux des spécifications (qu'elles soient conceptuelles, logiques ou physiques). Pour atteindre cet objectif, on dispose des schémas de spécifications aux différents niveaux d'abstraction ainsi que des historiques d'évolution (conceptuelle, logique ou physique) et de conception (logique et physique). Ces historiques retracent l'histoire des spécifications, c'est-à-dire les modifications [4] apportées aux spécifications pour obtenir les nouvelles spécifications physiques. Une analyse approfondie des historiques va permettre d'aboutir à la réalisation des scripts de conversion. Le processus de conversion est décomposé en quatre phases :

- l'analyse des historiques d'évolution et de conception de manière à construire des historiques simplifiés contenant les informations strictement nécessaires à la conversion (section 5.6.2),
- la traduction des modifications des historiques simplifiés d'évolution en modifications relatives aux spécifications physiques (section 5.6.3),
- la construction d'une liste restructurée des modifications sur la base de l'historique simplifié, obtenu précédemment, de manière à satisfaire les impératifs techniques du SGBDR choisi (section 5.6.4),
- la génération des scripts de conversion des données et structures de données à partir de la liste des modifications (section 5.6.5).

5.6.2 Construction des historiques simplifiés

Tout d'abord, le point 5.6.2.1 argumente la nécessité de simplifier les historiques. Ensuite, le principe de simplification des historiques est expliqué dans le point 5.6.2.2. Et finalement, les règles de comparaison utilisées pour la simplification d'un historique sont décrites dans le point 5.6.2.3.

5.6.2.1 Nécessité de la simplification des historiques

Une première étape dans la conversion des données consiste à analyser les informations contenues dans les historiques d'évolution et de conception. On constate que certaines informations ne sont pas nécessaires pour la conversion. Un historique peut contenir deux types d'informations non pertinentes qui peuvent engendrer des scripts de conversion inutilement compliqués :

1. A cause de la propriété d'exhaustivité, un historique enregistre toutes les informations concernant les modifications (pour permettre son inversion) dont une partie seulement est nécessaire pour la conversion. Par exemple, si un type d'entités est détruit, les informations le concernant ainsi que celles sur ses attributs et ses groupes sont enregistrées dans l'historique. Pour la conversion, seule la destruction du type d'entités est nécessaire car elle implique obligatoirement la destruction de ses éléments.
2. Le comportement du concepteur de la base de données est aussi à l'origine d'enregistrements inutiles pour la conversion. L'historique est brut et contient toutes les modifications

4. Les termes *transformation* et *modification* sont utilisés indifféremment comme terme générique pour désigner tout changement (création, destruction, modification ou transformation) apporté sur des spécifications.

effectuées sur des spécifications. Le concepteur procède parfois par tâtonnements (essais et erreurs), ce qui engendre des enregistrements superflus. Prenons deux exemples pour illustrer le problème.
Exemple 1 : si un objet est modifié plusieurs fois, toutes les modifications sont enregistrées dans l'historique. Les informations pertinentes sont celles concernant l'objet après toutes ses modifications. Les informations intermédiaires sont des étapes, dans le processus de modification du schéma, qui sont inutiles dans le cadre de la conversion.
Exemple 2 : le concepteur ajoute un attribut à un type d'entités puis il décide d'en faire un type d'entités qui est relié par un type d'associations à son ancien parent (transformation d'attribut en type d'entités). Seule la création d'un type d'entités (attribut transformé) et d'un type d'associations sont pertinentes pour la conversion.

Pour ces deux raisons, un historique n'est pas directement exploitable pour la conversion des structures de données. Il doit être retravaillé pour en extraire un historique simplifié contenant les informations strictement nécessaires et suffisantes pour la conversion des données. Pour chaque stratégie, il faut simplifier les historiques de conception et d'évolution suivants :

- Première stratégie : l'historique simplifié HS_{EC1} est construit à partir de l'historique d'évolution H_{EC1}, HS_{C0} à partir des anciens historiques de conception logique et physique (H_{CL0}, H_{CP0}) et HS_{C1} à partir des nouveaux historiques de conception logique et physique (H_{CL1}, H_{CP1}). Les historiques de conception sont regroupés dans un seul historique simplifié pour faciliter la deuxième phase du processus de conversion.
- Deuxième stratégie : l'historique simplifié HS_{EL1} est construit sur la base de l'historique d'évolution H_{EL1}, HS_{CP0} sur la base de l'ancien historique d'évolution physique H_{CP0} et HS_{CP1} sur la base du nouvel historique d'évolution physique H_{CP1}.
- Troisième stratégie : l'historique simplifié HS_{EP1} est construit à partir de l'historique d'évolution H_{EP1}.

5.6.2.2 Simplification d'un historique

Un historique simplifié est construit selon un principe d'incrémentation/comparaison. Soit H l'historique à simplifier et HS l'historique résultat de la simplification. Chaque transformation de l'historique H (désignée par T) est comparée à toutes les transformations déjà contenues dans HS. Lors de la phase de comparaison, quatre cas peuvent se présenter :

1. *T ne rentre pas en conflit avec les transformations de HS. Elle est ajoutée à la fin de HS.*
   Exemple : Soit HS = <E←creer-TE(S,n)> et T = E1←modifier-TE(E1,n'). Les deux transformations ne s'appliquent pas aux mêmes objets, le renommage de E1 est ajouté à la fin de HS.
2. *T annule une transformation présente dans HS. Les deux transformations n'apparaissent pas dans HS.*
   Exemple : Soit HS = <..., A←creer-Att(E,n), ...> et T = ()←supprimer-Att(E,A). Les deux transformations s'annulent, la création de A disparaît de HS et sa destruction n'est pas ajoutée.
3. *T est pertinente, elle est ajoutée à HS. Mais une transformation de HS n'est plus pertinente et doit disparaître.*
   Exemple : Soit HS = <..., A←creer-Att(E,n), ...> et T = ()←supprimer-TE(E). La destruction du type d'entités E entraîne la destruction de ses attributs dont A. La création de l'attribut n'est plus pertinente et doit être enlevée de HS car il est inutile de créer un attribut qui sera détruit par la suite. Par contre, T est ajoutée à la fin de HS.
4. *T est pertinente, mais elle n'est pas ajoutée à HS car elle modifie une transformation déjà présente dans HS.*
   Exemple : Soit HS = <..., E←creer-TE(S,n), ...> et T = E←modifier-TE(E,n'). La création de E avec n comme nom est remplacée par E←creer(S,n') dans HS. T n'est pas ajoutée à HS.

Lorsque T est comparée aux transformations de HS et qu'elle entre en conflit avec une de ces dernières (cas 2, 3 et 4 ci-dessus), T est (ou n'est pas suivant les cas) ajoutée à HS dont le contenu est éventuellement modifié. Généralement, la comparaison s'arrête là. Toutefois, lorsque T est la destruction d'un objet possédant des éléments (attributs ou groupes), il faut continuer la comparaison jusqu'à la fin de HS. Les règles de comparaison sont décrites dans le point 5.6.2.3.

Grâce aux propriétés de normalisation et de non-concurrence des historiques (point 3.3.4.4) et conformément à la définition de la propriété de minimalité (point 3.3.4.3), le principe de construction par incrémentation de HS assure qu'il est *minimal*. Il contient la liste des transformations strictement nécessaires et suffisantes de H (c'est-à-dire le contenu de H moins les transformations non pertinentes) de telle sorte que, si S1 = H(S0), alors on a également S1 = HS(S0).

5.6.2.3 Règles de comparaison

Chaque transformation T de H est comparée avec chaque transformation de HS (en cours de création). Pour extraire les règles de comparaison, tous les types de transformations ont été comparés entre eux, deux à deux, dans des tables de comparaisons qui sont présentées dans l'annexe D. Seules les comparaisons provoquant des conflits ont été prises en compte pour définir les règles de comparaison. Les règles sont scindées en deux parties : celles concernant les transformations sur un même objet (point a) et celles concernant les transformations sur des objets et leurs éléments (point b).

a) Règles de comparaison des transformations portant sur un même objet

Soit O un type d'objet pouvant être un type d'entités, un type d'associations, un attribut, un groupe, un rôle, une relation IS-A ou une collection. Soit P le propriétaire ou parent de O. Pour un type d'entités, un type d'associations et une collection, il s'agit d'un schéma. Pour un attribut et un groupe, il s'agit d'un type d'entités, d'un type d'associations ou d'un attribut décomposable. Pour un rôle, il s'agit d'un type d'associations. Pour une relation IS-A, il s'agit du surtype.

On considère quatre types de transformations sur O : la création (*creer*), la destruction (*supprimer*), la modification des propriétés (nom, type, longueur, ...) de O (*modifier*) et la transformation qui crée un nouvel objet (*transformer*).

Les règles extraites sont les suivantes :

1. *Si HS = <..., O←creer(P,n), ...> et T = ()←supprimer(P,O), alors O←creer(P,n) est enlevée de HS et T n'est pas ajoutée à HS.*
   La destruction de O annule sa création dans HS (la création est enlevée de HS et la destruction n'est pas ajoutée).
2. *Si HS = <..., O←creer(P,n), ...> et T = O←modifier(O,n'), alors O←creer(P,n) devient O←creer(P,n') dans HS et T n'est pas ajoutée.*
   La modification de O entraîne la mise à jour de la création de O dans HS sans être ajoutée à HS.
3. *Si HS = <..., O←modifier(O,n'), ...> et T = ()←supprimer(P,O), alors O←modifier(O,n') est enlevée de HS et T est ajoutée à HS.*
   La destruction de O annule sa modification dans HS et elle est ajoutée à HS en tant que destruction de O avant sa modification.
4. *Si HS = <..., O←modifier(O,n'), ...> et T = O←modifier(O,n"), alors O←modifier(O,n') est remplacée par O←modifier(O,n") dans HS et T n'est pas ajoutée à HS.*
   La modification de O entraîne la mise à jour de sa propre modification. Les deux modifications sont fusionnées dans HS.
5. *Si HS = <..., O←creer(P,n), ...> et T = (O1, ...)←transformer(O), alors O←creer(P,n) est remplacée par la création des nouveaux objets O1←creer(P,n1), ... dans HS et T n'est pas ajoutée à HS.*

La transformation de O annule sa création pour la remplacer par la création des nouveaux objets (O1, ...) créés par la transformation.

6. *Si HS = <..., O←modifier(O,n'), ...> et T = (O1, ...)←transformer(O), alors O←modifier(O,n') est enlevée de HS et (O1, ...)←transformer(O) est ajoutée dans HS.*
   La transformation de O annule et remplace la modification de O.
7. *Si HS = <..., (O1, ...)←transformer(O), ...> et T = ()←supprimer(P,O1), alors (O1, ...)←transformer(O) est enlevée de HS et ()←supprimer(P,O) est ajoutée dans HS.*
   La destruction des objets créés par la transformation annule la transformation de O qui est remplacée par la destruction de O.
8. *Si HS = <..., (O1, ...)←transformer(O), ...> et T = O1←modifier(O1,n1'), alors (O1, ...)←transformer(O) reste dans HS avec O1 modifié comme nouvel objet créé et T n'est pas ajoutée à HS.*
   La modification des objets créés par la transformation entraîne la mise à jour de cette transformation pour tenir compte de la modification.

b) Règles de comparaison des transformations concernant un objet et ses composants

Soit O un type d'objet pouvant être un type d'entités, un type d'associations ou un attribut décomposable. Soit C un composant de l'objet parent P. Pour un type d'entités, il s'agit d'un attribut, d'un groupe ou d'une relation IS-A. Pour un type d'associations, il s'agit d'un attribut, d'un groupe ou d'un rôle. Pour un attribut décomposable, il s'agit d'un attribut ou d'un groupe. La notion de composant englobe tous les objets dont l'existence dépend de l'existence de l'objet parent.

Les types de transformations considérés sont les mêmes que dans le point précédent. Les règles extraites des tables de comparaisons sont :

1. *Si HS = <..., C←creer(P,n), ...> et T = ()←supprimer(P), alors C←creer(P,n) est enlevé de HS.*
   La destruction d'un objet parent annule dans HS toutes les créations de ses composants. Seule la destruction du parent apparaît dans HS.
2. *Si HS = <..., ()←supprimer(P,C), ...> et T = ()←supprimer(P), alors ()←supprimer(P,C) est enlevé de HS.*
   La destruction d'un objet parent annule dans HS toutes les destructions de ses composants. Seule la destruction du parent apparaît dans HS.
3. *Si HS = <..., C←modifier(P,C,n'), ...> et T = ()←supprimer(P), alors C←modifier(P,C,n') est enlevé de HS.*
   La destruction d'un objet parent annule dans HS toutes les modifications de ses composants. Seule la destruction du parent apparaît dans HS.
4. *Si HS = <..., (C1, ...)←transformer(C), ...> et T = ()←supprimer(P,C), alors (C1, ...)←transformer(C) est enlevé de HS si la transformation crée des objets appartenant à P.*
   La destruction d'un objet parent annule les transformations de ses composants si ces transformations créent des objets appartenant à l'objet parent.
   Exemple : Si on transforme un attribut d'un type d'associations en un type d'entités, la destruction du type d'associations n'enlève pas la transformation de son attribut.
   Exemple : Si on désagrège un attribut décomposable dans un type d'associations, la destruction du type d'associations nécessite le retrait de la désagrégation de son attribut de HS.

5.6.3 Traduction des historiques d'évolution simplifiés

La deuxième phase du processus de conversion consiste à traduire les transformations enregistrées dans les historiques simplifiés construits sur la base des historiques d'évolution (conceptuelle ou logique) en transformations physiques. Cette phase n'est nécessaire que pour les deux

premières stratégies puisque, pour la troisième, l'historique simplifié d'évolution physique ne contient que des transformations sur les schémas physiques.

5.6.3.1 Représentation du problème

La figure 5.8 présente l'historique générique avant la phase de traduction. Pour la première stratégie, les schémas S0 et S1 représentent les schémas conceptuels SC0 et SC1; E1, les transformations d'évolution conceptuelle EC1; C0 et C1, l'ensemble des transformations de conception logique et physique (CL0 + CP0 et CL1 + CP1). Pour la deuxième stratégie, les schémas S0 et S1 représentent les schémas logiques SL0 et SL1; E1, les transformations d'évolution logique EL1; C0 et C1, l'ensemble des transformations de conception physique (CP0 et CP1). Le but de cette deuxième phase est de reconstruire les transformations d'évolution physique EP1 à partir de E1, C1 et C0. Pour reformuler le problème dans le contexte des historiques simplifiés, comment construire HS_{EP1} à partir de HS_{E1}, HS_{c0} et HS_{C1} ?

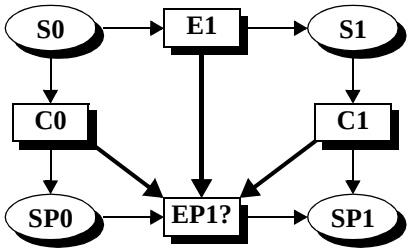


Figure 5.8 - Représentation du problème de la traduction des modifications conceptuelles ou logiques en modifications physiques.

Par l'analyse des stratégies de propagation, on sait que C1 est obtenu par application de C0 sur S1. Certaines transformations de C0 ne sont pas réappliquées et des nouvelles transformations sont exécutées. On a : SP1 = C1(E1(S0)) = EP1(C0(S0))

Supposons que C0' contienne les transformations appliquées telles quelles sur S1 et C0" les transformations qui ne sont pas appliquées. On a C0 = C0' + C0". On peut écrire [5] : C1 = C0' + C1' où C1' contient les nouvelles transformations appliquées sur S1. Cela donne : SP1 = C1'(C0'(E1(S0))) = EP1(C0"(C0'(S0))). Comme C0' ne contient que des transformations sur les objets non modifiés de S1, on peut logiquement en déduire que C0'(E1(S0)) = E1(C0'(S0)) puisque E1 ne contient que des transformations sur les objets modifiés de S1 (l'ordre d'application des transformations n'a pas d'importance). Ce qui donne : SP1 = C1'(E1(C0'(S0))) = EP1(C0"(C0'(S0))). Si on appelle SP0' le schéma physique construit à partir de S0 dans lequel seuls les objets non modifiés ont été transformés c'est-à-dire SP0' = C0'(S0). On obtient : SP1 = C1'(E1(SP0')) = EP1(C0"(SP0')) ce qui signifie que E1 + C1' = C0" + EP1. On peut conclure que l'historique de conception physique (EP1) est équivalent à l'historique d'évolution conceptuelle ou logique (E1) augmenté des nouvelles transformations appliquées sur S1.

5.6.3.2 Principe de traduction

Le principe de la traduction des transformations d'évolution conceptuelle ou logique en transformations physiques est le suivant :

1. Pour chaque transformation T de HS_{E1} :
   a) Si T est une transformation de création, on parcourt $HS_{C1'}$ pour trouver les transformations concernant l'objet créé par T. Si de telles transformations existent, les créations des nouvelles structures sont ajoutées à HS_{EP1}. Sinon T est ajoutée à HS_{EP1}.

5. Le sigle + est utilisé pour représenter la concaténation (mise bout à bout) de deux historiques. Si on a H2(H1(S)) : l'application de deux historiques (H1 et H2) sur un schéma S est équivalente à l'application de la concaténation des deux historiques sur S. On peut donc écrire : (H1 + H2)S = H2(H1(S)).

b) Si T est une transformation de destruction, on parcourt $HS_{C0"}$ pour trouver les transformations concernant l'objet détruit par T. Si de telles transformations existent, on ajoute à HS_{EP1} les destructions des anciennes structures de $HS_{C0"}$. Sinon T est ajoutée à HS_{EP1}.

c) Si T est une transformation de modification ou une transformation à sémantique constante sur un objet O, on parcourt $HS_{C0"}$ pour trouver les transformations appliquées sur O ainsi que $HS_{C1'}$ pour trouver les transformations concernant le même objet. On compare les objets trouvés dans SP0 et dans SP1. S'ils sont de mêmes structures (c'est-à-dire que les transformations utilisées dans $HS_{CO"}$ et $HS_{C1'}$ sont identiques), on ajoute à HS_{EP1} les modifications des objets représentant O dans SP1 par rapport à ceux le représentant dans SP0. S'ils ne sont pas de mêmes structures (c'est-à-dire que les transformations utilisées dans $HS_{CO"}$ et $HS_{C1'}$ ne sont pas identiques), on ajoute à HS_{EP1} la transformation qui permet de passer des structures représentant O dans SP0 à celles le représentant dans SP1.

2. Pour chaque transformation T de $HS_{C1'}$ ne concernant pas des objets modifiés dans HS_{E1} :

a) Si T est une transformation concernant les clés d'accès ou les collections, elle est ajoutée à HS_{EP1}.

b) Si T est une transformation à sémantique constante, cela signifie que le concepteur décide d'appliquer une nouvelle transformation sur un objet O non modifié de S1. On parcourt $HS_{C0"}$ pour trouver les transformations concernant O. On compare les objets représentant O dans SP0 et ceux le représentant dans SP1. S'ils sont de structures équivalentes (c'est-à-dire que les transformations utilisées dans $HS_{CO"}$ et $HS_{C1'}$ sont identiques), on ajoute à HS_{EP1} les modifications des objets représentant O dans SP1 par rapport à ceux le représentant dans SP0. S'ils ne sont pas de structures équivalentes (c'est-à-dire que les transformations utilisées dans $HS_{CO"}$ et $HS_{C1'}$ ne sont pas identiques), on ajoute à HS_{EP1} la transformation qui permet de passer des structures représentant O dans SP0 à celles le représentant dans SP1.

Par cet algorithme, HS_{EP1} est le nouvel historique simplifié d'évolution physique qui appliqué sur SP0 donnera le nouveau schéma physique SP1.

5.6.4 Construction de la liste des modifications

5.6.4.1 Introduction

L'historique simplifié des modifications physiques HS_{EP1} est très général. Jusqu'à présent, on n'a pas tenu compte de la technologie utilisée pour gérer la base de données. Chaque SGBDR a ses spécificités. Le changement de nom d'un type d'entités peut être possible dans un système, alors que, dans un autre, il faut passer par la création d'un nouveau type d'entités et la destruction de l'ancien après le transfert des instances. Et même, au sein d'une famille de SGBD, il existe des divergences. Pour s'en convaincre, il suffit de comparer les manuels de référence du langage SQL de différents SGBDR. Chaque constructeur adapte la norme à ses besoins en ajoutant de nouvelles fonctionnalités ou en passant sous silence certaines fonctionnalités décrites dans la norme.

A partir de ce constat, il semble nécessaire d'intégrer, dans le processus de conversion de données, une phase de restructuration des modifications dépendant des caractéristiques techniques du SGBD choisi de manière à produire des scripts de conversion efficaces.

Les réorganisations proposées sont adaptées à la famille des SGBD relationnels. C'est la norme SQL-92 qui a été prise en compte pour éviter au maximum les spécificités propres à chaque constructeur. Il est toujours possible d'améliorer la restructuration en vue de satisfaire les exigences d'un système propriétaire.

Dans un premier temps, on explique le besoin de construire une liste des modifications pour restructurer un historique simplifié (point 5.6.4.2). Ensuite, les différentes restructurations possibles

sur un historique simplifié sont détaillées (point 5.6.4.3). Finalement, le point 5.6.4.4 présente le principe de restructuration et de construction de la liste des modifications.

5.6.4.2 Nécessité de construire une liste des modifications

Au terme de cette phase, il faut fournir au générateur de script de conversion une liste de modifications. Le générateur parcourt cette liste et génère pour chaque modification une ou plusieurs requêtes SQL. Dans HS_{EP1}, il existe de fortes interactions entre certaines modifications. Pour illustrer un type d'intéractions, prenons l'exemple suivant. Soit HS = <..., E←creer-TE(S,n), ..., C←creer-Coll(S,n,{E,...}), ...>. L'espace de stockage C contient certaines tables dont E. En SQL, une table doit être créée dans l'espace de stockage (concrètement une table doit être vide pour être attachée à un espace de stockage). Si on garde l'historique comme tel, après la création de l'espace de stockage C, il faut détruire la table E (elle a déjà été créée précédemment) et la recréer dans l'espace C. Si la création de C était avant celle de E, on ne créerait qu'une seule fois la table. La meilleure solution consiste à mettre dans l'historique les créations des espaces de stockage avant celles des tables.

Les interactions entre modifications nécessitent la restructuration de l'historique ce qui va entraîner de profonds changements. Cette restructuration peut engendrer des simplifications qui empêchent l'historique simplifié d'être rejoué sur SP0 pour obtenir SP1. L'appellation historique pour le résultat de la restructuration est abusive. Le résultat de la restructuration constitue plutôt une liste de modifications. L_{EP1} est la liste des modifications qui est le résultat de la restructuration de HS_{EP1}. Elle est constituée comme HS_{EP1} des signatures des modifications.

5.6.4.3 Restructurations des modifications

Il existe trois types d'interactions nécessitant des restructurations :

- entre les modifications relatives à une table et celles relatives à ses éléments (point a),
- entre les modifications relatives à un espace de stockage et celles relatives à ses tables (point b),
- entre les modifications relatives à un groupe et celles relatives à ses composants ou les groupes avec qui le groupe est impliqué dans une contrainte référentielle (point c).

L'analyse de ces interactions a permis de dégager deux sortes de restructurations. La première consiste à réorganiser les modifications relatives aux objets qui interagissent de manière à éviter des redondances [6] dans les instructions générées par le convertisseur. La réorganisation est le terme générique désignant cette restructuration. La deuxième simplifie L_{EP1} en supprimant une ou plusieurs modifications relatives aux objets qui interagissent. Une simplification (terme employé pour désigner ce type de restructuration) engendre une liste où toutes les informations pour la génération des scripts ne sont plus présentes puisque HS_{EP1} est déjà un historique simplifié et minimal. Cela ne pose aucun problème puisqu'il y a moyen de retrouver les informations perdues dans les spécifications de SP0 et SP1 qui sont aussi disponibles pour la génération. C'est la raison pour laquelle on ne parle pas de la construction d'un historique restructuré mais bien de la construction d'une liste de modifications.

a) Modifications relatives à une table et ses éléments

Par éléments d'une table, on entend ses colonnes et ses groupes (identifiants, contraintes diverses et index). Le cas des contraintes référentielles est traité de manière spécifique au point c).

Pour les tables et leurs éléments, deux restructurations sont possibles :

1. En SQL, la commande de création d'une table doit contenir au moins la création d'une colonne. De plus, il est plus simple et efficace de créer la table avec tous ses éléments plutôt

6. Le terme de redondance est employé à ce niveau pour désigner une ou plusieurs instructions SQL identiques qui apparaissent à plusieurs endroits d'un script.

que de créer les colonnes et autres contraintes au fur et à mesure qu'elles sont rencontrées dans la liste des modifications par le biais de la requête ALTER TABLE. La règle est : si une table est créée dans L_{EP1}, les modifications (destruction, création ou modification) concernant ces éléments ne sont pas ajoutées à L_{EP1}.

2. La fonction qui renomme une table n'existe pas dans certains SGBDR. La modification consiste à créer la table avec le nouveau nom, transférer les instances et détruire l'ancienne table. La règle de restructuration qui en découle est : toutes les modifications relatives aux éléments (colonnes et contraintes) de la table ainsi que celles relatives aux clés étrangères qui la référencent sont inutiles car elles seront réalisées lors de la création de la nouvelle table. La modification d'une table enlève de L_{EP1} les modifications (destruction, création ou modification) relatives à ses éléments. Si le SGBDR possède une commande spécifique pour renommer une table, la règle énoncée ci-dessus n'est plus d'application.

b) Modifications relatives à un espace de stockage et ses tables

La gestion des modifications relatives aux espaces de stockage respecte les restrictions suivantes :

- Pour détruire un espace de stockage, il faut que les tables qu'il contient soient vides sinon les instances des tables sont perdues.
- Pour renommer un espace de stockage, il faut le détruire puis le recréer avec son nouveau nom.
- On ne peut transférer une table dans un espace de stockage que s'il est vide (donc au moment de sa création).

Il existe de fortes interactions entre les modifications relatives à un espace de stockage et aux tables lui appartenant. Pour avoir un script de conversion efficace, la liste des modifications doit respecter les deux règles de restructuration suivantes :

1. Une création d'espace doit être placée avant les modifications (création ou modification) relatives à ses tables et une destruction doit être placée après les modifications relatives à ses tables. Cela permet de créer les tables modifiées directement dans l'espace auquel elles appartiennent ou hors de l'espace auquel elles n'appartiennent plus.
2. Pour toutes les tables appartenant à un espace modifié (créé, renommé ou détruit) et qui ne font pas l'objet d'une modification (création ou modification), il faut ajouter à L_{EP1} la modification qui consiste à créer une table temporaire dans laquelle seront stockées les instances de la table, à détruire la table et la recréer dans son espace éventuel, à transférer les instances de la table temporaire dans la table recréée et à détruire la table temporaire.

Pour satisfaire ces règles, il faut mettre en début de L_{EP1} les créations des espaces modifiés dans HS_{EP1} et, à la fin, les destructions, ensuite, il faut ajouter à L_{EP1} les modifications des tables non modifiées dans HS_{EP1} mais qui appartiennent aux espaces modifiés de HS_{EP1}.

c) Modifications relatives à un groupe et ses composants

La gestion des groupes [7] est entravée par les restrictions suivantes :

- La modification d'une colonne appartenant à un groupe entraîne la destruction du groupe si cette modification passe par la destruction et la recréation de la colonne (ce qui est le cas pour la plupart des modifications relatives à une colonne).
- Pour modifier un groupe (renommer, ajouter ou enlever des composants, changer le type de contrainte), il faut le détruire puis le recréer.

7. La notion de groupe est utilisée pour désigner les contraintes (clé primaire, clé candidate, clé étrangère, index, ...) définies sur les colonnes d'une table.

Les modifications relatives à un groupe et les colonnes qui lui appartiennent sont dépendantes. L'exemple suivant illustre les problèmes engendrés par les restrictions décrites ci-dessus. Supposons que HS_{EP1} contient la création d'une clé étrangère F vers un clé primaire I et la modification d'une colonne A faisant partie de I. On a HS_{EP1} = <..., F←creer-FK(E1,n,{...}), ..., A←modifier-Att(E2,A,n'), ...>. F est créé et ensuite automatiquement détruit lorsque I est détruit pour être reconstruit avec A. Ce problème peut être résolu en déplaçant la création de F après la modification de A. Si plusieurs colonnes de I ou même de F sont modifiées, le déplacement de la création de F doit se faire après la dernière modification d'une colonne appartenant à I ou F.

Pour rendre le script de conversion efficace, il faut respecter deux règles de restructuration :

1. La création d'un groupe doit être placée après les modifications relatives à ses colonnes.
2. La destruction d'un groupe doit être placée avant les modifications relatives à ses colonnes car, en SQL, une colonne ne peut être modifiée si elle appartient à une contrainte (à moins d'utiliser la clause CASCADE qui détruit les contraintes auxquelles participe la colonne).

Pour satisfaire ces règles, il faut mettre au début de L_{EP1} les destructions des groupes détruits ou modifiés dans HS_{EP1} et, à la fin, les créations des groupes créés ou modifiés dans HS_{EP1}.

5.6.4.4 Principe de construction de la liste restructurée

Sur la base des règles énoncées au point 5.6.4.3, l'algorithme de création de L_{EP1} est décomposé en quatre parties :

- La première crée les listes temporaires (point a).
- La deuxième parcourt HS_{EP1} pour remplir les listes de travail en restructurant les modifications (point b).
- La troisième affine les listes temporaires pour régler les derniers problèmes (point c).
- La quatrième construit L_{EP1} par concaténation des listes temporaires (point d).

a) Création des listes temporaires

On définit les listes suivantes :

- L_{DC} : liste des destructions des espaces de stockage,
- L_{CC} : liste des créations des espaces de stockage,
- L_{TC} : liste des modifications des tables non modifiées dans HS_{EP1} mais qui appartiennent aux espaces créées dans L_{CC}.
- L_{DG} : liste des destructions des groupes,
- L_{CG} : liste des créations des groupes,
- L_{MOD} : liste de toutes les autres modifications.

b) Remplissage des listes temporaires à partir de HS_{EP1}

On parcourt HS_{EP1} et, pour chaque modification rencontrée, on réalise les restructurations suivantes :

- Création d'un espace de stockage :
  - Il faut ajouter la création dans L_{CC}.
  - Pour toutes les tables appartenant à l'espace créé qui ne sont pas créées ou renommées dans HS_{EP1}, il faut ajouter les modifications de ces tables dans L_{TC} si elles n'y sont pas déjà.
- Destruction d'un espace de stockage :

  - Il faut ajouter la destruction dans L_{DC}.
  - Pour toutes les tables appartenant à l'espace détruit qui ne sont pas créées ou renommées dans HS_{EP1}, il faut ajouter les modifications de ces tables dans L_{TC} si elles n'y sont pas déjà.
- Modification d'un espace de stockage :
  - L'espace est renommé : il faut ajouter la création du nouvel espace dans L_{CC} et la destruction de l'ancien dans L_{DC}. Pour toutes les tables appartenant à l'espace renommé et qui ne sont pas créées ou renommées dans HS_{EP1}, il faut ajouter les modifications de ces tables dans L_{TC} si elles n'y sont pas déjà.
  - Des tables sont ajoutées ou enlevées de l'espace : pour toutes ces tables qui ne sont pas créées ou renommées dans HS_{EP1}, il faut ajouter les modifications de ces tables dans L_{TC} si elles n'y sont pas déjà.
- Création ou modification d'une table :
  - Il faut ajouter la création ou modification de la table à L_{MOD}.
  - Il faut ajouter à L_{CG} la création des clés étrangères de la table ainsi que celles qui référencent la table créée. Les autres groupes (clés primaires et candidates, index et autres contraintes) sont créés avec la table.
- Transformation (passage d'une structure à une autre) :
  - Il faut ajouter la transformation de la table à L_{MOD}.
  - Il faut ajouter à L_{CG} la création des clés étrangères des éventuelles tables créées lors de la transformation ainsi que les clés étrangères référençant ces mêmes tables.
- Destruction d'un groupe : il faut ajouter la destruction du groupe à L_{DC} si la table contenant le groupe n'est pas renommée ou créée dans HS_{EP1}.
- Création d'un groupe : il faut ajouter la création du groupe à L_{CC} si la table contenant le groupe n'est pas renommée ou créée dans HS_{EP1} (excepté pour une clé étrangère qui est ajoutée à L_{CC}).
- Modification d'un groupe : il faut ajouter la création du groupe à L_{CC} et la destruction à L_{DC} si la table contenant le groupe n'est pas renommée ou créée dans HS_{EP1} (excepté pour une clé étrangère qui est ajoutée à L_{CC}).
- Création d'un attribut : il faut ajouter la création de l'attribut à L_{MOD} si la table contenant l'attribut n'est pas renommée ou créée dans HS_{EP1}.
- Modification d'un attribut :
  - Il faut ajouter la modification de l'attribut à L_{MOD}.
  - Il faut ajouter la création des groupes contenant l'attribut modifié à L_{CC} et la destruction de ces mêmes groupes à L_{DC} si la table contenant le groupe n'est pas renommée ou créée dans HS_{EP1} (excepté pour les clés étrangères contenant la colonne qui sont ajoutées à L_{CC}).
- Les autres modifications (non reprises ci-dessus) sont ajoutées telles quelles à L_{MOD}.

Précisons que l'ajout de créations de groupe à L_{CC} et de destructions de groupe à L_{DC} ne se fait pas toujours en fin de liste. En effet, la création des identifiants doit précéder la création des clés étrangères qui les référencent. Il faut donc ajouter la création des clés primaires et candidates en début de liste et celle des clés étrangères en fin de liste. Pour L_{DC}, c'est le contraire. Il faut d'abord détruire les clés étrangères avant les clés primaires et candidates (à moins d'utiliser la

clause CASCADE qui détruit les clés étrangères référençant la clé primaire ou candidate détruite). Les autres groupes (index, autres contraintes) peuvent être ajoutés en fin de liste.

c) Restructuration des listes temporaires

Deux restructurations doivent encore être effectuées sur les listes temporaires. Elles concernent les tables qui ont été ajoutées à L_{TC}. Tout d'abord, il faut parcourir L_{TC} pour ajouter à L_{CG} (si elles n'y sont pas déjà) les créations des clés étrangères qui appartiennent aux tables modifiées dans L_{TC} ou qui référencent un identifiant de ces mêmes tables. Ensuite, il faut parcourir L_{DC} et L_{CC} pour enlever la création et la destruction des autres groupes (exceptés les clés étrangères) appartenant aux tables modifiées dans L_{TC}.

d) Création de la liste définitive des modifications

Finalement, on obtient la liste définitive des modifications physiques utilisée pour la conversion des données : $L_{EP1} = L_{CC} + L_{DG} + L_{MOD} + L_{TC} + L_{CG} + L_{DC}$.

5.6.5 Génération des scripts de conversion

5.6.5.1 Principe

Sur la base des schémas physiques SP0 et SP1, un générateur parcourt la liste des modifications L_{EP1} et, pour chaque modification, il produit un script de conversion des structures de données et des données en SQL standard. L'ensemble de ces scripts forme le convertisseur que l'analyste doit exécuter pour modifier la base de données. Les scripts correspondant à toutes les modifications de L_{EP1} sont détaillés dans l'annexe C (points C.2.5, C.3.4 et C.4.4).

A ce stade de la conversion des données, il convient de tempérer le caractère automatique de la conversion. En effet, certaines instances sont dépendantes de modifications [8] de L_{EP1} qui causent la perte d'information ou la violation de contraintes dans la base de données.

Les scripts de conversion correspondant à de telles modifications fournissent des requêtes pour vérifier la violation des contraintes avant la conversion. Sur la base des résultats de ces requêtes, le concepteur doit assurer la cohérence de la base de données après sa modification. Il peut agir de trois manières différentes pour assurer cette cohérence :

1. Les données incriminées sont, soit supprimées car elles ne sont plus utiles, soit mises de côté pour être traitées par la suite. Par exemple, si un identifiant est créé sur une colonne existante, les instances de la table peuvent violer la contrainte d'unicité. Le script doit stocker dans une table temporaire, les lignes ne respectant pas la contrainte (une autre solution serait de laisser la première instance dans la table et transférer les autres dans une table temporaire).
2. Le concepteur corrige les données pour qu'elles vérifient à nouveau les contraintes. Les données erronées proviennent souvent de problèmes d'encodage (par exemple, un même client est enregistré deux fois dans la base de données avec le même numéro).
3. Le concepteur décide de ne pas exécuter le script d'une modification suite au résultat de la requête de vérification des contraintes. Dans ce cas, le script correspondant doit être enlevé du convertisseur et la modification traitée manuellement.

Pour assurer au concepteur une capacité de réaction suffisante, le convertisseur est scindé en deux parties. Dans une première partie, le générateur met toutes les requêtes de vérification des contraintes nécessitées par les modifications de L_{EP1} ainsi que les solutions proposées pour

8. Il existe des modifications indépendantes et dépendantes des données (voir point 4.1.3.1). Dans le chapitre 6, la table 6.2 (page 168) récapitule le caractère dépendant ou indépendant pour toutes les modifications répertoriées dans la typologie.

résoudre les conflits. Les scripts de conversion sont stockés dans la deuxième partie. Cette découpe permet de différencier les tâches nécessitant l'intervention du concepteur de celles complètement automatisées. Le concepteur exécute, d'abord, pas à pas, les requêtes de vérification des contraintes et résout les conflits. Ensuite, il lance l'exécution des scripts de conversion en ayant pris soin d'enlever les requêtes qui violeraient les nouvelles contraintes de la base de données.

5.6.5.2 Exemple

Pour illustrer le principe de conversion des données, reprenons l'exemple intuitif de l'introduction (section 1.4) en considérant la transformation de la figure 5.9, du type T^+ (extension de la cardinalité maximum d'un rôle).

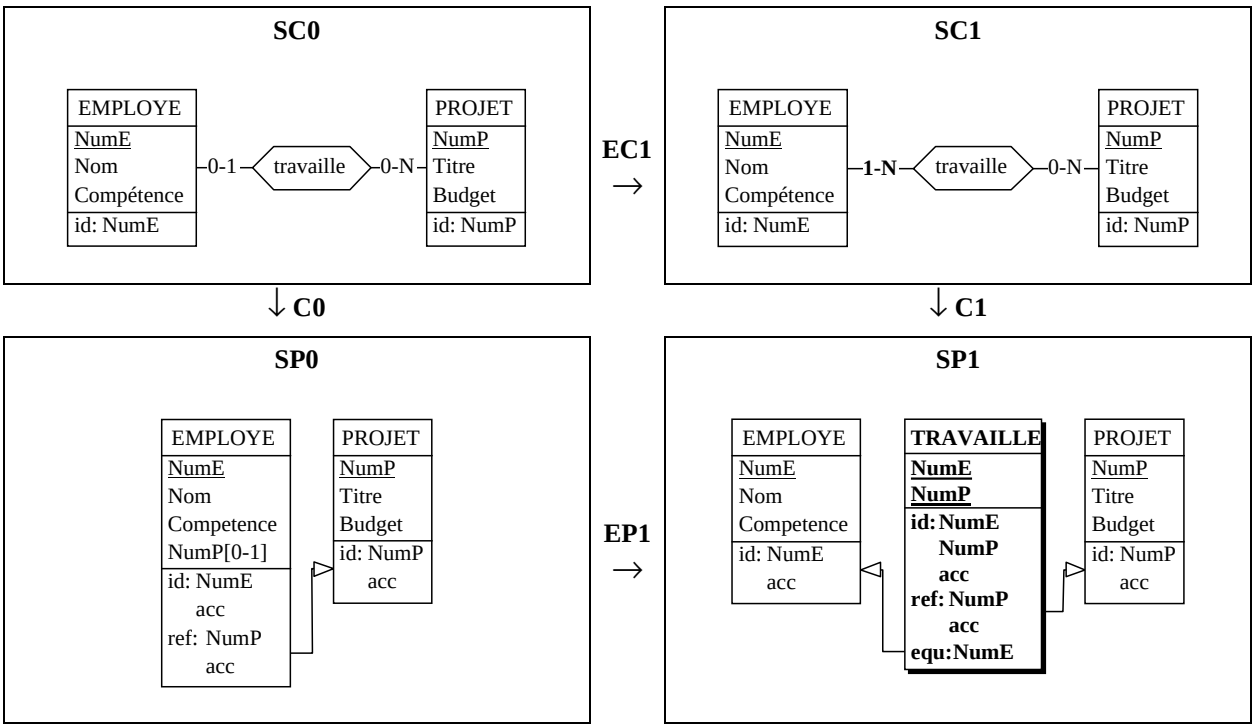


Figure 5.9 - Modification de la cardinalité du rôle *travaille.EMPLOYE* dans le schéma SC1 et propagation de cette modification en aval jusqu'au niveau physique (SP1).

La modification de la cardinalité a entraîné la modification du type d'associations fonctionnel (un à plusieurs) *travaille* en type d'associations complexe (plusieurs-à-plusieurs). *travaille* ne peut plus être transformé en clé étrangère mais doit être transformé en type d'entités. L'analyse de C0 indique que le type d'associations *travaille* était représenté dans SP0 par un attribut de référence *NumP* dans *EMPLOYE* et une clé étrangère vers l'identifiant de *PROJET*. L'analyse de C1 nous apprend que *travaille* est maintenant représenté par une table *TRAVAILLE* contenant deux clés étrangères (une vers *PROJET* et l'autre vers *EMPLOYE*).

On dispose des historiques simplifiés :

HS_{EC1} = <(travaille,EMPLOYE) ← modifier-Role(travaille,EMPLOYE,1-N)>
HS_{C0} = <(EMPLOYE,{NumP}) ← TA-en-FK(travaille)>
HS_{C1} = <
  (TRAVAILLE,{trav_pro,trav_emp}) ← TA-en-TE(travaille)
  (TRAVAILLE,{NumP}) ← TA-en-FK(trav_pro)
  (TRAVAILLE,{NumE}) ← TA-en-FK(trav_emp)
>

L'analyse des historiques HS_{CO} et HS_{C1} permet de traduire l'historique d'évolution conceptuelle HS_{EC1} en HS_{EP1} qui rejoué sur SP0 donne le nouveau schéma physique SP1 :

HS_{EP1} = <(TRAVAILLE) ← FK-en-TE(EMPLOYE,{NumP})>

La restructuration de HS_{EP1} donne la liste des modifications :

```
L_EP1 = <
  TRAVAILLE ← FK-en-TE(EMPLOYE,{NumP})
  fkprojet ← creer-FK(TRAVAILLE,{NumP},{PROJET.NumP})
  fkemploye ← creer-FK(TRAVAILLE,{NumE},{EMPLOYE.NumE},egal)
>
```

L'analyse de L_{EP1} avec SP0 et SP1 donne deux scripts (instructions SQL). Le premier script vérifie l'appartenance de chaque employé à un projet :

```
SELECT * FROM EMPLOYE WHERE NumP is NULL;
```

Le deuxième script convertit les structures de données et les données :

```
-- 1. TRAVAILLE ← FK-en-TE(EMPLOYE,{NumP})
-- Création de la table TRAVAILLE :
CREATE TABLE TRAVAILLE (NumP NUMERIC(5) NOT NULL,NumE NUMERIC(5) NOT NULL,
                        CONSTRAINT IDTRAVAILLE PRIMARY KEY(NumP, NumE));
CREATE UNIQUE INDEX IDXTRAVAILLE ON TRAVAILLE (NumP, NumE);
CREATE INDEX IDXPROJET ON TRAVAILLE (NumP);
-- Transfert des instances :
INSERT INTO TRAVAILLE (NumP,NumE) (SELECT Nump,NumE FROM EMPLOYE);
-- Destruction de la colonne NumP de la table EMPLOYE :
ALTER TABLE EMPLOYE DROP NumP;
-- 2. fkprojet ← creer-FK(TRAVAILLE,{NumP},{PROJET.NumP})
ALTER TABLE TRAVAILLE ADD CONSTRAINT FKPROJET
                      FOREIGN KEY (NumP) REFERENCES PROJET (NumP);
-- 3. fkemploye ← creer-FK(TRAVAILLE,{NumE},{EMPLOYE.NumE},egal)
ALTER TABLE TRAVAILLE ADD CONSTRAINT FKEMPLOYE
                      FOREIGN KEY (NumE) REFERENCES EMPLOYE (NumE);
ALTER TABLE EMPLOYE ADD CONSTRAINT EQTRAVAILLE
                    CHECK(EXISTS(SELECT * FROM TRAVAILLE
                                 WHERE TRAVAILLE.NumE = EMPLOYE.NumE));
```

5.7 Modification des programmes

5.7.1 Complexité du problème

Modifier les programmes s'avère une tâche beaucoup plus complexe que la conversion des données et structures de données, sauf dans des situations simples où les modifications sont mineures. Pour rappeler la complexité de la maintenance des programmes, reprenons la section de pseudo-code présentée dans l'exemple de l'introduction au point 1.4.3 (figure 1.7 de la page 29). Elle exprime un traitement TrP appliqué au projet de chaque employé qui a emprunté au moins un ouvrage et un traitement TrE appliqué à chacun de ces employés. Les sections TrP et TrE se présentent sous la forme de séquences d'instructions. La conversion consiste à transformer l'accès à l'unique projet de l'employé courant et la séquence TrP en une itération qui parcourt les projets de l'employé courant. La difficulté réside dans l'identification de la séquence TrP, dont la frontière avec TrE n'est pas toujours aisée à déterminer, d'autant plus que des instructions indépendantes du projet peuvent apparaître dans TrP. Idéalement, seules les instructions dépendant du curseur P devraient subsister dans le corps de la boucle mineure.

5.7.2 Outils de compréhension de programmes

L'automatisation de la propagation des modifications dans les programmes est impossible sauf dans des cas simples (modification de nom par exemple). Pour caractériser l'impact des modifications des structures de données sur les programmes, trois classes de modifications ont été définies dans le chapitre 4. Pour rappel, cette classification est établie sur le degré d'influence d'une modification de structures sur les programmes :

- *Indépendance totale* : certaines modifications n'entraînent aucune modification des programmes. Par exemple, la suppression d'un index altère les performances des requêtes mais n'entraîne pas de modifications des programmes.
- *Indépendance structurelle* : certaines modifications entraînent des modifications légères de programmes. Par exemple, pour renommer une table, un outil classique de recherche/remplacement sur des chaînes de caractères est suffisant pour remplacer l'ancien nom de la table par le nouveau.
- *Dépendance structurelle* : certaines modifications entraînent des modifications conséquentes des programmes. L'exemple de la figure 5.9, dans lequel une clé étrangère est transformée en table, nécessite une restructuration importante des programmes.

Les modifications engendrant une dépendance structurelle au niveau des programmes peuvent être très difficiles à gérer lorsqu'une application est composée de milliers de lignes de code (voire de millions dans le cas de grosses applications) réparties dans plusieurs fichiers. L'absence de techniques standards de programmation rend encore plus difficile l'analyse des sources des programmes. La programmation est tributaire entre autres du niveau de compétence des programmeurs, de l'âge des programmes, du langage de programmation, de la plateforme de travail; autant de contraintes qui gênent la compréhension des programmes.

Même si la propagation des modifications au niveau opérationnel n'est pas automatisable et reste à charge du programmeur, il est possible de préparer son travail par une analyse du code permettant de repérer et d'annoter les sections critiques des programmes. Le recouvrement des spécifications complètes du programme n'est pas nécessaire, il faut retrouver les endroits où le code est susceptible d'être modifié. Il existe des techniques de compréhension de programmes qui permettent de localiser avec une bonne précision le code à modifier. Trois techniques parmi les plus intéressantes ont été choisies pour la localisation des sections à modifier dans les programmes (elles sont approfondies au point 7.1.6) :

1. La recherche dans les textes sources de certains patrons. Un patron [9] est plus qu'une chaîne de caractères, c'est un ensemble de chaînes de caractères possibles. Il inclut des termes génériques (tels que *, ?, ...), des variables pouvant être instanciées, des suites de caractères (0-9, A-Z, ...), des structures multiples. Il peut également être basé sur d'autres patrons. L'utilisation de patrons permet de rechercher des requêtes où les variables contenues dans le patron sont instanciées avec les objets sur lesquels on opère des modifications.
2. Le graphe de dépendances : il s'agit d'un graphe où chaque variable du programme est représentée par un nœud et dont les arcs (orientés ou non) représentent une relation directe (assignation, comparaison, ...) entre deux variables. La signification des relations est définie par l'analyste. Par exemple, les variables ont les mêmes valeurs ou la structure d'une variable est inclue dans celle de l'autre. Le graphe de dépendances utilise les patrons (première technique).
3. La fragmentation de programmes (en anglais "program slicing") : cette technique sélectionne dans un programme toutes les instructions qui contribuent à définir l'état d'une variable à un point déterminé d'un programme. Un fragment de programme contient toutes les instructions et tous les prédicats qui peuvent affecter la valeur d'une variable à un point du programme.

5.7.3 Localisation des sections critiques

Pour aider le concepteur à localiser les sections des programmes où des modifications sont susceptibles d'intervenir, il est possible de générer les patrons nécessaires aux outils de compréhension de programmes et, plus précisément, à l'outil de recherche de patrons à partir des modifications de L_{EP1}.

Pour chaque modification de L_{EP1}, le générateur met dans un fichier les patrons nécessaires à la recherche. Le concepteur introduit les patrons pour une modification dans le moteur de recherche qui analyse les fichiers sources. On peut imaginer le résultat sous la forme de numéros de lignes de code vérifiant le patron ou de lignes colorées dans un éditeur de texte.

Pour certaines modifications, le concepteur est amené à trouver les variables des programmes dépendant des colonnes [10] localisées dans le code. Il utilise le graphe de dépendances qui calcule le graphe des variables et les arcs qui les relient à partir de patrons définissant les relations entre les variables du langage de programmation utilisé. Le concepteur retrouve ainsi les variables dépendant de certaines colonnes.

Finalement, le concepteur doit parfois retrouver les instructions déterminant l'état d'une variable à un endroit d'un programme lorsque le graphe de dépendances ne fournit pas de résultat satisfaisant. Il utilise alors un outil de fragmentation de programmes. Cette technique est très utile lorsqu'il faut, par exemple, introduire des boucles supplémentaires dans des sections de code et déterminer les instructions devant appartenir à la boucle.

Cette section ne présente que les grandes lignes du principe général de localisation des sections de code à modifier. Cette présentation est suffisante pour permettre une compréhension globale de la résolution du problème mais elle est scientifiquement insatisfaisante pour une mise en pratique efficace. En fait, le problème de la localisation est étroitement lié aux outils de compréhension de programmes qui sont expliqués dans l'approche DB-Main. C'est pourquoi une étude approfondie du principe de localisation est proposée dans le chapitre 7 (section 7.4).

Exemple

Pour illustrer ces techniques, reprenons la liste L_{EP1} des modifications apportées aux spécifications de la figure 5.9 et le code qui, pour chaque employé ayant emprunté au moins un ouvrage, affiche ses références et met à jour le budget de son projet en fonction du nombre d'emprunts qu'il a effectué :

9. de l'anglais "pattern" qui signifie patron, modèle.
10. On considère que les colonnes sont les seuls objets des spécifications pouvant être référencés par des variables.

```
1:  Declare cursor E for select EMPLOYE.NumE, count(*) from EMPLOYE, EMPRUNT
                          where EMPLOYE.NumE = EMPRUNT.NumE group by EMPLOYE.NumE;
2:  open E;
3:  fetch E into :ne, :nbr;
4:  while SQLCODE = 0 do
5:    select NumP, Titre into :np, :t from PROJET  where NumP =
                                    (select NumP from EMPLOYE where NumE = :ne);
6:    select Nom into :n from EMPLOYE where NumE := :ne;
7:    if :t then writeln(:ne,'. ',:n,' ',:t);
      else writeln(:ne,'. ',:n,' ','Pas de projet');
8:   update PROJET set Budget = Budget - (20 * :nbr) where NumP = :np;
9:    fetch E into :ne, :nbr;
10: endwhile;
11: close E;
```

Premièrement, l'analyse de L_{EP1} va fournir les patrons pour le moteur de recherche. La première modification `TRAVAILLE←FK-en-TE(EMPLOYE,{NumP})` de L_{EP1} nécessite une recherche sur la table et les colonnes de la clé étrangère, c'est-à-dire *EMPLOYE* et *NumP*. Les patrons sont composés d'une recherche de toutes les requêtes du type :

```
SELECT * FROM ... EMPLOYE ...
SELECT ... NumP ... FROM ... EMPLOYE ...
SELECT ... FROM ... EMPLOYE ... WHERE ... NumpP ...
SELECT ... FROM ... EMPLOYE ... GROUP BY ... NumpP ...
SELECT ... FROM ... EMPLOYE ... HAVING ... NumpP ...
DELETE FROM EMPLOYE WHERE ... NumP ...
UPDATE EMPLOYE SET ... NumP ...
UPDATE EMPLOYE ... WHERE ... NumP ...
INSERT INTO EMPLOYE ...
INSERT INTO EMPLOYE(... NumP ...) ...
```

Les deux autres modifications de L_{EP1} ne donnent aucun patron puisqu'il s'agit d'objets créés. Le moteur de recherche trouve, dans le code, l'instruction de la ligne 5.

Deuxièmement, dans les requêtes trouvées par le moteur de recherche, le concepteur doit chercher les variables qui dépendent de la colonne *NumP*. Il s'agit des requêtes du type :

```
DECLARE C CURSOR FOR SELECT ...
SELECT * INTO ... VAR ... FROM ...
SELECT ... INTO ... VAR ...
SELECT ... FROM ...  GROUP BY ... VAR ...
SELECT ... FROM ...  HAVING ... VAR ...
DELETE FROM ... WHERE ... VAR ...
UPDATE ... SET ... VAR ...
UPDATE ... WHERE ... VAR ...
INSERT INTO ... VALUES(... VAR ...)
```

Après le calcul du graphe de dépendances à l'aide de patrons définissant les relations entre variables, le programmeur dispose de toutes les lignes contenant des variables liées directement ou indirectement à *NumP*. Dans la requête trouvée par le moteur, les variables :np et :t dépendent de *NumP* de la table *EMPLOYE*. On trouve les variables :np et :t aux lignes 5, 7 et 8. Maintenant, la section de code critique est plus ou moins cernée. La modification à charge du programmeur consiste à transformer l'accès à l'unique projet de l'employé courant en une itération qui parcourt les projets de l'employé courant. Une difficulté réside dans l'instruction de la ligne 7 qui doit être scindée en deux parties : l'affichage des informations sur l'employé doit être en dehors de la nouvelle boucle sur les projets de l'employé et l'affichage des informations sur un projet doit être dans la nouvelle boucle.

Troisièmement, pour aider le concepteur à identifier la séquence d'instructions concernant un projet (TrP) dont la frontière avec celle concernant un employé (TrE) n'est pas bien déterminée, l'outil de fragmentation de programmes peut être utilisé pour mettre en évidence toutes les instructions concernant le projet de l'employé courant. L'identification de la séquence TrP permet de la mettre dans une boucle itérant sur les projets de l'employé courant. La technique de la fragmentation des programmes s'utilise lorsque le graphe de dépendances atteint ses limites dans la recherche des dépendances entre variables.

«Saisissons dans le moment présent
celui qui suit.»

Shakespeare

CHAPITRE 6

RECOMMANDATIONS MÉTHODOLOGIQUES

Dans le chapitre 5, des stratégies de propagation des modifications ont été développées pour aider le concepteur à faire évoluer son application. L'objectif est l'adaptation du système d'information pour mieux répondre aux besoins des utilisateurs. Une idée intéressante serait d'approfondir le raisonnement pour voir si les problèmes d'évolution ne peuvent pas être atténués en minimisant l'impact des modifications sur le système par des choix de représentation adéquats.

Le but poursuivi est d'extraire de l'étude réalisée des informations sous la forme de conseils pour aider les analystes à concevoir des applications de bases de données plus faciles à adapter lorsque les besoins évoluent.

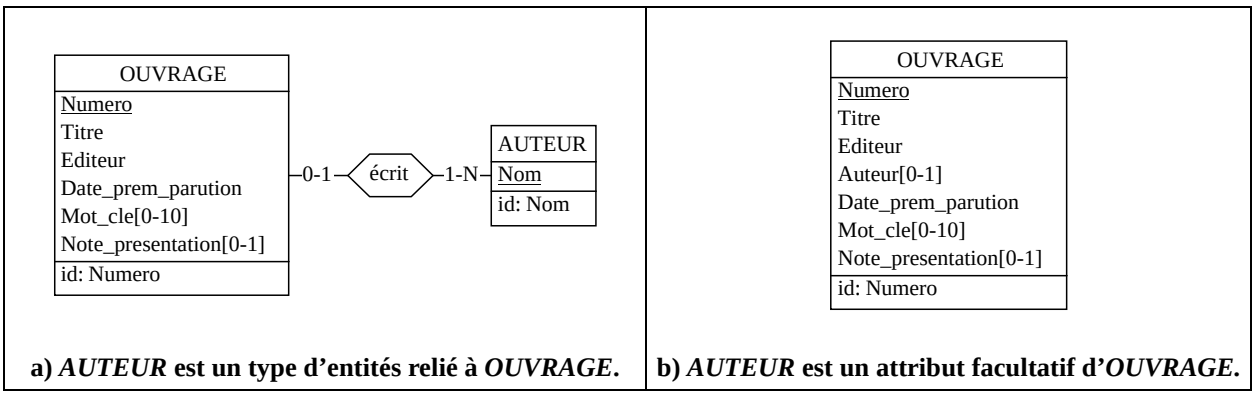
Certaines structures de données représentant le même univers de discours n'ont pas nécessairement la même flexibilité ou capacité d'évolution. Les transformations à sémantique constante utilisées en conception logique génèrent des structures de données qui sont plus ou moins adaptées à une évolution des besoins. Dans ce cadre, la typologie des modifications (chapitre 4) s'avère une source intéressante d'informations à interpréter pour extraire des recommandations pour la conception du composant base de données d'un système d'information. Les transformations décrites dans la section 3.2 sont également utiles pour déterminer la meilleure façon de représenter dans un modèle relationnel un concept en termes de flexibilité.

6.1 Introduction

Depuis les années septante, l'évolution des programmes est un important domaine de réflexion dans l'ingénierie du logiciel. Parnas [Par79] affirmait que les programmeurs devaient rendre leur application plus facile à maintenir en prenant en compte les changements futurs qui pourraient survenir. Pour réaliser cela, il proposait une technique d'encapsulation des informations pour séparer les éléments susceptibles de changer des éléments stables de l'application de manière à mieux localiser les modifications.

La recherche sur l'évolution du composant base de données d'un logiciel est plus récente. Auparavant, le rôle du concepteur d'une base de données était considéré comme passif dans le processus de développement d'un système. Actuellement, il est admis qu'il existe plusieurs schémas conceptuels corrects qui sont construits sur la base d'un ensemble de besoins identiques et qui décrivent le même univers de discours [Bat92] [Moo94]. Certaines façons de concevoir ou programmer une base de données peuvent être plus résistantes à l'évolution des besoins que d'autres. [Moo94] développe une méthode formelle d'évaluation de la meilleur représentation d'un domaine d'application. Il considère la capacité à s'adapter aux changements de son environnement comme une des qualités les plus importantes d'un système. Pour sa part, Verelst [Ver97] pense que le concepteur peut introduire des facteurs d'évolution dans son système d'information pendant le processus de modélisation. Il essaie d'identifier les facteurs de variabilité d'un système d'information, leur influence sur le système et une mesure de cette influence pour tirer des recommandations pour le concepteur.

D'une part, chaque concepteur peut construire son propre schéma conceptuel ayant des différences et des similitudes avec d'autres schémas conceptuels construits sur la base de besoins identiques. Dans la figure 6.1, le concept d'auteur est représenté par un attribut d'*OUVRAGE* ou un type d'entités à part entière. Les deux représentations respectent les mêmes besoins mais n'ont pas la même flexibilité ou capacité d'évolution [1]. La représentation en type d'entités est plus facile à modifier en cas d'ajout d'attributs ou de rôles au concept d'auteur.



a) *AUTEUR* est un type d'entités relié à *OUVRAGE*. **b) *AUTEUR* est un attribut facultatif d'*OUVRAGE*.**

Figure 6.1 - Deux représentations possibles du concept d'auteur.

D'autre part, lors de la conception logique, certaines structures peuvent être transformées de façons différentes tout en gardant la même sémantique. Dans la figure 6.2, l'attribut multivalué *Téléphone* est représenté au niveau logique par deux attributs ou par une table. Si les utilisateurs désirent augmenter la cardinalité maximum de l'attribut *Téléphone*, la première représentation nécessite l'introduction d'une nouvelle colonne tandis que la deuxième représentation est déjà adaptée à la modification. Que ce soit dans la base de données ou dans les programmes, la deuxième représentation est plus facile à modifier pour satisfaire les nouveaux besoins. Il faut remarquer que les deux représentations ne sont pas tout à fait équivalentes car la transformation

1. On entend par capacité d'évolution (ou flexibilité) le pouvoir de s'adapter plus ou moins facilement à des changements dans le domaine d'application.

de *Téléphone* en deux attributs monovalués introduit une notion de tableau qui n'est pas présente dans l'autre représentation.

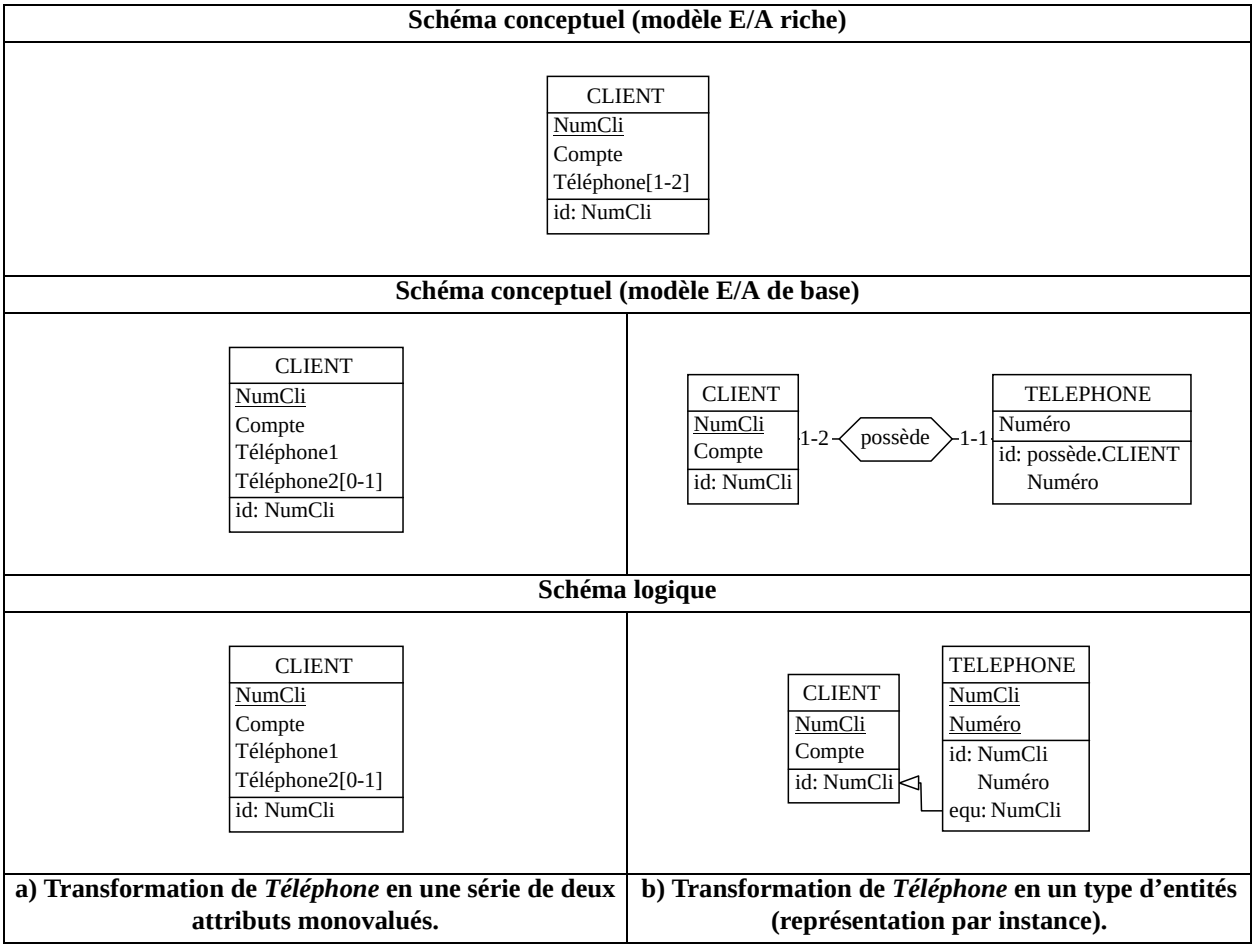


Figure 6.2 - Deux transformations à sémantique équivalente de l'attribut multivalué *Téléphone*.

Le concepteur fait des choix à deux niveaux : dans la phase d'analyse conceptuelle et dans la phase de conception logique. Ces choix multiples engendrent la création de schémas logiques (et physiques) très différents en fonction des analystes.

La section 6.2 décrit brièvement une méthodologie de conception d'un schéma conceptuel et montre la variété de solutions possibles construites à partir de besoins similaires. La section 6.3 analyse la flexibilité des spécifications engendrées par des transformations de conception logique. Finalement, la section 6.4 évalue la capacité d'évolution des modifications (décrites dans la typologie) en termes de structures.

6.2 Règles d'expression d'un concept sous la forme de spécifications E/A

6.2.1 Introduction

L'expression d'une notion réelle sous la forme de structures Entité/Association impose au concepteur la nécessité de réaliser des choix de conception. Telle notion représente-t-elle un concept important du domaine d'application ou simplement une propriété d'un autre concept ? De telles questions apparaissent régulièrement au cours du processus de conception. L'objectif de cette section est de montrer que les choix de conception peuvent avoir une influence sur la capacité d'évolution du système final et d'extraire de l'analyse des recommandations de conception.

Le point 6.2.2 présente brièvement une méthodologie de conception basée sur des stratégies incrémentales et par raffinement. Un exemple de conception est commenté dans le point 6.2.3. A partir d'un énoncé commun, deux schémas conceptuels très différents sont construits. Finalement, le point 6.2.4 énonce quelques recommandations pour le processus de conception qui tiennent compte de la capacité d'adaptation aux changements des besoins.

6.2.2 Elaboration d'un schéma conceptuel

Il existe différentes méthodologies pour la conception du composant base de données d'un système d'information. Citons, entre autres, [Bat92], [Bod89] et [Bla98] pour la construction d'un schéma conceptuel. Beaucoup de méthodes de conception de systèmes d'information, telles que MERISE [Nan96] sur le marché français, ont adopté le modèle E/A. D'autres modèles tels que NIAM [Hal95], basés sur des associations binaires entre objets, ont également fait leur place sur le marché. Plus particulièrement, [Bat92] définit la construction d'un schéma E/A comme un processus incrémental dans lequel la perception de la réalité est progressivement raffinée et enrichie, et le schéma conceptuel est graduellement développé. Sa méthodologie est basée sur des primitives de raffinement qui sont des ensembles de transformations qui sont appliquées à un schéma initial pour produire le schéma final.

La méthodologie présentée dans ce point s'apparente à celle développée dans [Bat92]. Les lecteurs intéressés en trouveront une analyse complète dans [Hai86], [Hai00a] et [Hai00b]. Son objectif est de formaliser sous la forme de schémas E/A les concepts sous-jacents aux diverses sources d'information concernant les besoins nouveaux des utilisateurs. Ces sources peuvent être des documents de toute nature. L'élaboration du schéma conceptuel est décomposée en trois phases :

1. La décomposition de l'énoncé : les sources d'informations de type texte sont décomposées en propositions élémentaires décrivant chacune un concept ou un type de fait. [Bod89] et [Nij89] proposent des techniques similaires.
2. L'analyse des propositions élémentaires :
   a) Pertinence : il faut discerner les propositions qui sont du bruit de celles qui sont pertinentes.
   b) Non-redondance : une proposition exprime l'existence de concepts et/ou l'existence de liens entre concepts. Il faut vérifier que ces faits ne sont pas déjà représentés totalement ou en partie dans le schéma en construction.
   c) Non-contradiction : les propositions ne doivent pas être en contradiction. Sinon il faut résoudre le conflit.
   d) Représentation : des propositions élémentaires, on peut extraire des concepts et des liens entre concepts. Un concept qui est important est représenté par un type d'entités alors qu'un concept moins important est représenté par un attribut. Les liens entre un type d'enti-

tés et un attribut se représentent par l'appartenance de l'attribut au type d'entités. Deux attributs sont liés par leur appartenance au même type d'entités. Entre deux types d'entités, on crée soit un type d'associations soit une relation hiérarchique.

3. La finalisation du schéma conceptuel :
   a) Complétude du schéma : on vérifie que les cardinalités des rôles et des attributs, le type des attributs et les identifiants des types d'entités sont bien définis. Les lacunes peuvent être solutionnées par une relecture de l'énoncé, des interviews, la connaissance du domaine ou le bon sens.
   b) Normalisation : un nettoyage du schéma est effectué soit dans un but de simplification soit pour éliminer les anomalies.
   c) Validation : il s'agit de faire valider le schéma par les utilisateurs pour s'assurer qu'il contient tous les concepts pertinents du domaine d'application.
   d) Documentation : chaque objet du schéma reçoit une description qui précise la signification exacte du concept qu'il représente.

6.2.3 Exemple de conception

A partir d'une même source d'information, nous avons construit deux schémas conceptuels différents (point 6.2.3.2) mais respectant les besoins décrits dans l'énoncé (point 6.2.3.1). Il ne s'agit pas de montrer la méthodologie présentée au point 6.2.2, mais de commenter du point de vue de la flexibilité deux schémas conceptuels dissemblables mais appréhendant la même réalité.

6.2.3.1 Enoncé

La gestion d'une compagnie de transports interurbain par autocar (modèle américain du type Greyhound) comporte la gestion des trajets, des véhicules et des clients, y compris les réservations. Cet exemple s'inspire de l'étude de cas de [Bat92].

Un trajet est effectué entre une ville de départ et une ville de destination, non nécessairement distincte de la première. Ce trajet est effectué certains jours de la semaine. On peut imaginer, par exemple, un trajet effectué chaque mardi et chaque jeudi. A chaque trajet est associé un code identifiant.

Outre les trajets réguliers, des trajets spéciaux peuvent être prévus à l'occasion de certains événements, organisés à des dates déterminées, tels que des manifestations sportives, culturelles ou religieuses.

Un trajet est composé de plusieurs tronçons successifs. Chaque tronçon est caractérisé par son départ (ville et heure de départ), et son arrivée (ville et heure d'arrivée). Son prix et sa distance sont connus. Pour chaque trajet, on connaît le nombre de tronçons qui le composent.

Un trajet régulier est effectué chaque semaine aux jours spécifiés de la semaine. Pour chaque trajet effectué, régulier ou spécial, on connaît la date de départ, l'autocar qui l'effectue et son conducteur. Pour chaque tronçon de ce trajet effectué, on connaît le nombre de sièges occupés (fumeur et non fumeur) et le nombre de sièges libres.

6.2.3.2 Deux schémas conceptuels possibles

Le schéma conceptuel de la figure 6.3 contient des trajets, des villes, des tronçons de trajets, des autocars, des conducteurs et des réservations. Un trajet type est découpé en plusieurs tronçons types. A chacun de ces deux concepts correspond le pendant effectué qui représente les trajets et les tronçons qui ont été réellement accomplis. Un trajet peut être de deux types (spécialisation) : régulier ou spécial. Un trajet a une ville d'arrivée et une ville de départ (idem pour les tronçons avec l'heure de départ et l'heure d'arrivée). Un trajet est effectué par un conducteur dans un autocar en suivant une série de tronçons. Pour un tronçon effectué, on peut faire des réservations.

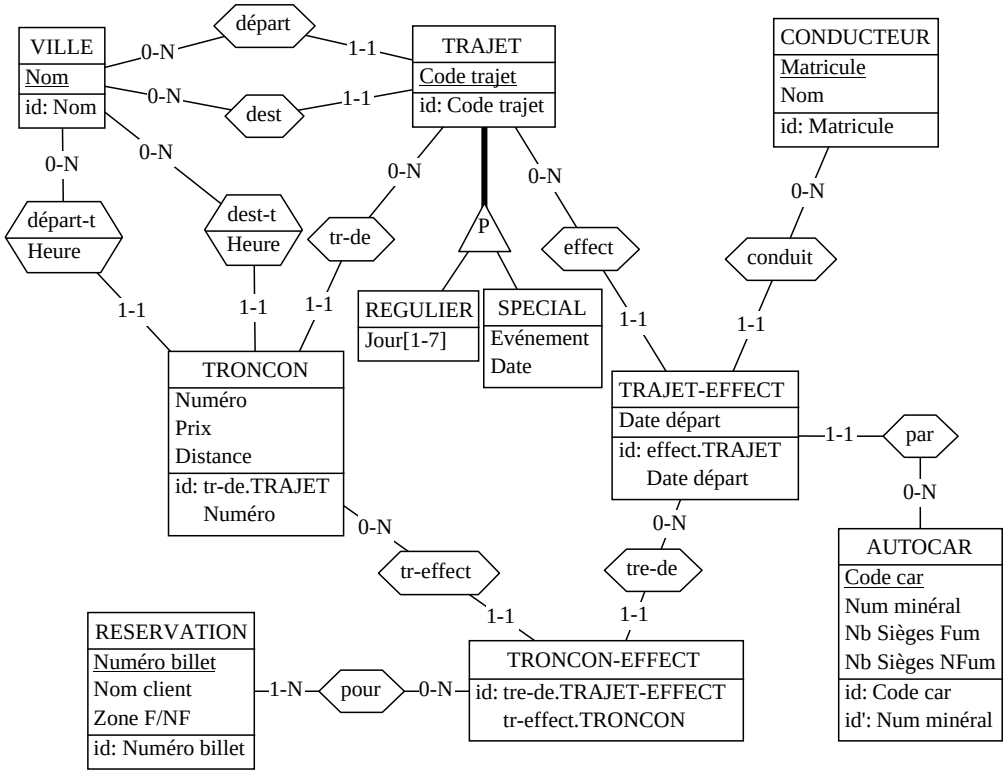


Figure 6.3 - Premier schéma conceptuel représentant les données nécessaires à la gestion d'une compagnie de transports interurbain par autocar.

Le schéma conceptuel de la figure 6.4 contient des trajets, des tronçons, et des réservations. Un trajet est soit régulier soit spécial. Un trajet est découpé en une série de tronçons pour lesquels on peut faire des réservations.

Le deuxième schéma conceptuel contient moins de types d'entités et de types d'associations. Pour son concepteur, les concepts d'autocar et de ville ne semblent pas suffisamment importants pour être représentés par des types d'entités. Intuitivement, le deuxième schéma pose des problèmes surtout au niveau de la flexibilité de ses composants.

Supposons que les utilisateurs du système souhaitent introduire la gestion des réparations des autocars. Dans le premier schéma, il faut simplement ajouter le concept réparation qui est relié au type d'entités *AUTOCAR*. L'existant n'est pas modifié. Seules de nouvelles structures sont ajoutées à la base de données et certaines fonctions sont ajoutées pour prendre en compte les modifications au niveau des programmes. Le deuxième schéma n'est pas adapté à cette modification. L'attribut *Autocar* du type d'entités *TRAJET-EFFECTUE* doit être extrait pour former un type d'entités. En effet, le concept de réparation ne peut pas être ajouté comme un type d'entités relié au type d'entités *TRAJET-EFFECTUE*, sinon le type d'associations entre ces types d'entités doit être de type plusieurs-à-plusieurs. Cela signifie que chaque trajet effectué est relié à toutes les réparations de l'autocar. Comme un autocar effectue plusieurs trajets, les liens avec les réparations sont répétés pour chacun d'eux. On comprend aisément la redondance d'informations dans une telle représentation du domaine d'application. La structure du schéma doit donc être revue en profondeur. Au niveau de la base de données et des programmes, le travail est beaucoup plus important que pour le premier schéma. En plus de la création des nouvelles structures pour la gestion des réparations, il faut créer les structures de données pour la table *AUTOCAR*, transférer les données de la table *TRAJET-EFFECTUE* dans *AUTOCAR* et détruire les colonnes concernant un autocar. Les programmes doivent également être modifiés pour prendre en compte les modifications.

Si les utilisateurs veulent ajouter l'adresse du dépôt des autocars dans chaque ville où ils passent, le premier schéma est également mieux adapté. L'attribut *Adresse* est ajouté au type d'entités *VILLE*. Au niveau de l'application, cela se traduit par l'ajout d'une colonne à une table et la gestion (insertion, suppression et affichage) de cette colonne dans les programmes. Dans le deuxième schéma, il faut répercuter la modification partout où l'on utilise le concept de ville. Non seulement, cela n'est pas très efficace en terme d'espace occupé mais, en plus, il y aura trois fois plus de modifications dans le système par rapport au premier schéma.

Dans le premier schéma, l'augmentation de la cardinalité maximum du rôle joué par *TRAJET-EFFECT* dans le type d'associations *conduit* (un trajet peut être effectué par plusieurs conducteurs) nécessite la transformation du type d'associations (du type plusieurs-à-plusieurs) en type d'entités. Le même problème se pose si on décide d'ajouter un attribut ou un rôle à *conduit*. Le deuxième schéma evolue beaucoup plus facilement puisque *CONDUIT* est déjà représenté par un type d'entités. Les modifications se limitent à augmenter la cardinalité maximum d'un rôle (le type d'associations devenant un-à-plusieurs), ajouter un attribut ou un rôle à un type d'entités. Dans un schéma conforme au modèle E/A de base, ces modifications n'engendrent aucune transformation complexe ce qui facilite leur propagation vers les données et les programmes.

Cet exemple a permis de montrer l'influence que pouvait avoir une représentation des spécifications sur la flexibilité d'un système. Il reste à voir les recommandations qui peuvent être données aux concepteurs de la base.

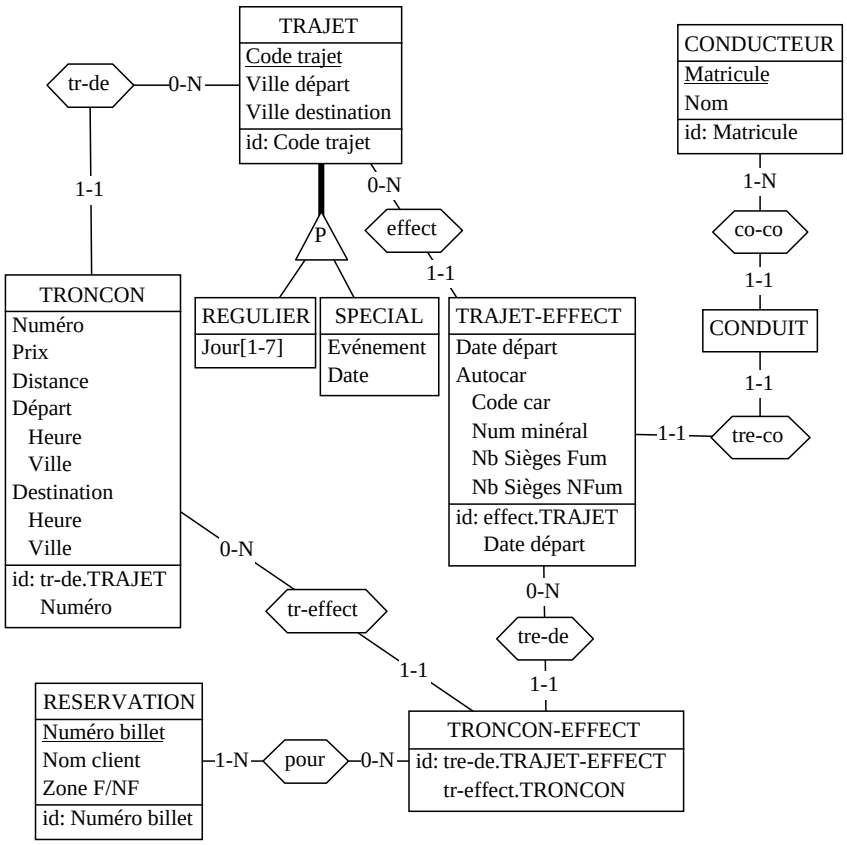


Figure 6.4 - Deuxième schéma conceptuel représentant les données nécessaires à la gestion d'une compagnie de transports interurbain par autocar.

6.2.4 Recommandations pour l'évolution

Il ne s'agit pas de donner des recommandations sur la conception d'un schéma. La littérature regorge d'ouvrages sur la conception de base de données qui tentent d'aider le concepteur dans

sa quête du "meilleur" schéma conceptuel possible ou, du moins, de celui qui lui convient le mieux en lui fournissant des règles de conception pour éviter les pièges courants. Toutefois, la flexibilité d'un schéma conceptuel est un aspect de la conception qui est moins développé. Comme cela a déjà été signalé précédemment, réfléchir à la capacité d'évolution d'un schéma de spécifications au moment de sa création n'est pas dénué de sens.

La flexibilité d'un schéma dépend essentiellement des modifications susceptibles d'intervenir lors du cycle de vie de l'application. Il est inutile de créer un schéma conceptuel très flexible qui donnera un système gourmand en ressources, lent du point de vue des accès et qui rend complexe la programmation. Toute l'habileté du concepteur résidera dans sa capacité à trouver un juste milieu entre la flexibilité et l'efficacité du système. Pour réaliser cet objectif, le concepteur doit posséder une connaissance approfondie du domaine d'application de manière à anticiper les modifications futures engendrées par un changement des besoins. Cette connaissance passe également par un dialogue avec les acteurs (directeurs qui définissent la politique d'une organisation et les utilisateurs du système) de l'entreprise qui vont influencer les modifications futures du système.

Les quelques recommandations énoncées ci-dessous ont pour seul mérite de mettre en évidence des points sensibles dans un schéma conceptuel qui pourraient avoir une influence sur la flexibilité de la base de données. Ces recommandations sont divisées en deux groupes : les choix de représentation d'un objet du domaine et les propriétés des objets.

6.2.4.1 Représentation d'un objet

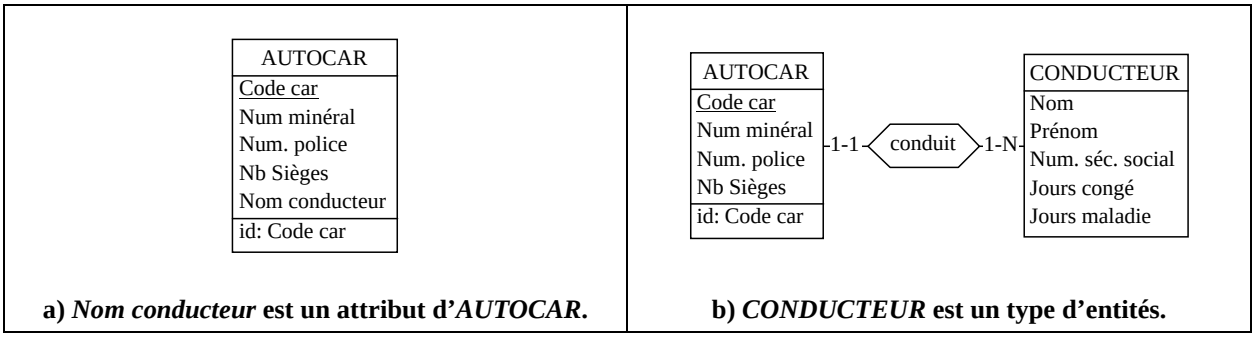
Dans un schéma conceptuel, une information peut être représentée soit par un type d'entités (un objet à part entière), soit un attribut (propriété d'un objet), soit un type d'associations (un objet qui fait le lien entre d'autres objets) ou soit une relation d'héritage (IS-A). Chaque type de représentation est comparé aux autres en termes de capacité d'évolution.

a) Type d'entités ou attribut ?

De manière générale, un concept se présente sous la forme d'un type d'entités s'il apparaît comme important, ou par un attribut s'il apparaît comme une simple propriété d'un autre concept. Deux cas sont envisageables :

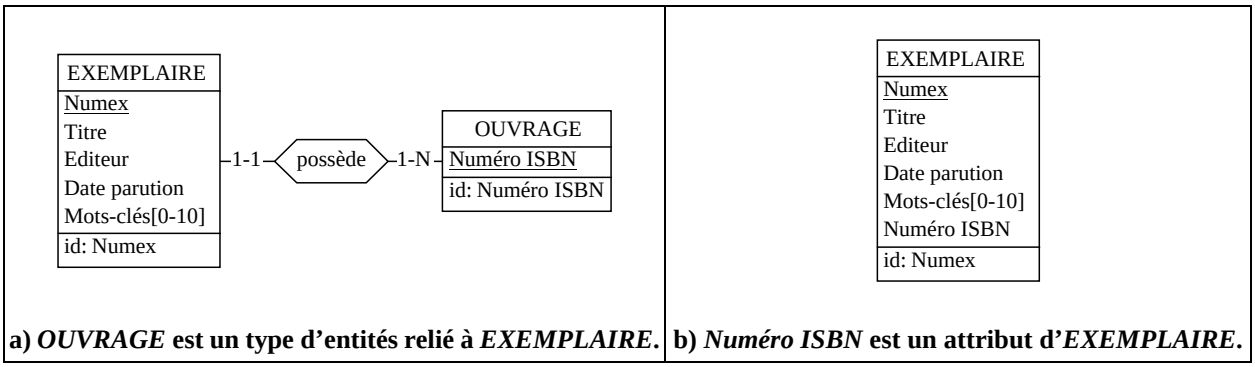
1. Une information est représentée comme un attribut d'un type d'entités. Si ce concept est amené à être enrichi par la suite (ajout de propriétés, augmentation de la cardinalité, possibilité de jouer des rôles dans des types d'associations), il convient d'en faire un type d'entités de manière à minimiser les impacts des modifications futures du schéma conceptuel sur le système d'information. Dans la figure 6.5 a), le concept de conducteur qui conduit un autocar est représenté par un attribut du type d'entités *AUTOCAR*. Pour le service qui organise les trajets, cette représentation est suffisante. Mais, si l'entreprise décide d'intégrer dans la base de données les informations nécessaires au service du personnel, le concept de conducteur devient primordial et nécessite une représentation plus adéquate (figure 6.5 b). C'est une analyse sur le long terme qui permet de dégager la deuxième représentation convenant mieux à la stratégie de l'entreprise.
   Dans le même contexte, un attribut qui est une propriété commune à plusieurs types d'entités peut devenir un type d'entités. La redondance de cette propriété en fait un concept important du point de vue de la flexibilité. Si l'attribut évolue, il faut modifier le schéma conceptuel à tous les endroits où il intervient. Cela multiplie aussi les modifications nécessaires en aval du schéma conceptuel. Dans la figure 6.4, le concept de ville est présent dans les types d'entités *TRAJET* et *TRONCON*. La représentation de la figure 6.3 est plus flexible puisqu'elle modélise la notion de ville comme un type d'entités.
2. Si un type d'entités n'apporte pas plus d'information qu'un attribut, il est préférable de le réduire à un attribut surtout s'il a peu de chance d'être modifié par l'ajout de nouveaux rôles ou d'attributs. Un type d'entités ne contenant qu'un seul attribut et jouant un seul rôle peut simplement être remplacé par un attribut. Dans la figure 6.6, le type d'entités *OUVRAGE* ne

joue qu'un rôle avec le type d'associations *possède* et n'a qu'un seul attribut représentant le numéro ISBN d'un exemplaire d'un ouvrage. Cette information est normalement immuable et, donc, pas susceptible d'évoluer. Il serait plus réaliste de la représenter sous la forme d'un attribut d'*EXEMPLAIRE*. Par contre, si on représente les maisons d'édition au lieu des numéros ISBN, un type d'entités *EDITEUR* s'avère une solution flexible si le schéma de spécifications évolue, par exemple, en ajoutant des propriétés comme l'adresse et le numéro de téléphone ou en introduisant la notion de catalogue proposé par les éditeurs.



a) *Nom conducteur* est un attribut d'*AUTOCAR*. **b) *CONDUCTEUR* est un type d'entités.**

Figure 6.5 - Deux représentations possibles du concept de conducteur d'un autocar.

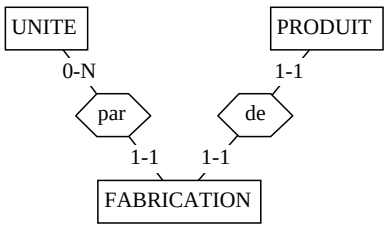


a) *OUVRAGE* est un type d'entités relié à *EXEMPLAIRE*. **b) *Numéro ISBN* est un attribut d'*EXEMPLAIRE*.**

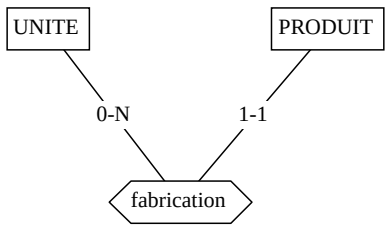
Figure 6.6 - Deux représentations possibles du concept de numéro ISBN d'un exemplaire d'ouvrage.

b) Type d'entités ou type d'associations ?

Une proposition qui établit un lien entre plusieurs concepts représentés par des types d'entités se traduit, en règle générale, par un type d'associations. Si un type d'entités n'apporte pas plus d'information qu'un type d'associations, il peut être réduit à un type d'associations. On repère une telle situation lorsqu'un type d'entités n'a pas d'attribut et qu'il joue des rôles de cardinalité [1-1]. Dans l'exemple de la figure 6.7 a), le type d'entités *FABRICATION* peut être transformé en type d'associations *fabrication* (figure 6.7 b). Si le concept de fabrication est modifié par l'ajout d'un attribut (numéro de fabrication, procédé, ...) ou si la cardinalité du rôle *de.PRODUIT* devient 1-N, le schéma a) ne nécessite pas de transformation dans le modèle E/A de base alors que le type d'associations *fabrication* devra être transformé en un type d'entités. Le schéma a) a une capacité d'évolution plus importante que le schéma b).



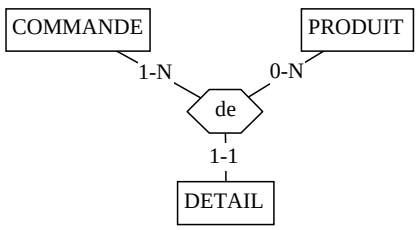
a) *FABRICATION* est un type d'entités relié à *UNITE* et *PRODUIT*.



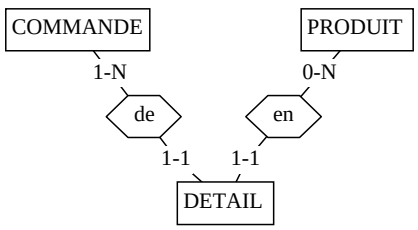
b) *FABRICATION* est un type d'associations.

Figure 6.7 - Deux représentations possibles de la notion de fabrication d'un produit par une unité de fabrication.

Un rôle [0-1] ou [1-1] d'un type d'associations n-aire suggère souvent une décomposition en types d'associations binaires. Dans la figure 6.8 a), le rôle joué par *DETAIL* est de cardinalité [1-1] dans le type d'associations *de*. La décomposition de la figure 6.8 b) représente le même domaine. L'analyse des deux représentations en termes de flexibilité montre que le schéma a) est plus flexible. L'ajout d'attribut au type d'associations *de* ou la modification de la cardinalité du rôle *de.DETAIL* ont peu d'impact sur le schéma a) dans le modèle E/A de base puisque le type d'associations était déjà transformé en un type d'entités. Par contre, les mêmes modifications relatives au schéma b) ont des conséquences plus importantes car les types d'associations *de* et *en* deviennent plusieurs-à-plusieurs (ou possèdent un attribut). Au niveau conceptuel de base, cela nécessite la transformation des types d'associations en types d'entités.



a) *de* est un type d'associations ternaire entre *COMMANDE, DETAIL* et *PRODUIT*.



b) *DETAIL* est relié à *COMMANDE* et *PRODUIT* par des types d'associations binaires.

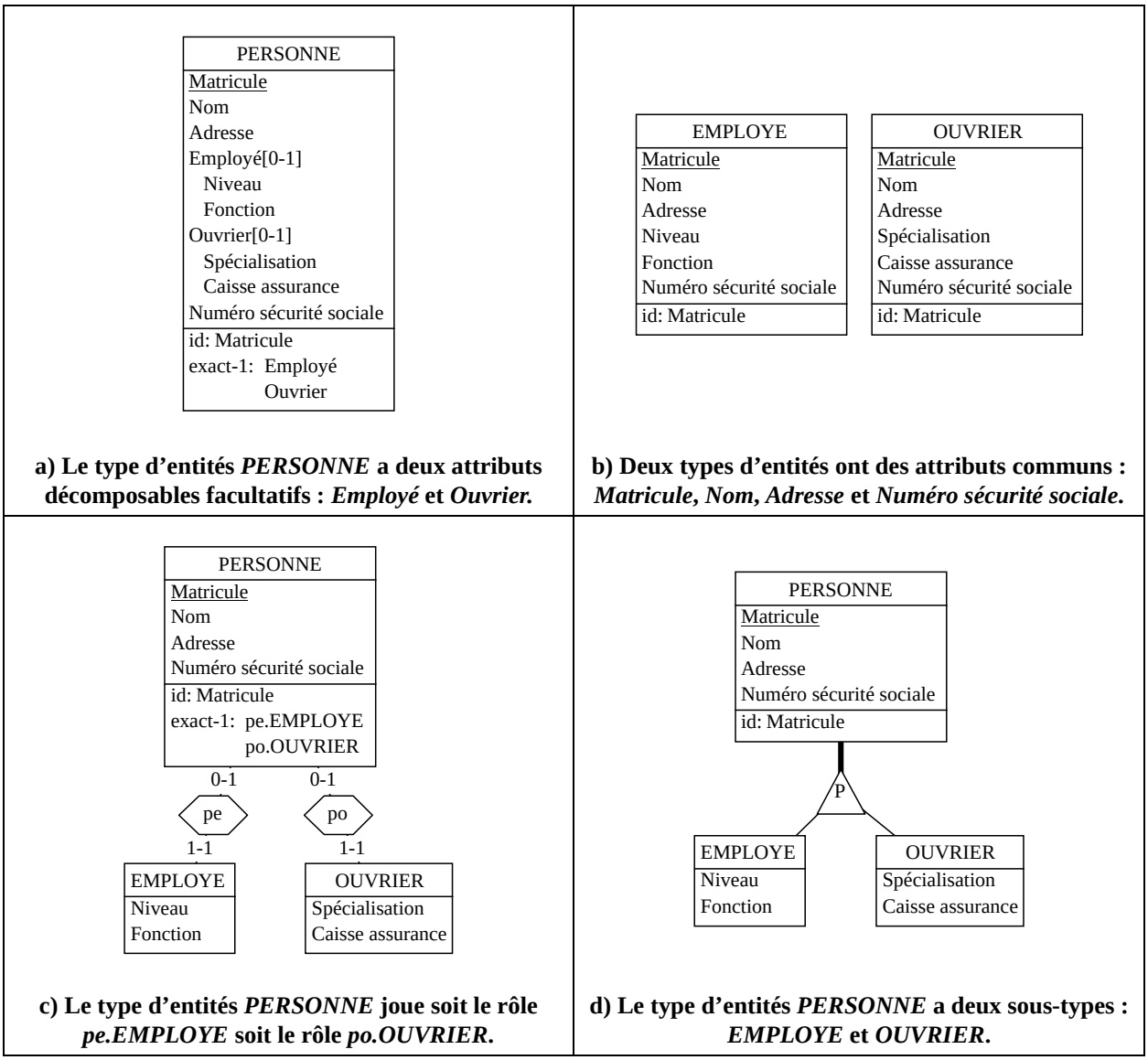
Figure 6.8 - Deux représentations possibles du concept de commande de produits.

c) Types d'entités, types d'associations, attributs ou relations IS-A ?

Certaines constructions cachent des relations IS-A qu'il peut être intéressant de rendre explicites. La figure 6.9 a) présente le type d'entités *PERSONNE* dans lequel deux attributs décomposables facultatifs (*Employé* et *Ouvrier*) sont définis. Une contrainte exactement-un englobe ces deux attributs. Cela signifie que, si *Employé* a une valeur pour une instance de *PERSONNE*, *Ouvrier* ne prend pas de valeur pour cette instance et inversement. Dans la figure 6.9 b), les types d'entités *EMPLOYE* et *OUVRIER* ont des attributs communs. La figure 6.9 c) propose un type d'entités *PERSONNE* jouant un et un seul des deux rôles *pe.EMPLOYE* ou *po.OUVRIER*. Manifestement, ces trois représentations traduisent une relation hiérarchique dans laquelle le surtype *PERSONNE* est soit un *EMPLOYE* soit un *OUVRIER* (figure 6.9 d).

En termes de flexibilité, les schémas 6.9 c) et d) sont très flexibles puisqu'ils permettent d'ajouter aisément des rôles et des attributs au surtype et aux sous-types. Par exemple, exprimer qu'un ouvrier participe à une équipe de travail est envisageable moyennant l'ajout d'un type d'associa-

tions entre *EQUIPE* et *OUVRIER*. Le schéma 6.9 a) est moins facile à modifier car l'ajout de rôles spécifiques aux sous-types s'accompagne d'une contrainte supplémentaire. Pour exprimer qu'un ouvrier participe à une équipe, il faut ajouter un type d'associations entre *PERSONNE* et *EQUIPE* dont la cardinalité est [0-1] pour le rôle joué par *PERSONNE* ainsi qu'une contrainte de coexistence entre l'attribut *Ouvrier* et le rôle *participe.EQUIPE*. Le schéma 6.9 b) a aussi une flexibilité moindre puisque tout ajout de rôles ou d'attributs au surtype engendre l'ajout de nouveaux composants dans tous les sous-types. Pour exprimer qu'une personne possède des informations comptables, il faut ajouter deux types d'associations entre *OUVRIER* et *INFORMATION* et entre *EMPLOYE* et *INFORMATION*. Les modifications relatives aux composants communs aux sous-types sont réalisées autant de fois qu'il y a de sous-types.



a) Le type d'entités *PERSONNE* a deux attributs décomposables facultatifs : *Employé* et *Ouvrier.*

b) Deux types d'entités ont des attributs communs : *Matricule, Nom, Adresse* et *Numéro sécurité sociale.*

c) Le type d'entités *PERSONNE* joue soit le rôle *pe.EMPLOYE* soit le rôle *po.OUVRIER*.

d) Le type d'entités *PERSONNE* a deux sous-types : *EMPLOYE* et *OUVRIER*.

Figure 6.9 - Quatre représentations possibles du concept de personne dans une société employant des ouvriers et des employés.

d) Type d'associations ou attribut ?

La comparaison entre les deux structures type d'associations et attribut n'a pas beaucoup de signification. Par convention, au niveau conceptuel, un lien entre type d'entités est toujours représenté par un type d'associations. En règle générale, un type d'associations fonctionnel est transformé en une clé étrangère (avec les colonnes de référence) au niveau logique.

6.2.4.2 Propriétés d'un objet

Seuls les attributs, les rôles et les relations IS-A d'un surtype possèdent des propriétés susceptibles d'influencer la flexibilité d'un schéma.

a) Cardinalités et domaine de valeurs d'un attribut

Le choix des cardinalités et du domaine de valeurs d'un attribut s'avère parfois crucial dans le contexte de la maintenance. Le concepteur doit attribuer à chaque attribut un domaine de valeurs adéquat. S'il est inadapté, cela peut avoir des conséquences néfastes sur l'évolution d'une base de données. Il s'agit de concilier deux points de vue distincts. Du point de vue du concepteur, le domaine de valeurs doit être suffisamment grand pour ne pas devoir le modifier par la suite et, du point de vue du technicien, le domaine de valeurs ne doit pas être trop grand de crainte d'altérer les performances du système (parcours d'index, temps d'accès, espace disque, ...). Supposons qu'un code client soit défini sur un type d'entités *CLIENT*, il est de type numérique de longueur 3. En supposant que le code soit identifiant, le nombre maximum de clients est 999. C'est trop petit dans le cadre d'une grosse société dont la clientèle est importante. Par contre, il est inutile d'avoir un attribut code de type numérique et de longueur 12 permettant de représenter un billion de valeurs distinctes. Il est probable que, dans le contexte de la société, un attribut de longueur 6 soit largement suffisant.

Une cardinalité [0-N] pour un attribut est la solution la plus générale car elle englobe toutes les cardinalités possibles. Bien sûr, il est utopique de donner cette cardinalité à tous les attributs d'un schéma car ils doivent être tous transformés dans le modèle E/A de base ce qui rend le schéma inutilement complexe. Le changement d'un attribut monovalué en attribut multivalué nécessite une ou plusieurs transformations dans le modèle E/A de base. Inversement, les transformations appliquées à un attribut multivalué doivent être inversées pour retrouver un attribut monovalué. La modification des cardinalités d'un attribut le rendant obligatoire ou monovalué a également des répercussions importantes au niveau des instances dont certaines peuvent ne plus respecter les nouvelles contraintes. Les choix des cardinalités des attributs sont primordiaux car ils peuvent impliquer des modifications importantes dans les niveaux d'abstraction en aval du schéma conceptuel.

b) Cardinalités d'un rôle

Le choix des cardinalités des rôles d'un type d'associations influence également la flexibilité d'un schéma. La cardinalité la plus générique d'un rôle est [0-N]. Cela permet de modéliser tous les cas possibles. Comme pour les cardinalités des attributs, il n'est pas réaliste de choisir cette cardinalité pour tous les rôles car elle oblige le concepteur à transformer tous les types d'associations en types d'entités dans le modèle E/A de base ce qui complexifie énormément le schéma. Si un rôle devient obligatoire ou de cardinalité maximum 1, certaines instances ne respectent pas les nouvelles contraintes ce qui gêne le concepteur au moment de l'implémentation des modifications. Les choix des cardinalités des rôles d'un type d'associations sont très importants car toute modification ultérieure peut changer la nature même du type d'associations et engendrer des modifications plus ou moins importantes.

c) Type des relations IS-A

L'ensemble des sous-types d'un type d'entités est soit sans type, disjoint, total ou forme une partition. Le choix du type est important car, dans le modèle E/A de base, les relations IS-A sont transformées en types d'associations et le type est traduit par une contrainte (exactement-un pour une partition, au-moins-un pour total, exclusive pour disjoint et aucune contrainte pour l'absence de type) entre les rôles joués par les sous-types. La figure 6.10 illustre les quatre types possibles pour des relations IS-A ainsi que leur traduction dans le modèle E/A de base. Les contraintes utilisées sont expliquées dans le point 3.1.2.6.

Le cas le plus simple est l'absence de type car il ne nécessite aucune gestion de contrainte. Si un type est ajouté ou modifié, une nouvelle contrainte est engendrée ce qui nécessite la vérification de l'intégrité des données et la modification des programmes. Le retrait d'un type est plus facile à gérer puisque les données restent intègres, mais la gestion de la contrainte doit être enlevée des programmes.

En règle générale, le choix du type s'impose par la réalité de l'organisation. Il doit être pertinent dès le départ pour éviter les modifications ultérieures. Le choix systématique de relations sans type peut poser des problèmes car il n'assure pas toujours la cohérence de la base par rapport au domaine réel.

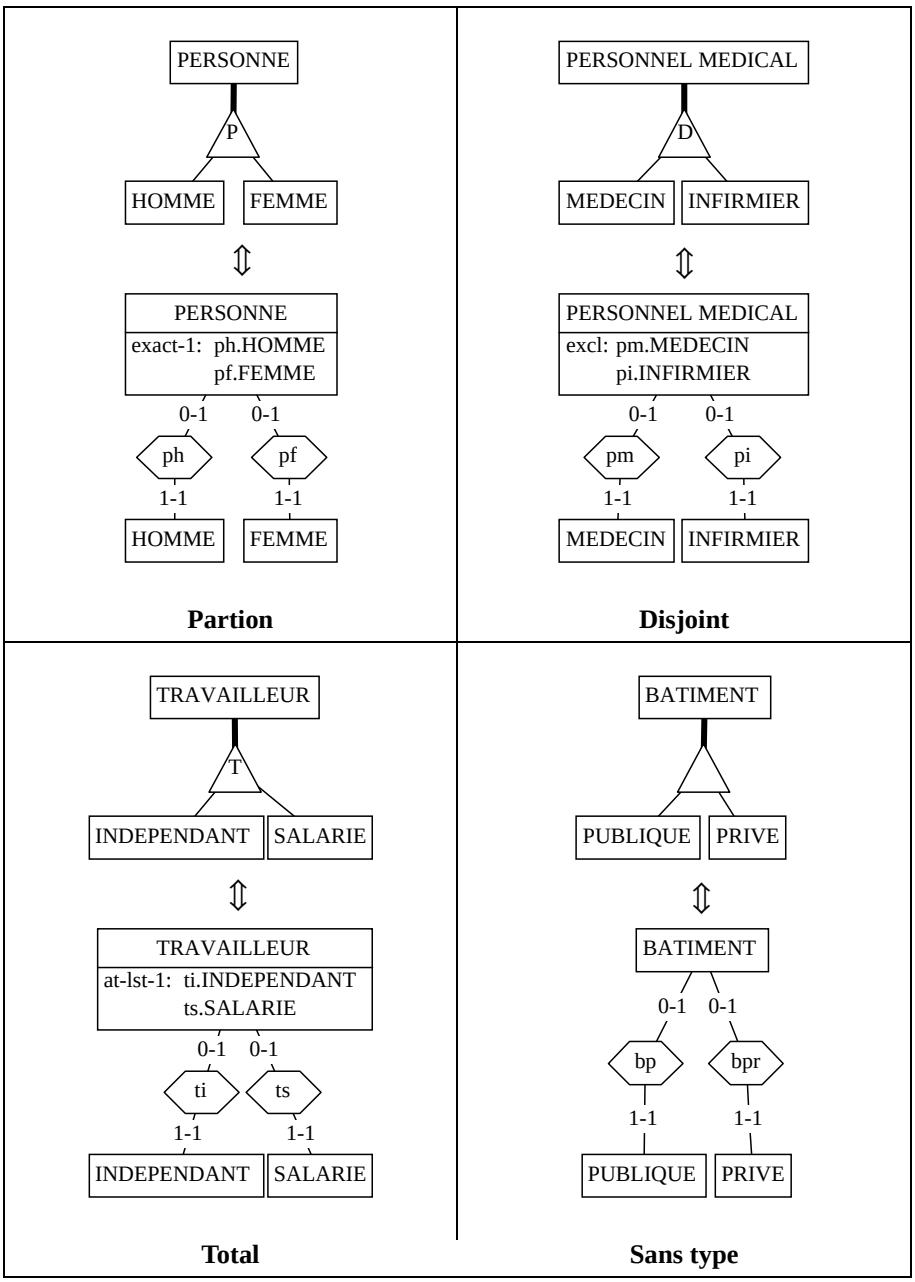


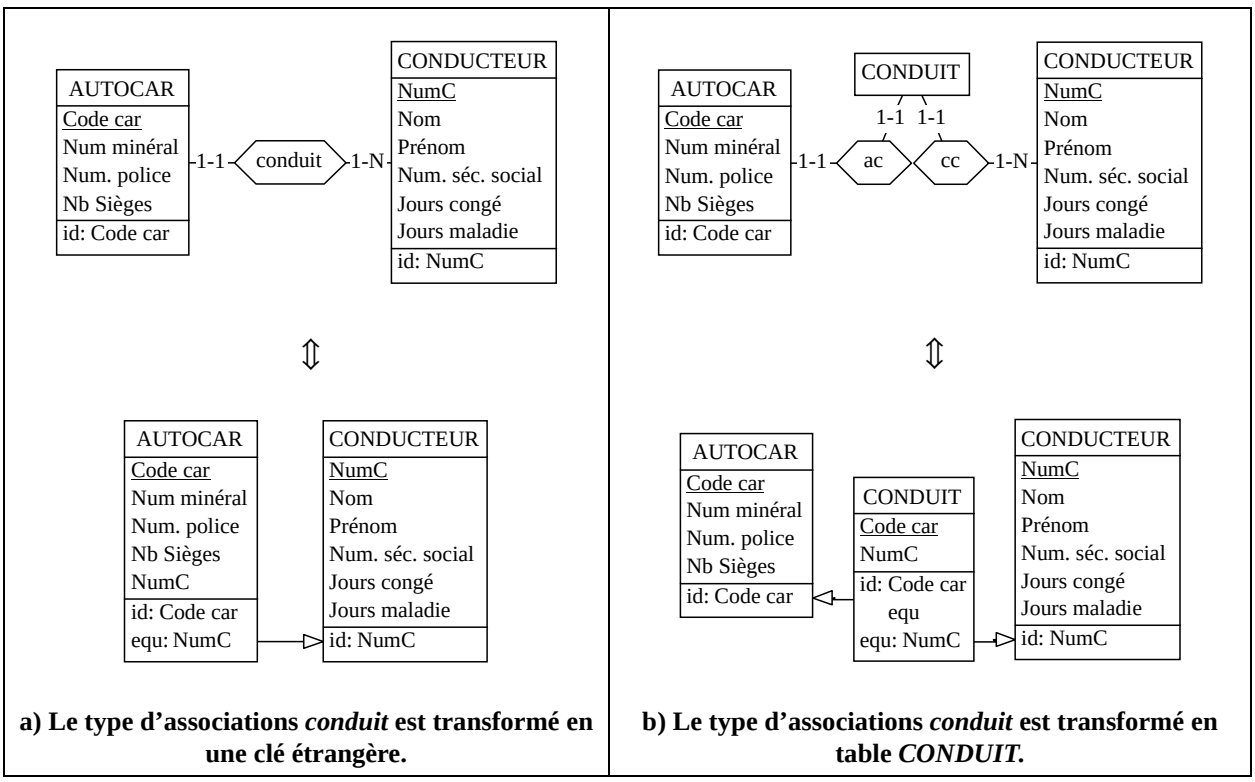
Figure 6.10 - Exemples de types possibles pour des relations IS-A.

6.3 Etude des caractéristiques des transformations sur chaque structure

Dans un schéma conceptuel, des structures complexes sont représentées. Ces structures (attributs multivalués et/ou décomposables, types d'associations ou relations IS-A) n'ont pas d'équivalent dans un schéma logique relationnel. Elles sont transformées lors de la conception logique pour correspondre à des structures qui sont directement traduisibles en instructions SQL. L'analyse des transformations de conception logique permet de formuler des recommandations de conception en termes de flexibilité sans perdre de vue certains aspects techniques liés à l'implémentation (comme les temps d'accès, la place occupée ou la facilité de programmation). Pour les types d'associations complexes (point 6.3.1), les attributs multivalués et/ou décomposables (point 6.3.2), les relations IS-A (point 6.3.3) et les types d'entités (point 6.3.4), analysons chaque transformation possible.

6.3.1 Types d'associations complexes

Supposons qu'un concept soit représenté par un type d'associations fonctionnel dans un schéma conceptuel. Dans les figures 6.11 a) et b), on représente le fait qu'un autocar est conduit par un chauffeur. Traditionnellement au niveau logique, le type d'associations *conduit* est transformé en une clé étrangère entre les tables *AUTOCAR* et *CONDUCTEUR* (figure 6.11 a). Rien n'empêche le concepteur de transformer le type d'associations conduit en type d'entités *CONDUIT* et de transformer les deux types d'associations *ac* et *cc* en clés étrangères (figure 6.11 b).



a) Le type d'associations *conduit* est transformé en une clé étrangère.

b) Le type d'associations *conduit* est transformé en table *CONDUIT*.

Figure 6.11 - Deux transformations possibles d'un type d'associations fonctionnel.

Les spécifications du schéma b) semblent inutilement complexes par rapport au schéma conceptuel. Pourtant, le schéma b) a une capacité d'évolution plus importante que le schéma a). Supposons que l'organisation de l'agence de voyages change en autorisant que plusieurs conducteurs conduisent un même autocar à des périodes différentes. Dans le schéma conceptuel, la cardina-

lité du rôle joué par *AUTOCAR* dans le type d'associations *conduit* devient [1-N]. Le schéma de la figure 6.11 a) n'est plus valable et doit être restructuré en profondeur . Par contre, le schéma b) peut traduire la nouvelle modification moyennant le retrait de l'identifiant sur l'attribut *Code car* de la table *CONDUIT*. Si un type d'associations fonctionnel devient complexe (ajout d'un rôle, augmentation d'une cardinalité maximum ou ajout d'un attribut), le schéma de la figure 6.11 b) est plus flexible puisque le type d'associations est déjà transformé en un type d'entités.

Au niveau des performances en terme d'accès et de stockage, la transformation du type d'associations en une clé étrangère est meilleure puisqu'il y a un accès en moins entre les tables et une table intermédiaire de moins à stocker. La programmation est également simplifiée puisqu'il y a une jointure et la gestion de la table intermédiaire en moins. Ce que l'on gagne en flexibilité, on le perd en efficacité.

6.3.2 Attributs décomposables et multivalués

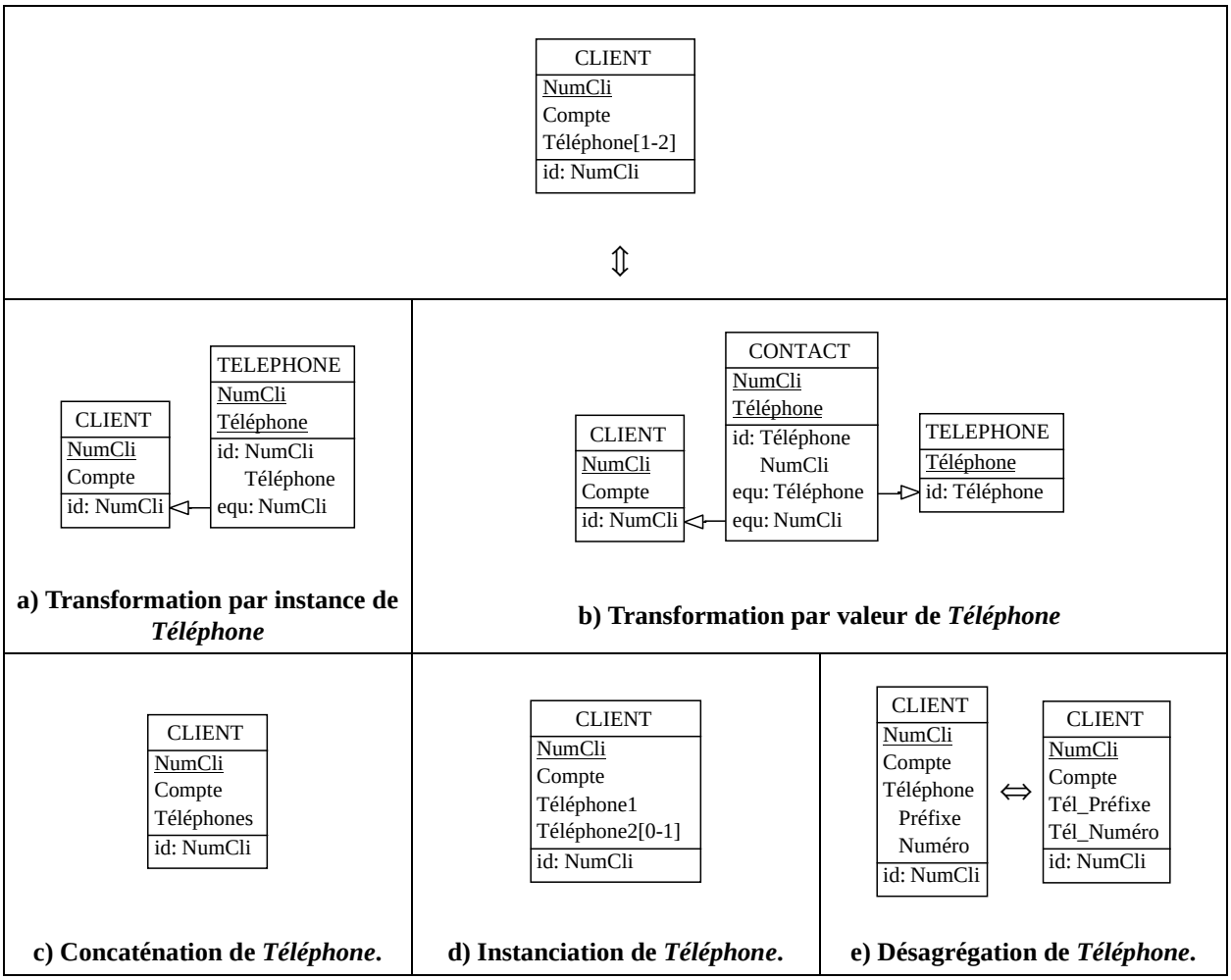
Les attributs multivalués et/ou décomposables du modèle conceptuel sont des structures qui n'ont pas de correspondant au niveau logique relationnel. Il existe cinq transformations pour de tels attributs : la transformation en un type d'entités (représentation par instance ou par valeur), la concaténation, l'instanciation et la désagrégation (voir point 3.2.3.3). Chacune des transformations présentées dans la figure 6.13 offre des avantages et des inconvénients dans le contexte de l'évolution.

Les deux transformations en un type d'entités permettent de représenter un attribut multivalué et/ou décomposable sous la forme d'une table reliée à la table d'origine. La transformation par instance (figure 6.13 a) représente toutes les instances de l'attribut transformé. Cette représentation est intéressante du point de vue de la flexibilité car elle facilite, au niveau conceptuel, l'ajout de propriétés qui sera traduit, au niveau logique, par l'ajout de colonnes et de contraintes sur la table représentant l'attribut transformé. Pour ce qui est des performances, un accès supplémentaire est nécessaire pour obtenir les téléphones d'un client et la base contient une table supplémentaire. La transformation par valeur (figure 6.13 b) représente les valeurs distinctes de l'attribut transformé. Au niveau de la flexibilité, il s'agit probablement de la structure la plus évolutive car, outre l'ajout de nouvelles propriétés ou de rôles, elle est insensible aux modifications des cardinalités des rôles. Dans l'exemple, la table *CONTACT* représente au niveau logique un lien plusieurs-à-plusieurs défini au niveau conceptuel ce qui constitue le cas le plus général. Les performances en temps d'accès sont moins bonnes car il faut deux accès pour obtenir les numéros de téléphone d'un client. Deux nouvelles tables sont également ajoutées à la base ce qui va augmenter sensiblement sa taille.

La transformation de concaténation consiste à réduire un attribut atomique multivalué en un attribut atomique monovalué dont le domaine de valeurs est égal au produit de la cardinalité maximum de l'attribut transformé et de la longueur de son domaine. Une telle transformation est intéressante au niveau des performances car, les informations sont disponibles sans accès supplémentaire et il n'y a pas de redondance d'informations. Malheureusement, la capacité d'évolution d'une telle structure est faible. Mis à part des modifications de cardinalités ou de domaine de valeurs, les autres modifications consistant par exemple à faire de téléphone un attribut décomposable ou à augmenter sa cardinalité de manière significative vont inévitablement nécessiter d'autres transformations.

La transformation d'instanciation représente un attribut multivalué dans son type d'entités par une série d'attributs monovalués. Comme la concaténation, l'instanciation est une transformation intéressante quand il s'agit de satisfaire des exigences de temps d'accès ou de place occupée sur les supports. Par contre, elle souffre d'un manque de capacité d'évolution si l'attribut devient décomposable ou qu'on augmente sa cardinalité maximum.

La transformation de désagrégation remplace un attribut décomposable monovalué par ses sous-attributs. Comme la concaténation et l'instanciation, cette transformation donne de bonnes performances mais elle manque de flexibilité lorsque l'attribut devient multivalué par exemple.



a) Transformation par instance de *Téléphone*

b) Transformation par valeur de *Téléphone*

c) Concaténation de *Téléphone*.

d) Instanciation de *Téléphone*.

e) Désagrégation de *Téléphone*.

Figure 6.12 - Cinq transformations possibles d'un attribut décomposable et/ou multivalué.

Le tableau 6.1 présente des exemples de modifications relatives à l'attribut *Téléphone* du type d'entités *CLIENT* (figure 6.12) et compare les transformations par instance (figure 6.12 a), par valeur (figure 6.12 b) et l'instanciation (figure 6.12 d) qui sont appliquées sur *Téléphone* pour le rendre conforme au modèle E/A de base. Le tableau analyse les impacts des modifications pour chacune des transformations sur les structures de données (S), les instances (D), les programmes (P) et donne une évaluation globale de la flexibilité des représentations choisies (E). La comparaison révèle que les transformations par valeur et par instance fournissent les meilleurs résultats en terme de flexibilité. Par contre, comme cela a déjà été mentionné précédemment, ces performances sont contrebalancées par un coût élevé en ce qui concerne les temps d'accès et l'espace consommé. Le concepteur doit donc faire le bon choix en fonction des besoins des utilisateurs et du contexte.

		[1-2] → [1-5]	[1-2] → [0-2]	char(12) → char(15)	"Téléphone" → "Num-Tel"	Téléphone devient décomposable
Transformation par instance	**E**	Moyen	Moyen	Bon	Bon	Moyen
	S	/	Enlever la contrainte d'égalité	Etendre le domaine	Renommer l'attribut	Désagréger l'attribut
	D	/	/	/	/	Désagréger les instances
	P	Changer la contrainte de cardinalité	Enlever la contrainte d'égalité	Indépendance structurelle	Indépendance structurelle	Dépendance structurelle
Transformatio par valeur	**E**	Moyen	Moyen	Bon	Bon	Moyen
	S	/	Enlever la contrainte d'égalité	Etendre le domaine	Renommer l'attribut	Désagréger l'attribut
	D	/	/	/	/	Désagréger les instances
	P	Changer la contrainte de cardinalité	Enlever la contrainte d'égalité	Indépendance structurelle	Indépendance structurelle	Dépendance structurelle
Instanciation	**E**	Mauvais	Moyen	Moyen	Moyen	Mauvais
	S	Créer les attributs	Rendre Téléphone1 facultatif	Etendre les domaines	Renommer les attributs	Désagréger les attributs + ajouter une contrainte de coexistence
	D	/	/	/	/	Désagréger les instances
	P	Dépendance structurelle	Dépendance structurelle	Indépendance structurelle	Indépendance structurelle	Dépendance structurelle

Tableau 6.1 - Comparaison des impacts de modifications relatives à l'attribut multivalué *Téléphone* sur trois transformations différentes.

6.3.3 Relations IS-A

Les relations IS-A peuvent être transformées de trois façons différentes. Il s'agit de l'héritage ascendant où les attributs et rôles des sous-types migrent dans le surtype (figure 6.13 a), de l'héritage descendant où les attributs et rôles du surtype sont transférés dans chaque sous-type (figure 6.13 b) et de la transformation en types d'associations puis en clés étrangères (figure 6.13 c). Du point de vue de la flexibilité, ces représentations ont déjà été discutées au point 6.2.4.1 c).

Au niveau des performances, l'héritage ascendant permet un accès direct à toutes les informations. La place occupée est importante puisqu'on en réserve pour tous les sous-types. Or, pour chaque instance, il n'y a le plus souvent qu'un seul groupe de colonnes qui ont des valeurs. Cette transformation introduit également des contraintes entre les colonnes des sous-types qui peuvent alourdir la programmation. Dans l'exemple de la figure 6.13 a), trois contraintes sont nécessaires pour gérer l'héritage. L'héritage descendant permet aussi un accès direct aux informations mais certaines parties des instances concernant les propriétés du surtype peuvent être redondantes en cas de recouvrement. La transformation en clé étrangère nécessite un accès supplémentaire pour obtenir les informations contenues dans les sous-types. Par contre, cette représentation n'engendre aucune redondance d'informations.

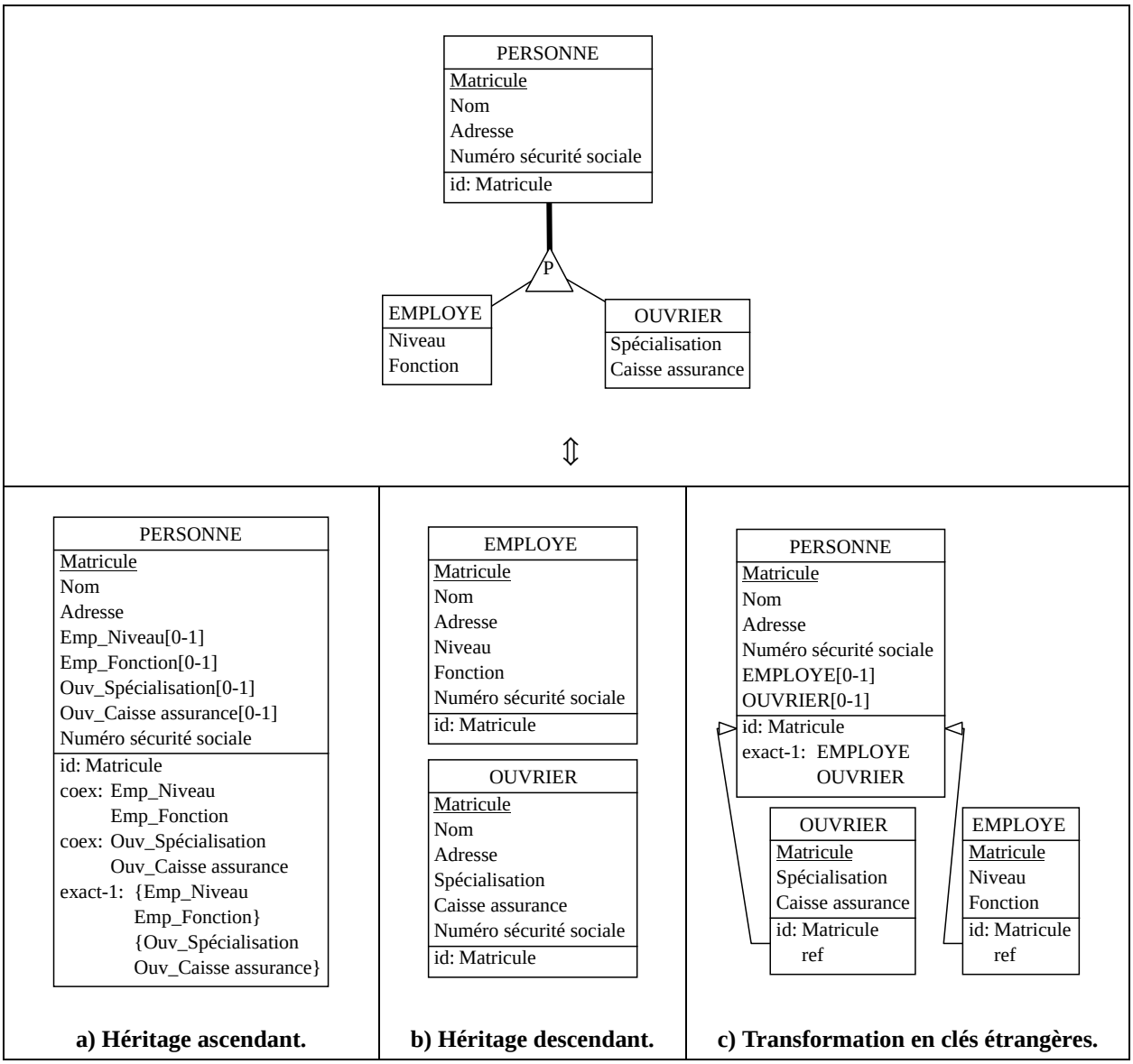


Figure 6.13 - Trois transformations possibles des relations IS-A au niveau logique.

6.3.4 Types d'entités

Un type d'entités d'un schéma conceptuel devient une table dans le schéma logique correspondant. Il n'y a pas de transformation, seulement un changement de terminologie. Il est toutefois possible d'éclater un type d'entités en transférant certaines propriétés dans un nouveau type d'entités qui est relié au premier par une clé étrangère. Cette technique permet d'isoler dans le nouveau type d'entités les propriétés moins importantes et d'accélérer l'accès aux données plus importantes.

Cette transformation peut altérer les performances puisqu'il faut un accès supplémentaire pour obtenir les données de la seconde table. Par contre, l'accès aux données de la table principale est amélioré puisque, par exemple, un tampon de lecture peut contenir plus d'instances de cette table. Au niveau de la flexibilité, tout ajout, retrait ou modification des composants se traduit de la même façon dans les deux représentations en changeant simplement la table sur laquelle les modifications sont réalisées.

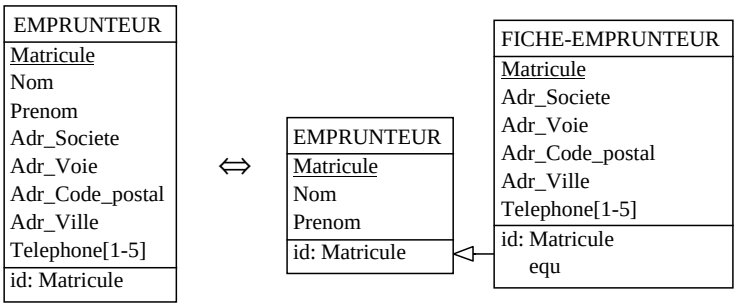


Figure 6.14 - Eclatement du type d'entités *EMPRUNTEUR*.

6.4 Evaluation des modifications en termes de structures

Les sections 6.2 et 6.3 ont mis en évidence un certain nombre de recommandations visant à améliorer la flexibilité d'une application. Pour affirmer qu'une structure est plus évolutive qu'une autre, les connaissances acquises en matière de dépendances des modifications par rapport aux données et aux programmes sont utilisées pour évaluer leurs implications. L'objectif de cette section est de synthétiser ces connaissances fournies par l'étude des modifications (chapitre 4 et annexe C). Le tableau 6.2 récapitule les dépendances des instances et des programmes par rapport aux modifications analysées. Pour rappel, les relations de dépendance entre les modifications et les données sont classées en deux catégories :

- Les modifications sont indépendantes (I) des données si les instances vérifient les contraintes des nouvelles structures de données.
- Les modifications sont dépendantes (D) des données si elles peuvent entraîner une inconsistance des données dans les nouvelles structures.

Les relations de dépendance entre les modifications et les programmes appartiennent à l'une des trois catégories suivantes :

- Lorsqu'un programme ne nécessite aucune modification pour fonctionner correctement avec les nouvelles structures, on parle d'indépendance totale (I).
- Si les programmes sont toujours fonctionnels moyennant quelques adaptations (comme le renommage ou la modification du domaine de valeurs d'un objet de la base), ils sont qualifiés de structurellement indépendants (IS).
- Si la modification des structures entraîne des modifications structurelles des programmes, les programmes sont structurellement dépendant des modifications (DS).

Le rôle de cette synthèse, matérialisée sous la forme du tableau 6.2 et de commentaires succincts (points 6.4.1 à 6.4.8), est double. Premièrement, elle permet aux concepteurs de choisir la meilleure représentation ou la meilleur transformation de conception dans un contexte de flexibilité. Deuxièmement, la table peut servir de catalogue pour aider les développeurs à évaluer les conséquences des changements réclamés par les utilisateurs ou les instances dirigeantes d'une organisation. Pour procéder, il faut d'abord propager les modifications futures jusqu'au niveau physique. Ensuite, pour toutes les modifications physiques correspondantes, le tableau 6.2 est consulté en interaction avec ses commentaires et l'analyse des modifications relatives aux données et aux programmes de l'annexe C (les références sont dans le tableau). Cela permet d'imaginer les conséquences au niveau opérationnel des changements apportés aux spécifications.

	Instances		**Programmes**		
	I	**D**	**I**	**IS**	**DS**
Modifications relatives aux tables					
Création (p. 74)	X		X		
Destruction (p. 75)	X				X
Renommage (p. 75)	X			X	
Modifications relatives aux colonnes					
Création (p. 76)	X				X
Destruction (p. 77)	X				X
Renommage (p. 78)	X			X	
Destruction d'une contrainte NOT NULL (p. 78)	X				X

Tableau 6.2 - Tableau récapitulatif des dépendances des instances et des programmes par rapport aux modifications.

	Instances		Programmes		
	I	**D**	**I**	**IS**	**DS**
Création d'une contrainte NOT NULL (p. 79)		X			X
Extension de domaine (p. 80)	X			X	
Restriction de domaine (p. 81)		X		X	
Changement de type (p. 82)		X		X	
Modifications relatives aux clés étrangères					
Création (p. 82)		X			X
Destruction (p. 83)	X				X
Renommage (p. 84)	X			X	
Destruction de la clé primaire définie sur les colonnes de référence (p. 84)	X				X
Création d'une clé primaire sur les colonnes de référence (p. 84)		X			X
Création de contraintes NOT NULL sur les colonnes de référence (p. 85)		X			X
Destruction des contraintes NOT NULL définies sur les colonnes de référence (p. 86)	X				X
Destruction d'une contrainte d'égalité (p. 115)	X				X
Création d'une contrainte d'égalité (p. 116)		X			X
Modifications relatives aux clés primaires et candidates					
Création (p. 86)		X			X
Destruction (p. 87)	X				X
Renommage (p. 88)	X		X		
Changement d'une clé primaire en candidate et inversement (p. 88)	X		X		
Ajout d'un composant (p. 89)	X				X
Retrait d'un composant (p. 90)		X			X
Modifications relatives aux contraintes de coexistence, exclusive, au-moins-un et exactement-un					
Création (p. 117)		X			X
Destruction (p. 118)	X				X
Renommage (p. 118)	X		X		
Changement de type (p. 119)		X			X
Ajout d'un composant (p. 119)		X			X
Retrait d'un composant (p. 120)		X			X
Modifications relatives aux index					
Création (p. 90)	X		X		
Destruction (p. 91)	X		X		
Renommage (p. 91)	X		X		
Ajout d'un composant (p. 91)	X		X		
Retrait d'un composant (p. 92)	X		X		
Modifications relatives aux espaces de stockage					
Création (p. 92)	X		X		
Destruction (p. 93)	X		X		
Renommage (p. 94)	X		X		
Ajout d'une table (p. 95)	X		X		
Retrait d'une table (p. 95)	X		X		
Changement de transformation de conception logique					
Changement de transformation sur les colonnes (p. 198)	X				X
Changement de transformation sur les clés étrangères (p. 222)	X				X
Changement de transformation sur les colonnes de référence (p. 224)	X				X

Tableau 6.2 - Tableau récapitulatif des dépendances des instances et des programmes par rapport aux modifications.

Dans les points 6.4.1 à 6.4.8, une analyse succincte du tableau 6.2 résume les impacts des modifications sur les structures de données, les instances et les programmes.

6.4.1 Modifications relatives aux tables

Les tables sont les structures les plus importantes dans une base de données. Toutes les modifications (excepté le renommage) sont cruciales aussi bien au niveau de la base de données que des programmes.

- Structures : La création et la suppression nécessite des modifications structurelles de la base de données.
- Instances : Le renommage peut être lourd à gérer si le SGBDR ne possède aucune instruction de renommage.
- Programmes : La suppression modifie la structure des programmes. Le renommage peut se faire au niveau de l'interface avec l'utilisateur (le nom de la table reste inchangé car les techniques des vues ou des synonymes sont utilisées) ou la table est renommée dans tous les programmes (commande chercher/remplacer). La création n'empêche pas les programmes de fonctionner correctement. Il faut cependant ajouter les fonctions qui gèrent la nouvelle table.

6.4.2 Modifications relatives aux colonnes

- Structures : Toutes les modifications relatives aux colonnes excepté la création et la destruction sont, dans certains SGBDR qui ne possèdent pas de requête spécifique pour altérer des colonnes, relativement lourdes car, elles nécessitent la création d'une colonne temporaire, la destruction de la colonne, la création de la nouvelle colonne et le transfert des instances.
- Instances : La modification du domaine, du caractère facultatif ou du type d'une colonne peuvent engendrer des incohérences dans les instances d'une base. Certains tests seront réalisés avant de modifier les structures de la base.
- Programmes : La création, la destruction et le changement de caractère facultatif d'un attribut engendrent des changements structurels dans les programmes. Les autres modifications nécessitent seulement de petites corrections (renommage, changement du type des variables, ...).

6.4.3 Modifications relatives aux clés étrangères

- Structures : Les modifications relatives aux clés étrangères sont du même ordre que les modifications relatives à une ou plusieurs colonnes avec la gestion de la clé étrangère en plus.
- Instances : La création d'une clé étrangère ainsi que les modifications qui ajoutent une contrainte sur les colonnes de références (primaire ou candidate, contrainte égalité ou contrainte NOT NULL) peuvent rendre la base incohérente.
- Programmes : Toutes les modifications excepté le renommage d'une clé étrangère ont un impact sur les structures des programmes.

6.4.4 Modifications relatives aux clés primaires et candidates

- Structures : Toutes les modifications excepté la création et la destruction nécessitent la destruction de la contrainte puis sa recréation avec les nouveaux paramètres.
- Instances : La création d'un identifiant et le retrait d'un composant peuvent entraîner la violation d'une contrainte d'unicité dans la base.
- Programmes : Toutes les modifications, exceptés le renommage et le changement de type d'une contrainte identifiante, ont des répercussions sur la structure des programmes.

6.4.5 Modifications relatives aux autres contraintes

- Structures : Toutes modifications, excepté la création et la destruction, nécessitent la destruction de la contrainte puis sa recréation avec les nouveaux paramètres.
- Instances : Excepté la destruction et le renommage d'une contrainte, toutes les autres modifications peuvent engendrer la violation de la contrainte dans la base.
- Programmes : Les programmes dépendent structurellement de toutes les modifications excepté le renommage d'une contrainte.

6.4.6 Modifications relatives aux index

Les modifications relatives aux index risquent d'être des opérations consommatrices de ressources dans le cas de bases de données volumineuses. Par conséquent, elles doivent être appliquées à bon escient.

- Structures : Les modifications différentes de la création ou de la destruction nécessitent la destruction de l'index et sa recréation. Ces modifications peuvent être très lourdes dans le cas de grosses bases de données.
- Instances : Toutes les instances de la base sont indépendantes des modifications.
- Programmes : Les programmes restent inchangés. L'ingénieur de maintenance doit toutefois rester attentif car les modifications relatives aux index peuvent influencer les performances de certaines requêtes.

6.4.7 Modifications relatives aux espaces de stockage

Les modifications relatives aux espaces de stockage consomment également beaucoup de ressources lorsqu'il s'agit de bases de données volumineuses.

- Structures : Excepté la création ou la destruction, les modifications nécessitent la destruction de l'espace et sa recréation. Les tables impliquées dans ces modification doivent être vides. Une solution consiste à créer une table temporaire stockant les instances de la table, à détruire puis recréer une nouvelle table et transférer les instances de la table temporaire dans cette nouvelle table.
- Instances : Toutes les modifications sont indépendantes de la base.
- Programmes : Les programmes restent inchangés. La modification des espaces de stockage peut avoir des conséquences en termes de temps d'accès en fonction du support où ils sont stockés.

6.4.8 Changement de transformation de conception

Ces modifications engendrent en général des modifications profondes tant au niveau des données qu'au niveau des programmes. Vu la complexité de certaines d'entre elles, l'ingénieur de maintenance doit bien maîtriser leurs impacts avant de réaliser les modifications nécessaires.

- Structures : L'application d'une autre transformation lors de la conception logique engendre la création de nouvelles structures, le transfert des instances des anciennes structures vers les nouvelles et la destruction des anciennes structures. Il s'agit d'opérations généralement lourdes.
- Instances : Pour les transformations de conception à sémantique constante, la base est toujours cohérente après la transformation. Les autres transformations nécessitent une phase de vérification des contraintes avant de modifier les structures.
- Programmes : Les programmes dépendent structurellement des changements de transformations, ils doivent être revus en profondeur.

6.5 Conclusion

La flexibilité d'une base de données est un des facteurs qui déterminent le coût de la maintenance d'un système. Le modèle de données constitue les fondations sur lesquelles le reste du système est construit. A ce titre, il doit être le plus stable possible car, l'expérience a souvent montré que les modifications les plus chères sont celles qui requièrent des changements au niveau de la structure de la base de données. Ces changements se propagent dans tous les programmes qui utilisent les structures modifiées [Moo94].

Le concepteur seul n'est pas habilité à prendre des décisions importantes concernant la flexibilité d'une base de données. Il doit travailler en étroite collaboration avec les dirigeants qui définissent les directions stratégiques de l'organisation ou les experts qui maîtrisent les nouvelles tendances dans le domaine d'application. Le concepteur intervient essentiellement pour choisir les représentations adéquates pour assurer une flexibilité suffisante à la base de données.

Ce chapitre présente des recommandations de conception qui visent à améliorer la flexibilité d'un système et qui, pour la plupart, relèvent plus du bon sens que de la nouveauté scientifique. Toutefois, il nous semble intéressant de garder une trace écrite des constatations découvertes lors de la rédaction de ce travail. Même si la liste est loin d'être exhaustive, elle permet de se faire une idée de la notion de flexibilité ainsi que de certaines techniques permettant de tendre vers cet objectif.

En règle générale, une structure plus flexible au niveau conceptuel se traduit au niveau logique par des structures plus complexes. Elle devient plus difficile à implémenter et, souvent, plus consommatrice de ressources. A charge du concepteur de trouver un juste milieu entre des critères aussi différents que la simplicité, la lisibilité, la flexibilité, les performances ou la facilité d'implémentation.

«Toute chose étant égale par ailleurs, seule la mise en perspective donne du sens à l'implémentation proactive.»

Pierre-André Leriche

CHAPITRE 7 IMPLÉMENTATION : APPROCHE DB-MAIN

Le but de ce travail est également de développer des outils prototypes qui aident le concepteur dans son activité de maintenance et d'évolution d'une application de base de données. Le projet de recherche DB-Main, qui a démarré en septembre 1993, est consacré aux problèmes de la maintenance et de l'évolution des applications de bases de données. Un des produits de ce programme est l'AGL générique DB-Main dédié à l'ingénierie des bases de données et, plus particulièrement, à la conception, la rétro-ingénierie, la réingénierie, l'intégration, la maintenance et l'évolution. L'objectif à long terme de ce programme est d'étudier les problèmes liés aux systèmes d'information, incluant ceux relatifs à l'évolution des besoins des applications de bases de données. Cette étude a mené à des propositions méthodologiques, en termes de méthodes et d'outils d'aide pour une grande variété d'activités d'ingénierie telles que la rétro-ingénierie, la compréhension de programmes, la modélisation des processus, la génération de code, ...

La section 7.1 présente l'architecture de l'atelier ainsi que des descriptions plus précises des fonctions nécessaires pour mettre en œuvre la méthodologie développée. La section 7.2 décrit l'implémentation des propagations dans l'atelier alors que les deux dernières sections expliquent l'architecture et le fonctionnement des outils développés pour aider le concepteur dans son travail de conversion des données (section 7.3) et de modifications des programmes (section 7.4).

7.1 Présentation de l'atelier de génie logiciel

L'une des activités principales du projet DB-Main [1] est la conception et le développement d'un AGL [2] en ingénierie de bases de données qui tente d'apporter des réponses aux lacunes des outils actuels. Cette section présente brièvement quelques aspects originaux et utiles pour la suite de ce travail :

- l'architecture de l'atelier (point 7.1.1),
- le modèle générique de représentation des spécifications (point 7.1.2),
- l'approche transformationnelle (point 7.1.3),
- la personnalisation méthodologique et la gestion des historiques (point 7.1.4),
- la personnalisation et l'extensibilité fonctionnelle de l'atelier (point 7.1.5),
- la compréhension de programmes et le moteur de patrons (point 7.1.6).

7.1.1 Architecture de l'atelier

L'architecture de DB-Main décrite à la figure 7.1 met en évidence les principaux composants de l'atelier :

- l'interface graphique, qui permet de visualiser le contenu du référentiel [3] et de demander l'exécution des opérations; elle est pilotée par le moteur méthodologique;
- le moteur méthodologique, qui guide l'analyste dans le suivi d'une méthode d'ingénierie propre à son entreprise (point 7.1.4);
- les assistants, qui sont des outils d'aide à la résolution de problèmes répétitifs et spécifiques (de transformation, de conformité, de rétro-ingénierie, ...) par l'intermédiaire de scripts prédéfinis ou personnalisés (point 7.1.3);
- la machine VOYAGER, interpréteur du langage de quatrième génération (L4G) Voyager 2, qui exécute des fonctions personnalisées développées par l'analyste (point 7.1.5);
- les outils de base, qui permettent l'accès au référentiel, la gestion de son contenu, l'importation/exportation de spécifications, la transformation de schéma (point 7.1.3), l'analyse de textes (extracteurs et moteur de patrons, point 7.1.6), ...

1. pour "Database Maintenance".
2. L'AGL (dont la sixième version est sortie en décembre 2000) est développé par une équipe de quatre chercheurs. Vincent Englebert développe le langage de quatrième génération Voyager 2 et sa machine abstraite. Jean Henrard travaille sur la rétro-ingénierie et les outils de compréhension de programmes. Didier Roland s'intéresse à la modélisation des processus et aux assistants d'analyse de schémas et de transformations globales. Pour ma part, je travaille sur le référentiel, les transformations ponctuelles, l'assistant d'intégration et les interfaces graphiques.
3. Base de données centrale qui stocke et gère les informations du système dans le but de servir de point de référence dans des phases ultérieures de traitement. On parle de "repository" en anglais.

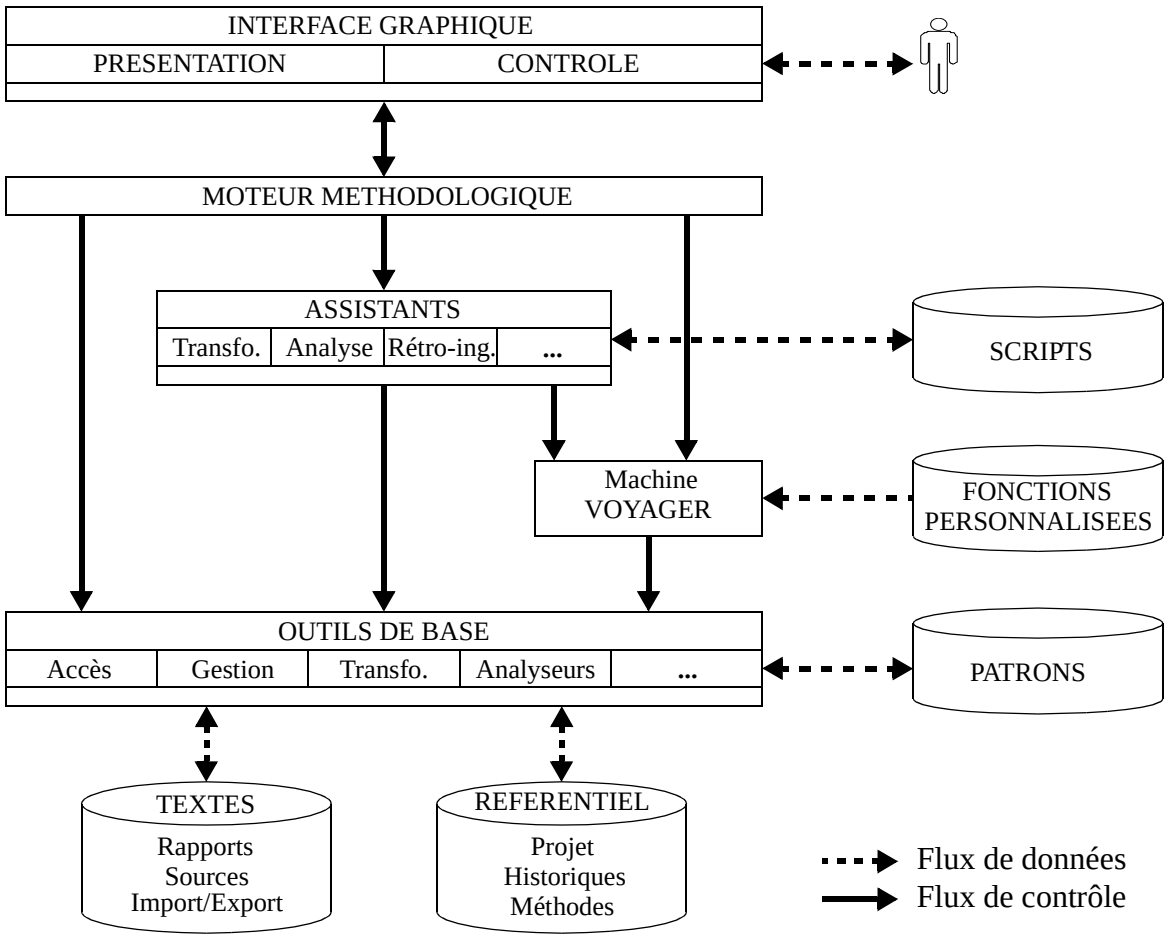


Figure 7.1 - Architecture de l'atelier DB-Main.

7.1.2 Modèle DB-Main de représentation des spécifications

L'atelier permet de représenter et de spécifier les informations et les structures de données qui composent un système d'information. Le modèle de spécification de DB-Main est conforme au modèle générique de représentation des spécifications présenté dans la section 3.1. Il contient neuf objets génériques classés en objets de haut niveau qui définissent des macro-structures (projet, schéma, fichier, processus) et en objets de bas niveau qui définissent des micro-structures (type d'entités, type d'associations, collection, attribut et groupe). Les micro-structures qui composent un schéma de données ont déjà été explicitées dans le point 3.1.2. Nous allons décrire les objets de la modélisation des processus qui permettent de décrire toutes les activités d'ingénierie sous la forme d'un historique structuré.

7.1.2.1 Projet

Un projet est fait d'une collection de produits et de processus. Un ensemble de spécifications apparaît comme un produit alors qu'un processus élabore des produits à partir d'autres produits. Les produits sont de deux types : les schémas (de données) et les fichiers (texte). Dans la figure 7.2, le rectangle grisé en traits forts est la représentation graphique d'un projet, les ellipses en traits forts représentent des schémas, les ellipses en traits fins des fichiers et les rectangles (en traits forts ou fins) des processus.

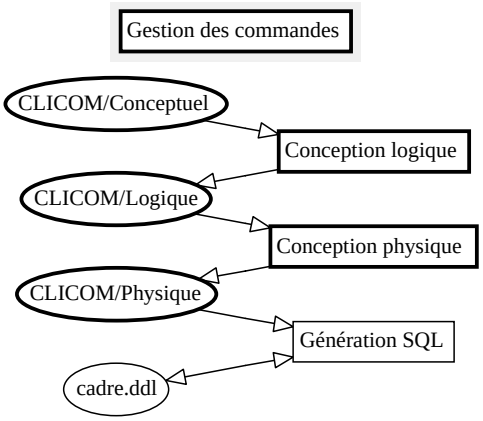


Figure 7.2 - Représentation graphique d'un projet avec ses produits et ses processus.

7.1.2.2 Schéma

Un schéma est une description complète ou partielle d'une structure de données. Il peut être constitué de types d'entités, de type d'associations et/ou de collections. Un schéma peut être lié à d'autres schémas par des liens de connexion ou des processus.

7.1.2.3 Fichier

Un fichier est un texte externe généralement dérivé d'un schéma (par exemple : un fichier de script SQL généré à partir d'un schéma physique relationnel) ou, duquel un schéma a été dérivé (par exemple : un rapport d'interview à partir duquel on crée un schéma conceptuel).

7.1.2.4 Processus

Un produit est le résultat d'une activité appelée processus. Intégrer des schémas, transformer un schéma conceptuel en schéma logique, optimiser un schéma sont des processus. Un processus appartient à un type de processus qui explicite comment résoudre un type spécifique de problèmes. Les processus peuvent être de deux types : d'ingénierie (traits forts) ou primitifs (traits fins). Un processus d'ingénierie est une phase du projet durant laquelle plusieurs sous-processus sont réalisés. Il peut y avoir des produits en entrée, en sortie et en interne. La figure 7.3 montre le processus d'ingénierie de conception logique ainsi que ses sous-processus et ses produits. Un processus primitif est une transformation de produits réalisée à l'aide des fonctions primitives (non décomposables). Copier un schéma, générer du code SQL sont des processus primitifs.

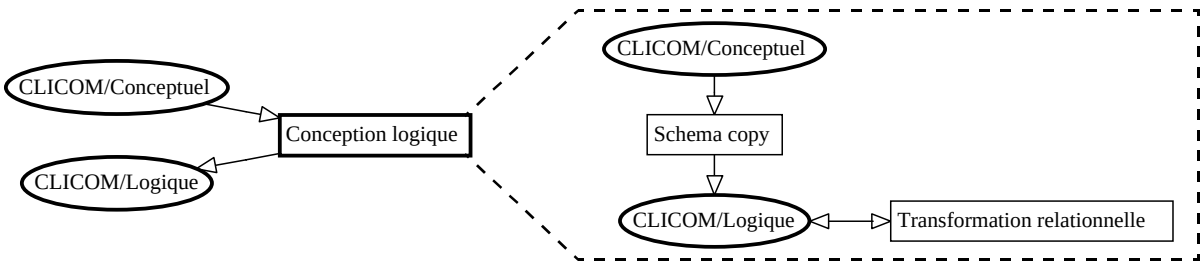
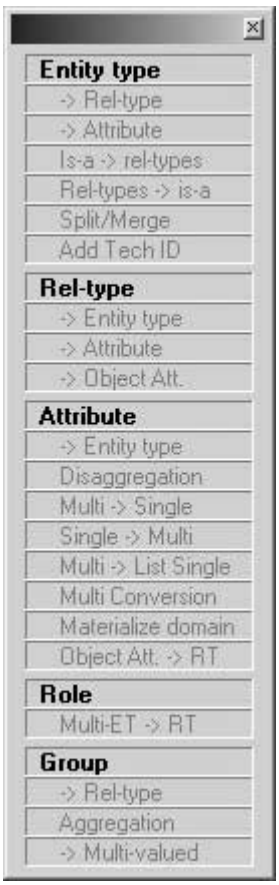


Figure 7.3 - Le processus d'ingénierie *Conception logique* et ses sous-processus primitifs *Schema copy* et *Transformation relationnelle*.

7.1.3 Approche transformationnelle

L'atelier offre une palette d'une vingtaine de transformations (figure 7.4). Elles ont été analysées en détail (pour la plupart) dans le point 3.2.3. Ces transformations sont appelées transformations ponctuelles car elles s'appliquent à un élément sélectionné. Cette façon de transformer un schéma peut être fastidieuse dans le cas d'un schéma volumineux. C'est pourquoi deux assistants de transformation ont été développés dans l'atelier.



Transformation sur les types d'entités
- Transformation d'un type d'entités en un type d'associations (p. 58)
- Transformation d'un type d'entités en un attribut (p. 59)
- Transformation de relations IS-A en types d'associations (p. 66)
- Transformation de types d'associations en relations IS-A (p. 65)
- Eclatement d'un type d'entités (p. 60) et Fusion de deux types d'entités (p. 60)
- Ajout d'un identifiant technique

Transformation sur les types d'associations
- Transformation d'un type d'associations en un type d'entités (p. 61)
- Transformation d'un type d'associations en une clé étrangère (p. 61)
- Transformation d'un type d'associations en un attribut objet

Transformation sur les attributs
- Transformation d'un attribut en un type d'entités (p. 61)
- Désagrégation d'un attribut décomposable (p. 64)
- Concaténation d'un attribut multivalué (p. 62)
- Transformation d'un attribut monovalué en un attribut multivalué (p. 63)
- Transformation d'un attribut multivalué en une série d'attributs (p. 63)
- Conversion d'un attribut multivalué
- Matérialisation du domaine utilisateur d'un attribut
- Transformation d'un attribut objet en un type d'associations

Transformation sur les rôles
- Transformation d'un rôle multi-TE en types d'associations (p. 65)

Transformation sur les groupes
- Transformation d'une clé étrangère en un type d'associations (p. 62)
- Agrégation d'une liste d'attributs en un attribut décomposable (p. 64)
- Transformation d'une série d'attributs en un attribut multivalué (p. 64)

Figure 7.4 - Palette de transformations ponctuelles de l'atelier.

Le premier assistant (appelé *assistant de transformations globales*) permet d'appliquer un script de transformations sur certains types d'objets en vue de résoudre certains problèmes. Pour chaque classe de construction (problème : partie gauche de la figure 7.5 en dessous de "Entity types"), l'assistant propose un ou plusieurs types de transformations qui remplacent la construction posant problème par une autre, équivalente ou non (solution : partie gauche de la figure 7.5 en dessous de "into"). L'utilisateur peut ainsi construire (sauver et réutiliser si nécessaire) des scripts de transformations personnalisés (partie droite de la figure 7.5) et dédiés à des problèmes spécifiques. La figure 7.5 propose la boîte de dialogue de l'assistant de transformations globales avec un script permettant de transformer un schéma conceptuel en un schéma logique relationnel.

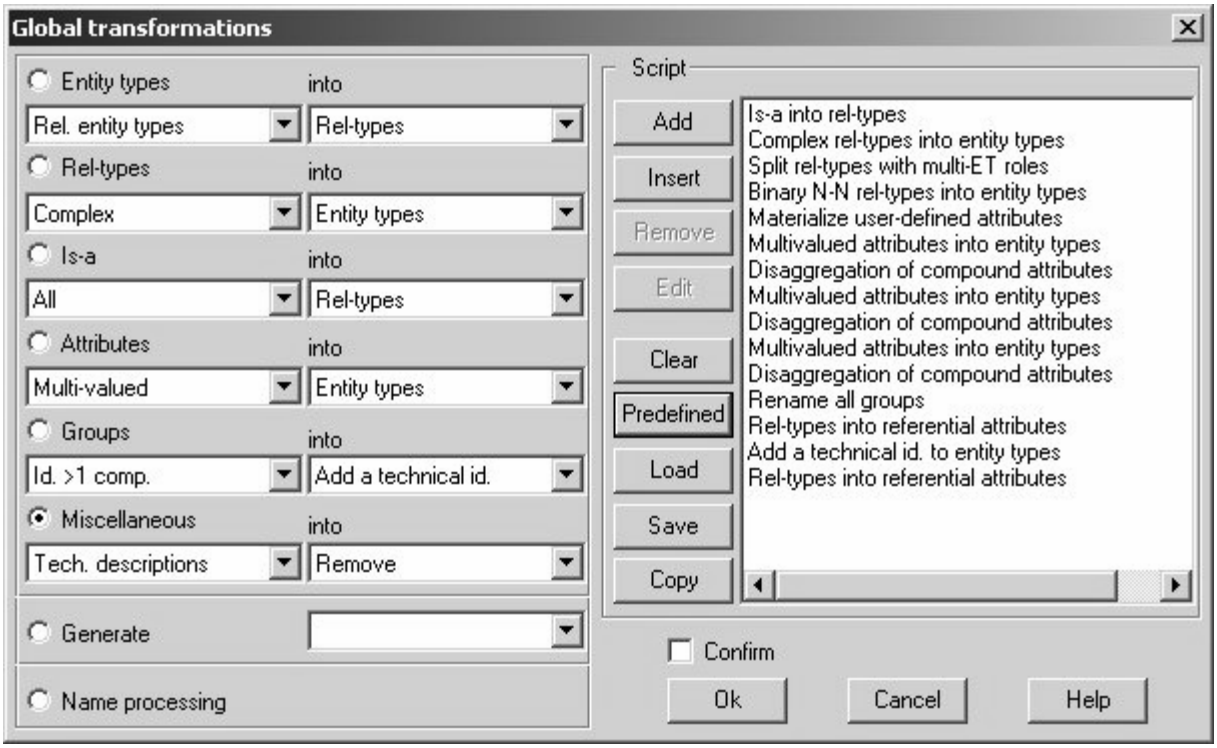


Figure 7.5 - Assistant de transformations globales (menu "Assist/Global transformation...").

Le deuxième assistant (appelé *assistant de transformations globales avancées*) est une version plus sophistiquée de l'assistant de transformations globales fournissant plus de flexibilité et de puissance dans le développement d'un script. Un script est composé de transformations et de structures de contrôle. Une transformation a la forme A(P) où A est une action (transformer, enlever, marquer, ...) et P un prédicat qui sélectionne des objects spécifiques dans un schéma de données. Le but est d'appliquer l'action A sur chaque objet vérifiant le prédicat P. Les structures de contrôle permettent de réduire le champ d'application d'un script ou de faire des boucles. L'utilisateur peut définir des librairies de transformations globales avancées et les réutiliser dans la définition d'autres librairies. La figure 7.6 montre la boîte de dialogue de l'assistant de transformations globales avancées avec un script permettant de transformer un schéma conceptuel en un schéma logique relationnel. Le script de cette figure est plus général que celui de la figure 7.5 car il contient une boucle permettant d'itérer sur les attributs multivalués et décomposables emboîtés. Il permet de transformer les attributs posant problème quelque soit leur niveau de profondeur. En dupliquant trois fois le couple de transformations permettant de désagréger les attributs décomposables et de transformer les attributs multivalués en type d'entités, le script de la figure 7.5 ne résout les problèmes que pour les attributs du premier au troisième niveau.

Pour plus de détails sur les transformations ainsi que les assistants, un aperçu des fonctionalités [Hic00b] de l'atelier ainsi qu'un tutoriel [Dbm99] sont disponibles à l'adresse http://www.db-main.be/references.html.

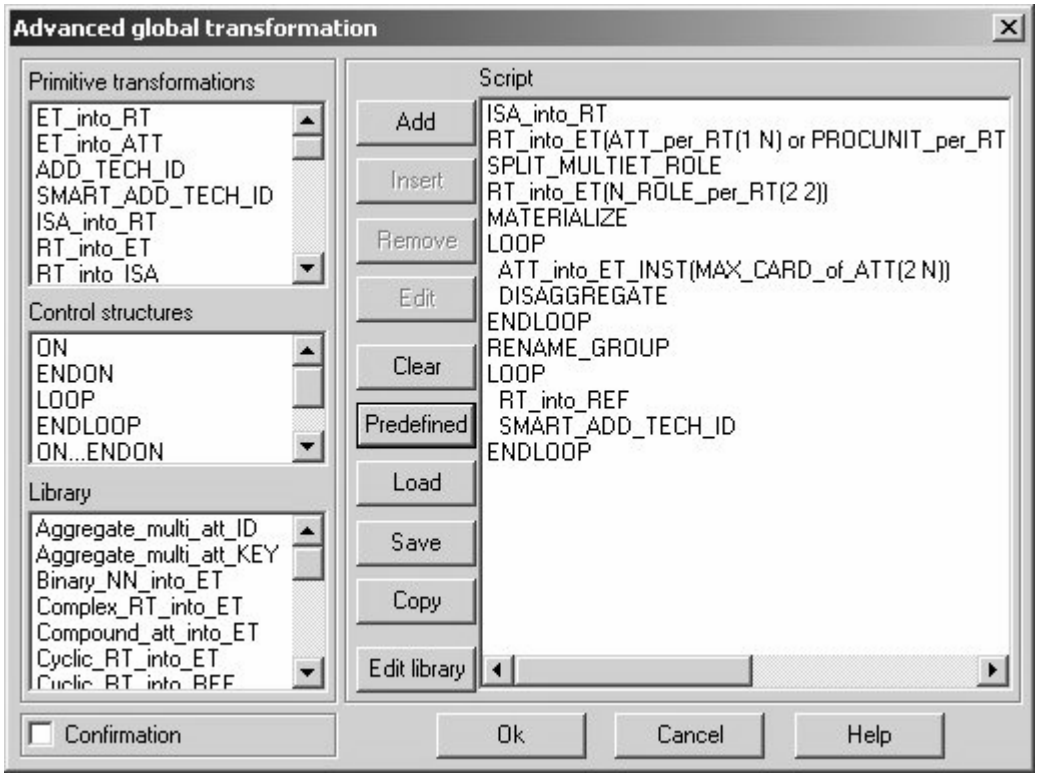


Figure 7.6 - Assistant de transformations globales avancées (menu "Assist/Advanced global transformation...").

7.1.4 Personnalisation méthodologique et gestion des historiques

L'atelier peut être utilisé dans des contextes très variés pour effectuer des activités comme la conception, la rétro-ingénierie, l'évolution, ... C'est pourquoi les fonctions de base de l'atelier (transformations, gestion, ...) sont méthodologiquement neutres. Toutefois, il peut être paramètré pour refléter une démarche particulière, liée, par exemple, à la culture méthodologique locale d'une entreprise. Toutes les activités qui sont réalisées dans l'atelier et qui suivent une méthode sont enregistrées dans un historique. Des informations précises sur la personnalisation méthodologique et les historiques sont disponibles dans [Rol97], [Rol99a] et [Rol00].

7.1.4.1 Personnalisation méthodologique

La spécialisation méthodologique de l'atelier est basée sur un modèle des processus de conception qui permet de spécifier, via un langage adéquat, une méthode particulière. Ce langage de modélisation exécutable, appelé *MDL* [4], permet de décrire des méthodes d'ingénierie de bases de données.

Les méthodes sont interprétées par le moteur méthodologique de l'atelier qui contrôle à la fois les actions permises à l'utilisateur et la présentation du contenu du référentiel. Lorsqu'on utilise une méthode, l'atelier guide l'utilisateur dans son utilisation des fonctionnalités disponibles. Il peut aussi bien exécuter de manière automatique des processus bien définis (par exemple, transformer un schéma conceptuel en schéma logique) ou présenter à l'analyste la liste des processus qu'il doit ou peut réaliser en le laissant accomplir le travail de manière semi-contrôlée (par exemple, concevoir un schéma conceptuel). Le moteur méthodologique agit sur l'interface

4. pour "Method Description Language".

utilisateur (sous-modèle actif, terminologie, ...) et sur l'accessibilité des différentes fonctionnalités de l'atelier. La figure 7.7 a) présente une méthode classique de conception de base de données.

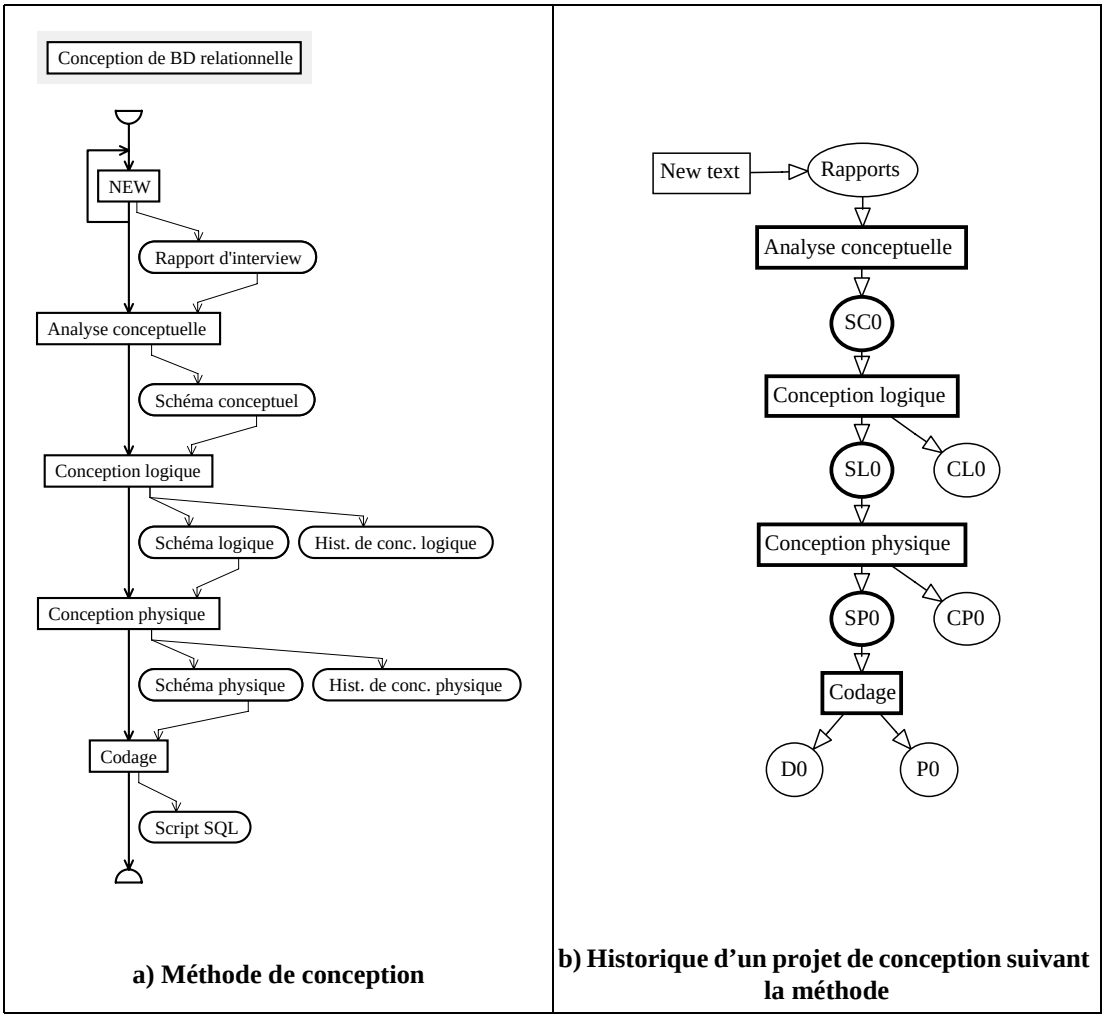


Figure 7.7 - Méthode classique de conception de bases de données relationnelles avec l'enregistrement des historiques de conception et un historique correspondant.

La procédure à suivre est la suivante : la méthode choisie est décrite dans le langage de spécification de méthodes, puis sa définition est chargée dans le référentiel du projet sous la forme, entre autres choses, de modèles et de types de produits et de types de processus. Dès cet instant, le projet est géré selon les concepts et la démarche de la méthode choisie.

La personnalisation méthodologique autorise essentiellement :

- une adaptation à la culture locale,
- une guidance méthodologique,
- un enregistrement automatique de l'historique des activités,
- un support des activités de maintenance et d'évolution des bases de données.

7.1.4.2 Le langage MDL

Le langage MDL a pour but de décrire la façon de travailler (appelée *méthode*) des analystes lorsqu'ils accomplissent des activités d'ingénierie. La personnalisation méthodologique est basée sur l'approche transformationnelle selon laquelle chaque processus transforme un ensemble de produits en un autre ensemble de produits. Un exemple de méthode décrite en MDL est disponible dans l'annexe E.1. Le langage est basé sur les concepts suivants :

- Un *produit* est un document utilisé, modifié ou produit durant le cycle de vie d'un projet. Il s'agit essentiellement de schémas de base de données et de textes tels que des scripts DDL, des rapports, ...
- Un *processus* est décrit par les opérations qui sont exécutées pour transformer des produits en entrée en produits en sortie, chaque opération étant à son tour un processus.
- Un *type de produits* décrit les propriétés d'une classe de produits jouant un rôle défini dans le système. Un produit est une instance d'un type de produits.
- Un *type de processus* décrit les propriétés générales d'une classe de processus qui ont le même objectif. Un processus est une instance d'un type de processus.
- Une *stratégie* d'un type de processus spécifie comment le processus doit être exécuté de manière à résoudre les problèmes pour produire les produits qui satisfont les exigences.
- Un *modèle de produits* définit une classe générale de produits en définissant les composants qui sont autorisés, les contraintes qui doivent être satisfaites et la terminologie qui doit être utilisée. Un type de produit est exprimé dans un modèle de produits.
- Une *méthode* est un ensemble de modèles, de types de produits et de types de processus.

Le langage MDL est non déterministe car il permet de décrire des processus exécutés par une intervention humaine. Il est également semi-procédural car il oblige l'analyste à respecter un ordre dans l'exécution des tâches. Une définition précise du langage peut être trouvée dans [Rol99b].

La description d'une méthode est scindée en deux parties. La première décrit les modèles de produits. Le modèle de représentation des schémas étant générique, il offre un très grand nombre de degrés de liberté, conduisant par exemple à mélanger des concepts de paradigmes ou de niveaux d'abstraction différents dans un même schéma. Un modèle de produits est composé d'un sous-ensemble du modèle de représentation. Il restreint le modèle générique en spécifiant une série de contraintes sur ce modèle et il adapte la terminologie des différents objets. Par exemple, pour définir un modèle de représentation des schémas relationnels, les contraintes suivantes doivent être imposées :

- pas de relation IS-A;
- pas de types d'associations;
- pas d'attributs décomposables;
- pas d'attributs multivalués;
- pas de types d'entités sans attributs.

Tout schéma qui respecte ces contraintes est conforme au modèle relationnel. Quant à la terminologie, elle devra être adaptée comme suit :

- collection $\rightarrow$ espace de stockage,
- type d'entités $\rightarrow$ table,
- attribut $\rightarrow$ colonne,
- identifiant primaire $\rightarrow$ clé primaire.

La deuxième partie de la description de la méthode s'occupe des types de processus. Un type de processus a un corps et une signature :

- La signature est faite d'un nom, d'un type de produits à recevoir en entrée, à fournir en sortie ou à modifier.
- Le corps est la stratégie qui doit être suivie. Une stratégie est une séquence d'opérations et de structures de contrôle.
- Une opération est soit :
  - Un appel à un autre type de processus.

  - Un appel à une fonction supportée par l'atelier.
  - Un appel à une fonction externe écrite en Voyager 2.
  - L'utilisation d'une *boîte à outils* qui est une liste d'outils de l'atelier que l'analyste peut utiliser à un moment donné. L'appel d'une boîte à outils suspend la méthode et passe la main à l'analyste qui peut faire ce qu'il veut avec les outils autorisés par la boîte à outils. Lorsqu'il a terminé, l'analyste rend la main à la méthode.
- Les structures de contrôle sont les traditionnels `if ... then ... else ...`, `while`, `repeat` qui peuvent prendre des décisions déterministes (basées sur une analyse des produits) ou non déterministes (décisions humaines) ainsi que des structures (non déterministes) plus spécifiques comme `one`, `some` et `each`.

7.1.4.3 Historiques

L'historique d'un processus d'ingénierie d'une base de données contient la trace des activités exécutées, les produits impliqués, les décisions et les hypothèses prises. DB-Main permet l'enregistrement sous la forme d'un historique de toutes les manipulations effectuées. Celui-ci peut avoir de nombreuses utilisations telles que :

  - documenter la conception d'une application en enregistrant toutes les actions qui ont conduit à sa réalisation;
  - annuler la ou les dernières actions effectuées;
  - rejouer une conception à partir du schéma de départ légèrement modifié (problème de l'évolution);
  - recouvrer une conception, c'est-à-dire reconstruire un historique possible d'une application à partir de l'historique de la rétro-ingénierie de cette application;
  - analyser la méthode suivie afin de la valider ou de l'améliorer.

Examinons les composants de base d'un historique dans l'atelier.

Processus

L'historique contient tous les processus exécutés et représente un arbre d'appel des processus. Chaque processus peut être composé de sous-processus. Il y a deux sortes de processus dans l'atelier :

- Les *processus primitifs* sont des processus exécutés en utilisant des primitives de base de l'atelier (soit internes comme les transformations, soit externes comme une fonction Voyager 2). L'exécution d'une primitive est enregistrée dans un historique plat, appelé aussi journal [5]. Cet historique est un fichier texte décrivant dans une syntaxe prédéfinie (propre à l'atelier DB-Main) l'exécution de primitives, c'est-à-dire, qu'il contient les enregistrements des opérations effectuées sur le schéma de spécifications. L'annexe E.2 donne un exemple de journal pour l'étude de cas du chapitre 8.
- Les *processus d'ingénierie* suivent une stratégie donnée par la méthode. Puisque l'analyste peut faire des hypothèses et prendre des décisions, il n'est pas possible de garder une trace de ces actions d'une façon linéaire dans un journal. L'historique d'un processus d'ingénierie est un graphe.

Produits

Les seconds éléments de base sont les produits. Dans l'atelier, un produit est soit un texte soit un schéma de données. Comme il est possible de générer plusieurs versions d'un produit, il est identifié par son nom et sa version. Un produit peut être utilisé par un ou plusieurs processus en

5. "Log file" en anglais.

tant qu'entrée (le produit est utilisé comme entrée du processus mais il ne peut être modifié), sortie (le produit est créé par un processus) ou modification (le produit est modifié par un processus). Un produit peut aussi avoir un verrou (lock) qui signifie que le produit est stabilisé et qu'il n'est plus possible de le modifier.

Décisions

Les décisions sont les troisièmes éléments de base des historiques. Une décision est un processus spécial qui ne peut ni modifier ni générer de produits. Il y a deux sortes de décisions :

- Une décision doit être prise selon la méthode suivie. C'est le cas lorsque la condition d'une instruction (if, while) nécessite une réponse de l'analyste.
- Une décision suit une hypothèse. Quand un analyste exécute un processus et qu'il est confronté à un problème, il peut faire différentes hypothèses et exécuter le même processus autant de fois qu'il y a d'hypothèses. Chaque exécution du processus génère ses propres versions de produits. L'analyste peut choisir une version, qu'il qualifiera de meilleure solution, avant de continuer son activité. Ce deuxième type de décision n'est pas lié à la méthode.

Dans l'atelier, l'historique complet est un arbre de graphes où les feuilles sont des processus primitifs avec leur journal et les autres nœuds sont des processus d'ingénierie avec leur graphe composé d'autres processus, de produits et de décisions. Les historiques sont enregistrés de manière automatique. Le journal peut être de deux types. Un journal simple contient uniquement les signatures des transformations exécutées sur un schéma. Il peut être rejoué facilement mais ne peut être inversé. Un journal complet contient la signature des transformations plus la description des opérations effectuées par la transformation. Par exemple, l'enregistrement de la destruction d'un type d'entités entraîne l'enregistrement de la destruction de ses groupes et de ses attributs. Ce type d'historique peut être inversé. L'atelier autorise plusieurs opérations de manipulation et de traitement sur les journaux :

- autoriser ou interdire l'enregistrement (menu "Log/Trace");
- ajouter des points de repères dans le fichier (menu "Log/Add check point");
- ajouter des commentaires (menu "Log/Add desc.");
- effacer le journal d'un schéma (menu "Log/Clear log");
- sauver le journal dans un fichier texte (menu "Log/Save log as...");
- rejouer un journal automatiquement (menu "Log/Replay/Automatic...") ou interactivement (menu "Log/Replay/Interactif...") c'est-à-dire pas à pas ou de point de repère en point de repère (en sautant éventuellement des enregistrements);
- inverser un journal (fonction Voyager 2).

La figure 7.7 b) présente un exemple d'historique d'une activité de conception correspondant à la méthodologie définie dans la figure 7.7 a).

7.1.5 Extensibilité (Voyager 2)

DB-Main est une plate-forme de base qui ne peut assumer à elle seule toutes les tâches rencontrées dans le cycle de vie d'une application. En effet, elle est d'une part souvent utilisée conjointement avec d'autres produits complémentaires (AGL, dictionnaires de données, SGBD, L4G, ...) et présente, d'autre part, des lacunes vis-à-vis des attentes de l'ingénieur méthode. Dès lors, il s'avère que pour assurer la viabilité d'un outil, il est nécessaire de lui adjoindre un moyen de personnalisation tel un environnement de programmation permettant l'ajout dynamique de nouvelles fonctionnalités. Non seulement l'outil pourra communiquer avec d'autres, mais l'utilisateur pourra développer, et inclure dans l'atelier, des fonctions qui lui sont personnelles. C'est sur la base de

cette constatation que le langage et l'environnement de programmation Voyager 2 ont été définis.

Voyager 2 allie puissance et simplicité pour rendre la personnalisation de l'outil aisée comme le montre ses caractéristiques (une description ainsi que la syntaxe de Voyager 2 est disponible dans [Eng99] et [Eng00]) :

- Langage faiblement typé : ce mécanisme accentue l'indépendance du langage vis-à-vis du référentiel utilisé dans DB-Main. Par ailleurs, la définition du référentiel étant orientée-objet, le langage supporte le typage dynamique.
- Procédural : le langage supporte les procédures/fonctions avec appels récursifs.
- Requêtes prédicatives et navigationnelles : des requêtes concises et efficaces rendent aisées aussi bien la consultation que la modification du référentiel de l'atelier.
- Définition d'un type liste : les listes peuvent contenir des informations de tout type et leur gestion est assurée par un "Garbage Collector" [6]. Elles sont utilisées aussi bien comme arguments que comme structures hôtes pour les résultats des requêtes.
- Analyseur lexical : le langage permet d'écrire très rapidement des fonctions d'importation de fichiers textes par l'utilisation d'expressions régulières.
- Accès aux outils de l'atelier : les fonctionnalités intrinsèques de l'atelier (outils de base) sont accessibles dans le langage.
- Fonctions d'interfaçage graphique : échange avec l'utilisateur via des boîtes de dialogue et une console.
- Communications avec des programmes Windows : possibilité d'appeler des programmes Windows.
- Instruction d'entrée/sortie : instructions classiques de gestion de fichiers.
- Programmation modulaire.
- Enrichissement de l'atelier avec des fonctionnalités externes : les fonctions et procédures d'un programme Voyager 2 peuvent être invoquées directement dans l'atelier par les assistants de transformations globales, d'analyse de schéma et de rétro-ingénierie.

Les programmes écrit en Voyager 2 sont précompilés en code binaire directement exécutable par la machine virtuelle VOYAGER (qui fait partie de l'atelier). Cela rend impossible la distinction (en temps et place occupée) de fonctionnalités écrites en code natif (C++) de celles écrites en Voyager 2. Voyager 2 doit donc être utilisé pour développer et distribuer certaines fonctionnalités de l'atelier que l'utilisateur pourrait désirer aménager selon ses besoins. Dans cette optique, il est opportun que le convertisseur de données et de structures de données développé dans le cadre de ce travail soit implémenté en Voyager 2. Vu les différences entre les syntaxes SQL acceptées par les SGBD disponibles sur le marché, ce programme peut aisément être modifié et recompilé afin de produire un nouveau générateur de scripts de conversion spécifique à un SGBD et aux standards propres à une entreprise.

L'exemple de la figure 7.8 illustre l'emploi d'un programme Voyager 2 pour étendre DB-Main avec des fonctions statistiques. Ce programme utilise le référentiel de l'atelier pour fournir des informations sur le nombre de types d'entités et la profondeur maximale des attributs décomposables. La procédure *stat* est déclarée *export* (ligne 2) pour la rendre visible dans l'atelier; elle apparaît sous forme d'un nouvel item dans un menu. La boucle d'itération (ligne 7) dans cette fonction parcourt tous les types d'entités du schéma ouvert (*GetCurrentSchema*) au moment de l'appel dans l'atelier. Pour chaque type d'entités, la fonction *prof* retourne la profondeur maximale des attributs du type d'entités. Les lignes 7, 12 et 19 montrent l'utilisation de constantes du type liste utilisées dans les boucles, comme arguments, et dans les requêtes.

6. Technique de gestion de la mémoire.

```
1:  /* Cette fonction peut être invoquée directement à partir de l'atelier */
2:  export procedure stat()
3:    entity: e;
4:    integer: ne, namax;
5:  { ne:=0;
6:    namax:=0;
7:    for e in ENTITY[e]{IN:[GetCurrentSchema()]} do {
8:      /* pour chaque entité e du schéma courant */
9:      ne:=ne+1;
10:     namax:=max(namax,prof(e));
11:   };
12:   print(["entités:",ne,"\nprofondeur:",namax,'\n']);
13: }
14: /* retourne la profondeur maximale des attributs qui composent l'objet o */
15: function integer prof(owner: o)
16:   integer: max_temp;
17:   attribute: a;
18: { max_temp:=0;
19:   for a in ATTRIBUTE[a]{OWN:[o]} do {
20:     /* pour chaque attribut a dont e est composé */
21:     max_temp:=max(max_temp,prof(a));
22:   };
23:   return max_temp+1;
24: }
```

Figure 7.8 - Un programme Voyager 2 comportant une procédure exportable (stat) qui permet d'étendre l'atelier DB-Main avec des fonctions statistiques.

7.1.6 Outils de compréhension de programmes

Plusieurs outils de compréhension de programmes sont également disponibles dans l'atelier : la recherche de patrons, le calcul de graphe de dépendances des variables et la fragmentation de programmes. Pour aider l'analyste dans la modification des programmes de son application de bases de données, ces outils sont le moyen idéal pour comprendre les programmes et générer des rapports de modifications de ceux-ci (section 7.4). Vu la multitude d'environnements présents sur le marché, il nous semble important de choisir un outil facilement adaptable à tous les contextes que l'analyste pourrait rencontrer. Les points suivants présentent la recherche de patrons, le graphe de dépendances des variables et la fragmentation de programmes. Pour plus de détails, les outils de compréhension de programmes de l'atelier DB-Main sont décrits dans [Hen96], [Hen98a] et [Hen98b].

7.1.6.1 Recherche de patrons textuels

Une des techniques de compréhension de programmes la plus simple est la recherche de patrons (ou clichés de programmation) dans un texte source. DB-Main dispose d'un moteur de recherche plus puissant qu'une simple recherche de chaînes de caractères comme le font la plupart des éditeurs de textes. Les patrons à rechercher sont définis dans un langage de définition de patrons (PDL - "Pattern Description Language") fort proche de la notation BNF.

La définition d'un patron peut faire appel à la définition d'un patron déjà défini. Par exemple, la définition du nom des variables COBOL ou d'une expression de comparaison est réutilisable pour définir des patrons plus complexes.

Le langage PDL dispose de variables (préfixées par @). Lorsqu'un patron est instancié, les variables de ce patron reçoivent une valeur qui peut être utilisée, par exemple, comme paramètre lors de l'exécution d'une procédure Voyager ou pour construire le graphe de dépendances (point 7.1.6.2). Ces variables peuvent également recevoir une valeur, ce qui permet d'instancier partiellement un patron avant d'effectuer la recherche.

Cet outil de recherche peut être utilisé pour effectuer une analyse fine de textes sources et des descriptions textuelles des objets du référentiel. La recherche permet une inspection visuelle des

textes ou, par couplage avec des fonctions développées en Voyager 2, de déclencher des actions sur le référentiel. L'analyste peut donc développer une base de connaissances relative à la phase d'extraction de structures de données en rétro-ingénierie. Dans la figure 7.9, le patron *joint* montre la définition d'un patron qui permet de retrouver les requêtes SQL basées sur la jointure de deux tables. @alias1, @alias2, @table1, @table2, @col1 et @col2 sont des variables qui seront instanciées lors de la recherche, elles doivent aussi être définies comme des patrons. Le caractère "-" représente un ou plusieurs espaces obligatoires, tandis que le caractère "~" représente des espaces facultatifs. Les caractères entre guillemets représentent des constantes qui sont recherchées telles quelles. "/g" introduit une expression régulière. Une définition complète du langage PDL est disponible dans l'aide de l'outil DB-Main.

```
mot ::= /g"[a-zA-Z][a-zA-Z0-9_]*";
table1 ::= mot;
table2 ::= mot;
col1 ::= mot;
col2 ::= mot;
alias1 ::= mot;
alias2 ::= mot;
joint ::= "from" - @alias1 - @table1 ~ "," ~ @alias2 - @table2 -
          "where" - @alias1"."@col1 ~ "=" ~ @alias2"."@col2 ;
```

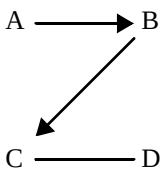
Figure 7.9 - Définition du patron "joint".

7.1.6.2 Graphe de dépendances

Le graphe de dépendances (qui est une généralisation du diagramme de flux) est un graphe où chaque variable du programme est représentée par un nœud et dont les arcs (orientés ou non) représentent une relation (assignation, comparaison,...) entre deux variables. Ces relations sont choisies par l'analyste sous la forme d'une liste de patrons à deux variables.

```
...
A = B;
...
B = C;
...
IF (C <= D) ...
```

a) Un fragment de programme C



b) Le graphe de dépendances du fragment

Figure 7.10 - Le graphe de dépendances d'un fragment de programme C.

La figure 7.10 b) est le graphe de dépendances de l'extrait du programme de la figure 7.10 a). Le graphe se parcourt dans le sens des flèches des arcs (un arc non orienté peut être traversé dans les deux sens). S'il y a un chemin entre la variable A et la variable C dans le graphe de dépendances, alors il existe dans le programme une séquence d'instructions telle que la valeur de A est en relation avec la valeur de C. Quand deux variables sont en relation, cela signifie que les deux variables ont la même structure ou au moins la structure de l'une des deux est incluse dans l'autre.

Dans le problème de l'évolution, le graphe de dépendances est un moyen très efficace pour retrouver les variables qui dépendent l'une de l'autre. S'il existe un chemin passant entre deux variables dont la structure d'une des deux est modifiée, alors on peut affirmer que la structure de l'autre devra également être modifiée.

Dans DB-Main, l'utilisation du graphe de dépendances se fait en deux parties : le calcul du graphe et son utilisation. Pour calculer le graphe de dépendances d'un texte source, l'analyste donne une liste de patrons. Chacun de ces patrons doit contenir deux variables (var_1 et var_2). Le graphe définit qu'il y a une relation entre var_1 et var_2, orientée de var_1 vers var_2 si l'ana-

lyste décide que le patron est orienté. Les nœuds du graphe qui sera construit seront les instanciations des variables var_1 et var_2. La figure 7.11 présente la liste des patrons nécessaires pour calculer le graphe de dépendances de la figure 7.10 ainsi que les instanciations des variables var_1 et var_2 pour les instructions du programme. Le patron *assignation* définit une relation orientée de var_1 vers var_2 et le patron *inferieur-egal* définit une relation non orientée.

```
mot ::= /g"[a-zA-Z][a-zA-Z0-9_]*";
var_1 ::= mot;
var_2 ::= mot;
assignation ::= @var_2 - "=" - @var_1
inferieur-egal ::= @var_1 - "<=" - @var_2
```

a) Liste des patrons

```
A = B;      : (var_1 = B) -> (var_2 = A)
B = C;      : (var_1 = C) -> (var_2 = B)
IF (C <= D) : (var_1 = C) <-> (var_2 = D)
```

b) Instanciation des variables

Figure 7.11 - Liste des patrons et instanciations des variables de la figure 7.10.

L'utilisation du graphe de dépendances est simple, toutes les occurrences des variables qui sont liées (directement ou indirectement) à une variable sélectionnée par l'utilisateur sont colorées dans le texte source. Le graphe n'est pas représenté sous une forme graphique mais il est présenté dans le texte source lui-même. Cela permet à l'analyste de visualiser le contexte dans lequel ces variables sont utilisées.

7.1.6.3 Fragmentation de programmes

Une autre technique fort intéressante pour la compréhension de programmes est la *fragmentation de programme*s qui permet d'extraire d'un programme le fragment nécessaire et suffisant (idéalement) pour comprendre et expliquer le comportement de ce programme à un point déterminé. Ce point, appelé *critère de fragmentation*, est constitué d'un couple [instruction, variable] tel que la variable est référencée dans l'instruction. Idéalement, un fragment de programme est composé de toutes les instructions qui affectent les valeurs calculées par rapport au critère de fragmentation et uniquement celles-là.

La figure 7.12 a) présente un extrait d'un programme COBOL qui demande le numéro d'un client et qui affiche son nom et le montant total de ses commandes. La figure 7.12 b) montre le fragment de programmes contenant les instructions qui ont contribué à l'affichage du nom du client.

```
FD CUSTOMER.
   01 CUS.
      02 CUS-NUM PIC 9(3).
      02 CUS-NAME PIC X(10).
      02 CUS-ORD PIC 9(2).
         OCCURS 10.
      ...
   01 ORDER PIC 9(3).
      ...
1  ACCEPT CUS-NUM.
2  READ CUS KEY IS CUS-NUM.
3  MOVE 1 TO IND.
4  MOVE 0 TO ORDER.
5  PERFORM UNTIL IND=10
6  ADD CUS-ORD(IND) TO ORDER
7  ADD 1 TO IND.
8  DISPLAY CUS-NAME.
9  DISPLAY ORDER.
```

a) Un extrait de programme COBOL.

```
FD CUSTOMER.
   01 CUS.
      02 CUS-NUM PIC 9(3).
      02 CUS-NAME PIC X(10).



1  ACCEPT CUS-NUM.
2  READ CUS KEY IS CUS-NUM.




8  DISPLAY CUS-NAME.
```

b) Le fragment du programme correspondant à la variable CUS-NAME et l'instruction de la ligne 8.

Figure 7.12 - Extrait d'un programme COBOL (a) et un fragment de ce programme (b).

Le concept de fragments de programmes a été introduit par Weiser [Wei84]. Celui-ci prétend qu'un fragment correspond à l'abstraction mentale que fait un programmeur quand il dissèque un programme. Différentes notions de fragmentation de programmes et méthodes de calcul ont été définies depuis lors, répondant chacune à des exigences particulières.

Les caractéristiques des différents langages de programmation comme les procédures, les contrôles de flux arbitraires (goto), les types de données composites et les pointeurs nécessitent des solutions spécifiques. L'outil de fragmentation de programmes de l'atelier a été développé dans le cadre de la rétro-ingénierie de bases de données, il analyse des programmes COBOL comportant des procédures, des types de données composées et des contrôles de flux arbitraires.

La fragmentation de programmes intervient lorsque l'on veut comprendre comment a été construite la valeur d'une variable à un point donné du programme. Dans le cadre de ce travail, on calculera souvent un fragment pour déterminer les séquences d'instructions qui doivent être mises dans une itération lors du processus de modification des programmes. Le critère de fragmentation est souvent constitué d'une requête SQL faisant intervenir des variables (ou un curseur) pour stocker un résultat (requête de sélection) ou insérer des valeurs dans la base de données (requête d'insertion ou de modification).

L'outil de fragmentation de programmes de DB-Main est inspiré de la technique proposée dans [Hor90] pour le calcul de fragments inter-procéduraux. Le calcul d'un fragment est défini en termes de parcours du *graphe de dépendances du système* (SDG - "System Dependence Graph"). Le SDG est un graphe orienté dont les nœuds correspondent aux instructions et les arcs représentent les dépendances de données, les dépendances de contrôle et les appels de procédures. Un critère de fragmentation est identifié à un nœud du graphe. Un fragment est l'ensemble des nœuds du SDG à partir desquels on peut atteindre le nœud représentant le critère de fragmentation. Pour adapter cet outil à d'autres langage (C, C++, Java, ...), il faut modifier l'algorithme de construction du SDG. La gestion du SDG par l'atelier est, quant à elle, indépendante du langage employé dans les programmes.

Dans DB-Main, l'outil de fragmentation de programmes permet de définir un critère de fragmentation constitué d'une instruction (nœud du SDG) et d'une ou plusieurs variables référencées dans cette instruction. Les instructions qui font partie du fragment sont colorées dans l'éditeur de texte source. Cela permet de visualiser le fragment dans son contexte. Les commentaires ou les instructions d'affichage de messages d'erreur ne font pas partie du fragment, car ils n'influencent pas la valeur des variables du critère de fragmentation, mais ils peuvent s'avérer particulièrement utiles à la compréhension du fragment. Il est également possible d'isoler les lignes du fragment et de les copier dans un fichier à part.

DB-Main dispose également d'un assistant qui permet de calculer l'union ou l'intersection de plusieurs fragments et de calculer les fragments pour toutes les instructions d'un type donné.

7.2 Implémentation des stratégies de propagations des modifications

En utilisant les fonctionnalités de l'atelier DB-Main, nous allons montrer la démarche à suivre pour mettre en œuvre les stratégies de propagation développées dans le chapitre 5. Cette section montre comment les trois stratégies sont implémentées grâce à la modélisation de processus. Pour ce qui est de la démarche de reconstruction des spécifications, sa modélisation sous la forme de processus est abordée au point 7.2.1.

7.2.1 Implémentation de la reconstruction des spécifications

Conformément au processus de rétro-ingénierie présenté dans la section 5.2, la méthodologie de reconstruction des spécifications est composée de deux processus principaux : l'extraction et la conceptualisation des structures de données. Seules les grands processus de la méthode sont présentés sans entrer dans les détails. Il s'agit de montrer le principe de la redocumentation satisfaisante d'une application pour permettre l'application des trois stratégies de propagation des modifications. La rétro-ingénierie n'est toutefois pas approfondie car elle nécessite de longs développements qui sortent du cadre de ce travail.

L'extraction des structures de données (figure 7.13 a) met en œuvre quatre processus. Le processus d'auto-extraction analyse les fichiers contenant du code DDL et produit les schémas physiques bruts. Dans le cas assez fréquent où on dispose de plusieurs schémas bruts, le processus d'intégration physique les fusionne dans un schéma physique augmenté SP0. Grâce à une analyse minutieuse de toutes les sources d'information et, notamment, des programmes, le processus de raffinement de schéma complète SP0 pour obtenir le schéma physique final. Le concepteur réalise une copie de SP0 nommée SL0 sur laquelle il applique le processus de nettoyage qui supprime les constructions techniques comme les index, les espaces de stockage, en enregistrant l'historique des transformations dans un journal CP0'. Il obtient le schéma logique final SL0 et l'historique de la rétro-ingénierie physique CP0' contenant les transformations de traduction de SP0 en SL0.

Le processus de conceptualisation ou interprétation des structures de données reconstruit le schéma conceptuel SC0 à partir de SL0 (figure 7.13 b). Après avoir demandé au concepteur de réaliser une copie de SL0 nommée SC0, il met en œuvre quatre processus et déclenche simultanément l'enregistrement des transformations dans un journal CL0'. Le processus de préparation supprime de SC0 les structures de données obsolètes ou techniques, renomme les objets ou réorganise les objets de SC0 pour une meilleur visualisation des spécifications. Les processus de détraduction et de désoptimisation éliminent ou transforment les structures non conceptuelles. Le processus de normalisation conceptuelle restructure SC0 pour satisfaire les exigences requises pour un schéma conceptuel. Le processus de conceptualisation retourne comme résultats le schéma conceptuel final SC0 et l'historique de rétro-ingénierie logique CL0' contenant les transformations de conceptualisation appliquées sur SL0 pour obtenir SC0.

Pour compléter la démarche, les historiques CP0' et CL0' sont inversés pour donner les processus de conception physique (CP0) et logique (CL0). Ainsi une documentation complète des spécifications a été recréée. Les schémas (conceptuel, logique et physique) et les historiques de conception (physique et logique) sont disponibles pour appliquer les stratégies de propagation développées dans les sections suivantes.

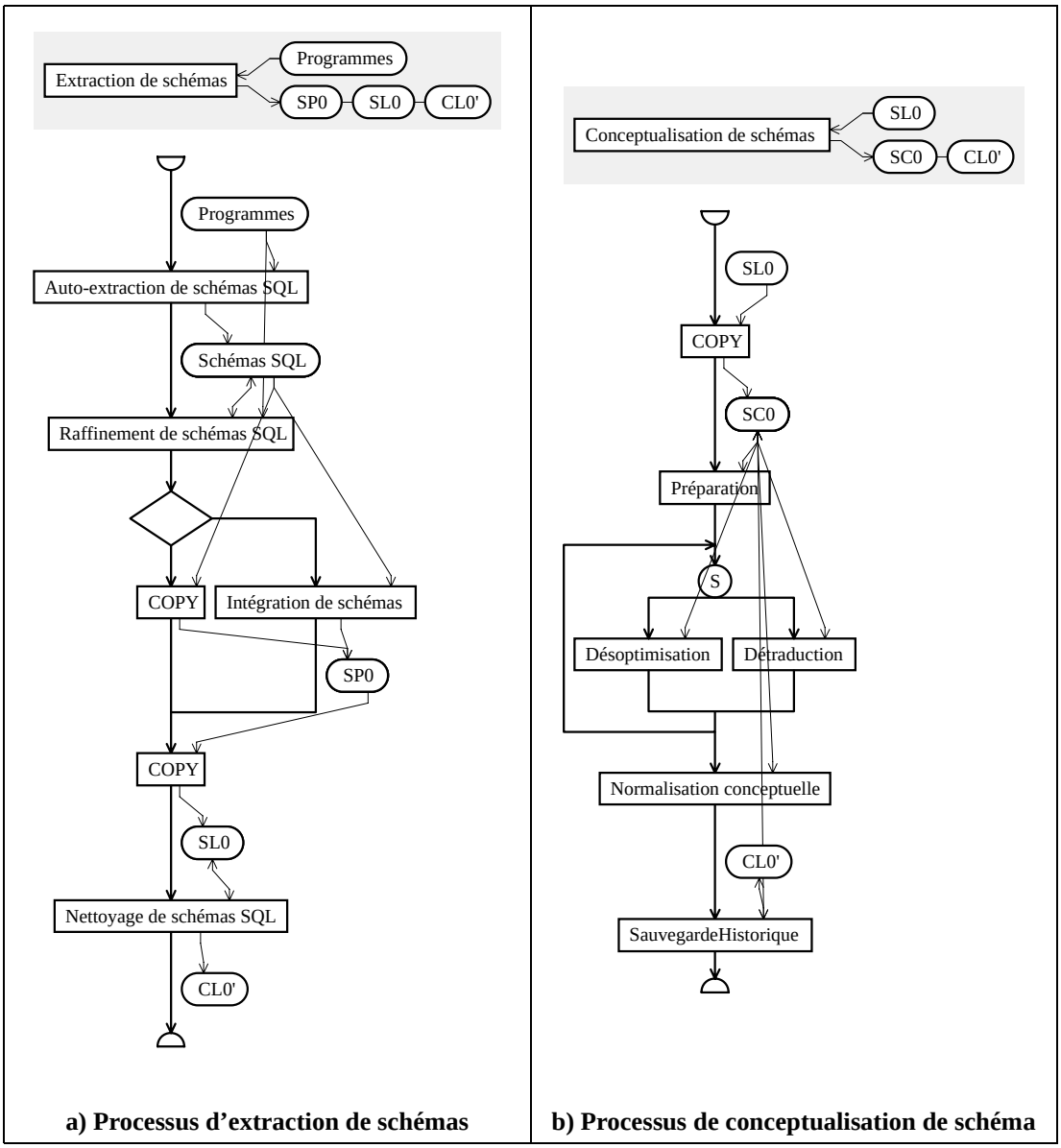


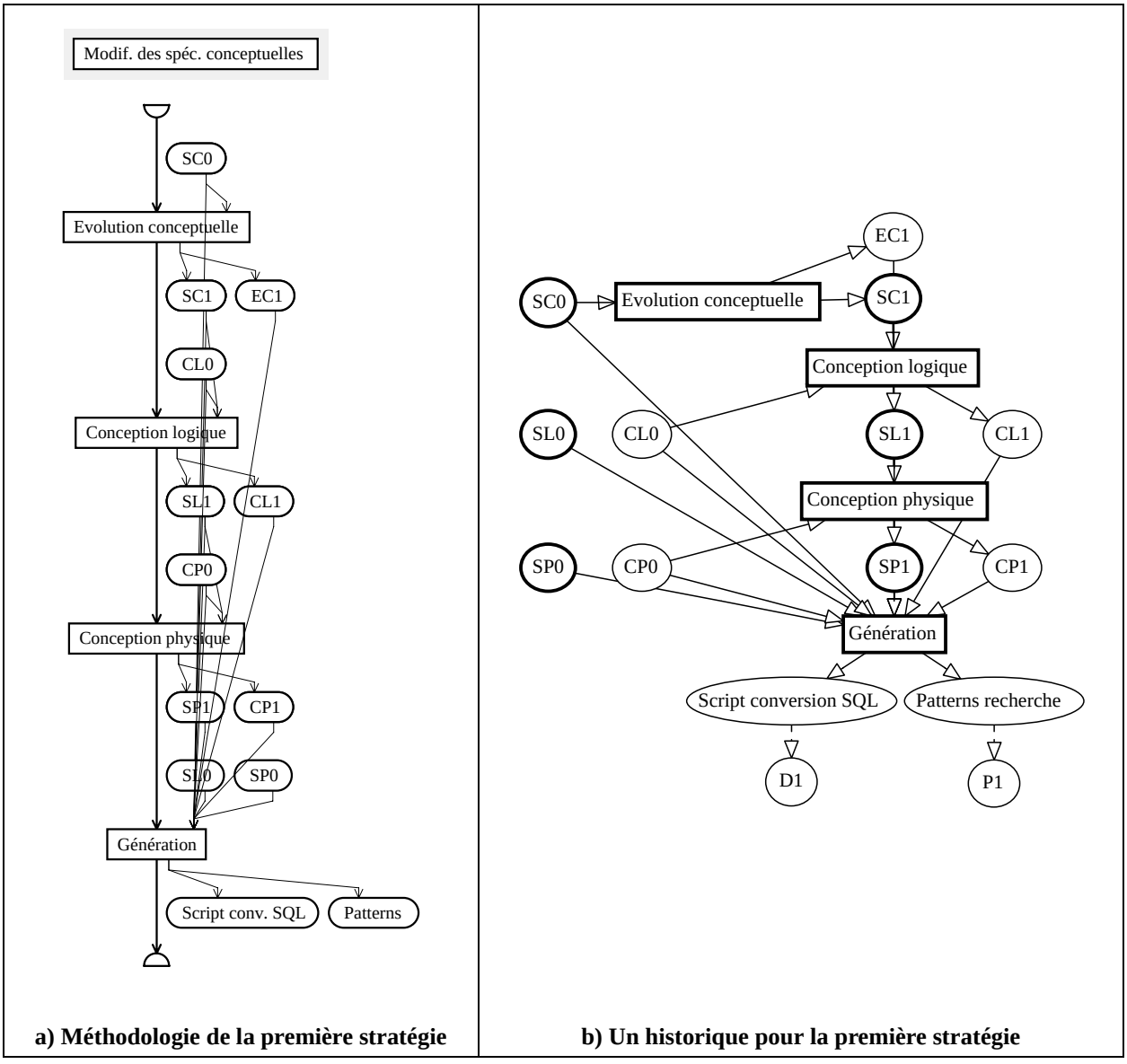
Figure 7.13 - Les deux processus principaux de la méthodologie de rétro-ingénierie.

7.2.2 Implémentation de la première stratégie : modifications des spécifications conceptuelles

Au départ, l'existant est composé du schéma conceptuel SC0, du schéma logique SL0, du schéma physique SP0, de l'historique CL0, de l'historique CP0, de la base de données D0 (scripts de création des structures de données relationnelles + données) et des programmes P0 (fichiers contenant les textes sources). Le processus d'analyse conceptuelle donne le schéma SC0 créé à partir des rapports d'interview. La conception logique transforme SC0 en un schéma logique relationnel SL0 et fournit l'historique CL0 contenant les transformations de traduction de SC0 en SL0. La conception physique transforme SL0 en un schéma physique relationnel SP0 avec CP0 l'historique des transformations de traduction de SL0 en SP0. Finalement, le processus de codage génère à partir de SP0 les instructions de création des structures de données. C'est aussi sur la base de SP0 que les programmes P0 sont construits. La méthode suivie est celle présentée à la figure 7.7 a) du point 7.1.4.

Les processeurs nécessaires à la gestion de l'évolution existent sous forme native (C++) dans le noyau de l'atelier (point 7.1) à l'exception du générateur de convertisseurs et de patrons de recherche qui sont développés en Voyager 2.

Examinons en détail les étapes de la démarche méthodologique pour la première stratégie. Ces étapes sont les principaux processus de la méthode de modifications des spécifications conceptuelles (figure 7.14a). L'historique (figure 7.14b) a été construit en suivant cette méthode. Pour information, le code MDL définissant cette méthode est disponible dans l'annexe E.1.



a) Méthodologie de la première stratégie **b) Un historique pour la première stratégie**

Figure 7.14 - Méthodologie de la stratégie des modifications des spécifications conceptuelles et un historique correspondant.

7.2.2.1 Processus d'évolution conceptuelle : production de la nouvelle version du schéma conceptuel SC1

Le processus d'évolution conceptuelle (figure 7.15) demande au concepteur de réaliser une copie de SC0, nommée SC1, et de la modifier en utilisant la boîte à outils des modifications conceptuelles autorisées (celles définies dans le point 5.3.1) et en déclenchant l'enregistrement des transformations. Lorsque ce processus primitif est terminé, l'historique est sauvegardé dans le journal EC1. Le processus d'évolution conceptuel retourne en sortie SC1 et EC1.

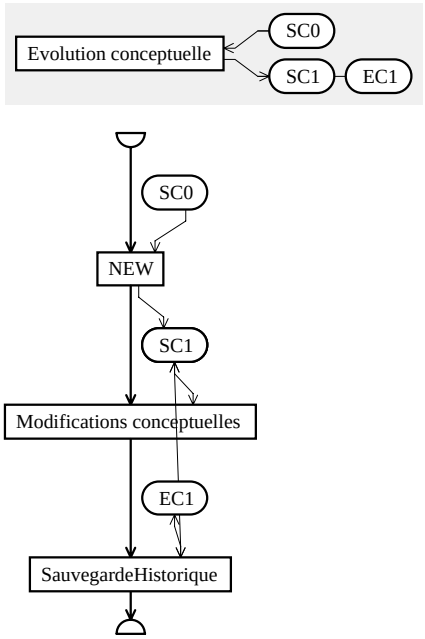


Figure 7.15 - Processus d'évolution conceptuel.

7.2.2.2 Processus de conception logique : production de la nouvelle version du schéma logique SL1

Le processus de conception logique (figure 7.16 a) demande une copie de SC1 (nommée SL1) sur laquelle il rejoue l'historique de conception logique original enregistré dans le journal CL0 en déclenchant l'enregistrement des opérations dans un nouvel historique. On obtient ainsi une première version incomplète du nouveau schéma logique SL1.

Les nouveaux composants de SC1 (provenant de la création de nouveaux objets ou de la modification d'objets de SC0) sont alors transformés spécifiquement grâce à la boîte à outils de conception logique (les opérations autorisées sont définies au point 5.3.2) et selon les heuristiques de transformation en vigueur dans l'entreprise. Ces opérations sont enregistrées dans l'historique de SL1 à la suite de celles enregistrées lors de l'exécution de CL0. Finalement, l'historique qui contient toutes les transformations exécutées pour obtenir SL1 à partir de SC1 est sauvegardé dans le journal CL1.

7.2.2.3 Processus de conception physique : production de la nouvelle version du schéma physique SP1

Le procesus de conception physique (figure 7.16 b) réalise une copie de SL1 (nommée SP1) sur laquelle il rejoue l'historique de conception physique original CP0 en déclenchant l'enregistrement des opérations dans l'historique. Il obtient ainsi un nouveau schéma physique incomplet SP1.

Les nouveaux composants de SL1 (provenant de la création de nouveaux objets ou de la modification d'objets du schéma conceptuel) sont alors transformés spécifiquement en utilisant la boîte à outils de conception physique (autorisant les opérations définies au point 5.3.3). Ces opérations (modifications des index et des espaces de stockage pour prendre en compte les modifications de SL1) sont enregistrées dans l'historique de SP1 à la suite des opérations déjà enregistrées. L'historique qui contient toutes les transformations exécutées pour obtenir SP1 à partir de SL1 est sauvegardé dans le journal CP1.

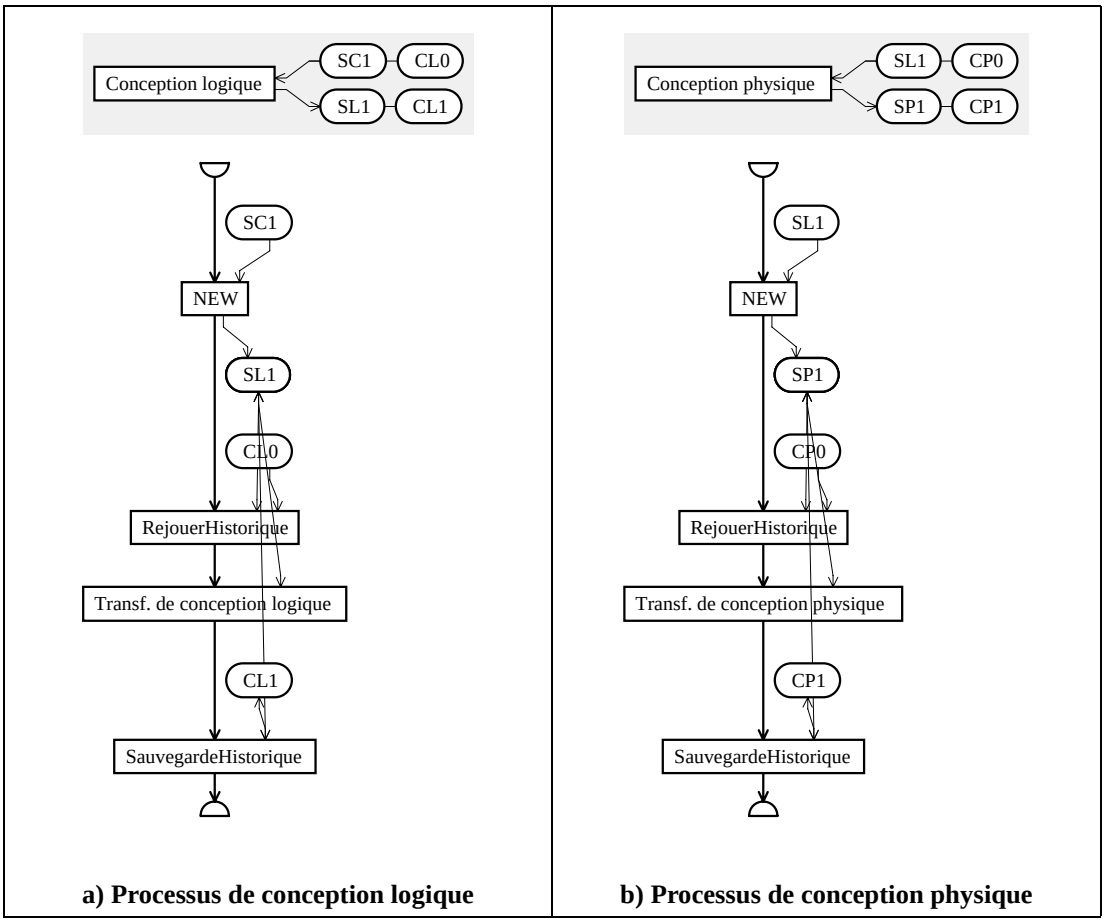


Figure 7.16 - Représentations des processus de conception logique et physique de la méthode de la première stratégie.

7.2.2.4 Processus de génération : conversion de la base de données et modifications des programmes

Le processus de génération (figure 7.17) exécute le générateur avec, comme arguments, les schémas conceptuels (SC0 et SC1), logiques (SL0 et SL1), physiques (SP0 et SP1) ainsi que des historiques de conception logique (CL0 et CL1), de conception physique (CP0 et CP1) et d'évolution conceptuelle (EC1). Le générateur est un module externe de l'atelier développé en Voyager 2. A la fin de son exécution, il fournit deux résultats :

1. *Le script de conversion des données et structures de données* qui se présente sous la forme de requête SQL et qui est exécuté par l'analyste pour adapter la base de données aux nouvelles modifications. L'exécution du script nécessite, dans certains cas, une intervention humaine pour prendre une décision concernant des données ne respectant pas les nouvelles contraintes de la base. La section 7.3 donne l'architecture et l'algorithme du générateur de scripts de conversion.
2. *Les patrons syntaxiques* qui vont constituer une base de recherche pour les analyseurs de programmes (constructeur de graphes de dépendances et/ou fragmenteur de programmes). Ces patrons dépendent du langage de programmation utilisé. L'analyste exécute un analyseur de programmes (avec les patrons de recherche comme arguments) qui repère les sections de code dépendant des composants de la base de données qui ont été modifiés. La section 7.4 analyse en détail la génération des patrons et, plus généralement, la modification des programmes.

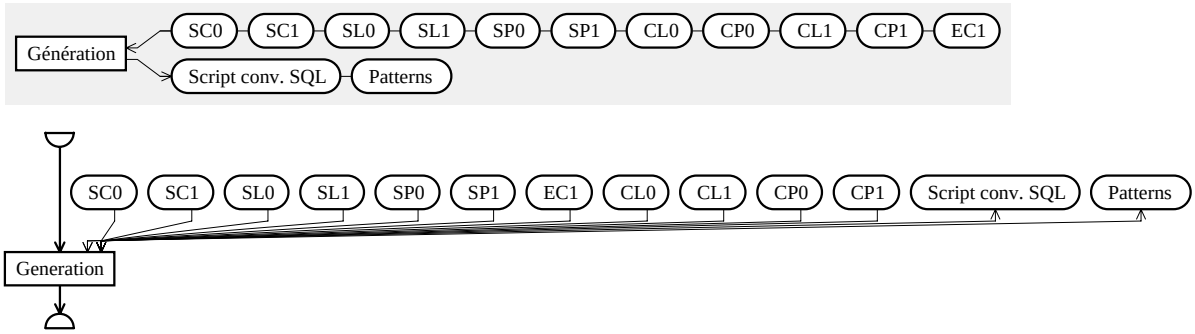


Figure 7.17 - Processus de génération du script de conversion des données et des patrons de recherche pour le graphe de dépendances (première stratégie).

7.2.3 Implémentation de la deuxième stratégie : modifications des spécifications logiques

Examinons en détail les étapes de la démarche méthodologique pour la deuxième stratégie. L'existant est le même que pour la première stratégie. Ces étapes sont les principaux processus de la méthode de modifications des spécifications logiques (figure 7.18 a). L'historique de la figure 7.18 b) a été construit en suivant cette méthode.

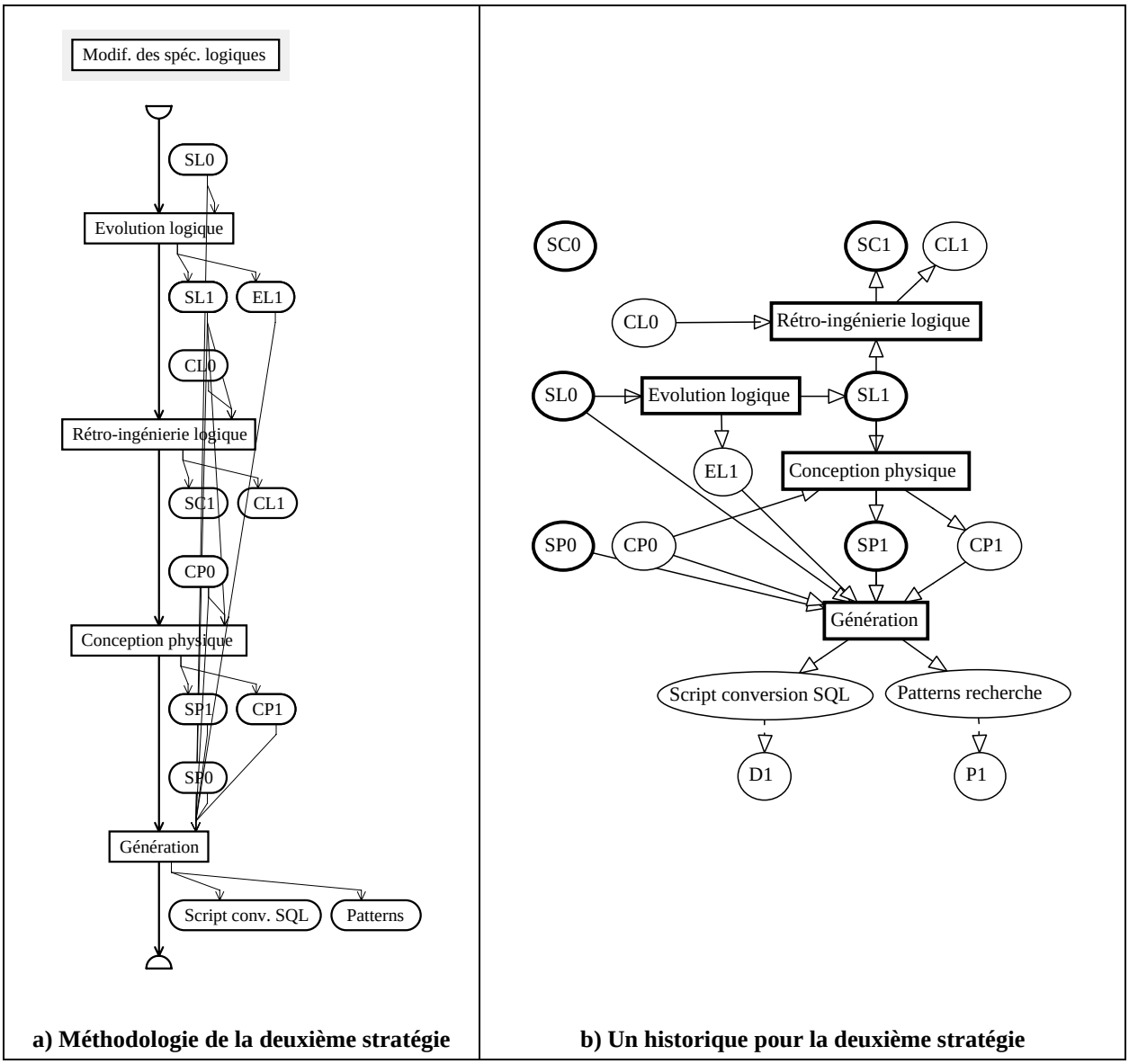


Figure 7.18 - Méthodologie de la stratégie des modifications des spécifications logiques et un historique correspondant.

7.2.3.1 Processus d'évolution logique : production de la nouvelle version du schéma logique SL1

Le processus d'évolution logique (figure 7.19 a) demande au concepteur de réaliser une copie du schéma logique SL0, nommée SL1, et de modifier SL1 en utilisant la boîte à outils des modifications logiques (voir les modifications autorisées au point 5.4.1) et en déclenchant l'enregistrement des transformations. Lorsque ce processus primitif est terminé, l'historique est sauvegardé dans le journal EL1. Le processus d'évolution logique a comme résultat SL1 et EC1.

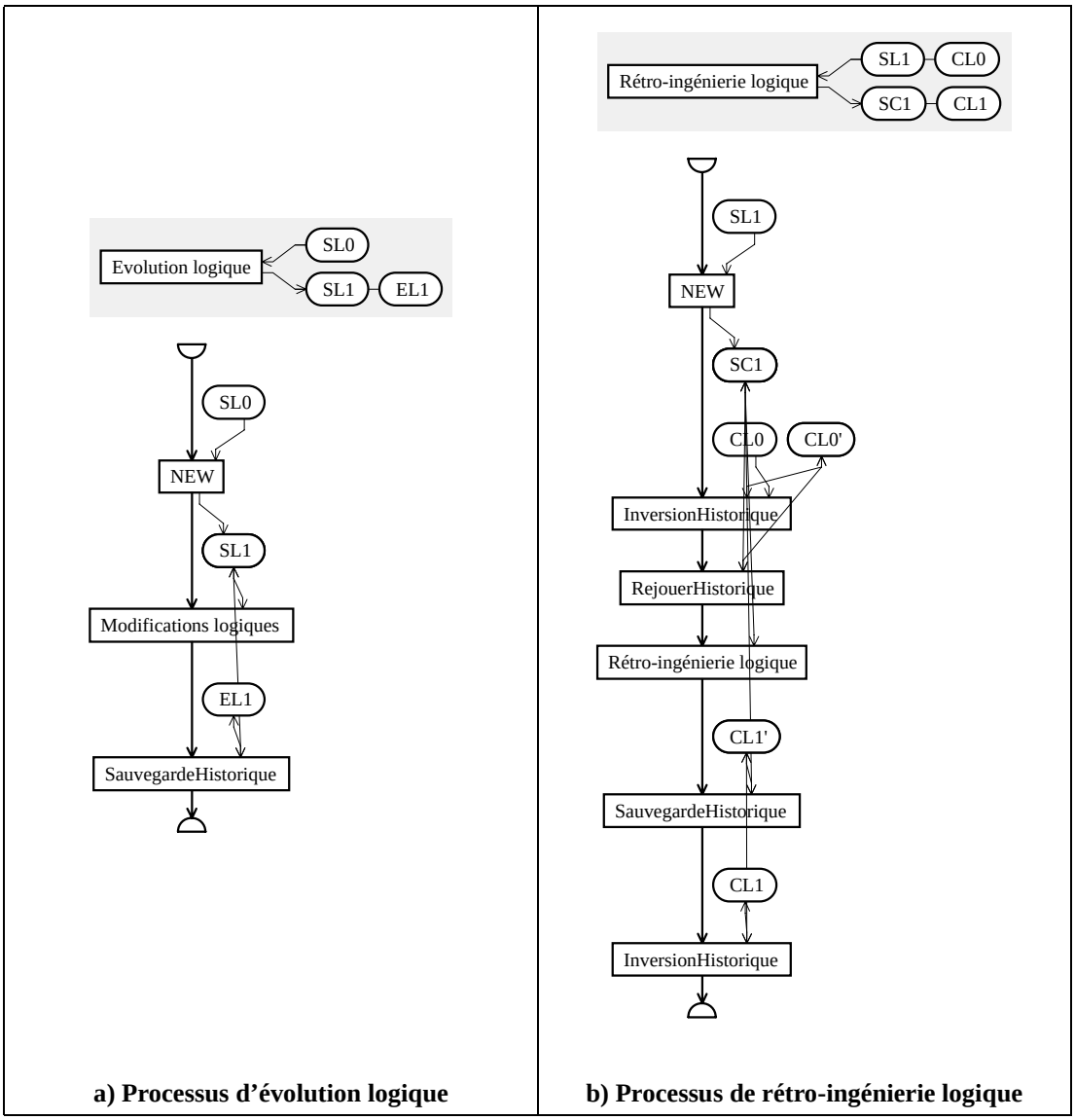


Figure 7.19 - Représentations des processus d'évolution logique et de rétro-ingénierie logique de la méthode de la deuxième stratégie.

7.2.3.2 Processus de rétro-ingénierie logique : production de la nouvelle version du schéma conceptuel SC1

Le processus de rétro-ingénierie logique (figure 7.19 b) a pour objectif de recouvrer un schéma conceptuel plausible intégrant les nouvelles modifications du schéma logique SL1 à partir de l'ancien historique de conception logique CL0. Il demande une copie de SL1 (nommée SC1) et inverse l'historique CL0 dans l'historique CL0'. Sur SC1, il rejoue l'historique de conception logique original inversé enregistré dans le journal CL0' en déclenchant l'enregistrement des opérations dans un nouvel historique. On obtient ainsi une première version incomplète du nouveau schéma conceptuel SC1.

Les nouveaux composants de SL1 (provenant de la création de nouveaux objets ou de la modification d'objets de SL0) sont alors transformés spécifiquement grâce à la boîte à outils de rétro-ingénierie logique (transformations autorisées au point 5.4.2). Ces opérations sont enregistrées dans l'historique de SC1 à la suite de celles enregistrées lors de l'exécution de CL0'. L'historique qui contient toutes les transformations exécutées pour obtenir SC1 à partir de SL1 est sauvegardé dans le journal CL1'. Ce dernier est inversé pour obtenir un historique de conception logi-

que CL1 contenant les transformations appliquées sur SC1 pour obtenir SL1. Ce processus aide le concepteur à garder une documentation complète des spécifications à tous les niveaux d'abstraction de manière à permettre la propagation de modifications futures dans un environnement favorable.

7.2.3.3 Processus de conception physique : production de la nouvelle version du schéma physique SP1

Ce processus est identique à celui du point 7.2.2.3.

7.2.3.4 Processus de génération : conversion de la base de données et modifications des programmes

Le processus de génération (figure 7.20) exécute le générateur avec comme arguments les schémas logiques (SL0 et SL1), physiques (SP0 et SP1) ainsi que des historiques de conception physique (CP0 et CP1) et d'évolution logique (EL1). Il fournit comme résultat le fichier de script de conversion SQL et celui avec les patrons de recherche (voir 7.2.2.4).

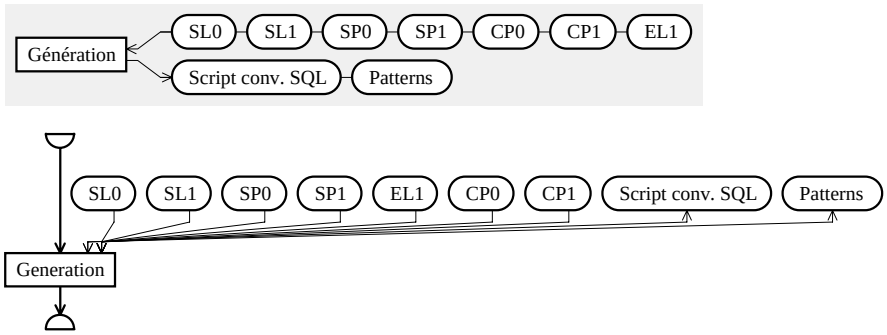
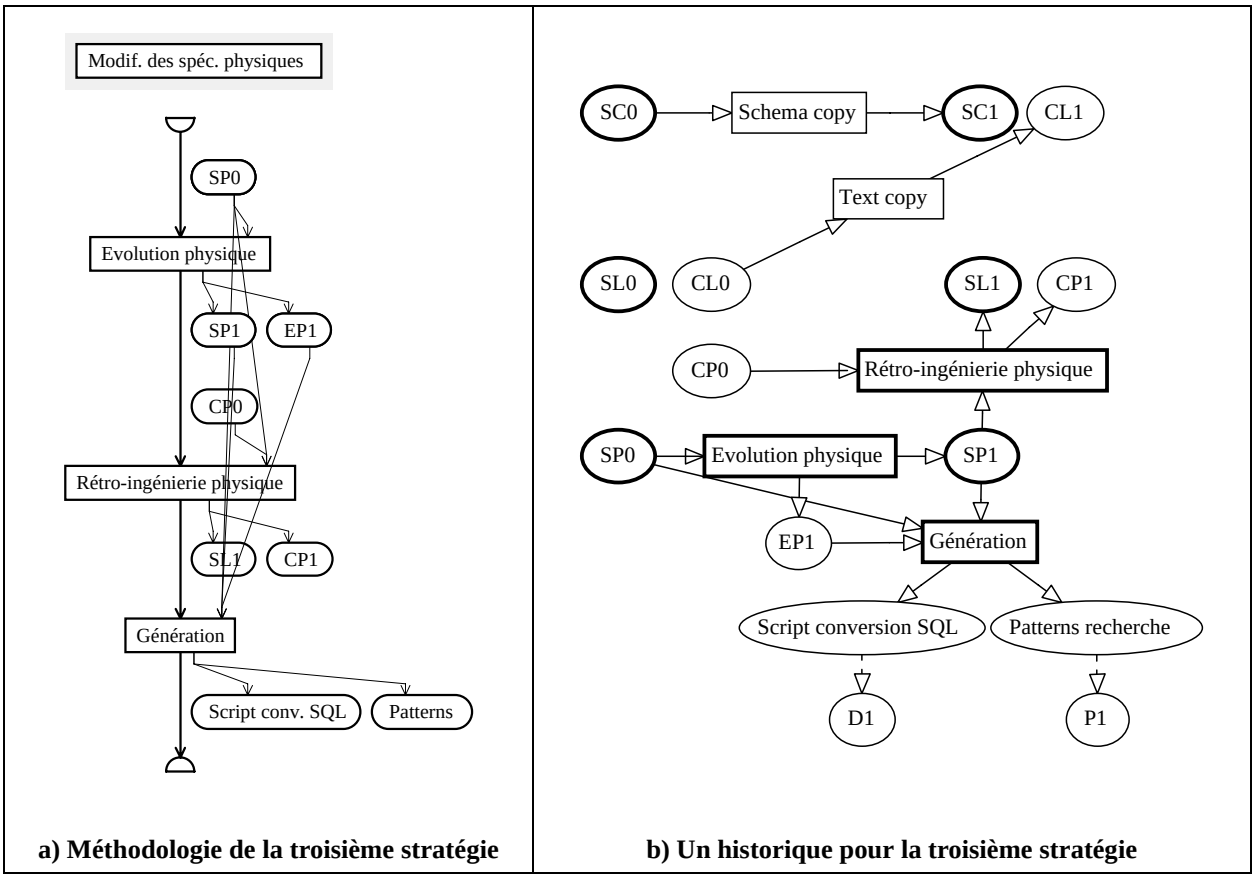


Figure 7.20 - Processus de génération du script de conversion des données et des patrons de recherche pour le graphe de dépendances (deuxième stratégie).

7.2.4 Implémentation de la troisième stratégie : modifications des spécifications physiques

La démarche méthodologique pour la deuxième stratégie se décompose en trois étapes. L'existant reste inchangé par rapport aux deux autres stratégies. Les étapes de la démarche sont les principaux processus de la méthode de modifications des spécifications physiques (figure 7.21 a). L'historique (figure 7.21 b) a été construit conformément à la méthode.

Rappelons qu'il n'est pas nécessaire de reconstruire l'historique de conception logique CL1 et le schéma conceptuel SC1. Ils sont équivalents à CL0 et SC0 puisque les structures physiques modifiées dans cette stratégie sont ignorées dans la conception logique. C'est pourquoi les processus primitifs *Schema copy* et *Text copy* ont été utilisés dans l'historique (figure 7.21 b) pour recréer SC1 et CL1.



a) Méthodologie de la troisième stratégie **b) Un historique pour la troisième stratégie**

Figure 7.21 - Méthodologie de la stratégie des modifications des spécifications physiques et un historique correspondant.

7.2.4.1 Processus d'évolution physique : production de la nouvelle version du schéma physique SP1

Le processus d'évolution physique (figure 7.22 a) demande au concepteur de réaliser une copie du schéma physique SP0, nommée SP1, et de la modifier en utilisant la boîte à outils des modifications physiques (transformations autorisées définies au point 5.5.1) et en déclenchant l'enregistrement des transformations exécutées. Lorsque ce processus primitif est terminé, l'historique est sauvegardé dans le journal EP1. Le processus d'évolution logique retourne comme résultats SP1 et EP1.

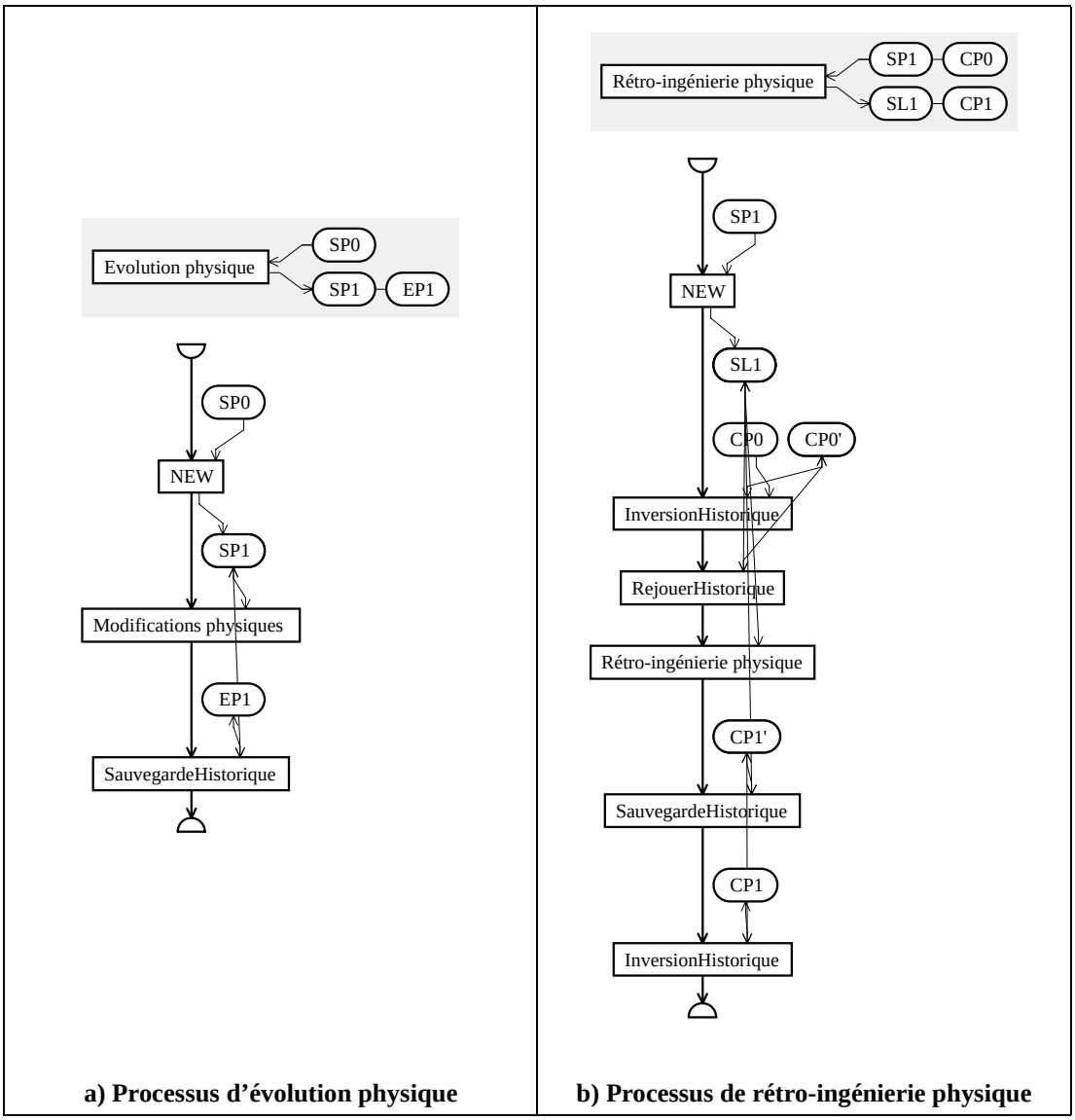


Figure 7.22 - Représentations des processus d'évolution physique et de rétro-ingénierie physique de la méthode de la troisième stratégie.

7.2.4.2 Processus de rétro-ingénierie physique : production de la nouvelle version du schéma logique SL1

Le processus de rétro-ingénierie physique (figure 7.22 b) a pour objectif de recouvrer un schéma logique plausible intégrant les nouvelles modifications du schéma physique SP1 à partir de l'ancien historique de conception physique CP0. Il demande une copie de SP1 (nommée SL1) et inverse l'historique CP0 dans l'historique CP0'. Sur SL1, il rejoue l'historique de conception physique original inversé, enregistré dans le journal CP0', en déclenchant l'enregistrement des opérations dans un nouvel historique. On obtient ainsi une première version incomplète du nouveau schéma logique SL1.

Les nouveaux composants de SP1 (provenant de la création de nouveaux objets ou de la modification d'objets de SP0) sont alors transformés spécifiquement grâce à la boîte à outils de rétro-ingénierie physique (transformations autorisées définies au point 5.5.2). Ces opérations sont enregistrées dans l'historique de SL1 à la suite de celles enregistrées lors de l'exécution de CP0'. L'historique qui contient toutes les transformations exécutées pour obtenir SL1 à partir de SP1 est sauvegardé dans le journal CP1'. Ce dernier est inversé pour obtenir un historique de

conception physique CP1 contenant les transformations appliquées sur SL1 pour obtenir SP1. On a, par ce processus, contribué à garder une documentation complète des spécifications.

7.2.4.3 Processus de génération : conversion de la base de données et modifications des programmes

Le processus de génération (figure 7.23) exécute le générateur avec comme arguments les schémas physiques (SP0 et SP1) ainsi que l'historique d'évolution physique (EP1). Il fournit comme résultat le fichier de script de conversion SQL et celui avec les patrons de recherche (voir 7.2.2.4).

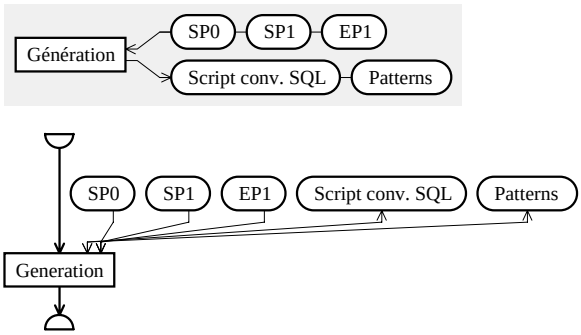


Figure 7.23 - Processus de génération du script de conversion des données et des patrons de recherche pour le graphe de dépendances (troisième stratégie).

7.3 Conversion des structures de données et des données

7.3.1 Introduction

Le but de cette section est d'expliquer le fonctionnement du programme[7] de génération des scripts de conversion des structures de données et des données au travers de son architecture (point 7.3.3). Les principales fonctions de ce programme, correspondant aux étapes du processus de génération définies dans la section 5.6, sont :

- la fonction "AnalyseFichierLog" qui construit un historique simplifié sur la base d'un historique brut (point 7.3.3);
- la fonction "IntegreListeModif" qui traduit les modifications enregistrées dans un historique d'évolution conceptuelle ou logique en modifications physiques (point 7.3.4);
- la fonction "ArrangeListeModif" qui construit une liste des modifcations restructurée en fonction de la technologie relationnelle à partir de l'historique simplifié des modifications physiques (point 7.3.5);
- la fonction "GenereScriptSQL" qui génère les scripts de conversion en analysant la liste des modifications (point 7.3.6).

7.3.2 Architecture du programme

Le programme est décomposé en trois parties principales (une par stratégie) :

- la procédure "EvolutionConceptuelle" (point 7.3.2.1) qui génère le script de conversion pour des modifications de spécifications conceptuelles (première stratégie);
- la procédure "EvolutionLogique" (point 7.3.2.2) qui génère le script de conversion pour des modifications de spécifications logiques (deuxième stratégie);
- la procédure "EvolutionPhysique" (point 7.3.2.3) qui génère le script de conversion pour des modifications de spécifications physiques (troisième stratégie).

7.3.2.1 Procédure "EvolutionConceptuelle"

La procédure dont l'architecture est présentée à la figure 7.24 est décomposée en quatre phases :

1. Les fichiers contenant les historiques sont analysés (fonction "AnalyseFichierLog") pour fournir les listes des modificiations (LEC1, LC0 et LC1). Pour chaque analyse, les schémas originaux (schémas avant l'application de l'historique) et les schémas résultats (schémas après l'application de l'historique sur le schéma original) sont passés comme arguments à la fonction ainsi que l'historique à analyser. L'analyse des historiques de conception CL0 et CP0 (respectivement CL1 et CP1) génère une seule liste des modifications LC0 (respectivement LC1) contenant les modifications appliquées sur SC0 (respectivement SC1) pour obtenir SP0 (respectivement SP1).
2. La fonction "IntegreListeModif" traduit les modifications conceptuelles de la liste LEC1 en modifications physiques dans la liste LEP1.
3. La fonction "ArrangeListeModif" restructure la liste LEP1 pour tenir compte des contraintes techniques de la famille du SGBD.

7. Le prototype a été développé en Voyager 2.

4. La fonction "GenereScriptSQL" traduit les modifications physiques de LEC1 en scripts de conversion SQL des données et structures de données.

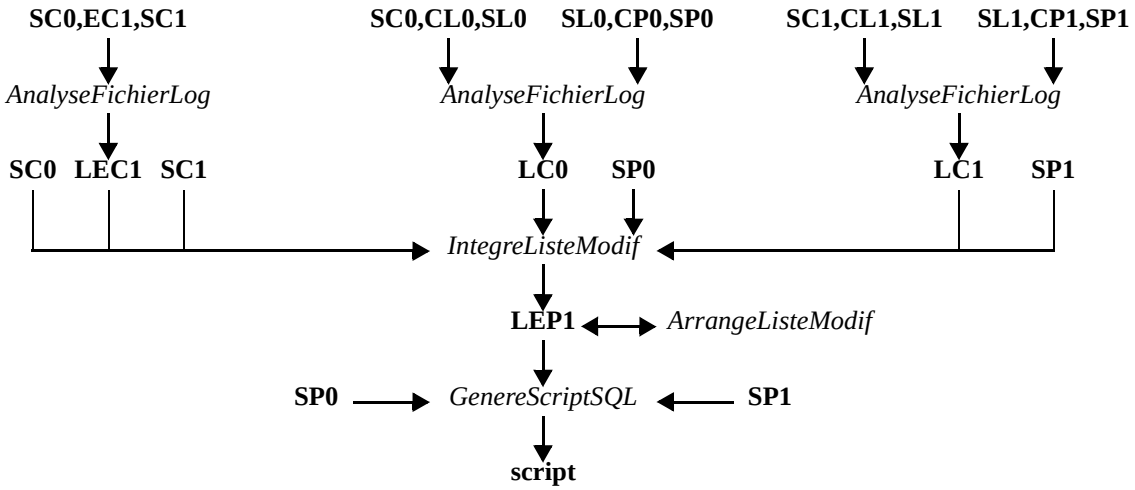


Figure 7.24 - Architecture de la procédure "EvolutionConceptuelle".

7.3.2.2 Procédure "EvolutionLogique"

La figure 7.25 présente l'architecture de la procédure décomposée en quatre phases identiques à la procédure "EvolutionConceptuelle" (point 7.3.2.1). Seul l'analyse de l'historique CP0 (respectivement CP1) est prise en compte pour créer LC0 (respectivement LC1).

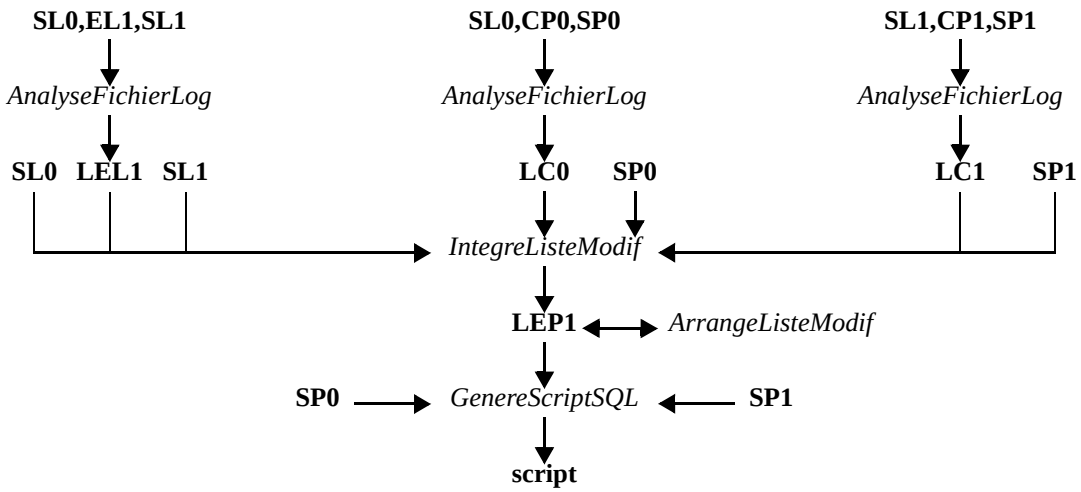


Figure 7.25 - Architecture de la procédure "EvolutionLogique".

7.3.2.3 Procédure "EvolutionPhysique"

La figure 7.26 présente l'architecture de la procédure décomposée en trois phases. La fonction "IntegreListeModif" n'est pas appelée car l'analyse de l'historique EP1 donne déjà la liste des modifications physiques. Cette liste est seulement restructurée pour générer les scripts de conversion.

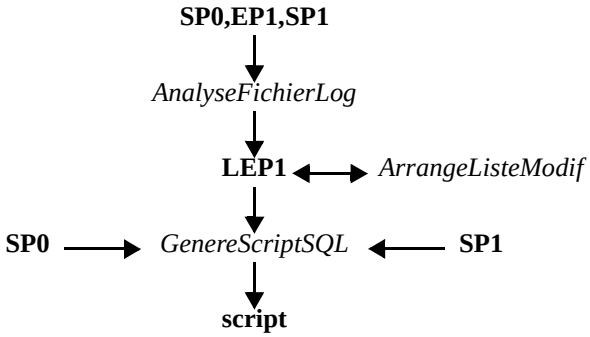


Figure 7.26 - Architecture de la procédure "EvolutionPhysique".

7.3.3 Fonction "AnalyseFichierLog"

7.3.3.1 Objectif

Cette fonction parcourt un historique (stocké sous la forme d'un journal) et enregistre dans un historique simplifié les transformations trouvées dans le journal. Pour faciliter la programmation, l'historique simplifié est stocké sous la forme d'une liste des modifications plus facile à manipuler qu'un journal, un historique pouvant être régénéré de manière assez systématique à partir d'une liste des modifications si cela s'avère nécessaire.

La fonction "AnalyseFichierLog" met à jour la liste des modifications avec les informations récoltées dans le journal. Chaque modification est enregistrée dans la liste sous la forme d'un triplet composé :

- d'une constante indiquant le type de la modification,
- d'un objet [8](ou une liste d'objets) cible qui est le résultat de la modification,
- d'un objet (ou une liste d'objets) origine de la modification.

La fonction reçoit aussi en argument le schéma original et le schéma cible (schéma original sur lequel l'historique contenu dans le journal a été rejoué). L'objet cible référence donc un objet du schéma cible et l'objet origine un objet du schéma original.

Tous les types de transformation n'ont pas d'objet cible ou origine. Les créations n'ont pas d'objet origine alors que les destructions ont un objet origine et, dans le cas des groupes et des attributs, un objet cible qui est le parent de l'objet détruit. Les destructions d'une collection, d'un type d'entités et d'un type d'associations n'ont pas d'objet cible car ils ne possèdent pas de parent autre que le schéma de spécifications. Les transformations sont représentées par une liste des objets cibles (le ou les objets créés par la transformation) et par une liste d'objets origines (le ou les objets transformés dans le schéma original). La table 7.1 donne, pour chaque type de modification, les informations susceptibles d'être extraites du journal. Pour simplifier le tableau, la constante TRANSFO représente tous les types de transformations à sémantique constante analysés dans le point 3.2.3.

Type objet	Type modification	Objet(s) cible(s)	Objet(s) origine(s)
Groupe	DETRUIREGR	Objet parent origine	Groupe origine
	CREERGR	Groupe cible	/
	MODIFGR	Groupe cible	Groupe origine

Tableau 7.1 - Informations susceptibles d'être extraites d'un journal et stockées dans une liste de modifications.

8. Techniquement, un objet est représenté par une référence (ou un pointeur) vers l'objet du schéma concerné.

Type objet	Type modification	Objet(s) cible(s)	Objet(s) origine(s)
Attribut	DETRUIREATT	Objet parent origine	Attribut origine
	CREERATT	Attribut cible	/
	MODIFATT	Attribut cible	Attribut origine
Type d'entités	DETRUIREENT	/	Type d'entités origine
	CREERENT	Type d'entités cible	/
	MODIFENT	Type d'entités cible	Type d'entités origine
Types d'associations	DETRUIREREL	/	Type d'associations origine
	CREERREL	Type d'associations cible	/
	MODIFREL	Type d'associations cible	Type d'associations origine
Rôles	DETRUIREROL	Type d'associations origine	Rôle origine
	CREERROL	Rôle cible	/
	MODIFROL	Rôle cible	Rôle origine
Collections	DETRUIRECOL	/	Collection origine
	CREERCOL	Collection cible	/
	MODIFCOL	Collection cible	Collection origine
	TRANSFO	Liste d'objets cibles	Liste d'objets origines

Tableau 7.1 - Informations susceptibles d'être extraites d'un journal et stockées dans une liste de modifications.

Prenons l'extrait de journal de la figure 7.27. La fonction parcourt le journal. D'abord, elle extrait le type de la modification (MOD ENT : modifier un type d'entités). Ensuite, elle obtient l'ancien nom du type d'entités dans S0 en recherchant le type d'entités PERSONNE dans S0. Finalement, elle recherche le type d'entités renommé EMPLOYE dans S1. L'élément suivant est ajouté à la liste des modifications : [MODIFENT, EMPLOYE, PERSONNE].

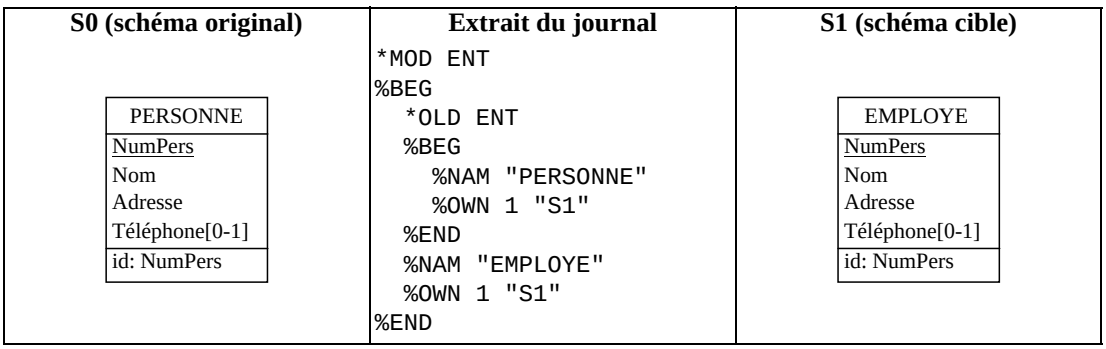


Figure 7.27 - Exemple de journal avec le schéma original (S0) et le schéma sur lequel l'historique a été rejoué (S1).

7.3.3.2 Résolution du problème des objets renommés

La mise en correspondance des objets (entre un journal et un schéma) se fait sur la base des noms des objets. Cela peut poser des problèmes si, par exemple, le journal contient la destruction du type d'entités EMPLOYE. Un type d'entités portant le nom EMPLOYE est recherché dans S0. Or, il faut rechercher le type d'entités PERSONNE dans S0 car c'est le type d'entités d'origine qui doit être détruit. Pour résoudre ce problème, il faut passer par une table des noms qui contient le plus ancien et le plus récent nom de chaque objet renommé. Lorsqu'un objet est renommé, on recherche dans la table le nom le plus récent correspondant au nom de l'objet avant la modification. S'il existe, on le remplace par le nouveau nom donné à l'objet renommé. Sinon, on crée une entrée dans la table avec l'ancien et le nouveau nom. Ainsi, on dispose, à tout instant, du nom d'origine (celui de S0) et du nouveau nom (celui de S1) pour tous les objets renommés dans un journal. Il reste à règler le problème des objets dont le nom est identifiant à

l'intérieur de son propriétaire. C'est le cas des attributs, des groupes et des rôles. Dans ce cas, la table des noms contient également l'ancien nom (de S0) du propriétaire. Par exemple, si on renomme un attribut, on ajoute dans la table un élément contenant ses deux noms (avant et après la modification) ainsi que le nom du type d'entités auquel il appartient. Le nom du type d'entités n'est pas nécessairement celui existant au moment de la modification, c'est le nom du type d'entités dans S0 (il faut peut-être le rechercher dans la table des noms si le type d'entités a déjà été renommé).

7.3.3.3 Algorithme

Supposons que F soit le fichier analysé, S0 le schéma original, S1 le schéma cible et L la liste des modifications créée sur la base de l'analyse de F. Pour chaque enregistrement de F, les informations sont extraites pour décrire la modification (appelée T) qui va être éventuellement ajoutée à L :

- Pour l'enregistrement d'une création, il faut trouver dans S1 l'objet créé (o1). Deux cas sont possibles :
  - Si o1 existe dans S1, alors T = [CREER,o1,/] est ajoutée à L sauf si o1 est un attribut, un rôle ou un groupe et que son parent (un type d'entités, un attribut décomposable ou un type d'associations) est déjà créé dans L.
  - Si o1 n'existe pas dans S1, cela signifie que o1 est renommé, détruit ou transformé dans la suite de F. Les modifications concernant o1 seront ajoutées à L au moment opportun.
- Pour l'enregistrement d'une destruction, il faut trouver dans S0 l'objet détruit (o0) sur la base de son ancien nom (stocké dans la table des noms). Si o0 est un attribut ou un groupe, il faut d'abord retrouver p0, le parent de o0 dans S0 (en utilisant la table des noms). Deux cas sont possibles :
  - Si o0 (et p0 si nécessaire) existe dans S0, alors T = [DETRUIRE,/,o0] ou [DETRUIRE,p0,o0] est ajoutée à L.
  - Si o0 (et p0 si nécessaire) n'existe pas dans S0, cela signifie que l'objet détruit a déjà été transformé ou créé dans L. S'il existe dans L une transformation ([TRANSFO,[...,o0,...],[...,o,...]] ou [MODIF,o0,o]) portant sur o0, il faut la remplacer par [DETRUIRE,/,o]. Sinon il s'agit d'un objet créé ou un objet dont le parent a été transformé, modifié ou créé précédemment dans F. Dans ce cas, L reste inchangé.
- Pour l'enregistrement d'une modification, il faut trouver dans S0 l'objet origine (o0) sur la base de son ancien nom et, dans S1, l'objet modifié (o1) sur la base de son nouveau nom. Si l'ancien et le nouveau nom sont différents (o1 est renommé), la table des noms est mise à jour comme cela a été décrit précédemment. Quatre cas sont possibles :
  - Si o0 (dans S0) et o1 (dans S1) existent : s'il y a déjà une modification ([MODIF,o1,o0]) ou une transformation ([TRANSFO,(..., no1, ...),(..., o0, ...)]) relative à o1 dans L, T n'est pas ajoutée à L et, dans le cas de la transformation, no1 est remplacé par o1 si c'est nécessaire. Sinon T = [MODIF,o1,o0] est ajoutée à L.
  - Si o1 existe et o0 n'existe pas, s'il y a dans L une transformation ou une création ayant o1 comme objet cible, T n'est pas ajoutée à L. Sinon il s'agit d'une création et T = [CREER,o1,/] est ajoutée à L sauf si o1 est un attribut, un groupe ou un rôle dont le parent (un type d'entités, un attribut décomposable ou un type d'associations) est déjà créé dans L.
  - Si o1 et o0 n'existent pas, o1 fait déjà l'objet d'une modification (création ou transformation) dans F et il est renommé, détruit ou transformé dans la suite de F. T n'est pas ajoutée à L. La modification relative à o1 sera enregistrée à la prochaine modification le concernant.

  - Si o1 n'existe pas et o0 existe, cela signifie que o1 est renommé, détruit ou transformé dans la suite de F. La modification relative à o1 sera enregistrée à la prochaine modification le concernant.
- Pour l'enregistrement d'une transformation à sémantique constante, il faut trouver dans S0 le ou les objets origines (o01, ..., o0n) sur la base de leur ancien nom et dans S1 le ou les objets transformés (o11, ..., o1m) sur la base de leur nouveau nom. Quatre cas sont possibles :
  - Si les objets cibles et origines existent, T = [TRANSFO,(o11, ..., o1m),(o01, ..., o0n)] est ajoutée à L.
  - Si les objets cibles existent mais pas les objets origines. S'il existe dans L une transformation concernant les objets origines, on remplace dans cette transformation les objets cibles par les nouveaux objets cibles de T. Sinon il s'agit de créations des objets cibles, T1 = [CREER,o11,/], ..., Tm = [CREER,o1m,/] sont ajoutées à L.
  - Si les objets cibles et origines n'existent pas, cela signifie que les objets cibles ont déjà été modifiés (créés, transformés) dans F et qu'ils sont modifiés (transformés, renommés ou détruits) dans la suite de F. S'il existe une transformation concernant les objets origines dans L, on remplace dans cette transformation les objets cibles par les objets cibles (pas les références mais les informations permettant de faire les correspondances) de l'enregistrement courant.
  - Si les objets origines existent mais pas les objets cibles, cela signifie que les objets cibles sont modifiés (transformés, renommés ou détruits) dans la suite de F. Il faut mettre dans L T = [TRANSFO,(no11, ..., no1m),(o01, ..., o0n)] où (no11, ..., no1m) ne sont pas des références à des objets mais bien les informations (type d'objet, nom, parent dans le cas d'attributs) permettant de faire la correspondance lorsqu'une modification relative aux objets concernés est rencontrée.

7.3.4 Procédure "IntegreListeModif"

7.3.4.1 Objectif

Pour les deux premières stratégies, les listes de modifications LEC1 ou LEL1, LC0 et LC1 vont être intégrées pour obtenir la liste des modifications physiques LEP1 à partir de laquelle les scripts sont générés. Concrètement, il s'agit de traduire les modifications conceptuelles de LEC1 ou logiques de LEL1 en modifications physiques ainsi que d'analyser les transformations de LC1 qui concernent des objets non modifiés dans LEC1 ou LEL1 mais qui n'ont pas d'équivalences dans LC0 (par exemple, si un attribut multivalué est transformé différemment dans LC1 et LC0).

7.3.4.2 Algorithme

Supposons que LE1 soit la liste de modifications d'évolution (LEC1 ou LEL1), LC0 et LC1 les listes de modifications de conception, S0 et S1 les schémas originaux, SP0 et SP1 les schémas physiques. LE1 contient les modifications appliquées à S0 pour obtenir S1, LC0 les transformations appliquées à S0 pour obtenir SP0 et LC1 les transformations appliquées à S1 pour obtenir SP1.

Pour la traduction des transformations, l'algorithme se base sur la recherche dans SP0 et SP1 des objets correspondant aux objets modifiés dans LE1. Pour réaliser cela, les listes de modifications de conception LC0 et LC1 sont exploitées. Supposons qu'un objet soit modifié dans LE1, on a [MODIF,o1,o0]. Pour traduire la modification relative à o1 en modification physique, il faut chercher les transformations le concernant dans LC0 (respectivement LC1). Si elle existe, l'objet cible op0 (respectivement op1) de la transformation est l'objet physique correspondant à o0 (respectivement o1). S'il n'y a pas de transformation, la recherche consiste à trouver l'objet op0 (respectivement op1) portant le même nom que o0 (respectivement o1). Dans ce cas, on peut traduire la modification d'évolution en [MODIF,op1,op0].

Les recherches des objets renommés se basent sur les tables de noms : Tn0 pour les objets renommés entre S0 et SP0, Tn1 pour les objets renommés entre S1 et SP1.

Pour chaque transformation T de LE1 :

- Si T = [DETRUIREENT,/,e0] :
  - Trouver, dans SP0, le type d'entités ep0 (ou les types d'entités) correspondant à e0 en utilisant LCO.
  - Si e0 n'est pas transformé dans LC0, il faut ajouter [DETRUIREENT,/,ep0] dans LEP1. Sinon e0 est transformé (éclatement) et, dans ce cas, il faut ajouter à LEP1 les destructions des types d'entités cibles de la transformation.
- Si T = [CREERENT,e1,/] :
  - Trouver, dans SP1, le type d'entités ep1 (ou les types d'entités) correspondant à e1 en utilisant LC1.
  - Si e0 n'est pas transformé dans LC1, il faut ajouter [CREERENT,ep1,/] dans LEP1. Sinon e0 est transformé (éclatement) dans LC1 et, dans ce cas, il faut ajouter à LEP1 les créations des types d'entités cibles de la transformation.
- Si T = [MODIFENT,e1,e0] :
  - Trouver, dans SP0, le type d'entités ep1 correspondant à e0 en utilisant LC0.
  - Trouver, dans SP1, le type d'entités ep1 correspondant à e1 en utilisant LC1.
  - Si e0 n'est pas transformé dans LC1, il faut ajouter [MODIFENT,ep1,ep0] dans LEP1. Sinon e0 est transformé dans LC1 et, dans ce cas, il faut ajouter à LEP1 les modifications des types d'entités cibles de la transformation.
- Si T = [DETRUIREATT,p,a0] :
  - Trouver, dans SP0, le type d'entités ep0 correspondant à p (type d'entités, type d'associations ou attribut décomposable) en utilisant LC0.
  - Trouver, dans SP1, le type d'entités ep1 correspondant à p en utilisant LC1.
  - Trouver, dans SP0, l'attribut ap0 appartenant à ep0 et correspondant à a0 en utilisant LC0.
  - Si ap0 existe, ajouter [DETRUIREATT,ep1,ap0] dans LEP1. Sinon il est transformé dans LC0, il faut ajouter à LEP1 les destructions des objets cibles de la transformation.
- Si T = [CREERATT,a1,/] :
  - Trouver, dans SP1, le type d'entités ep1 correspondant à l'objet auquel appartient a1 en utilisant LC1.
  - Trouver, dans SP1, l'attribut ap1 correspondant à a1 et appartenant à ep1 en utilisant LC1.
  - Si ap1 appartient à SP1, ajouter [CREERATT,ap1,/] dans LEP1. Sinon il est transformé dans LC1, il faut ajouter à LEP1 les créations des objets cibles de la transformation.
- Si T = [MODIFATT,a1,a0] :
  - Trouver, dans SP0, le type d'entités ep0 correspondant à l'objet auquel appartient a0 en utilisant LC0.
  - Trouver, dans SP1, le type d'entités ep1 correspondant à l'objet auquel appartient a1 en utilisant LC1.
  - Trouver, dans SP0, l'attribut ap0 appartenant à ep0 et correspondant à a0 en utilisant LC0.
  - Trouver, dans SP1, l'attribut ap1 appartenant à ep1 et correspondant à a1 en utilisant LC1.
  - Si ap0 appartient à ep0 et ap1 appartient à ep1, il faut ajouter [MODIFATT,ap1,ap0] dans LEP1. Sinon a0 (ou a1) a été transformé dans LC0 (ou LC1). S'il s'agit du même type de transformation, il faut ajouter à LEP1 les modifications relatives aux objets dans lesquels a1 a été transformé. S'il s'agit de transformations différentes, il faut ajouter la transforma-

tion à LEP1 permettant de passer des anciens objets dans lesquels a0 a été transformé aux nouveaux objets dans lesquels a1 a été transformé.

- Si T = [DETRUIREGR,p,g0] :
  - Trouver, dans SP0, le type d'entités ep0 correspondant à l'objet parent p (type d'entités, type d'associations ou attribut décomposable multivalué) en utilisant LC0.
  - Trouver, dans SP1, le type d'entités ep1 correspondant à l'objet parent p en utilisant LC1.
  - Trouver, dans SP1, le groupe gp0 appartenant à ep0 et correspondant à g0 en utilisant LC0.
  - Ajouter à LEP1 [DETRUIREGR,ep1,gp0].
- Si T = [CREERGR,g1,/] :
  - Trouver, dans SP1, le type d'entités ep1 correspondant à l'objet parent auquel appartient g1 en utilisant LC1.
  - Trouver, dans SP1, le groupe gp1 correspondant à g1 en utilisant LC1.
  - Ajouter [CREERGR,gp1,/] dans LEP1.
- Si T = [MODIFGR,g1,g0] :
  - Trouver, dans SP0, le type d'entités ep0 correspondant à l'objet parent auquel appartient g0 en utilisant LC0.
  - Trouver, dans SP1, le type d'entités ep1 correspondant à l'objet parent auquel appartient g1 en utilisant LC1.
  - Trouver, dans SP0, le groupe gp0 appartenant à ep0 et correspondant à g0 en utilisant LC0.
  - Trouver, dans SP1, le groupe gp1 appartenant à ep1 et correspondant à g1 en utilisant LC1.
  - Si gp0 appartient à ep0 et gp1 appartient à ep1, il faut ajouter [MODIFGR,gp1,gp0] dans LEP1. Si gp0 appartient à ep0 et gp1 n'existe pas dans ep1, il faut ajouter [DETRUIRE,/,gp0] dans LEP1. Si gp0 n'existe pas dans ep0 et gp1 appartient à ep1, il faut ajouter [CREER,gp1,/] dans LEP1.
- Si T = [DETRUIREREL,/,r0] :
  - Trouver, dans SP0, les objets dans lesquels r0 a été transformé en utilisant LC0.
  - Ajouter la destruction de ces objets (attributs de référence + clé étrangère ou type d'entités) dans LEP1.
- Si T = [CREEREL,r1,/] :
  - Trouver, dans SP1, les objets dans lesquels r1 a été transformé en utilisant LC1.
  - Ajouter la création de ces objets (attributs de référence + clé étrangère ou type d'entités) dans LEP1.
- Si T = [MODIFREL,r1,r0] :
  - Trouver, dans SP0, les objets dans lesquels r0 a été transformé en utilisant LC0.
  - Trouver, dans SP1, les objets dans lesquels r1 a été transformé en utilisant LC1.
  - S'il s'agit du même type de transformation, il faut ajouter les modifications des objets dans LEP1, sinon il faut ajouter la transformation à LEP1 permettant de passer des anciens objets dans lesquels r0 a été transformé aux nouveaux objets dans lesquels r1 a été transformé.
- Si T = [DETRUIREROL,r,rl0] ou T = [CREERROL,rl1,/] ou T = [MODIFROL,rl1,rl0], le traitement est identique à la transformation MODIFREL car on peut les considérer comme des modifications relatives à un type d'associations.

- Si T = [TRANSFO,[no1,...,non],[ao1,...,aom]] :
  - Trouver, dans SP0, les objets dans lesquels ao1, ..., aom ont été transformés en utilisant LC0.
  - Trouver, dans SP1, les objets dans lesquels no1, ..., non ont été transformés en utilisant LC1.
  - Ajouter à LEP1 la transformation permettant de passer des anciens objets dans lesquels ao1, ..., aom ont été transformés aux nouveaux objets dans lesquels no1, ..., non ont été transformés.

Pour chaque transformation T de LC1, T porte sur des objets non modifiés dans LE1 et elle n'a pas de transformation équivalente dans LC0 :

- Si T = [MODIFENT,ep1,e0] :
  - Trouver, dans SP0, le type d'entités ep0 correspondant à e0 en utilisant LC0.
  - Ajouter [MODIFENT,ep1,ep0] dans LEP1.
- Si T = [MODIFATT,ap1,a0] :
  - Trouver, dans SP0, le type d'entités ep0 correspondant à l'objet auquel appartient a0 en utilisant LC0.
  - Trouver l'attribut ap0 appartenant à ep0 et correspondant à a0 dans SP0 en utilisant LC0.
  - Si ap0 existe dans ep0, ajouter [MODIFATT,ap1,ap0] dans LEP1.
- Si T = [DETRUIREGR,/,g0] :
  - Trouver, dans SP0, le type d'entités ep0 correspondant à l'objet auquel appartient g0 en utilisant LC0.
  - Trouver, dans SP0, le groupe gp0 appartenant à ep0 et correspondant à g0 en utilisant LC0.
  - Si gp0 existe dans ep0, ajouter [DETRUIREGR,/,gp0] dans LEP1.
- Si T = [CREERGR,gp1,/], il faut ajouter la transformation telle quelle à LEP1.
- Si T = [MODIFGR,gp1,g0] :
  - Trouver, dans SP0, le type d'entités ep0 correspondant à l'objet auquel appartient g0 en utilisant LC0.
  - Trouver, dans SP0, le groupe gp0 appartenant à ep0 et correspondant à g0 en utilisant LC0.
  - Si gp0 existe dans ep0, ajouter [MODIFGR,gp1,gp0] dans LEP1.
- Si T = [TRANSFO,[no1,...,non],[ao1,...,aom]] :
  - Trouver dans LC0 les objets dans lesquels ao1, ..., aom ont été transformés dans SP0.
  - Ajouter la transformation à LEP1 permettant de passer des anciens objets dans lesquels ao1, ..., aom ont été transformés aux nouveaux objets no1, ..., non.
- Si T = [DETRUIRECOL,/,c0] :
  - Trouver, dans SP0, la collection c0 en utilisant LC0.
  - Ajouter [DETRUIRECOL,/,c0] dans LEP1.
- Si T = [CREERCOL,c1,/], il faut ajouter la transformation telle quelle à LEP1.
- Si T = [MODIFCOL,c1,c0] :
  - Trouver, dans SP0, la collection c0 en utilisant LC0.
  - Trouver, dans SP1, la collection c1 en utilisant LC1.
  - Si c0 et C1 existent, ajouter [MODIFCOL,c1,c0] dans LEP1.

7.3.5 Procedure "ArrangeListeModif"

La procédure "ArrangeListeModif" restructure LEP1 en fonction du SGBDR choisi. Si la technologie change, cette procédure doit être adaptée. La restructuration est décomposée en trois parties :

- la procédure "ArrangeModifTE" organise les transformations sur les types d'entités, leurs attributs et leurs groupes,
- la procédure "ArrangeModifColl" organise les transformations sur les collections et leurs types d'entités,
- la procédure "ArrangeModifGr" agence les transformations sur les groupes et les attributs qui les composent.

7.3.5.1 Procédure "ArrangeModifTE"

Comme cela a déjà été mentionné précédemment, la modification consistant à renommer une table n'existe pas dans certains SGBDR. Une technique consiste à détruire la table et toutes ses contraintes puis à la reconstruire avec son nouveau nom. Comme on l'a mentionné dans le point 4.2.3.3, il existe deux autres techniques : les vues et les synonymes. Si on choisit l'option du renommage par la technique de création / destruction, l'algorithme suivant doit être appliqué sur LEP1.

Pour chaque transformation T de LEP1, si T est une transformation sur un attribut ou un groupe et qu'il existe dans LEP1 une transformation qui renomme ou crée le type d'entités parent, alors T est enlevée de LEP1.

7.3.5.2 Procédure "ArrangeModifColl"

Toutes les modifications relatives aux collections (ou espace de stockage) nécessitent dans la plupart des systèmes relationnels des tables vides. Dans le cas contraire, il faut stocker les instances dans des tables temporaires. Ces manipulations coûteuses peuvent être allégées par une restructuration de LEP1. L'idée consiste à extraire les transformations sur les collections pour mettre les créations au début et les destructions à la fin de LEP1. Pour réaliser cela, deux listes temporaires sont créées : LCC pour les créations et LDC pour les destructions de collections.

Pour chaque transformation T de LEP1 :

- Si T = [CREERCOL,c1,/] :
  - Ajouter T à LCC.
  - Pour tous les types d'entités e appartenant à c1, ajouter [MODIFENT,e,/] à LEP1 si e n'est pas déjà créé, modifié ou transformé dans LEP1.
- Si T = [DETRUIRECOL,/,c0] :
  - Ajouter T à LDC.
  - Pour tous les types d'entités e appartenant à c0, ajouter [MODIFENT,e,/] à LEP1 si e n'est pas déjà créé, modifié ou transformé dans LEP1.
- Si T = [MODIFCOL,c1,c0] :
  - Si le nom de c1 est différent de celui de c0 (renommer une collection), il faut ajouter [CREERCOL,c1,/] à LCC et ajouter [DETRUIRECOL,/,c0] à LDC. Pour tous les types d'entités e appartenant à c0 et c1, ajouter [MODIFENT,e,/] à LEP1 si e n'est pas déjà créé, modifié ou transformé dans LEP1.
  - Sinon (transfert de types d'entités composants de la collection) pour tous les types d'entités e appartenant à c0 et n'appartenant pas à c1 (et inversement), ajouter [MODIFENT,e,/] à LEP1 si e n'est pas déjà créé, modifié ou transformé dans LEP1.

- Les autres transformations restent inchangées.

7.3.5.3 Procédure "ArrangeModifGroup"

Cette procédure réorganise les modifications relatives aux groupes dans LEP1. Les groupes sont des contraintes sur les attributs qui les composent. Si un attribut est modifié, les groupes dans lesquels il joue un rôle sont détruits. Les contraintes référentielles rendent le problème encore plus complexe puisqu'il s'agit de contraintes inter-groupes (si un groupe participant à une contrainte référentielle est modifié, l'autre groupe de la contrainte est détruit). Les modifications relatives aux tables entraînent aussi des modifications au niveau des groupes.

Pour résoudre le problème, les destructions de groupes sont placées au début de LEP1 et les créations à la fin. Il convient aussi de séparer les modifications relatives aux contraintes intra-groupe (identifiant, clé d'accès, coexistence, exclusion, au-moins-un et exactement-un) et les contraintes inter-groupes (contraintes référentielles). Le modèle générique de représentation permet d'attribuer plusieurs fonctions à un groupe mais, en SQL, chaque fonction doit être traduite séparément. Cette séparation est également nécessaire pour traiter les contraintes référentielles de manière particulière.

On définit une liste LCG pour les créations de contraintes intra-groupe, une liste LCR pour les créations de contraintes référentielles, une liste LDG pour les destructions de contraintes intra-groupe et LDR pour les destructions de contraintes référentielles.

Pour chaque transformation T de LEP1 :

- Si T = [DETRUIREGR,ep1,g0],
  - Ajouter (si elles n'y sont pas déjà) les destructions des contraintes intra-groupes de g0 à LDG si le type d'entités auquel appartient g0 n'est pas modifié dans LEP1 (modification provenant de la restructuration sur les collections).
  - Ajouter (si elles n'y sont pas déjà) les destructions des contraintes référentielles de g0 à LDR si le type d'entités auquel appartient g0 n'est pas modifié dans LEP1 (modification provenant de la restructuration sur les collections).
  - Enlever T de LEP1.
- Si T = [CREERGR,g1,/],
  - Ajouter (si elles n'y sont pas déjà) les créations des contraintes intra-groupes à LCG si le type d'entités auquel appartient g1 n'est pas modifié dans LEP1 (modification provenant de la restructuration sur les collections).
  - Ajouter (si elles n'y sont pas déjà) les créations des contraintes référentielles de g1 à LCR.
  - Enlever T de LEP1.
- Si T = [MODIFGR,g1,g0],
  - Ajouter (si elles n'y sont pas déjà) les destructions des contraintes intra-groupe de g0 à LDG si le type d'entités auquel appartient g0 n'est pas modifié dans LEP1 (modification provenant de la restructuration sur les collections). Si g1 et g0 ont les mêmes composants, il s'agit des contraintes de g0 non présentes dans g1. Sinon il s'agit de toutes les contraintes intra-groupe de g0.
  - Ajouter (si elles n'y sont pas déjà) les destructions des contraintes référentielles de g0 à LDR si le type d'entités auquel appartient g0 n'est pas modifié dans LEP1 (modification provenant de la restructuration sur les collections). Si g1 et g0 ont les mêmes composants, il s'agit des contraintes référentielles de g0 non présentes dans g1. Sinon il s'agit de toutes les contraintes référentielles de g0.
  - Ajouter (si elles n'y sont pas déjà) les créations des contraintes intra-groupes de g1 à LCG si le type d'entités auquel appartient g1 n'est pas modifié dans LEP1 (modification provenant de la restructuration sur les collections). Si g1 et g0 ont les mêmes composants, il

s'agit de la contraintes intra-groupe de g1 non présentes dans g0. Sinon il s'agit de toutes les contraintes intra-groupe de g1.

- Ajouter (si elles n'y sont pas déjà) les créations des contraintes référentielles de g1 à LCR. Si g1 et g0 ont les mêmes composants, il s'agit des contraintes référentielles de g1 non présentes dans g0. Sinon il s'agit de toutes les contraintes référentielles de g1.
- Enlever T de LEP1.

- Si T = [MODIFATT,a1,a0], pour tous les groupes (g) auxquels a1 appartient :
  - Ajouter (si elles n'y sont pas déjà) les destructions des contraintes intra-groupe de g à LDG si le type d'entités auquel appartient g n'est pas modifié dans LEP1 (modification provenant de la restructuration sur les collections).
  - Ajouter (si elles n'y sont pas déjà) les destructions des contraintes référentielles en provenance (g est une contrainte référentielle) ou à destination (g est un identifiant référencé) de g à LDR si le type d'entités auquel appartient g n'est pas modifié dans LEP1 (modification provenant de la restructuration sur les collections).
  - Ajouter (si elles n'y sont pas déjà) les créations des contraintes intra-groupes de g à LCG si le type d'entités auquel appartient g n'est pas modifié dans LEP1 (modification provenant de la restructuration sur les collections).
  - Ajouter (si elles n'y sont pas déjà) les créations des contraintes référentielles à destination ou en provenance de g dans LCR.
- Si T = [CREERENT,e1,/], pour tous les groupes (g) de e1 : ajouter (si elles n'y sont pas déjà) les créations des contraintes référentielles à destination ou en provenance de g dans LCR.
- Si T = [MODIFENT,e1,e0], pour tous les groupes (g) de e1 : ajouter (si elles n'y sont pas déjà) les créations des contraintes référentielles à destination ou en provenance de g dans LCR.
- Si T = [TRANSFO,[no1,...,non],[ao1,...,aom]], pour tous les groupes (g) appartenant aux nouveaux objets créés (no1,...,non), ajouter (si elles n'y sont pas déjà) les créations des contraintes référentielles à destination ou en provenance de g dans LCR.
- Les autres transformations restent inchangées.

7.3.5.4 Conclusion

Finalement, on reconstruit la liste LEP1 à partir des listes de travail. On obtient : LEP1 = LCC + LDG + LDR + LEP1 + LCG + LCR + LDC.

7.3.6 Procédure "GenereScriptSQL"

A partir de la liste des modifications LEP1, la procédure "GenereScriptSQL" produit les scripts SQL de conversion des structures de données et de leurs instances.

Pour tous les types de transformation de LEP1, le tableau 7.2 donne une description et les pages de référence dans l'annexe C des scripts de conversion correspondants.

Types de transformation	Descriptions	Références
CREERENT	Création de la table (`CREATE TABLE ...`) avec ses colonnes, ses clés primaires et candidates, ses index et ses autres contraintes.	page 74
CREERATT	Création de la colonne dans sa table (`ALTER TABLE ... ADD ...`).	page 76
CREERID	Création d'une clé primaire ou candidate (`ALTER TABLE ... ADD CONSTRAINT ... PRIMARY KEY` ou `UNIQUE ...`).	page 86

Tableau 7.2 - Descriptions et pages de référence (annexe C) des scripts de conversion associés à chaque type de transformation de LEP1.

Types de transformation	Descriptions	Références
CREERFK	Création d'une clé étrangère (`ALTER TABLE ... ADD CONSTRAINT ... FOREIGN KEY ...`). Création d'une contrainte d'égalité si nécessaire (`ALTER TABLE ... ADD CONSTRAINT ... CHECK(...)` ou `CREATE TRIGGER ...`).	page 82 page 116
CREERACC	Création d'un index (`CREATE INDEX ...`).	page 90
CREERCOL	Création de l'espace de stockage (`CREATE TABLESPACE ...`).	page 92
DETRUIREENT	Destruction d'une table (`DROP TABLE ...`).	page 75
DETRUIREATT	Destruction d'une colonne d'une table (`ALTER TABLE ... DROP ...`).	page 77
DETRUIREID	Destruction d'une clé primaire ou candidate (`ALTER TABLE ... DROP CONSTRAINT ...`).	page 87
DETRUIREFK	Destruction d'une clé étrangère (`ALTER TABLE ... DROP CONSTRAINT ...`). Destruction d'une contrainte d'égalité si nécessaire (`ALTER TABLE ... DROP CONSTRAINT ...` ou `DROP TRIGGER ...`).	page 83 page 115
DETRUIREACC	Destruction d'un index (`DROP INDEX ...`).	page 91
DETRUIRECOL	Destruction d'un espace de stockage (`DROP TABLESPACE ..`).	page 93
MODIFENT	Pour un renommage : création de la nouvelle table, transfert des instances et destruction de l'ancienne table	page 75
	Pour un changement d'espace de stockage : création d'une table temporaire avec les instances de la table à modifier, destruction de la table, création de la table dans ses espaces de stockage, transfert des instances de la table temporaire dans la table et destruction de la table temporaire.	page 95
MODIFATT	Pour le renommage : création d'une nouvelle colonne dans la table, transfert des instances de l'ancienne colonne dans la nouvelle et destruction de l'ancienne colonne.	page 78
	Pour les autres transformations : création d'une colonne temporaire dans la table, transfert des instances de la colonne à modifier vers la colonne temporaire, destruction de la colonne à modifier, création de la colonne modifée, transfert des instances de la colonne temporaire dans la colonne modifiée et destruction de la colonne temporaire.	page 78 à page 82
TRANSFO	Création des nouvelles structures, transfert des données des anciennes structures vers les nouvelles et destruction des anciennes structures.	page 198

Tableau 7.2 - Descriptions et pages de référence (annexe C) des scripts de conversion associés à chaque type de transformation de LEP1.

7.4 Modification des programmes

7.4.1 Introduction

La modification des programmes est une tâche complexe impossible à automatiser sauf dans certains cas simples (voir les chapitres 1 et 5 : point 1.4.3 et section 5.7). Cette section montre comment utiliser des outils de compréhension de programmes pour localiser les sections de codes susceptibles d'être modifiées suite aux changements des données de la base utilisées par les programmes analysés. Nous allons procéder en trois phases : la recherche des requêtes SQL concernées par la conversion des données (point 7.4.2), la recherche des variables des programmes impliquées dans ces requêtes (point 7.4.3) et l'utilisation de la fragmentation de programmes pour affiner et compléter le résultat (point 7.4.4).

7.4.2 Première phase : Recherche des requêtes

7.4.2.1 Objectif

Les requêtes SQL utilisées dans les programmes sont les seuls points d'entrée dont dispose le programmeur pour accéder aux données de la base. Dans ce contexte, la première phase dans la localisation des sections de codes à modifier est la recherche dans les programmes de toutes les requêtes susceptibles d'être modifiées suite aux changements apportés à la base de données.

A partir de la liste des modifications LEP1 créée au point 7.3.5, l'ingénieur de maintenance va utiliser un moteur de recherche de patrons syntaxiques pour repérer toutes les requêtes SQL utilisant des tables ou des colonnes impliquées dans les modifications de LEP1. Les quatre types de requêtes par lesquelles les programmes établissent des liens avec les données sont : la sélection (`SELECT`), l'insertion (`INSERT`), la suppression (`DELETE`) et la modification (`UPDATE`) d'instances d'une table.

7.4.2.2 Principe

Les requêtes contiennent, mis à part les mots-clés spécifiques à SQL, des noms de tables (éventuellement avec des alias), des noms de colonnes et des noms de variables ou des curseurs dans lesquels des valeurs de colonnes sont stockées. Le tableau 7.3 propose pour chaque modification possible de LEP1 (voir le tableau 7.2 pour une description des modifications) les éléments (table, colonne ou variable) sur lesquels la recherche doit être effectuée. Remarquons que, pour les transformations de structures (TRANSFO), les trois éléments sont cochés mais il faut une évaluation au cas par cas pour déterminer ce qu'il faut rechercher. Pour obtenir des informations complémentaires sur les recherches nécessaires, les pages mentionnées dans la dernière colonne référencent l'étude des modifications de l'annexe C.

Types de transformation	Table	Colonne	Variable	Références
CREERENT				page 74
CREERATT	X		X	page 76
CREERID		X	X	page 86
CREERFK		X	X	page 82 et page 116
CREERACC				page 90
CREERCOL				page 92
DETRUIREENT	X		X	page 75

Tableau 7.3 - Eléments (table, colonne ou variable) propres à chaque type de transformation de LEP1 sur lesquels la recherche de patrons doit s'effectuer.

Types de transformation	Table	Colonne	Variable	Références
DETRUIREATT		X	X	page 77
DETRUIREID		X	X	page 87
DETRUIREFK		X	X	page 83 et page 115
DETRUIREACC				page 91
DETRUIRECOL				page 93
MODIFENT (renommage)	X			page 75
MODIFENT (transfert d'espace de stockage)				page 95
MODIFATT (renommage)		X		page 78
MODIFATT (modification de longueur ou de type)		X	X	page 78 à page 82
TRANSFO	X	X	X	page 198

Tableau 7.3 - Eléments (table, colonne ou variable) propres à chaque type de transformation de LEP1 sur lesquels la recherche de patrons doit s'effectuer.

Premièrement, le nom des colonnes n'étant pas unique dans le schéma de la base de données, il faut rechercher, si nécessaire, les tables modifiées (*Table* dans le tableau 7.3) ou dont les colonnes sont modifiées (*Colonne* dans le tableau 7.3). A ce stade, une première simplification est possible. En effet, plusieurs modifications peuvent impliquer une même table ce qui rend inutile plusieurs exécutions de la même recherche. Par exemple, si deux colonnes sont créées dans une même table (deux CREERATT dans LEP1), on recherche une seul fois les requêtes impliquant la table. C'est la personne chargée de la modification des programmes qui doit veiller à la simplification des recherches.

Pour rechercher les tables dans les requêtes, les patrons [9] suivants sont utilisés :

```
- ::=   /g"[/n/t/r ]+";            /* un ou plusieurs espaces obligatoires */
any ::= /g"[^;]*" ;               /* tous les caractères sauf un point virgule */
pr ::= /g"[, /n/t/r]";
mot ::= /g"[a-zA-Z][a-zA-Z0-9_]*"; /* un chaîne de caractères commençant par une
                                      lettre majuscule ou minuscule */
table ::= mot;

from ::=  "from" any pr @table;
update ::= "update" - @table;
insert ::= "insert" - "into" - @table;
```

Il s'agit de retrouver tous les `FROM` contenant la table recherchée (dans les instructions `SELECT`, `DELETE` ou `INSERT`) ainsi que les `UPDATE` et `INSERT` sur cette table. Dans les patrons, la variable `@table` doit être instanciée avec le nom de la table recherchée. On obtient ainsi toutes les requêtes impliquant la table. Remarquons que, pour le renommage d'une table, le travail de localisation est déjà terminé. Il suffit de remplacer, dans toutes les requêtes trouvées, l'ancien nom par le nouveau.

Deuxièmement, pour toutes les modifications de LEP1 où la recherche doit s'effectuer sur une ou plusieurs colonnes, il faut trouver dans les requêtes localisées précédemment celles qui contiennent ces colonnes. Il s'agit des requêtes du type (où T est la table trouvée dans le premier point et C la colonne recherchée) :

```
SELECT * FROM ... T ...
SELECT ... C ... FROM ... T ...
SELECT ... FROM ... T ... WHERE ... C ...
SELECT ... FROM ... T ... GROUP BY ... C ...
SELECT ... FROM ... T ... HAVING ... C ...
DELETE FROM T WHERE ... C ...
UPDATE T SET ... C ...
UPDATE T ... WHERE ... C ...
```

9. La syntaxe du langage PDL est commentée au point 7.1.6.1.

```
INSERT INTO T ...
INSERT INTO T(... C ...) ...
```

Les patrons utilisés par le moteur de recherche pour localiser ces types de requêtes sont :

```
- ::=   /g"[/n/t/r ]+";
any ::= /g"[^;]*" ;
mot ::= /g"[a-zA-Z][a-zA-Z0-9_]*";
col ::= mot;

select_all ::= "select" - "*";
select ::= "select" any @col;
where ::= "where" any @col;
group_by ::= "group by" any @col;
having ::= "having" any @col;
set ::= "set" any @col;
insert_all ::= "insert" - "into" - @table - "values";
insert ::= "insert" - "into" - @table - "(" any @col;
```

Dans les patrons, la variable `@col` doit être instanciée avec le nom de chaque colonne recherchée. Par exemple, pour une modification de colonne, la variable `@col` est instanciée avec le nom de la colonne alors que, pour une création de clé primaire, elle est instanciée pour chaque colonne appartenant à la nouvelle clé.

Pratiquement, le générateur de script (section 7.3) va créer à partir de LEP1 un fichier contenant tous les patrons instanciés pour la recherche des tables et des colonnes en éliminant les redondances. L'ingénieur doit lancer la recherche sur la base du fichier de patrons généré.

Maintenant, on dispose de toutes les requêtes définissant un point d'entrée dans les programmes pour une modification de LEP1. Les phases suivantes vont compléter la localisation des instructions à modifier.

7.4.2.3 Evaluation

La recherche de patrons est une technique souple et facilement paramétrable (via la définition de patrons) pour d'autres technologies ou des syntaxes particulières. Malheureusement, la syntaxe utilisée pour définir les patrons n'est pas suffisamment puissante pour rechercher des colonnes dans une requête en une seule passe. En effet, pour trouver les requêtes du style `SELECT ... FROM ... T ... WHERE ... C ...`, il faudrait définir un patron : `"from" any pr @table any "where" any @col`. Malheureusement, ce genre de patron autorise que le `FROM` recherché soit dans une requête et le `WHERE` dans une autre requête sans lien avec la première. Pour une recherche efficace en une passe, il faudrait développer un analyseur de code spécifique plus difficilement adaptable.

7.4.3 Deuxième phase : Recherche des variables dépendantes

7.4.3.1 Objectif

Dans le contexte de ce travail, on considère que les liens entre les données et les programmes se font par le biais de variables et de curseurs. On distingue trois type de flux entre les données et les variables :

1. Le flux *donnée-variable* signifie que des valeurs des instances de la base de données sont stockées dans des variables ou des curseurs. Ce type de flux est présent dans les requêtes de sélection du type :

```
DECLARE C CURSOR FOR SELECT ... col ... FROM ...
DECLARE C CURSOR FOR SELECT * FROM ...
```

```
FETCH C INTO ... var ...
SELECT ... col ... INTO ... var ... FROM ...
SELECT * INTO ... var ... FROM ...
```

2. Le flux *variable-donnée* représente le stockage de valeurs des variables dans des instances de la base de données. On le trouve dans des requêtes d'insertion et de modification d'instances du type :

```
UPDATE EMPLOYE SET ... col op var ...
INSERT INTO EMPLOYE VALUES(... var ...)
INSERT INTO EMPLOYE(... col ...) VALUES(... var ...)
```

3. La *comparaison* définit les comparaisons réalisées dans des requêtes entre des instances de colonnes et des variables. Ce type de flux se présente dans des clauses `WHERE` et `HAVING` appartenant à des requêtes du type :

```
SELECT ... FROM ...  WHERE ... col op var ...
SELECT ... FROM ... EMPLOYE ... HAVING ... col op var ...
DELETE FROM ... WHERE ... col op var ...
UPDATE ... SET ... WHERE ... col op var ...
INSERT INTO ... SELECT ... FROM ... WHERE ... col op var ...
INSERT INTO ... SELECT ... FROM ... EMPLOYE ... HAVING ... col op var ...
```

L'objectif de cette phase consiste à trouver toutes les variables des programmes qui sont liées aux tables et aux colonnes recherchées dans la première étape. Cela concerne les variables présentent dans les requêtes mais aussi celles qui en dépendent. Cette recherche permet de localiser les principales instructions (autres que les accès à la base de données) des programmes susceptibles d'être modifiées par le programmeur.

Pour atteindre cet objectif, on a choisi la technique du graphe de dépendances qui est un graphe où chaque variable du programme est représentée par un nœud et dont les arcs (orientés ou non) représentent une relation (assignation, comparaison,...) entre deux variables. Les relations sont définies par l'ingénieur sous la forme d'une liste de patrons. La technique du graphe de dépendances est présentée au point 7.1.6.2.

7.4.3.2 Principe

Le calcul du graphe de dépendances des variables se fait sur la base de patrons définis au préalable. Ces patrons dépendent du langage de programmation utilisé. Puisque la démarche proposée ne fait aucune hypothèse sur le langage de programmation utilisé, la tâche de définition des patrons incombe aux ingénieurs responsable de la maintenance et qui ont une bonne connaissance du langage. Les patrons ci-dessous définissent des relations (`MOVE`, `WRITE`, `REDEFINES` et `IF`) entre variables pour des programmes COBOL :

```
- ::= /g"[/n/t/r ]+";
~ ::= /g"[/n/t/r ]*";
sep_cob ::= /g"[/n/t/r .]";
var ::= /g"[a-zA-Z][-a-zA-Z0-9]*";
rel_op ::= /g"[=<>]+";
var_1 ::= var;
var_2 ::= var;
redefines ::= @var_1 - "REDEFINES" - @var_2 ;
write ::= "WRITE" - @var_1 - "FROM" @var_2 ;
move ::= "MOVE" - ["CORRESPONDING" -] @var_1 - ["OF" - var -]
         "TO" - @var_2 [- "OF" - var];
if ::= "IF" - @var_1 - ["OF" - var - ] ["NOT" -] rel_op - @var_2 [- "OF" - var];
```

Une fois le graphe de dépendances construit, il faut, dans chaque requête obtenue dans la première phase et pour les variables ou curseurs liés aux modifications [10], localiser les variables qui en dépendent grâce au graphe. Si les variables trouvées sont, à leur tour, utilisées dans des requêtes qui ne font pas partie des requêtes trouvées dans la première phase, le processus est réappliqué sur les variables en relation avec les variables trouvées précédemment et ainsi de suite jusqu'à ce qu'on ne trouve plus de variables dans des requêtes non répertoriées. On obtient ainsi les instructions contenant les variables dépendantes des requêtes trouvées lors de la première phase.

Pratiquement, un analyseur passe en revue les requêtes pour dégager toutes les variables intéressantes. Il parcourt le graphe de dépendances pour trouver les variables dépendantes et localise les instructions contenant ces variables.

7.4.3.3 Evaluation

Cette technique est générique car elle peut être facilement adaptable à d'autres langages de programmation. Elle s'intègre donc bien dans l'approche proposée. D'autant plus que l'apprentissage des règles de définition des patrons (sur lesquels se base le constructeur du graphe de dépendances) est relativement aisé.

Par contre, elle peut générer du bruit ou du silence au niveau du résultat. Cette technique peut rater certaines dépendances (silence) ou, au contraire, trouver des dépendances erronées (bruit). Les lecteurs intéressés par une évaluation précise des outils de compréhension de programmes peuvent consulter [Hen01]. L'auteur repère trois types de silence et deux types de bruit pour le graphe de dépendances des variables.

Silences

1. Il faut absolument définir toutes les relations possibles dans le langage de programmation concerné. Il ne faut pas seulement tenir compte des assignations car il existe d'autres instructions qui contribuent au flux des données.
   Exemple : Dans les patrons COBOL définis dans le point 7.4.3.2, on ne tient pas compte d'instructions du type `COMPUTE`, `MULTIPLY`, ...
2. Le graphe ne tient pas compte de la structure des variables.
   Exemple : La figure 7.28 présente un exemple en COBOL où une variable composite (B) empêche la découverte de la dépendance entre (A1,A2) et C.

```
01 A1 PIC X(10).
01 A2 PIC X(10).
01 B.
  02 B1 PIC X(10)
  02 B2 PIC X(10)
01 C PIC X(20).

MOVE A1 TO B1
MOVE A2 TO B2
MOVE B TO C
```

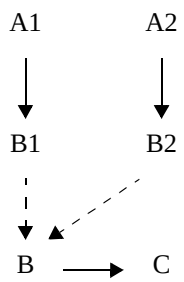


Figure 7.28 - Un exemple de variable composite (B) qui génère des silences dans un graphe de dépendances.

3. L'ignorance des flux de contrôle génère également des silences.
   Exemple : Une intruction du type `IF A = "X" THEN B := "Y" ELSE B := "N"` implique une dépendance entre les variables `A` et `B`.

10. Il s'agit de toutes les variables si la modification concerne une table ou seulement les variables en relation avec les colonnes impliquées dans la modification.

Les deux derniers types de silence sont très difficile à combler avec la technique des graphes de dépendances et celle de définition des patrons.

Bruits

1. Des bruits peuvent être générés parce que les graphes ne tiennent pas compte des flux de contrôle.
   Exemple : Si on considère le code de la figure 7.29 a), le graphe de dépendances montre deux dépendances (une entre A et C et une autre entre E et D) qui sont erronées d'après le code.
2. Une variable peut contenir une valeur qui n'apparaît pas dans la base.
   Exemple : Si on considère le code de la figure 7.29 b) où A1 est un champ de A et B1 un champ de B. Le graphe montre une dépendance entre A1 et B1 dont on pourrait conclure érronément que B dépend de A.

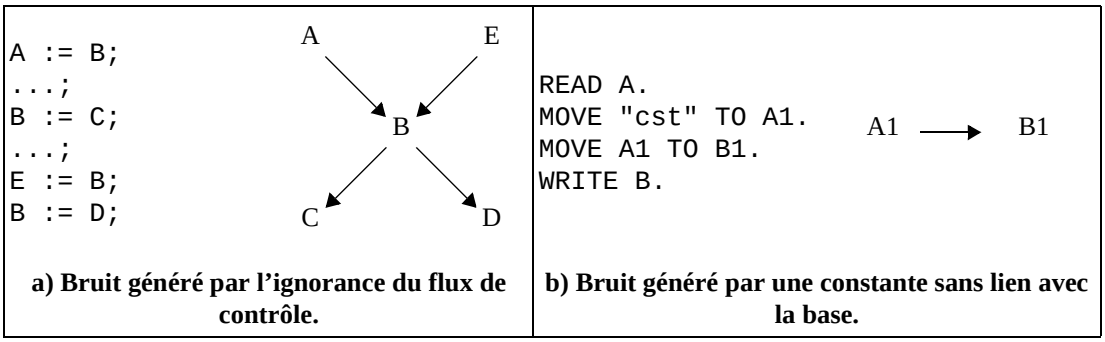


Figure 7.29 - Deux exemples de bruit dans un graphe de dépendances des variables.

En conclusion, le graphe de dépendances des variables est une technique intéressante car elle peut être facilement mise en œuvre et adaptée mais elle souffre de lacunes qui peuvent poser problème dans certains programmes. Sachant que toutes ces lacunes ne peuvent être comblée par des définitions précises et complètes des patrons de recherche, une troisième étape a été définie pour affiner la délimitation des sections de code à modifier.

7.4.4 Troisième phase : Fragmentation des programmes

7.4.4.1 Objectif

Comme on l'a vu dans la deuxième phase du processus d'identification des instructions, le graphe de dépendances ne suffit pas toujours pour déterminer les instructions susceptibles d'être modifiées. Les caractéristiques des différents langages de programmation comme les procédures, les contrôles de flux arbitraires (goto), les types de données composites et les pointeurs nécessitent des solutions spécifiques. Pour affiner la recherche, on propose la technique de fragmentation de programmes. Cette phase peut se révéler facultative, dans certains cas, où le calcul du graphe de dépendances est suffisant. C'est le cas de programmes dans lesquel les instructions d'accès à la base de données sont encapsulées dans des modules spécifiques ce qui rend la tâche de maintenance plus aisée. Dans les cas plus complexe, la fragmentation de programmes devient une phase nécessaire du processus.

7.4.4.2 Principe

La fragmentation de programmes (voir point 7.1.6.3) permet d'extraire d'un programme le fragment nécessaire et suffisant pour comprendre et expliquer le comportement de ce programme à un point (appelé critère de fragmentation) constitué d'un couple [instruction, variable] tel que la variable est référencée dans l'instruction.

La fragmentation de programmes analyse un programme et construit le graphe de dépendances du système (ou SDG). Le SDG est un graphe orienté dont les nœuds correspondent aux instructions et les arcs représentent les dépendances de données, les dépendances de contrôle et les appels de procédures.

Un critère de fragmentation correspond à un nœud du graphe. Un fragment est l'ensemble des nœuds du SDG à partir desquels on peut atteindre le nœud correspondant au critère de fragmentation. Le calcul d'un fragment est défini en termes de parcours du SDG.

Le choix des critères de fragmentation représente le travail essentiel une fois le SDG construit. La meilleure solution consiste à choisir toutes les instructions où la variable dont on veut calculer le fragment est référencée. Il s'agit le plus souvent d'instructions d'écriture ou de requêtes stockant la valeur de la variable dans la base de données. L'union des fragments calculés pour chaque instructions considérées donne l'ensemble des lignes de codes à vérifier pour intégrer les modifications.

7.4.4.3 Evaluation

La fragmentation de programmes tient compte du flux de contrôle, des types de données composites et d'autres structures qui posent des problèmes avec la technique du graphe de dépendances des variables. Cela en fait une technique plus sûr car elle génère moins de bruit ou de silences.

Par contre, la fragmentation de programmes est fortement dépendante du langage de programmation choisi en ce qui concerne la construction du SDG. Pour adapter cet outil à d'autres langage, il faut modifier l'algorithme complexe de construction du SDG. La gestion du SDG est, quant à elle, indépendante du langage employé dans les programmes. C'est une technique nettement moins souple que le graphe de dépendances des variables.

7.4.5 Conclusion

Face à la complexité des programmes qui empêche, dans la plupart des cas, leur modification automatique, les outils présentés dans cette section permettent de localiser automatiquement et le plus finement possible les instructions qui sont susceptibles d'être modifiées suite aux changements intervenus dans une base de données. Dans ce contexte, la modification des lignes de code reste une tâche prise en charge par les programmeurs, sauf dans les cas simples où les programmes ne dépendent pas structurellement des modifications.

Bien sûr, cette solution ne résout pas tous les problèmes. Par exemple, la création d'une table n'empêche pas les programmes de fonctionner normalement, mais elle engendre souvent des modifications sous la forme de nouvelles fonctions de gestion de la table. Dans ce cas, l'outil de localisation est inutile. Toutefois, il semble être un bon compromis dans le cadre d'applications de base de données héritées.

«Tous les conseils de bonne foi ne valent pas un bon exemple.»

E. Prévot

CHAPITRE 8 ETUDE DE CAS

L'objectif majeur de ce chapitre est de démontrer la faisabilité de la méthode ainsi que l'efficacité du support (inclus dans l'AGL DB-Main) mis à disposition de l'ingénieur de maintenance. Pour atteindre cet objectif, une application réaliste a été développée de manière à illustrer au mieux les principes de la méthode et les difficultés que l'ingénieur peut rencontrer.

Le point de départ de l'étude de cas (section 8.1) est une petite application de base de données relationnelles ainsi que la documentation correspondante pour chaque niveau d'abstraction. Il s'agit d'une application fictive de gestion des clients, des commandes et des représentants d'une entreprise commerciale. Les informations de la base de données sont suffisantes pour établir les quotas de ventes des représentants d'une région et le budget publicitaire alloué à celle-ci.

La deuxième partie de l'étude de cas (section 8.2) présente les changements apportés à l'application par le biais de modifications de la partie base de données. Ces dernières vont être propagées au travers de toute la documentation pour aboutir à la conversion des structures de données, des données et des programmes. La propagation est illustrée au travers de la première stratégie consistant à modifier les spécifications conceptuelles (voir chapitres 5 et 7). Les principes de base de la méthode étant abondamment utilisés par la première stratégie, les deux autres stratégies ne sont pas abordées dans ce chapitre.

8.1 Présentation de l'existant

Cette section présente les principales phases de la conception de l'application à la base de l'étude de cas. La description de chaque phase est légère puisque l'objectif n'est pas de détailler une conception mais bien la modification d'une application existante. Après un bref énoncé introduisant les principales données et certains objectifs de l'application (8.1.1), le point (8.1.2) donne l'historique de la conception afin de bien visualiser l'existant. Les points 8.1.3 à 8.1.5 passent en revue les trois premières phases de la conception, c'est-à-dire, l'analyse conceptuelle, les conceptions logique et physique. Le point 8.1.6 décrit les structures de données, les instances et quelques fonctions intéressantes de l'application.

8.1.1 Enoncé

Le point de départ de l'étude de cas est un énoncé tiré de [For91] décrivant les spécifications de la gestion des clients, des représentants et des produits d'une société commerciale. Elles sont, entre autres, définies de manière à permettre l'établissement du quota de vente des représentants d'une région et du budget publicitaire alloué à celle-ci.

Une société commerciale dispose d'un certain nombre de représentants. Ceux-ci sont répartis en régions et s'occupent de clients. Ces représentants mettent en vente toute la gamme de produits disponibles pour la région. Un client appartient à une région et est visité par un et un seul représentant. Il passe des commandes à celui-ci. Chaque client commande les produits disponibles dans sa région.

Ces commandes portent sur des produits mis en vente dans la région. Chaque produit est caractérisé par un prix propre à la région où il est mis en vente.

Au début de chaque année, la société désire fixer un quota de vente pour chacun de ses représentants et un budget publicitaire pour la région. Ces valeurs seront calculées sur une période d'analyse. La détermination de la publicité et des quotas était précédemment effectuées par des experts.

8.1.2 Historique de la conception

La figure 8.1 présente l'historique de la conception de la partie base de données de l'application *QUOTA VENTES*.

Il contient les quatre processus d'ingénierie classique :

1. L'analyse conceptuelle reçoit en entrée l'énoncé de l'application et fournit comme produit résultat, le schéma conceptuel *QUOTA VENTES/Conceptuel*.
2. La conception logique prend en entrée le schéma conceptuel et fournit en sortie comme produits résultats, le schéma logique *QUOTA VENTES/Logique* et l'historique de conception logique matérialisé dans le journal *cl0.log*.
3. La conception physique prend en entrée le schéma logique et fournit en sortie le schéma physique *QUOTA VENTES/Physique* et l'historique de conception physique matérialisé dans le journal *cp0.log*.
4. Le codage donne à partir du schéma physique le script de création des structures de données (*quota.sql*) et les programmes de l'application.

Les points 8.1.3 à 8.1.5 décrivent sommairement chacun des processus.

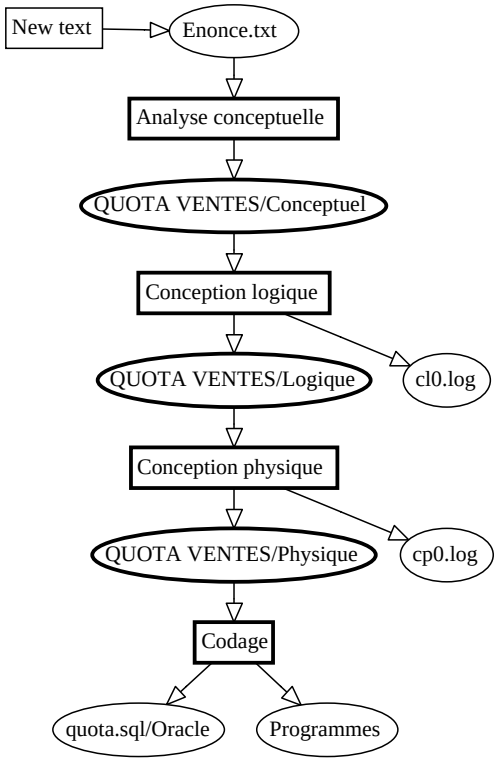


Figure 8.1 - Historique de la conception de la base de données *QUOTA VENTES*.

8.1.3 Analyse conceptuelle

A partir de l'énoncé, le schéma conceptuel de la figure 8.2 est créé. Le but n'étant pas de détailler la conception, le processus de création du schéma conceptuel à partir des spécifications est passé sous silence. Les types d'entités ont été enrichis avec des attributs afin d'obtenir un schéma conceptuel réaliste. Ce schéma contient quelques structures complexes intéressantes : les attributs décomposables *Adresse* des types d'entités *REPRESENTANT* et *CLIENT*, l'attribut multivalué *Telephone* de *REPRESENTANT* et les types d'associations complexes (plusieurs-à-plusieurs) *vente* et *detail*.

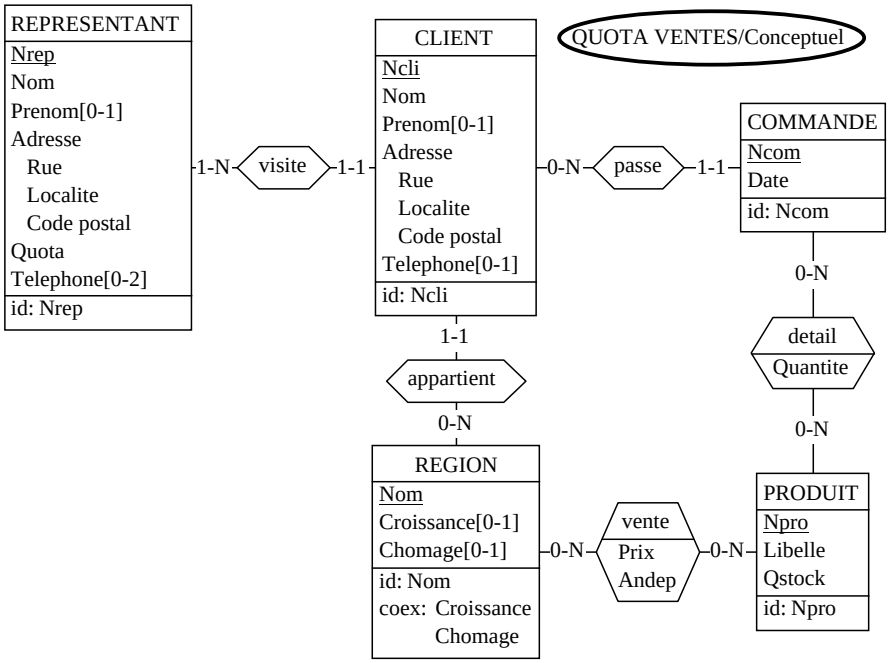


Figure 8.2 - Schéma conceptuel *QUOTA VENTES*.

8.1.4 Conception logique

La conception logique consiste à transformer le schéma conceptuel de la figure 8.2 en un schéma logique relationnel (figure 8.3). Les transformations de conception logique utilisées sont enregistrées dans un historique matérialisé sous la forme d'un journal (cl0.log). Cet historique contient la trace des transformations nécessaires pour transformer le schéma conceptuel en un schéma conforme à un modèle relationnel. En bref, les attributs *Adresse* des types d'entités *REPRESENTANT* et *CLIENT* sont désagrégés. Les types d'associations *vente* et *détail* sont transformés en types d'entités. Tous les types d'associations sont transformés en clés étrangères. L'historique H_{CL0} donne les signatures des transformations exécutées :

```
H_CL0 = <
  (REPRESENTANT,{Adr_Rue,Adr_Localite,Adr_Code postal})<-
                                              desagreger-Att(REPRESENTANT,Adresse)
  (CLIENT,{Adr_Rue,Adr_Localite,Adr_Code postal})<-desagreger-Att(CLIENT,Adresse)
  Adr_cp<-modifier-Att(REPRESENTANT,Adr_Code postal,"Adr_cp")
  Adr_cp<-modifier-Att(CLIENT,Adr_Code postal,"Adr_cp")
  (REPRESENTANT,{Telephone1,Telephone2})<-Attmult-en-Serie-Att(REPRESENTANT,Telephone)
  (DETAIL,{cd,pd})<-TA-en-TE(detail)
  (VENTE,{vr,pv})<-TA-en-TE(vente)
  (CLIENT,{Nrep})<-TA-en-FK(visite)
  (CLIENT,{Nomreg})<-TA-en-FK(appartient)
  (VENTE,{Nomreg})<-TA-en-FK(rv)
  (VENTE,{Npro})<-TA-en-FK(pv)
  (COMMANDE,{Ncli})<-TA-en-FK(passe)
  (DETAIL,{Ncom})<-TA-en-FK(cd)
  (DETAIL,{Npro})<-TA-en-FK(pd)
>
```

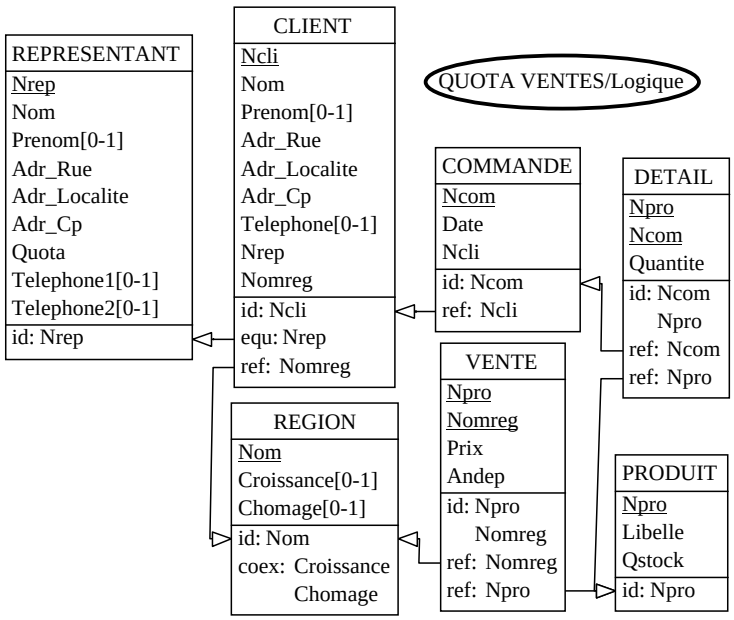


Figure 8.3 - Schéma logique relationnel *QUOTA VENTES*.

8.1.5 Conception physique

Le schéma logique relationnel (figure 8.3) est transformé en un schéma physique relationnel (figure 8.4). Les transformations exécutées sont enregistrées dans un historique matérialisé dans le journal *cp0.log*. Il contient les transformations concernant les structures physiques relationnelles (index et espaces de stockage) introduites au niveau physique. Les espaces de stockage *CLIENTS* et *PRODUITS* sont créés. Ils contiennent les tables du schéma physique. La colonne *Date* de la table *COMMANDE* est renommée en *Datecom* car *Date* est un mot réservé dans certains SGBDR (comme Oracle par exemple). Tous les groupes deviennent des index excepté les groupes préfixes et le groupe de coexistence de *REGION*. L'historique de la conception physique H_{CP0} est :

```
H_CP0 = <
  Datecom<-modifier-Att(COMMANDE,Date,"Datecom")
  CLIENTS<-creer-Coll("CLIENTS")
  (CLIENTS,{CLIENT,REPRESENTANT,REGION})<-
                            modifier-Coll(CLIENTS,{CLIENT,REPRESENTANT,REGION})
  PRODUITS<-creer-Coll("PRODUITS")
  (PRODUITS,{COMMANDE,DETAIL,PRODUIT,VENTE})<-
                            modifier-Coll(PRODUITS,{COMMANDE,DETAIL,PRODUIT,VENTE})
  IDCLIENT<-creer-Index(CLIENT,"IDCLIENT",{Ncli})
  IDREPRESENTANT<-creer-Index(REPRESENTANT,"IDREPRESENTANT",{Nrep})
  FKvisite<-creer-Index(CLIENT,"FKvisite",{Nrep})
  FKappartient<-creer-Index(CLIENT,"FKappartient",{Nomreg})
  IDREGION<-creer-Index(REGION,"IDREGION",{Nom})
  IDCOMMANDE<-creer-Index(COMMANDE,"IDCOMMANDE",{Ncom})
  FKpasse<-creer-Index(COMMANDE,"FKpasse",{Ncli})
  IDVENTE<-creer-Index(VENTE,"IDVENTE",{Npro,Nomreg})
  FKrv<-creer-Index(VENTE,"FKrv",{Nomreg})
  IDPRODUIT<-creer-Index(PRODUIT,"IDPRODUIT",{Npro})
  IDDETAIL<-creer-Index(DETAIL,"IDDETAIL",{Ncom,Npro})
  FKpd<-creer-Index(DETAIL,"FKpd",{Npro})
>
```

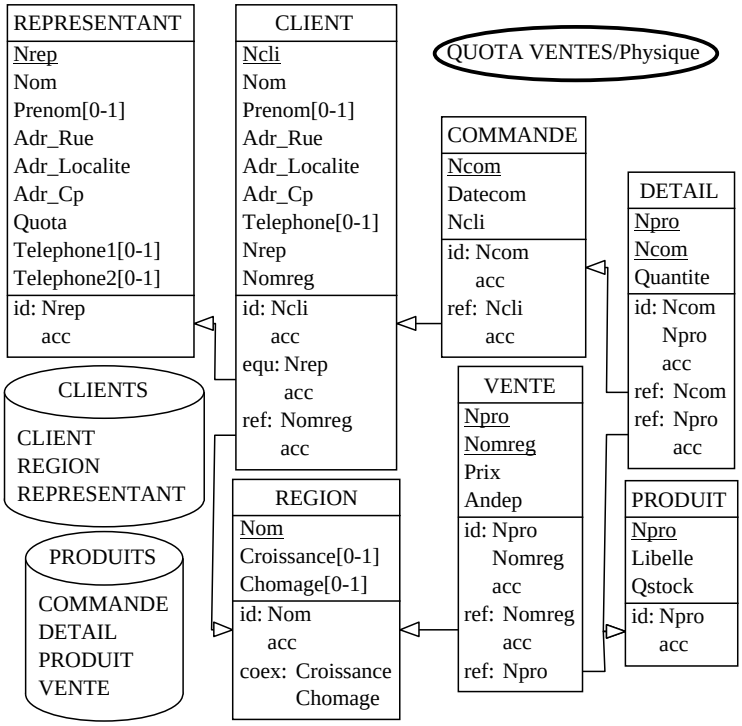


Figure 8.4 - Schéma physique relationnel *QUOTA VENTES*.

8.1.6 Structures de données, instances et programmes

8.1.6.1 Structures de données

L'application utilise le système de gestion de bases de données relationnel Oracle 8. Le script ci-dessous permet de créer les structures de données de la base. Les index sur les clés primaires sont créés automatiquement par Oracle. La contrainte référentielle d'égalité de *CLIENT*(*Nrep*) vers *REPRESENTANT*(*Nrep*) est traduite par une clé étrangère classique, plus une procédure stockée ("stored procedure") qui implémente la contrainte d'égalité selon laquelle un représentant visite au moins un client. Les clés étrangères peuvent être définies dans la création des tables. Toutefois, une clé étrangère doit toujours référencer une table déjà créée dans la base. Par conséquent, le script doit respecter un ordre bien déterminé pour la définition des structures des tables. De plus, lorsque deux tables se référencent mutuellement, il est impossible de créer les clés étrangères dans leur table respective. Pour éviter le problème des références en avant, la définition des clés étrangères est placée après celle des tables.

```
------------------------
-- Tablespace section --
------------------------
create tablespace CLIENTS datafile 'CLIENTS.DAT' size 100K reuse;
create tablespace PRODUITS datafile 'PRODUITS.DAT' size 100K reuse;
-------------------
-- Table section --
-------------------
create table CLIENT (Ncli numeric(5) not null,Nom char(50) not null,
  Prenom char(30),Adr_Rue char(40) not null,
  Adr_Localite char(30) not null,Adr_Cp numeric(4) not null,
  Telephone char(12),Nrep numeric(5) not null,Nomreg char(30) not null,
  constraint IDCLIENT primary key (Ncli))
  tablespace CLIENTS;
create table COMMANDE (Ncom numeric(5) not null,Datecom date not null,
```

```
  Ncli numeric(5) not null,constraint IDCOMMANDE primary key (Ncom))
  tablespace PRODUITS;
create table DETAIL (Npro char(5) not null,Ncom numeric(5) not null,
  Quantite numeric(6,2) not null,
  constraint IDDETAIL primary key (Ncom, Npro))
  tablespace PRODUITS;
create table PRODUIT (Npro char(5) not null,Libelle char(50) not null,
  Qstock numeric(10) not null,constraint IDPRODUIT primary key (Npro))
  tablespace PRODUITS;
create table REGION (Nom char(30) not null,Croissance float(3),
  Chomage float(3),constraint IDREGION primary key (Nom))
  tablespace CLIENTS;
create table REPRESENTANT (Nrep numeric(5) not null,Nom char(50) not null,
  Prenom char(30),Adr_Rue char(40) not null,
  Adr_Localite char(30) not null,Adr_Cp numeric(4) not null,
  Quota numeric(10,2) not null,Telephone1 char(12),Telephone2 char(12),
  constraint IDREPRESENTANT primary key (Nrep))
  tablespace CLIENTS;
create table VENTE (Npro char(5) not null,Nomreg char(30) not null,
  Prix numeric(8) not null,Andep numeric(2) not null,
  constraint IDVENTE primary key (Npro, Nomreg))
  tablespace PRODUITS;
-------------------------
-- Constraints Section --
-------------------------
alter table CLIENT add constraint FKvisite
     foreign key (Nrep) references REPRESENTANT;
alter table CLIENT add constraint FKappartient
     foreign key (Nomreg) references REGION;
alter table COMMANDE add constraint FKpasse
     foreign key (Ncli) references CLIENT;
alter table REGION add constraint GRREGION
     check((Croissance is not null and Chomage is not null)
           or (Croissance is null and Chomage is null));
create or replace procedure verifier_equ_FKvisite is
    n number;
    contrainte_egalite_violee exception;
  begin
    select count(*) into n from REPRESENTANT
                           where Nrep not in (select Nrep from CLIENT);
    if n > 0 then raise contrainte_egalite_violee; end if;
    exception when contrainte_egalite_violee then
    raise_application_error(-20303,
      'La contrainte d''egalite sur CLIENT.Nrep est violee.');
  end;
alter table DETAIL add constraint FKcd
     foreign key (Ncom) references COMMANDE;
alter table DETAIL add constraint FKpd
     foreign key (Npro) references PRODUIT;
alter table VENTE add constraint FKrv
     foreign key (Nomreg) references REGION;
alter table VENTE add constraint FKpv
     foreign key (Npro) references PRODUIT;
-------------------
-- Index Section --
-------------------
create index FKvisite on CLIENT (Nrep);
create index FKappartient on CLIENT (Nomreg);
create index FKpasse on COMMANDE (Ncli);
create index FKpd on DETAIL (Npro);
create index FKrv on VENTE (Nomreg);
```

8.1.6.2 Instances de la base de données

Les tableaux suivants donnent des exemples d'instances pour les tables de la base. Elles permettront d'illustrer les requêtes du script de conversion des données.

Table CLIENT

NCLI	*NOM*	*PRENOM*	*ADR_RUE*	*ADR_LOCALITE*	*ADR_CP*	*TELEPHONE*	*NREP*	*NOMREG*
112	HANSENNE	Albert	23, av. Dumont	Bastogne	6600	063/33.15.64	3	LUXEMBOURG
123	MERCIER	Jean	25, r. Lemaitre	Namur	5000	081/72.26.80	4	NAMUR
332	MONTI	Nadine	112, r. Neuve	Verviers	4800	087/33.55.86	2	LIEGE
10	TOUSSAINT	Jean-Pierre	5, r. Godefroid	Arlon	6700	063/23.15.89	3	LUXEMBOURG
111	VANBIST	Gérard	180, r. Florimont	Mons	7000	065/36.05.56	5	HAINAUT
127	VANDERKA	Paul	3, av. des roses	Namur	5000	081/72.18.96	4	NAMUR
512	GILLET	Francoise	14, r. de l'Ete	Bruxelles	1000	02/770.26.85	6	BRUXELLES
62	GOFFIN	Jean-Marie	72, r. de la gare	Namur	5000	081/72.35.56	4	NAMUR
400	FERARD	Jose	65, r. du Tertre	Arlon	6700	063/23.45.62	3	LUXEMBOURG
3	AVRON	Christophe	8, ch. de la Cure	Bruxelles	1000	02/780.35.25	1	BRUXELLES
729	NEUMAN	Alfred	40, r. Bransart	Bruxelles	1000	02/652.23.41	1	BRUXELLES
11	PONCELET		17, Clos des Erables	Liege	4000	041/77.62.74	2	LIEGE
422	FRANCK	Anne	60, r. de Wepion	Namur	5000	081/72.49.21	4	NAMUR
712	GUILLAUME	Pierre	14a, ch. des Roses	Liege	4000	041/77.23.58	2	LIEGE
63	MERCIER	Luc	201, bvd du Nord	Liege	4000	041/67.25.65	2	LIEGE
410	JACOB		78, ch. du Moulin	Bruxelles	1000	02/344.23.74	6	BRUXELLES

Table REPRESENTANT

NREP	*NOM*	*PRENOM*	*ADR_RUE*	*ADR_LOCALITE*	*ADR_CP*	*QUOTA*	*TELEPHONE1*	*TELEPHONE2*
1	VANHOUT	Jules	68, av. Dewandre	WAVRE	1300	1000000	02/336.12.54	02/335.15.59
2	ABSIL	Marc	50A, v. des Hougnes	SERAING	4100	2000000	041/96.23.54	075/33.12.24
3	VANDENBERG		16, r. G. Thiry	ATHUS	6791	500000	063/25.35.55	095/25.65.48
4	CLOOS	Jacques	29, av. des Fraises	WEPION	5100	750000	081/77.10.50	
5	DURANGO	Mauri	12, r. de Lilles	ATH	7812	500000	068/24.66.98	068/29.53.21
6	ELOI	Jean	36, blv. de Tournai	BRUXELLES	1010	1500000		

Table COMMANDE

NCOM	*DATECOM*	*NCLI*
178	12/03/2001	111
179	12/03/2001	400
182	14/03/2001	127
184	27/03/2001	400
185	02/04/2001	11
186	02/04/2001	400
188	03/04/2001	512

Table PRODUIT

NPRO	*LIBELLE*	*QSTOCK*
CS262	CHEV. SAPIN 200x6x2	45
CS264	CHEV. SAPIN 200x6x4	2690
CS464	CHEV. SAPIN 400x6x4	450
PA45	POINTE ACIER 45 (2K)	580
PA60	POINTE ACIER 60 (1K)	134
PH222	PL. HETRE 200x20x2	782
PS222	PL. SAPIN 200x20x2	1220

Table DETAIL

NPRO	*NCOM*	*QUANTITE*
CS464	178	250
PA60	179	200
CS262	179	600
PA60	182	300
CS464	184	1200
PA45	184	200
PA60	185	150
PS222	185	6000
CS464	185	2600
PA45	186	30
PA60	188	700
PH222	188	920
CS464	188	1800
PA45	188	220

Table VENTE

NPRO	*NOMREG*	*PRIX*	*ANDEP*
CS464	HAINAUT	220	98
PA60	LUXEMBOURG	90	98
CS464	LUXEMBOURG	215	99
CS262	LUXEMBOURG	70	99
PA45	LUXEMBOURG	100	00
PA60	NAMUR	95	98
PA60	LIEGE	90	98
PS222	LIEGE	190	00
CS464	LIEGE	220	00
PA60	BRUXELLES	95	00
PH222	BRUXELLES	235	00
CS464	BRUXELLES	225	01
PA45	BRUXELLES	110	01

Table REGION

NOM	*CROISSANCE*	*CHOMAGE*
BRUXELLES	0,041	0,075
LIEGE	0,043	0,076
NAMUR	0,03	0,079
HAINAUT	0,019	0,085
LUXEMBOURG	0,015	0,07

8.1.6.3 Programmes

Des extraits (écrits dans un pseudo-code facilement compréhensible) représentatifs de certaines fonctionnalités de l'application *QUOTA VENTES* sont maintenant présentés. Ils seront utilisés pour illustrer la localisation des sections de code à modifier. Les appels à la base de données se font par le biais des requêtes incluses [1] dans le code.

a) Affichage de toutes les informations concernant un représentant

La procédure *AFFICHEINFOREPRESENTANT* affiche à l'écran les informations concernant un représentant donné : le nom, le prénom, le numéro, l'adresse, le quota, les numéros téléphoniques, les informations sur les clients visités, le montant de ses ventes par région et le montant total de ses ventes.

```
Procedure AFFICHEINFOREPRESENTANT(NUMREP : integer)
 1  declaration
 2    NUM,NR,CODE,VEN,TOTVEN : integer;
 3    QUOT : real;
 4    NOM,NOMR,PRE,RUE,LOC,TEL1,TEL2 : string;
 5  begin
 6    -- afficher les informations sur le representant NUMREP
 7    select * into NUM, NOM, PRE, RUE, LOC, CODE, QUOT, TEL1, TEL2
 8      from REPRESENTANT where NREP = NUMREP;
 9    writeln(NOM,' ',PRE,' (',NUM,') Tél. : ',TEL1);
10    writeln('Adresse : ',RUE,' ',CODE,' ',LOC);
11    writeln('Quota : ',QUOT);
12    writeln('Autre tél. : ',TEL2);
13    -- afficher les informations sur les clients du representant NUMREP
14    writeln('Les clients du représentant sont :');
15    declare C cursor for select * from CLIENT where NREP = NUMREP;
16    open C;
17    fetch C into NUM,NOM,PRE,RUE,LOC,CODE,TEL1,NR,NOMR;
18    while SQLCODE = 0 do
19      writeln('- ',NOM,' ',PRE,' (',NUM,') Tél. : ',TEL1);
20      writeln('  Adresse : ',RUE,' ',CODE,' ',LOC);
21      writeln('  Région : ',NOMR);
22      fetch C into NUM,NOM,PRE,RUE,LOC,CODE,TEL1,NR,NOMR;
23    endwhile;
24    close C;
25    -- calculer les montants des ventes du représentant NUMREP par région
26    writeln('Les montants des ventes par région du représentant sont : ');
27    declare V cursor for
28      select sum(V.PRIX * D.QUANTITE), C.NOM_REG
29        from CLIENT C,COMMANDE CO,DETAIL D,PRODUIT P,VENTE V
30        where C.NREP = NUMREP and C.NCLI = CO.NCLI and CO.NCOM = D.NCOM and
31              D.NPRO = P.NPRO and P.NPRO = V.NPRO and V.NOM = C.NOM_REG
32        group by C.NOM_REG;
33    open V;
34    fetch V into VEN, NOMR;
35    while SQLCODE = 0 do
36      writeln('- Pour le région ',NOMR,' : ',VEN,' Bef.');
37      TOTVEN := TOTVEN + VEN;
38      fetch V into VEN, NOMR;
39    endwhile;
40    close V;
41    -- calculer le montant total des ventes du représentant NUMREP
42    writeln('Le montant total des ventes du représentant est de ',TOTVEN,' Bef.');
43  end
```

1. En anglais, l'expression "embedded SQL" est régulièrement utilisée.

b) Calcul du nouveau quota de vente d'un représentant

La procédure *CALCULNOUVQUOTAREPRESENTANT* fixe le nouveau quota de vente d'un représentant donné. Ce quota est fonction des ventes du représentant dans chaque région où il visite un client ainsi que du budget publicitaire, des facteurs publicitaires et économiques de ces régions. Le facteur économique dépend des taux de croissance et de chômage d'une région. Le facteur publicitaire d'une région est calculé sur base du budget publicitaire, des taux de croissance et des taux de chômage, alors que le budget publicitaire d'une région est défini comme la multiplication du facteur économique correspondant et de la somme des quotas des réprésentants de cette région.

```
Procedure CALCULNOUVQUOTAREPRESENTANT(NUMREP : integer)
 1  declaration
 2    FACECO, CROIS, CHOM, BASE, FACPUB : real;
 3    QUOT, ECO, PUBREG, MONTVEN, QUOTREG, NOUVQUOT : integer;
 4    NOMREG : string;
 5  begin
 6    NOUVQUOT := 0;
 7    declare R cursor for select R.NOM, R.CROISSANCE, R.CHOMAGE
 8                          from REGION R, CLIENT C
 9                          where NUMREP = C.NREP and C.NOM_REG = R.NOM
10                          group by R.NOM, R.CROISSANCE, R.CHOMAGE;
11    open R;
12    fetch R into NOMREG, CROIS, CHOM;
13    while SQLCODE = 0 do
14      -- calcul du facteur economique de la region
15      if CROIS >= 0,04 and CHOM < 0,076 then ECO := 2;
16      if 0,02 <= CROIS < 0,04 and 0,076 <= CHOM < 0,082 then ECO := 1;
17      if CROIS < 0,02 and 0,082 <= CHOM then ECO := 0;
18      if ECO = 2 then FACECO := CROIS;
19      if ECO = 1 then FACECO := CROIS / 3;
20      if ECO = 0 then FACECO := CROIS / 5;
21      -- calcul de la somme des quotas des representants d'une region
22      select sum(R.QUOTA) into QUOT
23        from CLIENT C, REPRESENTANT R
24        where NOMREG = C.NOM_REG and C.NREP = R.NREP;
25      -- calcul du budget publicitaire d'une région
26      PUBREG := FACECO * QUOT;
27      -- calcul de la somme des ventes du representant NUMREP dans la region NOMREG
28      select sum(V.PRIX * D.QUANTITE) into MONTVEN
29        from CLIENT C, COMMANDE CO, DETAIL D, PRODUIT P, VENTE V
30        where C.NREP = NUMREP and C.NOM_REG = NOMREG and C.NCLI = CO.NCLI and
31              CO.NCOM = D.NCOM and D.NPRO = P.NPRO and P.NPRO = V.NPRO and
32              V.NOM = NOMREG;
33      if MONTVEN > (1,15 * QUOT) then BASE := QUOT+(MONTVEN-(1,15*QUOT));
34      else BASE := QUOT;
35      -- calcul du facteur publicitaire de la region NOMREG
36      if PUBREG > 7000 and ECO = 0 then FACPUB := PUBREG;
37      else if PUBREG < 5000 and ECO = 2 then FACPUB := -0,015;
38           else FACPUB := 0;
39      NOUVQUOT := NOUVQUOT + ((1 + ((FACPUB + FACECO) / 2)) * BASE);
40      fetch R into NOMREG, CROIS, CHOM;
41    endwhile;
42    close R;
43    update REPRESENTANT set QUOTA = NOUVQUOT where NREP = NUMREP;
44    writeln('Le nouveau quota du représentant ',NUMREP,' est de ',NOUVQUOT,' Bef.');
45  end
```

c) Insertion d'un nouveau client

La procédure *INSERTIONCLIENT* insère dans la base de données toutes les informations recueillies sur un nouveau client. Elle attache également le nouveau client à un représentant et une région. Si le représentant ou la région n'existe pas, ils sont également créés.

```
Procedure INSERTIONCLIENT()
 1  declaration
 2    CROIS,CHOM : real;
 3    NUM,CODE,NUMR,CODER,NR : integer;
 4    NOM,PRE,RUE,LOC,NOMR,PRER,RUER,LOCR,NOMREG,NREG,TEL,TEL1,TEL2 : string;
 5  begin
 6    -- insertion d'un nouveau client
 7    writeln('Insertion d'un nouveau client :');
 8    writeln('  Numéro : '); read(NUM);
 9    writeln('  Nom : '); read(NOM);
10    writeln('  Prénom : '); read(PRE);
11    writeln('  Rue : '); read(RUE);
12    writeln('  Localité : '); read(LOC);
13    writeln('  Code postal : '); read(CODE);
14    writeln('  Téléphone : '); read(TEL);
15    writeln('  Num. rep. : '); read(NUMR);
16    select NREP into NR where NREP = NUMR;
17    if NR = 0 then
18      begin
19        -- insertion d'un nouveau representant
20        writeln('Insertion du représentant ',NUMR,' :');
21        writeln('  Nom : '); read(NOMR);
22        writeln('  Prénom : '); read(PRER);
23        writeln('  Rue : '); read(RUER);
24        writeln('  Localité : '); read(LOCR);
25        writeln('  Code postal : '); read(CODER);
26        writeln('  Quota : '); read(QUOT);
27        writeln('  Téléphone (1) : '); read(TEL1);
28        writeln('  Téléphone (2) : '); read(TEL2);
29        insert into REPRESENTANT values
30                    (NUMR,NOMR,PRER,RUER,LOCR,CODER,QUOT,TEL1,TEL2);
31      end;
32    writeln('  Nom rég. : '); read(NOMREG);
33    select NOM into NREG where NOM = NOMREG;
34    if NREG = '' then
35      begin
36        -- insertion d'une nouvelle region
37        writeln('Insertion de la nouvelle région ',NOMREG,' :');
38        writeln('  Croissance : '); read(CROIS);
39        writeln('  Chômage : '); read(CHOM);
40        insert into REGION values (NOMREG,CROIS,CHOM);
41      end;
42    insert into CLIENT values (NUM,NOM,PRE,RUE,LOC,CODE,TEL,NUMR,NOMREG);
43  end
```

d) Ajustement des prix des nouveaux produits

La procédure *AJUSTERPRIXNOUVPRODUIT* ajuste, en fin d'exercice comptable, les prix de lancement des nouveaux produits vendus (c'est-à-dire ceux dont l'année de lancement est identique à l'année de l'exercice comptable considéré) dans une région en fonction de la quantité des autres produits vendus dans cette région. Si la quantité moyenne vendue des anciens produits est supérieure à la quantité vendue du nouveau produit, le prix de ce dernier est diminué de dix pour-cent.

```
Procedure AJUSTERPRIXNOUVPRODUIT(ANNEECOURANTE : integer)
 1  declaration
```

```
 2    SOM,SOMAUT,SOMNUMPRO,MOYAUT : real;
 3    NUMPRO,NP,I : integer;
 4    NOMREG : string;
 5    D,V : cursor;
 6  begin
 7    NOUVQUOT := 0;
 8    declare V cursor for select V.NPRO,V.NOMREG from VENTE V
 9                         where ANDEP = ANNEECOURANTE;
10    open V;
11    fetch V into NOMREG,NUMPRO;
12    while SQLCODE = 0 do
13      SOMAUT := 0;
14      I := 0;
15      -- calculer la quantité vendue des produits dans la région NOMREG
16      declare D cursor for select D.NPRO,sum(D.QUANTITE)
17                           from REGION R,CLIENT C,COMMANDE CO,DETAIL D
18                           where R.NOM = NOMREG and C.NOM_REG = R.NOM and
19                                 CO.NCLI = C.NCLI and D.NCOM = CO.NCOM
20                           group by D.NPRO;
21      open D;
22      fetch D into NP,SOM;
23      while SQLCODE = 0 do
24        -- calculer la quantité vendue du nouveau produit NUMPRO dans la région NOMREG
25        if NP = NUMPRO then SOMNUMPRO = SOM;
26        else
27          begin
28            -- calculer la quantité vendue des autres produits dans la région NOMREG
29            I := I + 1;
30            SOMAUT := SOMAUT + SOM;
31          end;
32        fetch D into NP,SOM;
33      endwhile;
34      close D;
35      -- calculer la moyenne des quantités vendues des produits différents de NUMPRO
36      -- dans la région NOMREG
37      MOYAUT := SOMAUT / I;
38      -- Mise à jour du prix du produit NUMPRO dans la région NOMREG
39      if MOYAUT > SOMNUMPRO then
40        update VENTE set PRIX = PRIX * 0,9 where NPRO = NUMPRO and NOM_REG = NOMREG;
41      fetch V into NOMREG,NUMPRO;
42    endwhile;
43    close V;
44  end
```

8.2 Modification des spécifications conceptuelles

Dans cette étude de cas, seule la première stratégie va être prise en compte. Les autres stratégies sont basées sur les mêmes principes de base, il n'est donc pas nécessaire de les illustrer de manière détaillée. La première stratégie consiste à modifier le schéma conceptuel et à propager les modifications vers les niveaux en aval. Cela suppose que tous les niveaux sont documentés comme cela a été fait dans le point 8.1. Si on applique la méthodologie définie au point 7.2.2, la stratégie se décompose en quatre étapes (points 8.2.3 à 8.2.6). Mais, avant d'analyser chaque étape, le point 8.2.1 présente l'évolution des besoins qui ont engendré les modifications des spécifications conceptuelles et le point (8.2.2) donne l'historique de l'évolution qui permet de visualiser graphiquement les quatre étapes de la méthode.

8.2.1 Evolution des besoins

Suite à des changements environnementaux, les besoins auxquels le schéma conceptuel satisfaisait sont modifiés. Cinq changements ont été déterminés :

1. Suite au développement croissant des technologies de télécommunication et, plus particulièrement, dans le domaine des téléphones portables, on constate une multiplication des points de contact pour les clients. Il n'est plus rare à notre époque qu'une personne dispose de quatre numéros de téléphone (téléphones fixes ou portables, privés ou professionnels).
2. Le nombre de clients de la société ne cesse d'augmenter ce qui rend la découpe en région beaucoup trop vaste et inadaptée. Les régions devront à l'avenir se décomposer en plusieurs zones.
3. Vu la spécialisation des représentants dans des gammes de produits, il est plus réaliste qu'un client soit visité par plusieurs représentants pour des produits bien distincts. Cette modification répond à un besoin de la société d'offrir un meilleur service aux clients et de mieux répondre aux exigences du marché.
4. L'application doit être modifiée pour assurer le passage à l'euro officiellement prévu en janvier 2002. Cela implique une conversion des prix en franc belge vers l'euro (1 euro = 40,3399 BEF).
5. La diversification des produits vendus entraîne une multiplication des fournisseurs dont il faut garder une trace pour mieux gérer les stocks.

8.2.2 Historique de l'évolution

La figure 8.5 présente l'historique de l'évolution de la partie base de données de l'application *QUOTA VENTES*.

Il contient les quatre processus de la propagation des modifications des spécifications conceptuelles :

1. La modification du schéma conceptuel reçoit en entrée le schéma conceptuel *QUOTA VENTES/Conceptuel* et fournit comme produit résultat le nouveau schéma conceptuel *QUOTA VENTES/Conceptuel-1* et l'historique d'évolution conceptuelle matérialisé dans le journal *ec1.log*.
2. La nouvelle conception logique prend en entrée le nouveau schéma conceptuel et l'ancien historique de conception logique (journal *cl0.log*) et fournit en sortie les produits résultats que sont le nouveau schéma logique *QUOTA VENTES/Logique-1* et le nouvel historique de conception logique matérialisé dans le journal *cl1.log*.
3. La nouvelle conception physique prend en entrée le nouveau schéma logique et l'ancien historique de conception physique (journal *cp0.log*) et fournit en sortie le nouveau schéma physi-

que *QUOTA VENTES/Physique-1* et le nouvel historique de conception physique matérialisé dans le journal *cp1.log*.

4. La génération donne à partir du nouveau schéma physique le script de conversion des structures de données (*Script SQL*) et les patrons (*Patterns*) de localisation des instructions à modifier dans les programmes de l'application.

Les points 8.2.3 à 8.2.6 approfondissent chacun des processus.

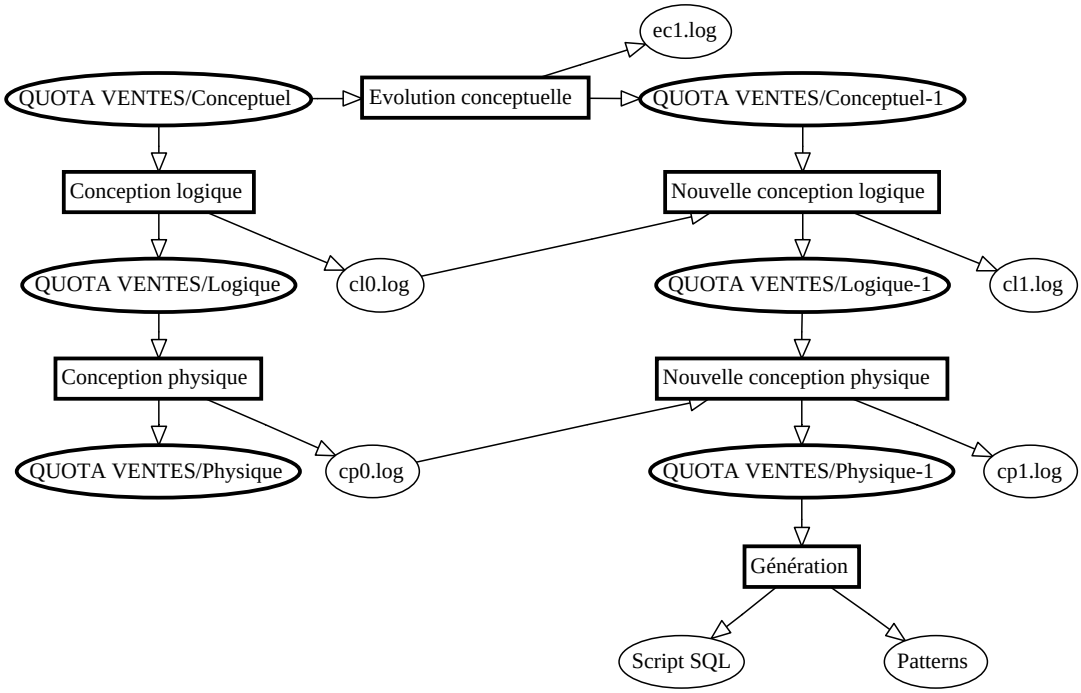


Figure 8.5 - Historique de l'évolution conceptuelle de la base de données *QUOTA VENTES*.

8.2.3 Etape 1 : Modification du schéma conceptuel

Suite à l'évolution des besoins, l'ingénieur de maintenance va modifier le schéma conceptuel *QUOTA VENTES/Conceptuel*. Pour réaliser cela, la méthode fait une copie du schéma nommée *QUOTA VENTES/Conceptuel-1*. En utilisant la boîte à outils des modifications conceptuelles, l'ingénieur modifie la copie. Cinq modifications correspondant aux changements du point 8.2.1 sont réalisées sur *QUOTA VENTES/Conceptuel-1* :

1. La cardinalité maximum de l'attribut *Telephone* du type d'entités *CLIENT* devient 5.
2. Pour tenir compte de la nouvelle découpe en zones, on décide de renommer le type d'entités *REGION* en *ZONE* car une zone a les mêmes propriétés et joue les mêmes rôles qu'une région. Simplement, il y aura dans la base plus de zones que de régions. Le renommage permet de rendre les spécifications conformes à la réalité du terrain. Afin de garder une trace de la région à laquelle appartient une zone, un attribut *Nomregion* est ajouté à la table renommée. Lors de la conversion des données initiales, l'attribut *Nomregion* prendra les mêmes valeurs que l'attribut *Nom*.
3. Puisqu'un client peut être visité par plusieurs représentants, le type d'associations *visite* entre *CLIENT* et *REPRESENTANT* devient plusieurs-à-plusieurs.
4. L'attribut *Prix* (entier de longueur 8 représentant le prix de vente d'un produit dans une région) du type d'associations *VENTE* devient un numérique de longueur 10 dont 2 décimales. Dorénavant, tous les traitements mettant en œuvre ce prix devront tenir compte des deux chiffres après la virgule qui sont importants lorsqu'on calcule en euros.

5. Le type d'entités *FOURNISSEUR* est créé. Il permet de stocker les informations concernant les fournisseurs des produits vendus par la société. Il est relié au type d'entités *PRODUIT* via le type d'associations un-à-plusieurs *fournit*.

Le nouveau schéma conceptuel *QUOTA VENTES/Conceptuel-1* (figure 8.6) englobe les modifications apportées au schéma original. La méthode a enregistré dans un historique les transformations de modifications. Il est matérialisé sous la forme d'un journal *ec1.log* dont le détail se trouve dans l'annexe E.2. Le processus d'évolution conceptuelle retourne comme résultat les produits *QUOTA VENTES/Conceptuel-1* et *ec1.log*. Les signatures des transformations de l'historique d'évolution conceptuelle sont :

```
H_EC1 = <
  Telephone<-modifier-Att(CLIENT,Telephone,0-5)
  ZONE<-modifier-TE(REGION,"ZONE")
  Nomregion<-creer-Att(ZONE,"Nomregion",char,30,1-1)
  Visite<-modifier-Role(CLIENT,Visite,0-N)
  Prix<-modifier-Att(VENTE,Prix,10,2)
  FOURNISSEUR<-creer-TE("FOURNISSEUR")
  Nfour<-creer-Att(FOURNISSEUR,"Nfour",numeric,5,1-1)
  Nom<-creer-Att(FOURNISSEUR,"Nom",char,50,1-1)
  Adresse<-creer-Att(FOURNISSEUR,"Adresse",char,70,1-1)
  Telephone<-creer-Att(FOURNISSEUR,"Telephone",char,12,1-1)
  Jour livraison<-creer-Att(FOURNISSEUR,"Jour livraison",char,10,0-1)
  IDFOURNISSEUR<-creer-Id(FOURNISSEUR,"IDFOURNISSEUR",{Nfour},primaire)
  fournit<-creer-TA("fournit")
  PRODUIT.fournit<-creer-Role(PRODUIT,fournit,1-1)
  FOURNISSEUR.fournit<-creer-Role(FOURNISSEUR,fournit,0-N)
>
```

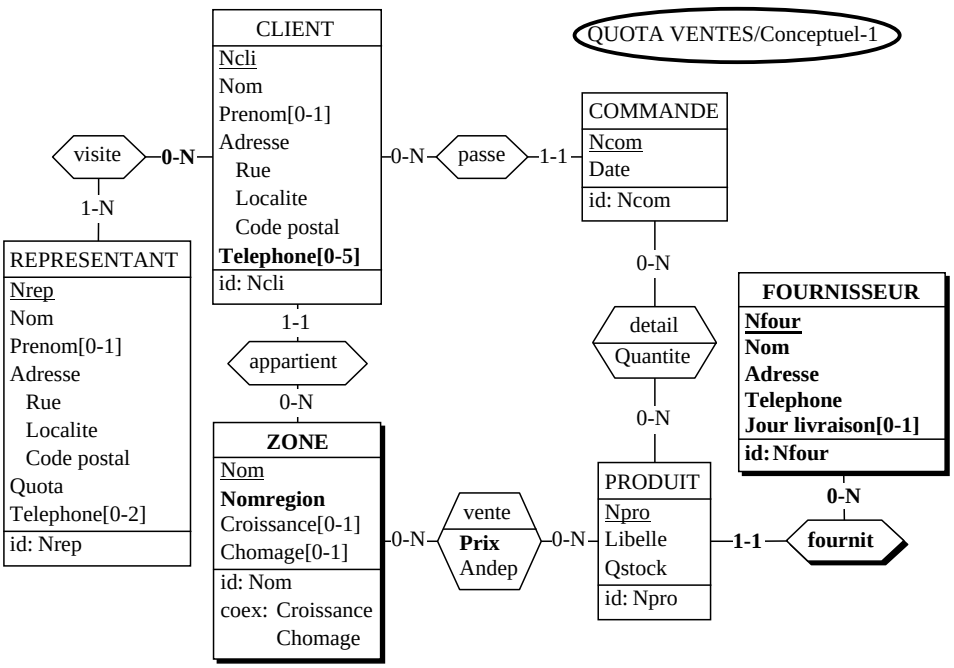


Figure 8.6 - Nouveau schéma conceptuel : les objets modifiés ou créés par rapport au schéma conceptuel original sont en gras.

8.2.4 Etape 2 : Propagation des modifications en aval vers le niveau logique

Le processus de conception logique demande une copie du schéma conceptuel *QUOTA VENTES/Conceptuel-1* (nommée *QUOTA VENTES/Logique-1*). L'historique de conception logique original enregistré dans le journal *cl0.log* est rejoué sur ce schéma en déclenchant l'enregistrement des transformations. Une première version incomplète du nouveau schéma logique relationnel est ainsi obtenue.

Les nouveaux composants de *QUOTA VENTES/Conceptuel-1* (provenant de la création de nouveaux objets ou de la modification d'objets de *QUOTA VENTES/Conceptuel*) sont ensuite transformés spécifiquement grâce à la boîte à outils de conception logique et selon les heuristiques de transformation en vigueur dans l'entreprise. Ces transformations sont enregistrées dans l'historique du nouveau schéma logique à la suite de celles enregistrées lors de l'exécution des transformations de *cl0.log*. Finalement, l'historique contenant toutes les transformations exécutées pour obtenir *QUOTA VENTES/Logique-1* (figure 8.7) à partir de *QUOTA VENTES/Conceptuel-1* est sauvegardé dans le journal *cl1.log*. L'historique H_{CL1} contient les signatures des transformations enregistrées dans *cl1.log* :

```
H_CL1 = <
  (REPRESENTANT,{Adr_Rue,Adr_Localite,Adr_Code postal})<-
                                              desagreger-Att(REPRESENTANT,Adresse)
  (CLIENT,{Adr_Rue,Adr_Localite,Adr_Code postal})<-desagreger-Att(CLIENT,Adresse)
  Adr_cp<-modifier-Att(REPRESENTANT,Adr_Code postal,"Adr_cp")
  Adr_cp<-modifier-Att(CLIENT,Adr_Code postal,"Adr_cp")
  (REPRESENTANT,{Telephone1,Telephone2})<-Attmult-en-Serie-Att(REPRESENTANT,Telephone)
  (DETAIL,{cd,pd})<-TA-en-TE(detail)
  (VENTE,{rv,pv})<-TA-en-TE(vente)
  ~~(CLIENT,{Nrep})<-TA-en-FK(visite)~~
  ~~(CLIENT,{Nomreg})<-TA-en-FK(appartient)~~
  ~~(VENTE,{Nomreg})<-TA-en-FK(rv)~~
  (VENTE,{Npro})<-TA-en-FK(pv)
  (COMMANDE,{Ncli})<-TA-en-FK(passe)
  (DETAIL,{Ncom})<-TA-en-FK(cd)
  (DETAIL,{Npro})<-TA-en-FK(pd)
  (TELEPHONE,ct)<-Att-en-TE(CLIENT.Telephone)
  (VISITE,{vr,cv})<-TA-en-TE(visite)
  (VISITE,{Nrep})<-TA-en-FK(vr)
  (VISITE,{Ncli})<-TA-en-FK(cv)
  (TELEPHONE,{Ncli})<-TA-en-FK(ct)
  (CLIENT,{Nomreg})<-TA-en-FK(appartient)
  (VENTE,{Nomreg})<-TA-en-FK(rv)
  (PRODUIT,{Nfour})<-TA-en-FK(fournit)
  Jour_livraison<-modifier-Att(FOURNISSEUR,Jour livraison,"Jour_livraison")
>
```

Les signatures barrées sont celles des transformations qui n'ont pas été rejouées sur la copie du nouveau schéma conceptuel. Elles disparaissent dans *cl1.log*. Ce sont les transformations en clés étrangères des types d'associations *visite* entre *REPRESENTANT* et *CLIENT*, *appartient* entre *CLIENT* et *REGION* et *rv* entre *VENTE* et *REGION* (comme le repérage des objets se base sur les noms, le type d'entités *REGION* renommé *ZONE* n'est pas retrouvé dans le nouveau schéma conceptuel).

Les signatures en gras représentent les nouvelles transformations engendrées par les modifications. L'attribut *Telephone* de *CLIENT* est transformé en un type d'entités, le type d'associations v*isite* est transformé en un type d'entités et tous les types d'associations fonctionnels restant sont transformés en clés étrangères. La colonne *Jour livraison* de la table *FOURNISSEUR* est renommée *Jour_livraison*.

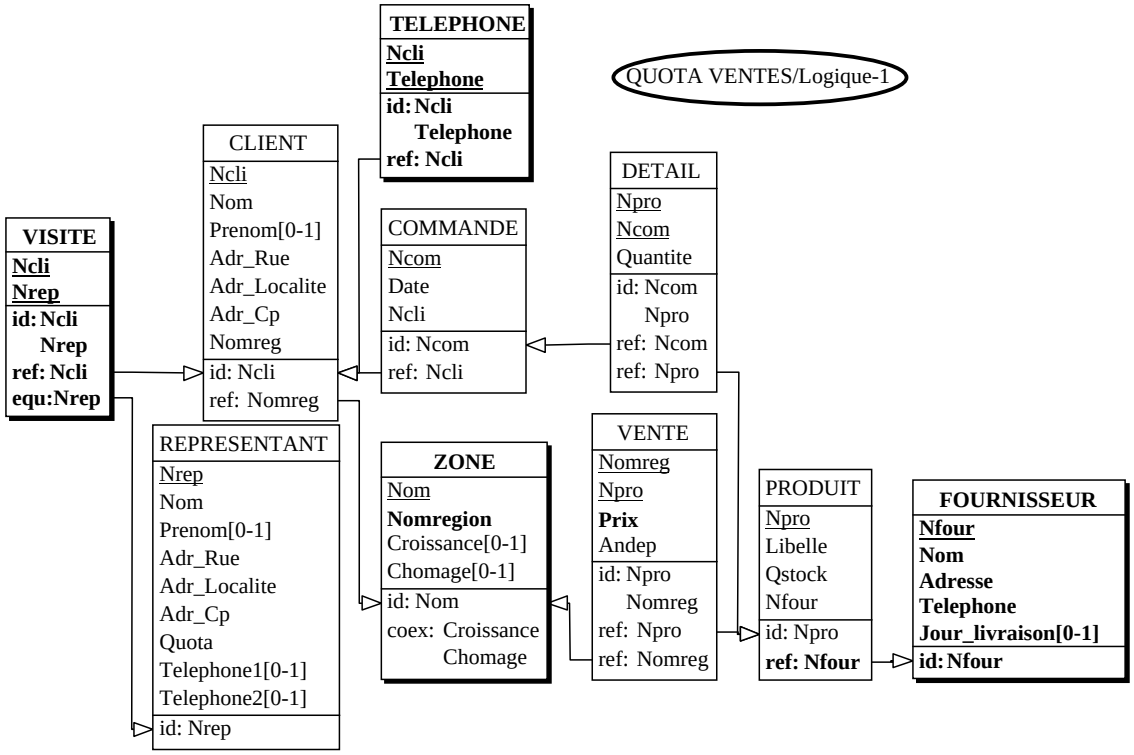


Figure 8.7 - Nouveau schéma logique relationnel : les objets modifiés ou créés par rapport au schéma logique originale sont en gras.

8.2.5 Etape 3 : Propagation des modifications en aval vers le niveau physique

Le processus de conception physique réalise une copie de *QUOTA VENTES\Logique-1* (nommée *QUOTA VENTES\Physique-1*) sur laquelle il rejoue l'historique de conception physique original contenu dans le journal *cp0.log*. L'enregistrement des transformations est déclenché. On obtient ainsi un nouveau schéma physique incomplet.

Les composants du nouveau schéma physique (provenant de la création de nouveaux objets ou de la modification d'objets du schéma conceptuel) sont alors transformés spécifiquement en utilisant la boîte à outils de conception physique. Ces transformations (modifications relatives aux index et aux espaces de stockage) sont enregistrées dans l'historique du schéma physique à la suite des transformations déjà enregistrées. L'historique qui contient toutes les transformations exécutées pour obtenir *QUOTA VENTES\Physique-1* (figure 8.8) à partir de *QUOTA VENTES\Logique-1* est sauvegardé dans le journal *cp1.log*. Les signatures des transformations enregistrées dans *cp1.log* sont :

```
H_CP1 = <
  Datecom<-modifier-Att(COMMANDE,Date,"Datecom")
  CLIENTS<-creer-Coll("CLIENTS")
  (CLIENTS,{CLIENT,REPRESENTANT,REGION})<-
                              modifier-Coll(CLIENTS,{CLIENT,REPRESENTANT,REGION})
  PRODUITS<-creer-Coll("PRODUITS")
  (PRODUITS,{COMMANDE,DETAIL,PRODUIT,VENTE})<-
                              modifier-Coll(PRODUITS,{COMMANDE,DETAIL,PRODUIT,VENTE})
  IDCLIENT<-creer-Index(CLIENT,"IDCLIENT",{Ncli})
  IDREPRESENTANT<-creer-Index(REPRESENTANT,"IDREPRESENTANT",{Nrep})
  FKvisite<-creer-Index(CLIENT,"FKvisite",{Nrep})
  FKappartient<-creer-Index(CLIENT,"FKappartient",{Nomreg})
```

```
  ~~IDREGION<-creer-Index(REGION,"IDREGION",{Nom})~~
  IDCOMMANDE<-creer-Index(COMMANDE,"IDCOMMANDE",{Ncom})
  FKpasse<-creer-Index(COMMANDE,"FKpasse",{Ncli})
  IDVENTE<-creer-Index(VENTE,"IDVENTE",{Npro,Nomreg})
  FKrv<-creer-Index(VENTE,"FKrv",{Nomreg})
  IDPRODUIT<-creer-Index(PRODUIT,"IDPRODUIT",{Npro})
  IDDETAIL<-creer-Index(DETAIL,"IDDETAIL",{Ncom,Npro})
  FKpd<-creer-Index(DETAIL,"FKpd",{Npro})
  (CLIENTS,{CLIENT,REPRESENTANT,ZONE,TELEPHONE,VISITE})<-
                    modifier-Coll(CLIENTS,{CLIENT,REPRESENTANT,ZONE,TELEPHONE,VISITE})
  (PRODUITS,{COMMANDE,DETAIL,PRODUIT,VENTE,FOURNISSEUR})<-
                                                  modifier-Coll(PRODUITS,{FOURNISSEUR})
  IDVISITE<-creer-Index(VISITE,"IDVISITE",{Ncli,Nrep})
  FKvr<-creer-Index(VISITE,"FKvr",{Nrep})
  IDTELEPHONE<-creer-Index(TELEPHONE,"IDTELEPHONE",{Ncli,Telephone})
  IDREGION<-creer-Index(ZONE,"IDREGION",{Nom})
  FKfournit<-creer-Index(PRODUIT,"FKfournit",{Nfour})
  IDFOURNISSEUR<-creer-Index(FOURNISSEUR,"IDFOURNISSEUR",{Nfour})
>
```

Les signatures barrées sont celles des transformations de *cp0.log* qui n'ont pas été rejouées sur la copie du nouveau schéma logique. Elles disparaissent de *cp1.log*. Il s'agit des transformations qui ajoutent un index sur les groupes *FKvisite* (la clé étrangère de *CLIENT* vers *REPRESENTANT* a disparu) et *IDREGION* (*REGION* est renommé *ZONE*).

Les signatures en gras représentent les transformations engendrées par les modifications. Les tables *ZONE*, *TELEPHONE* et *VISITE* sont ajoutées à l'espace de stockage *CLIENTS*. La table *FOURNISSEUR* est ajoutée à l'espace de stockage *PRODUITS*. Les groupes *IDVISITE*, *FKvr*, *IDTELEPHONE*, *IDREGION*, *FKfournit* et *IDFOURNISSEUR* deviennent des index.

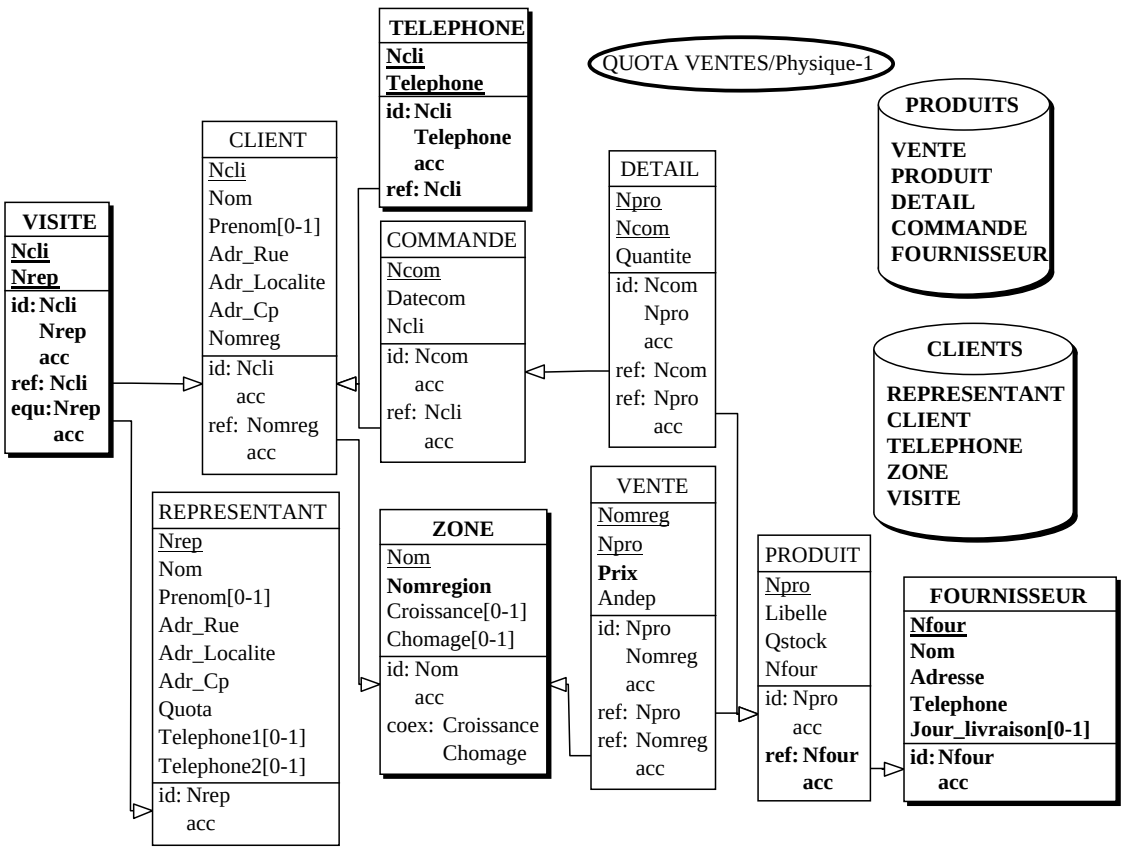


Figure 8.8 - Nouveau schéma physique relationnel : les objets modifiés ou créés sont en gras.

8.2.6 Etape 4 : Propagation des modifications vers les données et les traitements

8.2.6.1 Conversion des structures de données et des instances

Le processus de conversion des données analyse les journaux obtenus dans les étapes précédentes pour générer un script de conversion des structures de données et des données. Il est décomposé en quatre phases (voir point 7.3) :

- l'analyse des journaux et la construction des listes de modifications,
- l'intégration des listes de modifications,
- la restructuration de la liste des modifications physiques,
- la génération du script de conversion.

Bien que les scripts soient générés automatiquement par le module d'évolution de l'atelier DB-Main, chaque phase va être analysée afin de démonter les rouages du générateur de scripts.

a) Analyse des journaux

Les historiques stockés dans les journaux (*cl0.log*, *cp0.log*, *ec1.log*, *cl1.log* et *cp1.log*) sont analysés pour en extraire les historiques simplifiés. Ces historiques sont matérialisés sous la forme de listes des modifications. Le tableau 8.1 donne la liste (L_{EC1}) des modifications conceptuelles créées à partir du journal *ec1.log*. H_{EC1} étant minimal (pas d'essai ou d'erreur), la liste contient le même nombre de modifications que H_{EC1}. La première colonne précise le type de transformation, la deuxième colonne donne le ou les objets cibles (appartenant au schéma *QUOTA VENTES\Conceptuel*) de la transformation et la troisième colonne donne le ou les objets origines (appartenant au schéma *QUOTA VENTES\Conceptuel-1*) de la transformation.

L_{EC1}		
Type de transformation	**QUOTA VENTES\Conceptuel-1**	**QUOTA VENTES\Conceptuel**
MODIFATT	CLIENT.Telephone	CLIENT.Telephone
MODIFENT	ZONE	REGION
CREERATT	ZONE.Nomregion	
MODIFROL	visite.CLIENT	visite.CLIENT
MODIFATT	VENTE.Prix	VENTE.Prix
CREERENT	FOURNISSEUR	
CREERATT	FOURNISSEUR.Nfour	
CREERATT	FOURNISSEUR.Nom	
CREERATT	FOURNISSEUR.Adresse	
CREERATT	FOURNISSEUR.Telephone	
CREERATT	FOURNISSEUR.Jour Livraison	
CREERGR	IDFOURNISSEUR	
CREERREL	fournit	
CREERROL	fournit.PRODUIT	
CREERROL	fournit.FOURNISSEUR	

Tableau 8.1 - Liste des modifications construite à partir de H_{EC1}.

Le tableau 8.2 donne la liste des transformations (L_{C0}) de conception créée à partir des journaux *cl0.log* et *cp0.log*. La deuxième colonne contient le ou les objets cibles appartenant au schéma *QUOTA VENTES\Conceptuel* et la troisième colonne contient le ou les objets origines appartenant au schéma *QUOTA VENTES\Physique*. Pour simplifier, les listes L_{CL0} et L_{CP0} sont concaténées dans la liste L_{C0}. Certaines simplifications sont réalisées :

a) Les modifications des espaces de stockage *CLIENTS* et *PRODUITS* sont enlevées car ces espaces sont déjà créés dans L_{C0}. Par exemple, la transformation `[CREERCOL,CLIENTS,/]` est suffisante pour retrouver dans le schéma *QUOTA VENTES\Physique* toutes les informations concernant l'espace *CLIENTS* et les tables qui lui appartiennent.

b) Les modifications (renommage) relatives aux attributs *REPRESENTANT.Adr_Code postal* et *CLIENT.Adr_Code postal* sont intégrées dans la désagrégation de leur attribut parent. Les deux désagrégations ont, parmi les objets cibles, les attributs renommés *REPRESENTANT.Adr_Cp* et *CLIENT.Adr_Cp*. Les deux modifications sont enlevées de L_{C0}.

L_{C0}			
	Type de transformation	**QUOTA VENTES\Physique**	**QUOTA VENTES\Conceptuel**
L_{CL0}	DESAGREGER-ATT	{Adr_Rue,Adr_Localite,Adr_cp}	REPRESENTANT.Adresse
	DESAGREGER-ATT	{Adr_Rue,Adr_Localite,Adr_cp}	CLIENT.Adresse
	ATTMULT-EN-SERIE-ATT	{Telephone1,telephone2}	REPRESENTANT.Telephone
	TA-EN-TE	DETAIL	detail
	TA-EN-TE	VENTE	vente
	TA-EN-FK	CLIENT.Nrep	visite
	TA-EN-FK	CLIENT.Nomreg	appartient
	TA-EN-FK	VENTE.Nomreg	rv
	TA-EN-FK	VENTE.Npro	pv
	TA-EN-FK	COMMANDE.Ncli	passe
	TA-EN-FK	DETAIL.Ncom	cd
	TA-EN-FK	DETAIL.Npro	pd
L_{CP0}	MODIFATT	COMMANDE.Datecom	COMMANDE.Date
	CREERCOL	CLIENTS	
	CREERCOL	PRODUITS	
	MODIFGR	IDCLIENT	IDCLIENT
	MODIFGR	IDREPRESENTANT	IDREPRESENTANT
	MODIFGR	FKvisite	FKvisite
	MODIFGR	FKappartient	FKappartient
	MODIFGR	IDREGION	IDREGION
	MODIFGR	IDCOMMANDE	IDCOMMANDE
	MODIFGR	FKpasse	FKpasse
	MODIFGR	IDVENTE	IDVENTE
	MODIFGR	FKrv	FKrv
	MODIFGR	IDPRODUIT	IDPRODUIT
	MODIFGR	IDDETAIL	IDDETAIL
	MODIFGR	FKpd	FKpd

Tableau 8.2 - Liste des modifications L_{C0} construite à partir de H_{CL0} et H_{CP0}.

Le tableau 8.3 donne la liste des transformations (L_{C1}) de conception créée à partir des journaux *cl1.log* et *cp1.log*. La deuxième colonne contient le ou les objets cibles appartenant au schéma *QUOTA VENTES\Conceptuel-1* et la troisième colonne contient le ou les objets origines appartenant au schéma *QUOTA VENTES\Physique-1*. Les listes L_{CL1} et L_{CP1} sont construites à partir de H_{CL1} et H_{CP1}. Elles sont concaténées dans la liste L_{C1}. Les simplifications réalisées dans L_{C0} sont également appliquées à L_{C1}.

L_{C1}			
	Type de transformation	**QUOTA VENTES\Physique-1**	**QUOTA VENTES\Conceptuel-1**
L_{CL1}	DESAGREGER-ATT	{Adr_Rue,Adr_Localite,Adr_cp}	REPRESENTANT.Adresse
	DESAGREGER-ATT	{Adr_Rue,Adr_Localite,Adr_cp}	CLIENT.Adresse
	ATTMULT-EN-SERIE-ATT	{Telephone1,telephone2}	REPRESENTANT.Telephone
	TA-EN-TE	DETAIL	detail
	TA-EN-TE	VENTE	vente
	TA-EN-FK	VENTE.Npro	pv
	TA-EN-FK	COMMANDE.Ncli	passe
	TA-EN-FK	DETAIL.Ncom	cd
	TA-EN-FK	DETAIL.Npro	pd
	ATT-EN-TE	{TELEPHONE,ct}	CLIENT.Telephone
	TA-EN-TE	VISITE	visite
	TA-EN-FK	VISITE.Nrep	vr
	TA-EN-FK	VISITE.Ncli	cv
	TA-EN-FK	TELEPHONE.Ncli	ct
	TA-EN-FK	CLIENT.Nomreg	appartient
	TA-EN-FK	VENTE.Nomreg	rv
	TA-EN-FK	PRODUIT.Nfour	fournit
	MODIFATT	Jour_livraison	Jour livraison
L_{CP1}	MODIFATT	COMMANDE.Datecom	COMMANDE.Date
	CREERCOL	CLIENTS	
	CREERCOL	PRODUITS	
	MODIFGR	IDCLIENT	IDCLIENT
	MODIFGR	IDREPRESENTANT	IDREPRESENTANT
	MODIFGR	FKappartient	FKappartient
	MODIFGR	IDCOMMANDE	IDCOMMANDE
	MODIFGR	FKpasse	FKpasse
	MODIFGR	IDVENTE	IDVENTE
	MODIFGR	FKrv	FKrv
	MODIFGR	IDPRODUIT	IDPRODUIT
	MODIFGR	IDDETAIL	IDDETAIL
	MODIFGR	FKpd	FKpd
	MODIFGR	IDVISITE	IDVISITE
	MODIFGR	FKvr	FKvr
	MODIFGR	IDTELEPHONE	IDTELEPHONE
	MODIFGR	IDREGION	IDREGION
	MODIFGR	FKfournit	FKfournit
	MODIFGR	IDFOURNISSEUR	IDFOURNISSEUR

Tableau 8.3 - Liste des modifications L_{C1} construite à partir de H_{CL1} et H_{CP1}.

Les listes de modifications L_{EC1}, L_{C0} et L_{C1} ont été construites en simplifiant les historiques H_{EC1}, H_{CL0}, H_{CP0}, H_{CL1} et H_{CP1}. Elles constituent les bases de la deuxième phase du processus de conversion.

b) Intégration des listes des modifications

L'objectif de cette phase est de traduire la liste des modifications conceptuelles L_{EC1} en une liste de modifications physiques L_{EP1}. Le tableau 8.4 présente pour chaque transformation T de L_{EC1} les transformations de L_{C0} impliquant le ou les objets origines de T et les transformations de L_{C1} impliquant le ou les objets cibles de T. On obtient les anciennes et les nouvelles structures physi-

ques représentant l'objet modifié au niveau conceptuel. Il faut traduire le changement de structures dans L_{EP1} de la manière suivante :

1. Augmentation de la cardinalité maximum de l'attribut *CLIENT.Telephone* :
   - Aucune transformation ne concerne *Telephone* dans L_{C0}.
   - Dans L_{C1}, l'attribut *Telephone* est transformé en un type d'entités *TELEPHONE* et le type d'associations *ct* est transformé en une clé étrangère.
   - Il faut ajouter, dans L_{EP1}, la transformation de la colonne *CLIENT.Telephone* en une table *TELEPHONE*.
2. Augmentation de la cardinalité maximum du rôle *visite.CLIENT* :
   - Dans L_{C0}, le type d'association *visite* est transformé en une clé étrangère.
   - Dans L_{C1}, le type d'association *visite* est transformé en un type d'entités *VISITE* et les types d'associations *vr* et *cv* sont transformés en clés étrangères.
   - Il faut ajouter, dans L_{EP1}, la transformation de la colonne de référence *CLIENT.Nrep* en une table *VISITE*.
3. Création de l'attribut *FOURNISSEUR.Jour livraison* :
   - Aucune transformation possible dans L_{C0}.
   - Dans L_{C1}, l'attribut *Jour livraison* est renommé *Jour_livraison*.
   - Il faut ajouter, dans L_{EP1}, le création de la colonne *Jour_livraison* de la table *FOURNISSEUR*.
4. Création du groupe *IDFOURNISSEUR* dans *FOURNISSEUR* :
   - Aucune transformation possible dans L_{C0}.
   - Dans L_{C1}, le groupe *IDFOURNISSEUR* est modifié.
   - Il faut ajouter, dans L_{EP1}, le création du groupe *IDFORUNISSEUR* dans la table *FOURNISSEUR*.
5. Création du type d'associations *fournit* :
   - Aucune transformation possible dans L_{C0}.
   - Dans L_{C1}, le type d'associations *fournit* est transformé en une clé étrangère.
   - Il faut ajouter, dans L_{EP1}, la création de la colonne *Nfour* dans la table *PRODUIT* et la création de la clé étrangère vers la table *FOURNISSEUR*.
6. Création des rôles *fournit.PRODUIT* et *fournit.FOURNISSEUR* :
   - Aucune transformation possible dans L_{C0}.
   - Dans L_{C1}, le type d'associations *fournit* est transformé en une clé étrangère.
   - Il ne faut rien ajouter dans L_{EP1} car on a déjà traité la transformation de *fournit* précédement.
7. Autres transformations :
   - Aucune transformation ne les concerne dans L_{C0}.
   - Aucune transformation ne les concerne dans L_{C1}.
   - Elles sont ajoutées à L_{EP1}.

L_{C0}	L_{C1}
[MODIFATT,CLIENT.Telephone,CLIENT.Telephone]	
/	[ATT-EN-TE,{TELEPHONE,ct},CLIENT.Telephone] [TA-EN-FK,TELEPHONE.Ncli,ct]
[MODIFENT,ZONE,REGION] **[CREERATT,ZONE.Nomregion,/]**	
/	/
[MODIFROL,visite.CLIENT,visite.CLIENT]	
[TA-EN-FK,CLIENT.Nrep,visite]	[TA-EN-TE,VISITE,visite] [TA-EN-FK,VISITE.Nrep,vr] [TA-EN-FK,VISITE.Ncli,cv]
[MODIFATT,VENTE.Prix,VENTE.Prix] **[CREERENT,FOURNISSEUR,/]** **[CREERATT,FOURNISSEUR.Nfour,/]** **[CREERATT,FOURNISSEUR.Nom,/]** **[CREERATT,FOURNISSEUR.Adresse,/]** **[CREERATTT,FOURNISSEUR.Telephone,/]**	
/	/
[CREERATT,FOURNISSEUR.Jour Livraison,/]	
/	[MODIFATT,Jour_livraison,Jour livraison]
[CREERGR,IDFOURNISSEUR,/]	
/	[MODIFGR,IDFOURNISSEUR,IDFOURNISSEUR]
[CREERREL,fournit,/]	
/	[TA-EN-FK,PRODUIT.Nfour,fournit]
[CREERROL,fournit.PRODUIT,/] **[CREERROL,fournit.FOURNISSEUR,/]**	
/	/

Tableau 8.4 - Traduction des modifications conceptuelles de L_{EC1} en modifications physiques dans L_{EP1}.

Il faut encore comparer L_{C0} et L_{C1} pour trouver les transformations (autres que celles répertoriées dans le tableau 8.4) qui ne sont pas présentes dans les deux listes. On trouve la modification du groupe *FKvisite* dans L_{C0} et les modifications des groupes *IDVISITE*, *FKvr* et *IDTELEPHONE* dans L_{C1}. La destruction du groupe *FKvisite* est ajoutée à L_{EP1} car la colonne *Nrep* de *CLIENT* (appartenant à *FKvisite*) est détruite dans le nouveau schéma physique par la transformation `[FK-EN-TE,VISITE,CLIENT.Nrep]`. Les créations des groupes *IDVISITE*, *FKvr* et *IDTELEPHONE* sont ajoutées à L_{EP1} car les tables (*VISITE* et *TELEPHONE*), auxquelles ils appartiennent, sont créées dans le nouveau schéma physique. Le tableau 8.5 donne la liste complète des modifications physiques de L_{EP1}.

L_{EP1}		
Type de transformation	**QUOTA VENTES\Physique-1**	**QUOTA VENTES\Physique**
ATT-EN-TE	TELEPHONE	CLIENT.Telephone
MODIFENT	ZONE	REGION
CREERATT	ZONE.Nomregion	
FK-EN-TE	VISITE	CLIENT.Nrep
MODIFATT	VENTE.Prix	VENTE.Prix
CREERENT	FOURNISSEUR	
CREERATT	FOURNISSEUR.Nfour	
CREERATT	FOURNISSEUR.Nom	
CREERATT	FOURNISSEUR.Adresse	
CREERATT	FOURNISSEUR.Telephone	
CREERATT	FOURNISSEUR.Jour_livraison	

Tableau 8.5 - Liste des modifications physiques L_{EP1}.

L_{EP1}		
Type de transformation	**QUOTA VENTES\Physique-1**	**QUOTA VENTES\Physique**
CREERGR	IDFOURNISSEUR	
CREERATT	PRODUIT.Nfour	
CREERGR	FKfournit	
DETRUIREGR		FKvisite
CREERGR	IDVISITE	
CREERGR	FKvr	
CREERGR	IDTELEPHONE	

Tableau 8.5 - Liste des modifications physiques L_{EP1}.

c) Arrangement de la liste des modifications

La liste L_{EP1} est restructurée en fonction du SGBD choisi de manière à obtenir un script de conversion efficace. Dans l'étude de cas, les restructurations suivantes sont nécessaires :

a) Il faut enlever toutes les modifications relatives à des attributs ou des groupes appartenant à des types d'entités créés ou renommés dans L_{EP1}. Cela concerne l'attribut *Nomregion* de *ZONE*, les attributs (*Nfour*, *Nom*, *Adresse*, *Telephone* et *Jour_livraison*) et le groupe (*IDFOURNISSEUR*) de *FOURNISSEUR*, les groupes *IDVISITE* et *FKvr* de *VISITE* ainsi que le groupe *IDTELEPHONE* de *TELEPHONE*.

b) Il faut enlever les destructions des groupes dont un des composants est détruit. C'est le cas du groupe *FKvisite* dont la colonne *Nrep* est détruite.

c) Il faut ajouter la création des clés étrangères en provenance ou à destination des tables créées ou renommées [2] à la fin de L_{EP1}. Il s'agit des clés étrangères *FKappartient* et *FKrv* qui référencent le types d'entités *ZONE*, la clé étrangère *FKpossede* de *TELEPHONE* et *FKcv* et *FKvr* de *VISITE*.

d) La création du groupe *FKfournit* est décomposée en deux modifications : la création de la clé étrangère (CREERFK) et la création de l'index (CREERACC).

Le tableau 8.6 présente la liste définitive des modifications physiques L_{EP1}.

L_{EP1}		
Type de transformation	**QUOTA VENTES\Physique-1**	**QUOTA VENTES\Physique**
ATT-EN-TE	TELEPHONE	CLIENT.Telephone
MODIFENT	ZONE	REGION
FK-EN-TE	VISITE	CLIENT.Nrep
MODIFATT	VENTE.Prix	VENTE.Prix
CREERENT	FOURNISSEUR	
CREERATT	PRODUIT.Nfour	
CREERACC	FKfournit	
CREERFK	FKappartient	
CREERFK	FKrv	
CREERFK	FKpossede	
CREERFK	FKcv	
CREERFK	FKvr	
CREERFK	FKfournit	

Tableau 8.6 - Liste définitive des modifications physiques L_{EP1}.

2. Pour renommer une table, on choisit de recréer une nouvelle table. La commande RENAME d'Oracle n'est pas utilisée.

d) Génération du script de conversion

Pour chaque modifications de L_{EP1}, on génère les requêtes SQL de conversion. Il faut remarquer que la création de la colonne *Nfour* dans la table *PRODUIT* est problématique car elle nécessite, pour chaque produit, des valeurs qui référencent un fournisseur existant. Une intervention de l'ingénieur est nécessaire pour compléter la base de données (création d'instances de *FOURNISSEUR* et de *PRODUIT.Nfour*) avant de créer la clé étrangère vers la table *FOURNISSEUR*. Lors du transfert des instances de la table REGION vers la table ZONE, les valeurs de la colonne *Nom* sont insérées dans les colonnes *Nom* et *Nomregion* de la table *ZONE* puisque, provisoirement, les zones sont les régions. Il doit aussi déterminer les requêtes de conversion des prix pour la colonne *Prix* de la table *VENTE*. On obtient le script de conversion suivant :

```
-- [ATT-EN-TE,TELEPHONE,CLIENT.Telephone]
create table TELEPHONE (Ncli numeric(5) not null,Telephone char(12) not null,
  constraint IDTELEPHONE primary key (Ncli,Telephone))
  tablespace CLIENTS;
insert into TELEPHONE SELECT Ncli,Telephone from CLIENT where Telephone is not null;
alter table CLIENT drop Telephone cascade constraint;

-- [MODIFENT,ZONE,REGION]
create table ZONE (Nom char(30) not null,Nomregion char(30) not null,
  Croissance float(3),Chomage float(3)) tablespace CLIENTS;
insert into ZONE select Nom,Nom,Croissance,Chomage from REGION;
drop table REGION cascade constraint;
alter table ZONE add constraint IDREGION primary key (Nom);
alter table ZONE add constraint GRREGION
     check((Croissance is not null and Chomage is not null)
           or (Croissance is null and Chomage is null));

-- [FK-EN-TE,VISITE,CLIENT.Nrep]
create table VISITE (Ncli numeric(5) not null,Nrep numeric(5) not null,
  constraint IDVISITE primary key (Ncli,Nrep))
  tablespace CLIENTS;
create index FKvr on VISITE (Nrep);
insert into VISITE (Ncli,Nrep) select Ncli,Nrep from CLIENT;
alter table CLIENT drop Nrep cascade constraint;
drop procedure verifier_equ_FKvisite;

-- [MODIFATT,VENTE.Prix,VENTE.Prix]
alter table VENTE modify Prix numeric(10,2);
-- + modifier les instances de Prix pour être conforme à l'euro.
update VENTE set Prix = Prix % 40.3399;

-- [CREERENT,FOURNISSEUR,/]
create table FOURNISSEUR (Nfour numeric(5) not null,Nom char(50) not null,
  Adresse char(70) not null,Telephone char(12) not null,Jour_livraison char(10),
  constraint IDFOURNISSEUR primary key (Nfour))
  tablespace PRODUITS;

-- [CREERATT,PRODUIT.Nfour,/]
alter table PRODUIT add Nfour numeric(5) default 0 not null;
-- + insérer des valeurs significatives dans la colonne Nfour de PRODUIT
--   et dans la table FOURNISSEUR.

-- [CREERACC,FKfournit,/]
create index FKfournit on PRODUIT (Nfour);

-- [CREERFK,FKappartient,/]
alter table CLIENT add constraint FKappartient foreign key (Nomreg) references ZONE;

-- [CREERFK,FKrv,/]
alter table VENTE add constraint FKrv foreign key (Nomreg) references ZONE;
```

```
-- [CREERFK,FKpossede,/]
alter table TELEPHONE add constraint FKpossede foreign key (Ncli) references CLIENT;

-- [CREERFK,FKcv,/]
alter table VISITE add constraint FKcv foreign key (Ncli) references CLIENT;

-- [CREERFK,FKvr,/]
alter table VISITE add constraint FKvr foreign key (Nrep) references REPRESENTANT;
create or replace procedure verifier_equ_FKvr is
    n number;
    contrainte_egalite_violee exception;
  begin
    select count(*) into n from REPRESENTANT
                          where Nrep not in (select Nrep from VISITE);
    if n > 0 then raise contrainte_egalite_violee; end if;
    exception when contrainte_egalite_violee then
    raise_application_error(-20303,
      'La contrainte d''egalite sur VISITE.Nrep est violee.');
  end;

-- [CREERFK,FKfournit,/]
alter table PRODUIT add constraint FKfournit foreign key (Nfour)
                                          references FOURNISSEUR;
```

8.2.6.2 Modification des programmes

La modification des programmes est, dans la plupart des cas, à la charge du programmeur. Pour l'aider dans cette tâche, un processus de localisation des sections de code susceptibles d'être modifiées a été mis au point. Il se décompose en trois phases :

- la recherche des requêtes touchées par les modifications des données,
- la recherche des variables dépendantes,
- la fragmentation de programme.

a) Recherche des requêtes

L'objectif de cette phase est de localiser les requêtes des programmes qui utilisent les objets modifiés par les transformations de L_{EP1}. Le tableau 8.7 présente, pour chaque transformation de L_{EP1}, les objets modifiés (une table ou une colonne), les procédures contenant des requêtes problématiques et les numéros de ligne de ces requêtes. Ces requêtes ont été trouvées grâce à un outil de recherche basé sur des patrons présentés au point 7.4.2. Quand l'objet modifié est une colonne, il faut d'abord rechercher les requêtes utilisant la table de la colonne puis, ensuite, rechercher, parmi les requêtes trouvées, celles qui utilisent la colonne modifiée.

Type de la transformation	Table ou colonne	Procédures	Numéros de lignes
[ATT-EN-TE, TELEPHONE, CLIENT.Telephone]	CLIENT	AFFICHEINFOREPRESENTANT CALCULNOUVQUOTAREPRESENTANT INSERTIONCLIENT AJUSTERPRIXNOUVPRODUIT	15,27 7,22,28 42 16
	Telephone	AFFICHEINFOREPRESENTANT INSERTIONCLIENT	15 42
[MODIFENT, ZONE, REGION]	REGION	CALCULNOUVQUOTAREPRESENTANT INSERTIONCLIENT AJUSTERPRIXNOUVPRODUIT	7 40 16

Tableau 8.7 - Les requêtes utilisant les objets modifiés dans L_{EP1}.

Type de la transformation	Table ou colonne	Procédures	Numéros de lignes
[FK-EN-TE, VISITE, CLIENT.Nrep]	CLIENT	AFFICHEINFOREPRESENTANT CALCULNOUVQUOTAREPRESENTANT INSERTIONCLIENT AJUSTERPRIXNOUVPRODUIT	15,27 7,22,28 42 16
	Nrep	AFFICHEINFOREPRESENTANT CALCULNOUVQUOTAREPRESENTANT INSERTIONCLIENT	15,27 7,22,28 42
[MODIFATT, VENTE.Prix, VENTE.Prix]	VENTE	AFFICHEINFOREPRESENTANT CALCULNOUVQUOTAREPRESENTANT AJUSTERPRIXNOUVPRODUIT	27 28 8,40
	Prix	AFFICHEINFOREPRESENTANT CALCULNOUVQUOTAREPRESENTANT AJUSTERPRIXNOUVPRODUIT	27 28 40
[CREERENT, FOURNISSEUR, /]	/		
[CREERATT, PRODUIT.Nfour, /]	PRODUIT	AFFICHEINFOREPRESENTANT CALCULNOUVQUOTAREPRESENTANT	27 28
[CREERACC, FKfournit, /]	/		
[CREERFK, FKappartient, /]	Transformation liée au renommage de REGION.		
[CREERFK, FKrv, /]	Transformation liée au renommage de REGION.		
[CREERFK, FKpossede, /]	Transformation liée à la création de la table TELEPHONE.		
[CREERFK, FKcv, /]	Transformation liée à la création de la table VISITE.		
[CREERFK, FKvr, /]	Transformation liée à la création de la table VISITE.		
[CREERFK,FKfournit[	Transformation liée à la création de la colonne Nfour dans PRODUIT.		

Tableau 8.7 - Les requêtes utilisant les objets modifiés dans L_{EP1}.

Les créations de l'index *FKfournit* et de la table *FOURNISSEUR* ne donnent lieu à aucune recherche car ces modifications sont indépendantes des programmes. Toutefois, dans le cas de la table *FOURNISSEUR*, il faudra implémenter sa gestion pour que les programmes tiennent compte des fournisseurs. Les créations des clés étrangères sont des transformations liées à d'autres transformations pour lesquelles les recherches ont déjà été effectuées.

b) Recherche des variables dépendantes

L'objectif de cette phase consiste à trouver toutes les variables des programmes qui sont liées aux tables et aux colonnes recherchées dans la première étape. Le graphe de dépendances est construit à partir de patrons définis pour le langage utilisé. Voici quelques patrons définissant des relations (comparaison et assignation) entre deux variables dans le pseudo-code utilisé :

```
- ::= /g"[/n/t/r ]+";
any ::= /g"[^;]*" ;
var ::= /g"[a-zA-Z][-a-zA-Z0-9]*";
rel_op ::= /g"[=<>]+";
var_1 ::= var;
var_2 ::= var;

assignation ::= @var_1 - ":=" any @var_2 ;
comparaison ::= ["(" -] @var_1 - rel_op - @var_2 [- ")"] ;
```

Il faut, dans chaque requête trouvée dans la première phase et, pour les variables ou curseurs liés aux objets modifiés, localiser les variables qui en dépendent grâce au graphe. Le tableau 8.8 présente les variables dépendant des objets modifiés ainsi que les numéros des lignes où elles sont utilisées. Dans les procédures analysées, il n'y a qu'un seul niveau de dépendance excepté

pour les variables NR de la procédure INSERTIONCLIENT et MONTVEN de la procédure CALCULNOUVQUOTAREPRESENTANT.

Dans le cas de NR, la variable NUMR trouvée dans la requête de la ligne 42 est utilisée dans la requête de la ligne 16. Cette requête, qui ne fait pas partie des requêtes trouvées dans la première phase, établit une dépendance entre NUMR et NR. En réappliquant le processus, on recherche les variables dépendantes de NR et on trouve les lignes où cette variable intervient.

Dans le cas de la variable MONTVEN, elle dépend d'autres variables à la ligne 33 dont la variable QUOT qui apparaît notamment dans la requête de la ligne 22. La variable QUOT contient les quotas des représentants d'une région. Il s'avère donc que la colonne *Quota* de la table *REPRESENTANT* est exprimée en francs belges. L'utilisation du graphe de dépendance a mis en évidence une modification de la base qui a été oubliée dans le cadre du passage à l'euro. Cela ne porte pas à conséquence sur la structure de cette colonne de type numérique avec deux chiffres après la virgule. Par contre, les instances doivent être converties en euro pour assurer une cohérence dans le calcul des nouveaux quotas. Le concepteur doit exécuter la requête de modification des données suivantes : `update REPRESENTANT set Quota = Quota % 40.3399`. Remarquons également que les variables MONTVEN et QUOT dépendent d'autres variables. Mais, après un bref examen de la procédure, on constate que ces dépendances n'entraînent aucune modification de code.

Certaines requêtes, impliquant la colonne CLIENT.Nrep, ne contiennent aucune variable en rapport avec *Nrep*. Pour corriger les programmes, il suffit de modifier ces requêtes pour qu'elles prennent en compte la table *VISITE* qui fait le lien entre les tables *CLIENT* et *REPRESENTANT*. Les requêtes, impliquant la table *PRODUIT*, ne doivent pas être modifiées.

Le travail avec le graphe de dépendances a permis de localiser les instructions susceptibles d'être modifiées dans les procédures. Mais, la lourde tâche de modification des programmes incombe toujours aux programmeurs.

Table ou colonne	Procédures	N° ligne requêtes	Variables	N° ligne variables
CLIENT.Telephone	AFFICHEINFOREPRESENTANT INSERTIONCLIENT	15 42	TEL1 (via curseur C) TEL	4,7,9,17,19,22 4,14,42
CLIENT.Nrep	AFFICHEINFOREPRESENTANT CALCULNOUVQUOTAREPRESENTANT INSERTIONCLIENT	15 27 7 22 28 42	NR (via curseur C) / / / / NUMR NR	2,17,22 3,15,16,20,29,42 3,16,17
VENTE.Prix	AFFICHEINFOREPRESENTANT CALCULNOUVQUOTAREPRESENTANT AJUSTERPRIXNOUVPRODUIT	27 28 40	VEN (via curseur V) TOTVEN MONTVEN QUOT /	2,34,36,37,38 2,37,42 3,28,33 3,22,26,33
PRODUIT	AFFICHEINFOREPRESENTANT CALCULNOUVQUOTAREPRESENTANT	27 28	/ /	

Tableau 8.8 - Les numéros de ligne des instructions contenant des variables dépendant des objets modifiés dans L_{EP1}.

c) Fragmentation de programmes

Les résultats fournis par le graphe de dépendances ne sont pas toujours précis. Pour affiner la recherche, on peut utiliser la technique plus fiable de la fragmentation de programmes. En supposant qu'on dispose du moteur permettant de calculer le SDG des procédures analysées dans cette étude de cas, la fragmentation permettrait de détecter du bruit dans les instructions four-

nies par le graphe de dépendances pour la variable TEL1, utilisée avec la requête de la ligne 15 dans la procédure AFFICHEINFOREPRESENTANT. En effet, les lignes 7 et 9 ne feraient pas partie du fragment de programme calculé sur le critère de fragmentation [ligne 19, TEL1]. Ces lignes concernent l'utilisation de la variable TEL1 pour stocker un numéro de téléphone d'un représentant et non d'un client.

8.3 Conclusion

L'objectif de ce chapitre était d'illustrer l'utilisation des méthodes et des outils développés dans les chapitres précédents. Malgré sa petite taille, l'application analysée a mis en évidence des problèmes non triviaux que l'utilisation d'une méthodologie et d'outils de propagation des modifications peuvent facilement résoudre.

L'automatisation de la conversion des structures de données et des données prend tout son sens lorsque l'ingénieur de maintenance est confronté à de très grandes quantités d'informations. L'utilisation des historiques lui permet de créer une nouvelle documentation de manière très efficace. Grâce au générateur de scripts de conversion, il dispose d'un outil de génération automatique bien plus fiable qu'une création de scripts au cas par cas.

Vu la complexité de la modification des programmes, les outils d'aide à la localisation des instructions à modifier s'avèrent indispensables lorsque le nombre de lignes de code se chiffre en milliers voire en millions d'instructions, surtout si les développeurs ne sont plus là pour modifier leurs programmes.

« Il y a quelque chose de pire dans la vie que de n'avoir pas réussi : c'est de n'avoir pas essayé.»

Roosevelt

CHAPITRE 9 CONCLUSIONS

La maintenance d'un système d'information engendre souvent le problème de l'évolution des données lorsqu'elles constituent la partie centrale des applications. L'évolution des besoins se traduit techniquement par la modification des spécifications de la base de données dans un des trois niveaux d'abstraction. La difficulté réside dans la propagation de cette modification vers les autres niveaux et vers les composants opérationnels, c'est-à-dire les structures de données, les données et les programmes.

Les concepts de l'approche DB-Main, dédiée à l'ingénierie des données, forment une base formelle favorable à la compréhension et à la résolution du problème de l'évolution : modélisation transformationnelle des processus, représentation uniforme rigoureuse des spécifications aux différents niveaux d'abstraction et selon différents paradigmes, traçabilité des processus. Si les documents de la conception d'un système sont encore disponibles ou, dans le cas contraire, s'ils peuvent être reconstitués par rétro-ingénierie, alors le contrôle de l'évolution devient un processus formellement défini, et donc largement automatisable pour ce qui concerne la base de données.

En revanche, la modification des programmes reste un problème ouvert dans le cas général. Il est cependant possible d'aider le développeur à modifier le code de ces programmes par le repérage des sections où des instances des types d'objets modifiés sont traitées.

Ce chapitre est organisé de la manière suivante. La section 9.1 résume les apports de la thèse dans le cadre du problème de l'évolution. La section 9.2 présente une analyse économétrique de deux cas réels qui illustrent l'efficacité de l'approche proposée. Les limites de la démarche sont décrites dans la section 9.3. Finalement, la section 9.4 explore une nouvelle perspective d'utilisation de la méthode pour résoudre des problèmes de migrations de données.

9.1 Contribution

Résumons les principaux résultats et les apports de ce travail à la résolution du problème de la maintenance et l'évolution d'applications de bases de données.

Cadre de travail

Nous avons affiné des concepts et des méthodes développés depuis plus de 10 ans dans le laboratoire de bases de données de l'Université de Namur pour définir un cadre méthodologique établissant les bases de cette thèse. Ce cadre comprend quatre piliers sur lesquels repose notre solution au problème de l'évolution :

1. Le phénomène de l'évolution est analysé dans le cadre de la modélisation classique qui envisage la conception d'une base de données comme une activité complexe décomposable en processus plus élémentaires. Ces processus, allant de la collecte des informations sur les besoins des utilisateurs jusqu'à la génération de code, transforment l'expression des besoins des utilisateurs en une collection de schémas et de programmes respectant des critères bien définis. La démarche de conception choisie est décomposée en trois niveaux d'abstraction (conceptuel, logique et physique). Elle est abondamment décrite et utilisée dans la littérature, et, par conséquent, familière à la plupart des concepteurs.
2. Un modèle générique basé sur un modèle Entité/Association étendu permet de représenter des spécifications dans les différents niveaux d'abstraction et de couvrir, pour chaque niveau, les principaux paradigmes de modélisations et technologies existants. Le choix du modèle E/A se justifie par sa grande popularité et une représentation graphique des concepts particulièrement attrayante. Il se prête aussi bien à la compréhension de schémas conceptuels par des utilisateurs non-initiés qu'à la conception de schémas physiques optimisés par des concepteurs chevronnés.
3. Tout processus d'ingénierie des bases de données peut être défini comme une suite de modifications de structures de données. Les relations entre les spécifications doivent être formalisées, analysées et manipulées de manière à permettre le raisonnement et la dérivation de nouvelles spécifications. Une transformation est considérée comme la présentation formelle de relations entre spécifications, c'est-à-dire comme un opérateur qui remplace une structure de données par une autre ayant certaines relations sémantiques (augmentation, diminution ou préservation) avec la structure initiale. Tous les processus d'ingénierie utilisés et décrits sont basés sur cette approche transformationnelle.
4. L'historique d'un processus complexe est la trace de toutes les opérations qui ont été exécutées pour le réaliser. Il permet entre autres de documenter une activité pour éviter qu'elle n'échappe à tout contrôle et pour qu'elle puisse être manipulée en rejouant les opérations historisées par exemple. La documentation de la conception d'un système doit être disponible car elle forme la base de sa maintenance et de son évolution.

Typologie des modifications

Pour développer une méthodologie débouchant sur des réalisations concrètes, la démarche se base sur la technologie relationnelle pour dégager une typologie des modifications. Cette typologie analyse de manière semi-formelle toutes les modifications possibles sur des spécifications décrites dans le modèle E/A étendu. Elle est partitionnée selon un enrichissement des modèles définis pour chaque niveau d'abstraction et qui sont une spécialisation du modèle générique.

Outre le développement de la méthodologie, l'étude typologique a permis de concevoir des outils de conversion pour la partie opérationnelle (développée à l'aide de langages standards de troisième génération tels que COBOL/SQL ou C/SQL) et l'expression de quelques recommandations pour la conception de systèmes plus flexibles.

Méthodologie

L'objectif de la thèse est d'analyser le phénomène de l'évolution du composant base de données des applications au travers de stratégies liées au niveau d'abstraction des besoins dont la modification induit cette évolution. Suite à l'évolution des besoins des utilisateurs, l'analyste est amené à modifier les spécifications de la base de données à un certain niveau d'abstraction. Il faut l'aider à propager les modifications dans tous les niveaux d'abstraction ainsi que dans le niveau opérationnel.

Le problème de la propagation est d'autant plus complexe qu'il peut y avoir plusieurs contextes d'évolution. L'hypothèse selon laquelle une application de bases de données s'inscrit dans un contexte où les trois niveaux d'abstraction sont présents et documentés est généralement irréaliste. Pour pallier ces lacunes, il faut d'abord reconstruire la documentation d'une application grâce à des techniques de rétro-ingénierie.

La méthodologie développée comprend trois stratégies de référence répondant chacune à des problèmes de modification des spécifications à un niveau d'abstraction. L'étude de ces stratégies est limitée au contrôle de la propagation des modifications vers les autres niveaux. Elle met également en place des mécanismes pour la génération de scripts de conversion capables de modifier les structures de données (et leurs instances) de l'application, de manière à satisfaire les nouveaux besoins. Ces convertisseurs sont construits à partir des schémas de spécifications des différents niveaux d'abstraction ainsi que des historiques d'évolution et de conception. Sachant que la modification des programmes n'est généralement pas automatisable, la méthodologie décrit et met en œuvre des techniques de localisation de zones de code à modifier.

Recommandations de conception

Pour mieux répondre aux besoins des utilisateurs, le problème de l'évolution devrait être envisagé au moment de la conception d'un système en essayant de minimiser l'impact des modifications sur la base de données par des choix adéquats de représentations.

A partir des choix de représentations de l'analyse conceptuelle, des transformations de conception et de l'étude des modifications, nous tentons d'extraire des recommandations de conception améliorant la flexibilité d'une base de données. Sans perdre de vue que ce qu'on gagne en flexibilité, on le perd souvent en termes de lisibilité, de performance ou de compréhension des programmes.

AGL supportant l'évolution

Les stratégies de propagation, les outils de conversion des données et de modification des programmes sont implémentés dans l'atelier de génie logiciel DB-Main. Il est développé à l'Université de Namur et a acquis, depuis quelques années, une certaine renommée. L'atelier, programmé en Borland C++ sous Windows (3.1, 95, 98, NT et 2000), met à disposition des fonctions et des composants d'usage très général qui permettent en particulier le développement de processeurs spécialisés assurant la gestion de l'évolution :

- un référentiel basé sur le modèle générique de représentation des spécifications autorisant la représentation des structures de données à tous les niveaux d'abstraction;
- une interface graphique assurant la visualisation du contenu du référentiel selon plusieurs formats (graphiques et textuels) ainsi que l'exécution d'opérations;
- une boîte à outils transformationnelle concrétisée par une trentaine de transformations élémentaires applicables aussi bien en ingénierie qu'en rétro-ingénierie;
- des assistants programmables permettant la résolution de problèmes (de transformation, de rétro-ingénierie, d'analyse de conformité de schémas, ...) répétitifs et spécifiques;
- une personnalisation méthodologique autorisant la spécification de démarches particulières interprétées par un moteur méthodologique contrôlant les actions permises et la présentation du référentiel;

- des fonctions de gestion des historiques comprenant l'enregistrement, la sauvegarde, l'inversion ou la réexécution des opérations enregistrées;
- des outils d'analyse de programmes permettant de construire et consulter des graphes de dépendance des variables ainsi que des fragments de programmes;
- un langage de quatrième génération (Voyager 2) permettant au concepteur de personnaliser l'atelier et de développer ses propres fonctions.

Nous disposons actuellement d'un prototype de contrôle de l'évolution de bases de données relationnelles selon les stratégies de propagation étudiées. Ce prototype a été développé en Voyager 2 sur la plate-forme générique DB-Main. Il est actuellement possible de générer de manière automatique les convertisseurs de bases de données correspondant à une séquence de modifications quelconques des types T-, T+ ou T=. Les règles relatives aux modifications des programmes sont définies, mais les outils sont en cours d'implémentation.

9.2 Mesures économétriques

Concrètement, que peut retirer une organisation de l'utilisation de la méthode proposée et des outils qui l'implémentent ? Voila une question que ne manqueront pas de se poser les décideurs confrontés à des problèmes de maintenance. Pour donner un début de réponse, nous allons examiner deux cas réels (provenant de grandes sociétés de distribution) qui, pour des raisons évidentes de confidentialité, seront appelés *application 1* et *application 2*.

Le tableau 9.1 donne une indication chiffrée de la taille des applications. L'application 1 est la plus grande en terme de lignes de code mais les modules de l'application 2 sont plus importants. Le découpage modulaire semble plus harmonieux dans l'application 2 où les tables sont utilisées dans 7 modules en moyenne. Au niveau physique, on constate que l'application 1 a deux fois plus de tables que l'application 2 pour presque le même nombre de colonnes. Toutes ces indications vont évidemment influencer la mise à jour des applications.

	Application 1	**Application 2**
Structure des bases de données		
Schéma conceptuel	500 types d'entités, 800 types d'associations, 4.000 attributs	96 types d'entités, 109 types d'associations, 1.086 attributs
Schéma physique	700 tables, 5.500 colonnes	326 tables, 4.850 colonnes
Programmes		
Nombre total de lignes de code	1.750.000	180.000
Nombre total de modules	3.500	270
Apparition moyenne d'une table dans les modules	20	7
Nombre moyen de lignes de code par module	500	700

Tableau 9.1 - Représentation chiffrée de deux applications analysées.

L'analyse porte sur 20 modifications annuelles portant sur chacune des applications. On prend comme hypothèse qu'elles se traduisent par la conversion des bases de données qui engendre la modification des programmes. Les chiffres de la table 9.2 sont des estimations basées sur des constatations réelles en ce qui concerne la localisation des lignes de code dans le cadre de problèmes similaires à la problématique de l'évolution. Pour chaque application, on distingue le processus d'évolution manuel qui n'utilise ni méthode, ni outil spécifique, et le processus assisté qui utilise la démarche et les outils présentés dans ce travail.

Dans les deux applications, la durée de la conversion des bases de données est multipliée par un facteur trois lorsque le processus est manuel. Cela s'explique par l'absence de documentation du processus de conception des spécifications. Les ingénieurs de la maintenance travaillent directement sur les structures physiques qui manquent de lisibilité. Le repérage des spécifications à modifier dure plus temps. L'écriture des scripts de conversion est manuel, ce qui augmente le risque d'erreur et nécessite une phase de test importante. Par conséquent, il est logique que la durée de la conversion manuelle soit trois fois plus longue que celle de la conversion assistée.

La localisation des modules des applications susceptibles d'être modifiés dure aussi plus longtemps dans la méthode manuelle. Cette durée est fonction de la quantité de modules à analyser. Pour la méthode assistée, la localisation nécessite uniquement une phase de préparation des outils et de mise en route en arrière-plan, les ingénieurs restant libres pour accomplir d'autres tâches ou analyser les premiers modules résultats de la recherche. Pour chaque module modifiable (en moyenne 20 pour l'application 1 et 7 pour l'application 2), le temps de localisation manuelle des lignes de code susceptibles d'être modifiées est six fois supérieure à celui de la méthode assistée. Ce résultat est basé sur des tests réels de localisation dans des modules comprenant entre 500 et 1000 lignes. Le temps de modification des lignes de code est identique

dans les deux méthodes de travail. Au total, on obtient une durée de réalisation de la méthode manuelle qui est trois fois supérieure à celle de la méthode assistée. Ce résultat est très intéressant quand on voit que la méthode assistée permet pour l'application 1 d'épargner quatre mois de travail d'un ingénieur de maintenance ce qui représente un gain minimum d'un million de francs belges. Précisons que l'analyse ne tient pas compte de deux facteurs importants :

- Les risques d'erreurs sont plus importants dans la méthode manuelle, ce qui accentue encore la différence entre les deux méthodes.
- L'investissement de départ de la méthode assistée nécessite, pour les systèmes hérités, la redocumentation de la base et l'adaptation des outils de conversion de données et d'analyse de programmes. On peut évaluer cet investissement entre quinze jours (application 2) et un mois de travail (application 1), ce qui n'empêche pas cette méthode d'être toujours plus efficace surtout si elle est utilisée pendant plusieurs années.

	Application 1		**Application 2**	
	Manuel	**Assistée**	**Manuel**	**Assistée**
Structures + Données	1 h 30'	30'	1 h	20'
Localisation modules	8 h	30'	1 h	30'
Localisation et modification code	20 * (90' + 30')= 40 h	20 * (15' + 30')= 15 h	7 * (90' + 30')= 17 h	7 * (15' + 30')= 5 h 15'
Temps estimé pour une modification	49 h 30'	16 h	19 h	6 h 05'
Temps total estimé pour 20 modifications annuelles	**123,75 jours**	**40 jours**	**47,5 jours**	**15,2 jours**

Tableau 9.2 - Estimation du temps de réalisation des modifications des applications analysées

Le graphique de la figure 9.1 montre que, jusqu'à une certaine taille (T), il n'est pas intéressant d'investir dans la méthode d'évolution assistée. Cela s'explique par le coût de mise en place de la méthode et des outils. Tout le problème réside dans la détermination de T, c'est-à-dire à partir de quel moment devient-il intéressant d'utiliser le processus assisté ? T est d'autant plus difficile à déterminer que, à priori, on ne connaît pas le nombre de modifications à réaliser et la durée de vie de l'application. L'analyse de l'application 2 indique que, pour une taille raisonnable (180.000 lignes de code), l'investissement est déjà rentable après une dizaine de modifications (manuel : 19h * 10 = 23,75 jours; assisté : 6h05' * 10 + 15 jours = 22,6 jours).

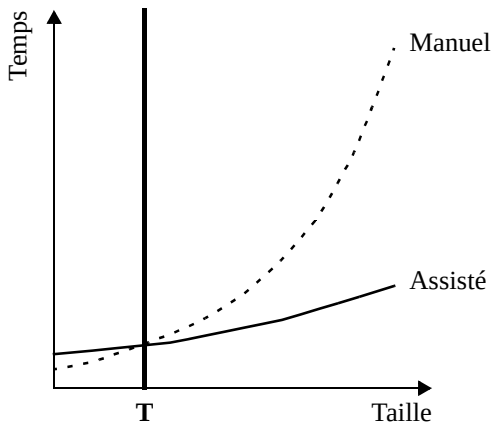


Figure 9.1 - Evolution du temps de mise à jour assistée et manuelle d'une application en fonction de sa taille.

9.3 Limites de l'approche

Une des missions initiales de cette thèse était d'automatiser au maximum le travail de l'ingénieur dans le cadre de sa tâche de maintenance et d'évolution d'applications de bases de données. Très vite, deux obstacles majeurs sont apparus :

- Pour aboutir à une méthodologie et des outils concrets supportant l'évolution, la démarche développée a dû se fonder sur un modèle de données précis et une technologie d'implémentation de base de données qui lui correspondait.
- Dans beaucoup de cas, la modification des programmes ne peut être réalisée automatiquement.

Non-généricité

Chaque modèle de données a ses propres caractéristiques qui induisent des modifications spécifiques. Dans ce contexte, il est difficile voire impossible de réaliser une étude des modifications sans s'attacher à un modèle de données concret. Mis à part les éléments de base du cadre méthodologique (une démarche de conception classique, un modèle générique de représentation des spécifications, une approche transformationnelle et la notion de traçabilité), les autres composants de la démarche dépendent plus ou moins du modèle de données choisi.

Bien que la méthodologie ait été développée dans un souci d'indépendance vis-à-vis d'un modèle de données, elle est légèrement influencée par le choix du modèle relationnel.

Le prototype réalisé dépend plus fortement du modèle de données. S'il est amené à dépasser le cadre du prototype pour devenir un outil vraiment opérationnel, il est en plus tributaire d'une technologie et d'un langage spécifique dans lequel les scripts de conversion seront exprimés. Le même constat intervient pour les outils d'analyse de programmes qui connaissent le langage de programmation utilisé afin de localiser assez précisément le code susceptible d'être modifié.

Toutefois, nous avons essayé de regrouper les parties de la démarche connotées par une dépendance au modèle de données afin de permettre une meilleur adaptation à d'autres modèles de données :

- Au niveau de la méthodologie, seules les transformations autorisées à chaque niveau d'abstraction sont définies en fonction du modèle choisi. Dans l'atelier DB-Main, il s'agit des boîtes à outils spécifiées dans la méthode.
- Le prototype de conversion des données est décomposé selon un processus en quatre phases : l'analyse des historiques créant les historiques simplifiés, la traduction des modifications des historiques simplifiés en modifications sur les spécifications physiques, la construction d'une liste de modifications à partir de l'historique simplifié obtenu dans la phase précédente et la génération des scripts de conversion sur la base de la liste de modifications. L'avant-dernière phase dépend du modèle de données car la construction de la liste est combinée avec une simplification des modifications de manière à tirer parti des caractéristiques du modèle. Quant à la dernière phase, elle dépend incontestablement de la technologie utilisée.
- Les outils d'analyse de programmes sont totalement dépendants du langage de programmation utilisé. Pour la recherche de patrons et le graphe de dépendances des variables, la construction de nouveaux patrons est relativement aisée. Par contre, dans l'outil de fragmentation de programmes qui analyse le code en profondeur, l'analyseur syntaxique doit être complètement réécrit. Les personnes initiées comprendront tout de suite l'ampleur de la tâche.

De la sorte, nous avons essayé de minimiser les impacts du changement de modèle de données sur la démarche. Il est clair que l'adaptation des outils mobilisera plus de ressources que l'ajustement de la méthode.

Modification non-automatisable des programmes

D'une part, la complexité des programmes, les degrés de liberté offerts par les langages de programmation ou les multiples façons de programmer sont autant de raisons qui ont mis en évidence la non-faisabilité d'outils supportant de manière automatique la propagation des modifications qui altèrent la structure (l'agencement des instructions) des programmes. Une modification automatique des programmes serait envisageable s'ils pouvaient être représentés dans un modèle abstrait de même niveau de formalité que le composant base de données d'une application. La mise à disposition, pour chaque niveau d'abstraction, de spécifications des fonctionnalités d'une application permettrait d'appliquer des transformations sur ces spécifications et d'en garder une trace. A l'heure actuelle, la recherche dans ce domaine reste ouverte et constitue un travail qui dépasse le cadre de cette thèse.

D'autre part, notre expérience du terrain ainsi que les constats émanant de la littérature scientifique nous permettent d'affirmer que la majorité des applications existantes sont trop peu ou pas du tout documentées. Pour pallier à ces manquements, il est toujours possible de reconstruire une documentation plausible du composant base de données de toute application. Malheureusement, la redocumentation des programmes est un domaine de recherche dans lequel les résultats ne sont pas aussi probants.

Sur la base de ces constats, un début de solution a été ébauché à partir de techniques d'analyses de programmes bien maîtrisées. La localisation des zones de code susceptibles (on ne peut empêcher le bruit et les silences) d'être modifiées est une solution intermédiaire qui devrait satisfaire la plupart des programmeurs face à l'absence d'outils spécialisés dans le domaine.

9.4 Perspective future : migration des données

Les principes développés sont applicables à d'autres domaines de l'ingénierie des données. La migration de données consiste à consolider des données venant de systèmes hérités et à les transférer vers d'autres applications ou des entrepôts de données [1]. Le processus de la migration des données présente des similitudes avec celui de l'évolution, notamment au niveau des transferts des instances d'une structure origine vers une structure cible. Ne pourrait-on pas voir la migration de données comme un cas particulier de notre démarche ? Ou plus précisément :

- Le cadre méthodologique peut-il servir de base pour résoudre le problème de la migration de données ?
- La méthodologie peut-elle être adaptée pour tenir compte des spécificités de la migration ?
- Les outils de génération peuvent-il être repris totalement ou partiellement pour générer des scripts de migration de données ?

Le cadre méthodologique définit les piliers indispensables à la réalisation de toutes les activités d'ingénierie de bases de données dont la migration fait partie. La migration de données concerne souvent deux modèles de données distincts, qui sont exprimables dans le modèle générique, quelque soit le paradigme utilisé. La construction des spécifications des deux modèles peut être formalisée grâce à l'approche transformationnelle. Les historiques gardent une trace des transformations exécutées entre le schéma origine et le schéma cible.

La figure 9.2 présente un historique d'une méthodologie pour la migration de données qui est une adaptation de la méthodologie de propagation des modifications. La première étape consiste à extraire les structures de données dans un schéma logique SP0 qui, après un nettoyage (CP0'), devient le schéma logique SL0'. Le processus CL0' prépare, conceptualise et normalise SL0 afin d'obtenir le schéma conceptuel SC0. Cette première phase s'apparente à la phase de reconstruction des spécifications définie dans la méthodologie. La deuxième étape construit les nouvelles spécifications (SL1 et SP1) grâce aux processus de conception logique et physique (CL1 et CP1). Le processus de génération crée les structures de données et celui de migration construit le script de transfert des instances de D0 vers D1. Cette deuxième phase correspond à une phase de conception classique.

Certains modules du convertisseur de données développé pour l'évolution sont réutilisables pour l'outil de génération de scripts de migration [2]. Les différentes étapes du générateur sont :

a) A partir du schéma physique SP0, le générateur produit le programme d'extraction (écrit dans le langage du système source) des instances de D0 dans un support intermédiaire (un ou plusieurs fichiers texte par exemple). Cette étape n'est pas nécessaire pour les systèmes récents qui supportent des techniques de communication avec d'autres applications telles que ODBC [3] par exemple. Pour les systèmes hermétiques, c'est la seule façon d'extraire l'information pour pouvoir la migrer.

b) Le générateur analyse les historiques CP0' et CL0' (respectivement CL1 et CP1) pour créer la liste des transformations C0' (respectivement C1) qui, appliquée à SP0 (respectivement SC0), donne SC0 (respectivement SP1). Le module d'analyse des historiques est identique à celui du convertisseur de données.

c) Le générateur compare les deux listes de transformations C0' (CP0' et CL0') et C1 (CP1 et CL1) pour générer une liste de transformations des structures de données originales (SP0)

1. Un entrepôt de données est une structure informatique dans laquelle est centralisé un volume important de données consolidées à partir des différentes sources de renseignements d'une entreprise et qui est conçue de telle manière que les personnes intéressées aient accès rapidement et sous forme synthétique à l'information stratégique dont elles ont besoin pour la prise de décision ("data warehouse" en anglais).
2. Le générateur du script de construction des structures de données de SP1 est un processeur à part.
3. pour "Open DataBase Connectivity" : interface standardisée, définie par Microsoft, permettant d'accéder à des bases de données ayant des formats différents.

vers les nouvelles structures (SP1). Le module de traduction des structures d'origine vers les structures cibles présente beaucoup de similitudes avec celui du convertisseur de données.

d) A partir de la liste de transformations, le générateur construit le programme qui va lire les données dans les fichiers intermédiaires (*Fichiers*) générés à l'étape a) (ou directement dans le système d'origine grâce à des techniques comme ODBC) et il les transfère dans les nouvelles structures.

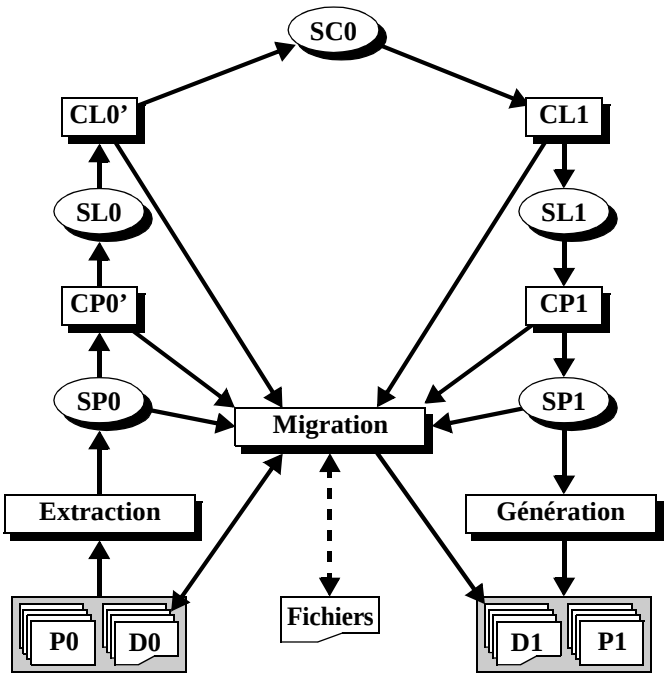


Figure 9.2 - Historique d'une méthodologie de migration de données.

Il existe sur le marché des outils de migration permettant de migrer d'une technologie vers une autre sans possibilité de l'appliquer dans d'autres contextes (migration "one shot"). Ces outils présentent le désavantage de migrer également les problèmes d'implémentation et les structures techniques propres à la technologie originale. La structure de la nouvelle base bien que conforme au nouveau modèle conserve des spécifications inutilement complexes, redondantes ou parfois même erronées. Ces anomalies rendent le système peu efficace, difficile à exploiter et peu flexible.

La méthodologie de migration ébauchée ici s'inscrit dans un contexte plus général où, moyennant certaines adaptations (semblables à celles de la méthodologie de propagation des modifications), n'importe quel type de migration peut être envisagé. La rétro-ingénierie fournit comme résultat un schéma conceptuel lisible sans artifices techniques. Un excellent point de départ pour une nouvelle conception dont le but est de construire une base de données efficace et conforme au nouveau modèle sans reproduire les spécificités de la technologie originale. Le passage par le schéma conceptuel autorise même la correction d'erreurs éventuelles ou l'évolution des spécifications pour satisfaire aux changements des besoins de l'organisation.

BIBLIOGRAPHIE

[Abd95] A. Abdellatif, M. Limame, A. Zeroual, *Oracle 7 : Langages - Architecture - Administration (troisième édition)*, Eyrolles, 1995.

[Adi93] M. Adiba, C. Collet, *Objets et bases de données - le SGBD O2*, Hermès, 1993.

[Alj95] L. Al-Jadir, T. Estier, G. Falquet, M. Léonard, *Evolution features of the F2 OODBMS*, in Proceedings of the International Conference on Database Systems for Advanced Applications, World Scientific Publishing, Singapore, April 1995.

[And91] J. Andany, M. Léonard, C. Palisser, *Management of Schema Evolution in Databases*, in Proceedings of the International Conference on Very Large DataBases, Barcelona, September 1991.

[Ari91] G. Ariav, *Temporally oriented data definitions: managing schema evolution in temporally oriented databases*, Data & Knowledge Engineering, no. 6, pp. 451-467, 1991.

[Bal81] R. Balzer, *Transformational implementation: an example*, IEEE Transactions on Software Engineering, vol. 7, no. 1, 1981.

[Ban81] F. Bancilhon, N. Spyratos, *Update semantics of relational views*, ACM Transactions on Database Systems, vol.6, no. 4, pp. 557-575, December 1981.

[Ban87] J. Banerjee, W. Kim, H.-J. Kim, H. F. Korth, *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*, in Proceedings of the Annual Conference on ACM Special Interest Group on Management of Data, San Francisco, May 1987.

[Bat92] C. Batini, S. Ceri, S. B. Navathe, *Conceptual Database Design - An Entity-Relationship Approach*, Benjamin-Cummings, 1992.

[Bel93] Z. Bellahsene, *An Active Meta-Model for Knowledge Evolution in an Object-oriented Database*, in Proceedings of the International Conference on Advanced Information Systems Engineering, Springer-Verlag, 1993.

[Bel96] Z. Bellahsene, P. Poncelet, M. Teisseire, *Views for information system design without reorganization*, in Proceedings of the International Conference on Advanced Information Systems Engineering, Heraklion, May 1996.

[Bel00] Z. Bellahsene, *Updates and object-generating views in ODBS*, Data & Knowledge Engineering, no. 34, pp. 125-163, 2000.

[Big89] T. J. Biggerstaff, *Design recovery for maintenance and reuse*, IEEE Computer, July 1989.

[Bla98] M. Blaha, W. Permerlani, *Object-oriented modeling and design for database applications*, Prentice-Hall, 1998.

[Bod89] F. Bodard, Y. Pigneur, *Conception assistée des systèmes d'information : méthodes, modèles et outils*, Masson, 1989.

[Boo94] G. Booch, *Analyse & Conception Orientées Objets*, Addison-Wesley, 1994.

[Bow91] J. P. Bowen, P. T. Breuer, K. C. Lano, *The Redo project: final report*, technical report, Oxford University, 1991.

[Bow93] J. Bowen, P. Breuer, K. Lano, *A Compendium of Formal Techniques for Software Maintenance*, IEE/BCS Software Engineering Journal, vol. 8, no. 5, pp 253-262, September 1993.

[Bro86] F. P. Brooks Jr., *No silver bullet - Essence and accidents of software engineering*, in Information Processing, North-Holland, pp. 1069-1076, 1986.

[Bro96] A. Brown, E. Morris, S. Tilley, *Assessing the evolvability of a legacy system*, technical report, Software Engineering Institute, Carnegie Mellon University, 1996.

[Bur00] E. Burd, M. Munro, *Using evolution to evaluate reverse engineering technologies: mapping the process of software change*, Journal of Systems and Software, vol. 53, no. 1, pp. 43-51, July 2000.

[Cas93] M. A. Casanova, L. Tucherman, A. H. F. Laender, *On the design and maintenance of optimized relational representations of entity-relationship schemas*, Data & Knowledge Engineering, North-Holland, no. 11, pp. 1-20, 1993.

[Cat97] R. G. G. Cattell, *Bases de donneés orienteés objets (deuxième édition)*, Addison-Wesley, 1997.

[Cat00] R. G. G. Cattel et al., *The Object Database Standard: ODMG 3.0*, Morgan Kaufmann, 2000.

[Cel97] J. Celko, *SQL avancé*, International Thomson Publishing France, 1997.

[Cha96] D. Chamberlin, *Using the new DB2, IBM's object-relational database system*, Morgan Kaufmann, 1996.

[Cha98] D. Chamberlin, *A complete guide to DB2 universal database*, Morgan Kaufmann, 1998.

[Che76] P. P. Chen, *The Entity-Relationship Model - Toward a Unified View of Data*, ACM Transactions on Database Systems, vol. 1, no. 1, pp. 9-36, 1976.

[Chu91a] L. Chung, *Representation and Utilization of Non-Functional Requirements for Information System Design*, in Proceedings of the International Conference on Advanced Information Systems Engineering, Trondheim, 1991.

[Chu91b] K. L. Chung, P. Katalagarianos, M. Marakakis, M. Mertikas, J. Mylopoulos, Y. Vassiliou, *From Information System Requirements to Designs: A Mapping Framework*, Information Systems, vol. 16, 1991.

[Cla91] A. Clarinval, *Comprendre et connaître le COBOL 85*, Presses universitaires de Namur, 1991.

[Cod70] E. Codd, *A relational model for large shared data banks*, Communications of the ACM, June 1970.

[Con96] T. Connolly, C. Begg, A. Strachan, *Database systems, a practical approach to design, implementation and management*, Addison-Wesley, 1996.

[Coo97] R. Cooper, *Object databases - An ODMG approach*, Thomson Computer Press, 1997.

[Dat94] C. J. Date, H. Darwen, *A guide to the SQL standard (third edition)*, Addison-Wesley, 1994.

[Dat00] C. J. Date, *An introduction to database systems (seventh edition)*, Addison-Wesley, 2000.

[Day82] U. Dayal, P. A. Bernstein, *On the correct translation of update operations on relational views*, ACM transactions on Database Systems, vol. 8, no. 3, pp. 381-416, September 1982.

[DB298] DB2 Universal Database - SQL Reference Version 5.2, IBM, 1998.

[Dbm99] DB-Main project, *Computer-aided Database Engineering - Volume 1: Database Models (fourth edition)*, technical document, Institut d'Informatique, FUNDP, 1999.

[Del95] P. Delmal, *SQL 2: Application à Oracle, Access et Rdb*, Deboek, 1995.

[Des91] C. Desclaux, M. Ribault, *MACS: Maintenance Assistance Capability for Software Maintenance*, in Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, pp. 2-11, 1991.

[Des92] C. Desclaux, M. Ribault, S. Cochinal, *RE-ORDER: A reverse engineering methodology*, in Proceedings of the International Conference on Software Engineering and Applications, pp. 517-529, Toulouse, December 1992.

[DTG71] Database Task Group, *CODASYL*, technical report, 1971.

[Eis99] A. Eisenberg, J. Melton, *SQL:1999, formerly known as SQL3*, SIGMOD Record, vol. 28, no. 1, March 1999.

[Elm94] R. Elmasri, S. B. Navathe, *Fundamentals of Database Systems (second edition)*, Benjamin-Cummings, 1994.

[Ema95] K. E. Emam, K. Goldenson, *SPICE: An empiricist's perspective*, in Proceedings of the International Conference of Software Engineering Standards Symposium, Montreal, August 1995.

[Eng99] V. Englebert, J.-L. Hainaut, *{DB-Main}: A Next Generation Meta-{CASE}*, Journal of Information Systems - Special Issue on Meta-CASEs, vol. 24, no. 2, pp 99-112, 1999.

[Eng00] V. Englebert, *Voyager 2 (version 6.0) - Reference manual*, technical document, Institut d'Informatique, FUNDP, December 2000.

[Ewa93] C. A. Ewald, M. E. Orlowska, *A procedural approach to schema evolution*, in Proceedings of the International Conference on Advanced Information Systems Engineering, Paris, 1993.

[Ewa96] C. A. Ewald, M. E. Orlowska, *Characterization of the effects of schema change*, Information Sciences, Elsevier Science, vol. 94, pp 23-29, 1996.

[Fic85] S. Fickas, *Automating the transformational development of software*, IEEE Transactions on Software Engineering, vol. SE-11, pp. 1268-1277, 1985.

[For91] J. Fortemps, J.-M. Hick, *Contribution à une méthodologie de développement d'applications d'aide à la décision*, mémoire de troisième maîtrise, Institut d'Informatique, FUNDP, 1991.

[Fra97] *Adobe Framemaker Version 5.5 : manuel d'utilisation*, Adobe Systems Incorporated, 1997.

[Gal91] K. B. Gallagher, J. R. Lyle, *Using program slicing in software maintenance*, IEEE Transactions on Software Engineering, vol. 17, no. 8, August 1991.

[Gel89] J. R. Geller, *IMS: Administration programming and database design*, John Wiley & Sons, 1989.

[Gib94] W. Gibbs, *Software's Chronic Crisis*, Scientific American, September 1994.

[Got94] O. C. Z. Gotel, A. C. W. Finkelstein, *An analysis of the requirements traceability problem*, in Proceedings of the International Conference on Requirements Engineering, IEEE CS press, p. 94-101, 1994.

[Hai86] J.-L. Hainaut, *Conception assistée des applications informatiques - Conception de la base de données*, Masson, 1986.

[Hai93] J.-L. Hainaut, M. Chandelon, C. Tonneau, M. Joris, *Contribution to a theory of database reverse engineering*, in Proceedings of the IEEE Working Conference on Reverse Engineering, IEEE Computer Press, Baltimore, May 1993.

[Hai94] J-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, D. Roland, *Evolution of database Applications: the DB-Main Approach*, in Proceedings of the International Conference on Entity-Relationship Approach, Springer-Verlag, Manchester, 1994.

[Hai96a] J.-L. Hainaut, *Specification Preservation in Schema transformations: Application to Semantics and Statistics*, Data & Knowledge Engineering, Elsevier Science, vol. 19, pp. 99-134, 1996.

[Hai96b] J.-L. Hainaut, J. Henrard, J.-M. Hick, D. Roland, V. Englebert, *Database Design Recovery*, in Proceedings of the International Conference on Advanced Information Systems Engineering, Springer-Verlag, 1996.

[Hai97] J.-L. Hainaut, V. Englebert, J.-M. Hick, J. Henrard, D. Roland, *Knowledge Transfer in Database Reverse Engineering - A Supporting Case Study*, in Proceedings of the IEEE Working Conference on Reverse Engineering, IEEE Computer Society Press, Amsterdam, October 1997.

[Hai99] J.-L. Hainaut, *Ingénierie des bases de données*, cours première licence, Institut d'Informatique, FUNDP, 1999.

[Hai00a] J.-L. Hainaut, *Bases de données et modèles de calcul - Outils et méthodes pour l'utilisateur*, Dunod, 2000.

[Hai00b] J.-L. Hainaut, *Analyse de l'information et conception de bases de données*, document technique, Institut d'Informatique, FUNDP, 2000.

[Hal95] T. A. Halpin, *Conceptual schema and relational database design*, Prentice-Hall, 1995.

[Har95] J. V. Harrison, *Incremental view maintenance in extended relational databases*, Information and Software Technology, vol. 37, no. 9, pp. 479-491, 1995.

[Har98] P. Harmon, M. Watson, *Understanding UML: the developpers's guide with a web-based application in Java*, Morgan Kaufmann, 1998.

[Hen96] J. Henrard, J.-M. Hick, D. Roland, V. Englebert, J.-L. Hainaut, *Technique d'analyse de programmes pour la rétro-ingénierie de bases de données*, dans Actes du Congrès INFORSID, Bordeaux, Juin 1996.

[Hen98a] J. Henrard, D. Roland, V. Englebert, J.-M. Hick, J.-L. Hainaut, *Outils d'analyse de programmes pour la rétro-ingénierie de bases de données*, dans Actes du Congrès INFORSID, Montpellier, Mai 1998.

[Hen98b] J. Henrard, V. Englebert, J.-M. Hick, D. Roland, J.-L. Hainaut, *Program understanding in databases reverse engineering*, in Proceedings of the International Conference on Database and Expert Systems Applications, Vienna, August 1998.

[Hen01] J. Henrard, J.-L. Hainaut, *Data dependency elicitation in database reverse engineering*, in Proceedings of the European Conference on Software Maintenance and Reengineering, Lisbon, March 2001.

[Hic98] J.-M. Hick, J.-L. Hainaut, *Maintenance et evolution d'applications de bases de donnees*, Journées sur la Réingénierie des Systèmes d'Information, Lyon, Avril 1998.

[Hic99] J.-M. Hick, J.-L. Hainaut, V. Englebert, D. Roland, J. Henrard, *Stratégies pour l'évolution des applications de bases de données relationnelles : l'approche DB-Main*, dans Actes du Congrès INFORSID, La Garde, juin 1999.

[Hic00a] J.-M. Hick, *DB-Main Project: Transformations*, technical document, Institut d'Informatique, FUNDP, Namur, 2000.

[Hic00b] J.-M. Hick, V. Englebert, J. Henrard, D. Roland, J.-L. Hainaut, *The DB-Main Database Engineering CASE Tool (version 6) - Functions Overview*, technical manual, Institut d'Informatique, FUNDP, November 2000.

[Hor90] S. Horwitz, T. Reps, D. Binkley, *Interprocedural slicing using dependence graphs*, ACM Transactions on Programming Languages and Systems, vol. 12, no. 1, pp. 26-60, January 1990.

[Hut90] R. Hutty, *COBOL 85 programming*, Macmillan, 1990.

[Inf99a] *Informix guide to SQL: Reference*, Informix Press, December 1999.

[Inf99b] *Informix guide to SQL: Syntax*, Informix Press, December 1999.

[Jah99a] J.-H. Jahnke, J. P. Wadsack, *Integration of analysis and redesign activities in information system reengineering*, In Proceedings of the European Conference on Software Maintenance and Reengineering, Amsterdam, 1999.

[Jah99b] J.-H. Jahnke, J. P. Wadsack, *Varlet: Human-Centered Tool Support for Database Reengineering*, in Proceedings of Workshop on Software-Reengineering, May 1999.

[Jar92] M. Jarke, J. Mylopoulos, J. W. Schmidt, Y. Vassiliou, *DAIDA: An Environment for Evolving Information Systems*, ACM Transactions on Information Systems, vol. 10, no. 1, pp. 1-50, January 1992.

[Jar94a] M. Jarke, H. W. Nissen, K. Pohl, *Tool integration in evolving information systems environments*, in Proceedings of GI Workshop Information Systems and Artificial Intelligence: Administration and Processing of Complex Structures, Hamburg, February 1994.

[Jar94b] M. Jarke, K. Pohl, R. Dömges, S. Jacobs, H. W. Nissen, *Requirements Information Management: The NATURE Approach*, Ingénierie des Systèmes d'Informations - Special Issue on Requirements Engineering, vol. 2, no. 6, 1994.

[Jen94] C. Jensen and al., *A consensus glossary of temporal database concepts*, in Proceedings of the International Workshop on an Infrastructure for Temporal Databases, Arlington, 1994.

[Jor92] M. Joris, R. Van Hoe, J.-L. Hainaut, M. Chandelon, C. Tonneau, F. Bodart and al., *PHENIX: methods and tools for database reverse engineering*, in Proceedings of the International Conference on Software Engineering and Applications, Toulouse, December 1992.

[Kaf87] D. Kafura, G. Reddy, *The use of software complexity metrics in software maintenance*, IEEE Transactions on Software Engineering, vol. SE-13, no. 3, pp. 335-343, 1987.

[Kan84] N. Kano, N. Seraku, F. Takahashi, S. Tsuji, *Attractive and must-be quality*, (Japanese) Hinshitsu, vol. 14, no. 2, pp. 39-48, 1984.

[Kem97] C. F. Kemerer, S. Slaughter, *Determinants of Software Maintenance Profiles: An Empirical Investigation*, Journal of Software Maintenance, vol. 9, pp. 235-251, 1997.

[Kim88] W. Kim, H.-T. Chou, *Versions of Schema for Object-Oriented Databases*, in Proceedings of the International Conference on Very Large DataBases, Los Angeles, pp. 148-159, August 1988.

[Kob86] I. Kobayashi, *Losslessness and semantic correctness of database schema transformation: another look of schema equivalence*, Information Systems, vol. 11, no. 1, pp. 41-59, 1986.

[Kor91] H. F. Korth, A. Silberschatz, *Database system concepts (second edition)*, McGraw-Hill, 1991.

[Koz87] W. Kozaczynski, L. Lilien, *An Extended Entity-Relationship (E2R) Database Specification and its Automatic Verification and Transformation into the Logical Relational Design*, in Proceedings of the International Conference on Entity-Relationship Approach, North-Holland, pp. 533-549, 1987.

[Kro92] D. Kroenke, *Database processing: fundamentals - design - implementation (fourth edition)*, MacMillan, 1992.

[Kro95] J. Krogstie, *Conceptual Modeling for Computerized Information Systems Support in Organisations*, PhD Thesis, NTH-University of Trondheim, 1995.

[Lar93] *Le petit Larousse (grand format)*, Larousse, 1993.

[Lel59] F. Lelotte, S. J., *Etincelles : choix de 1200 pensées recueillies et classées*, Editions Foyer Notre-Dame, 1959.

[Leo99] M. Léonard, *M7 : une approche évolutive des Systèmes d'Information*, Tutorial du Congrès INFORSID, La Garde, juin 1999.

[Lie80] B. P. Lientz, E. B. Swanson, *Software maintenance management*, Reading MA: Addison-Wesley, 1980.

[Lin96] T. W. Ling, M. L. Lee, *View update in entity-relationship approach*, Data & Knowledge Engineering, North-Holland, no. 19, pp. 135-169, 1996.

[Liu94a] C.-T. Liu, S.-K. Chang, P. K. Chrysanthis, *Database schema evolution using EVER diagrams*, in Proceedings of Workshops on Advanced Visual Interface, 1994.

[Liu94b] C.-T. Liu, P. K. Chrysanthis, S.-K. Chang, *Database schema evolution thought the specification and maintenance of changes on entities and relationships*, in Proceedings of the International Conference on Entity-Relationship Approach, LNCS 881, pp.132-149, December 1994.

[Loc97] D. Lockman, *Oracle 8 : Développement de bases de données*, Macmillan, 1997.

[Lyn90] E. Lynch, *Understanding SQL*, Macmillan, 1990.

[Mac96] L. A. Macaulay, *Requirements engineering*, Springer-Verlag, 1996.

[Mar87] V. M. Markowitz, J. A. Makowsky, *Incremental reorganization of relational databases*, in Proceedings of the International Conference of Very Large DataBases, Brighton, 1987.

[Mar93] J. Martin, *Principles of object-oriented: analysis and design*, Prentice-Hall, 1993.

[Mar94] C. Marée, G. Ledant, *SQL2 : Initiation, Programmation*, Armand Colin, 1994.

[Mck90] E. McKenzie, R. Snodgrass, *Schema evolution and the relational algebra*, Information Systems, vol. 15, no. 2, pp. 207-232, 1990.

[Mel93] J. Melton, A. R. Simon, *Understanding the new SQL: a complete guide*, Addison-Wesley, 1993.

[Mul97a] P.-A. Muller, *Modélisation objet avec UML*, Eyrolles, 1997.

[Mul97b] C. S. Mullins, *DB2: Developper's guide (third edition)*, Sams publishing, 1997.

[Moo94] D. Moody, G. Shanks, *What Makes a Good Data Model? Evaluating the Quality of Entity Relationship Models*, in Proceedings of the International Conference on the Entity-Relationship Approach, pp. 94-111, 1994.

[Nan96] D. Nancy, B. Espinasse, *Ingénierie des systèmes d'information Merise, Deuxième génération*, Sybex, 1996.

[Nav80] S. B. Navathe, *Schema Analysis for Database Restructuring*, ACM Transactions on Database Systems, vol. 5, no. 2, June 1980.

[Ngu89] G. T. Nguyen, D. Rieu, *Schema evolution in object-oriented database systems*, in Data & Knowledge Engineering, Elsevier Science, no. 4, 1989.

[Nij89] G. M. Nijssen, T. A. Halpin, *Conceptual schema and relational database design - a fact-oriented approach*, Prentice-Hall, 1989.

[One94] P. O'Neil, *Database: Principles - Programming - Performance*, Morgan Kaufmann, 1994.

[Ora97] *Oracle: SQL Reference Release 8.0*, Oracle Corporation, December 1997.

[Owe96] K. T. Owens, *Building intelligent databases with Oracle PL/SQL, triggers, & stored procedures*, Prentice-Hall, 1996.

[Par72] D. Parnas, *On the Criteria to be used in Decomposing Systems into Modules*, Communications of the ACM, vol. 15, no. 12, pp. 1053-1058, 1972.

[Par79] Parnas, D., *Designing Software for Ease of Extension and Contraction*, IEEE Transactions on Software Engineering, Mars 1979.

[Pau93] M. C. Paulk, B. Curtis, M. B. Chrissis, C. V. Weber, *Capability Maturity Model for Software (Version 1.1)*, technical document (CMU/SEI-93-TR-24), Software Engineering Institute, Carnegie Mellon University, February 1993.

[Per81] R. Perron, *Design guide for CODASYL database management systems*, Q.E.D. Information Sciences, 1981.

[Pot88] C. Poots, G. Bruns, *Recording the reasons for design decisions*, in Proceedings of the International Conference on Software Engineering, IEEE, 1988.

[Pro94] H. A. Proper, T. P. van der Weide, *EVORM: a conceptual modelling technique for evolving application domains*, Data & knowledge Engineering, no. 12, pp. 313-359, 1994.

[Pro97] H. A. Proper, *Data schema design as a schema evolution process*, Data & knowledge Engineering, no. 22, pp. 159-189, 1997.

[Ram97] B. Ramesh, C. Stubbs, T. Powers, M. Edwards, *Requirements Traceability: Theory and Practice*, Annals of Software Engineering, no. 3, p.397-415, 1997.

[Rod92a] J. F. Roddick, *SQL/SE - A Query Language Extension for Databases Supporting Schema Evolution*, SIGMOD Record, vol. 21, no. 3, 1992.

[Rod92b] J. F. Roddick, *Schema evolution in Database Systems - An annotated bibliography*, SIGMOD Record, vol. 21, no. 4, pp. 34-40, December 1992.

[Rod93] J. F. Roddick, N. G. Crashe, T. J. Richards, *A Taxonomy for Schema Versioning Based on the Relational and Entity Relationship Models*, in Proceedings of the International Conference on the Entity-Relationship Approach, Arlington, 1993.

[Rod95] J. F. Roddick, *A survey of schema versioning issues for database systems*, Information and Software Technology, vol. 37, no. 7, pp. 383-393, 1995.

[Rol93] C. Rolland, *Modeling the Requirements Engineering Process*, in Proceedings of the European-Japanese Seminar on Information Modelling and Knowledge Bases, Budapest, June 1993.

[Rol97] D. Roland, J.-L. Hainaut, *Database Engineering Process Modeling*, in Proceedings of the International Conference on the Many Facets of Process Engineering, Gammarth, September 1997.

[Rol99a] D. Roland, J.-L. Hainaut, J. Henrard, J.-M. Hick, V. Englebert, *Database Engineering Process History*, in Proceedings of the International Conference on the Many Facets of Process Engineering, Gammarth, May 1999.

[Rol99b] D. Roland, J.-L. Hainaut, *Database Engineering Process Modeling: the MDL Language*, technical document, Institut d'Informatique, FUNDP, October 1999.

[Rol00] D. Roland, J.-L. Hainaut, J.-M. Hick, J. Henrard, V. Englebert, *Database engineering processes with DB-Main*, in Proceedings of the European Conference on Information Systems, Vienne, June 2000.

[Ros87] A. Rosenthal, D. Reiner, *Theoretically Sound Transformations for Practical Database Design*, in Proceedings of the International Conference on Entity-Relationship Approach, Elsevier Science, pp. 115-131, New-York, 1987.

[Rug95] S. Rugaber, *Program Comprehension*, in Encyclopedia of Computer Science and Technology, Marcel Dekker Inc., vol. 35, no. 20, pp. 341-368, 1995.

[Sab92] N. Sabanis, N. Stevenson, *Tools and Techniques for Data Remodelling COBOL Applications*, in Proceedings of the International Conference on Software Engineering and Applications, Toulouse, 1992.

[Sch87] Schneidewind, N. F., *The State of Software Maintenance*, IEEE Transactions on Software Engineering, vol. SE-13, no. 3, pp. 303-310, 1987.

[Shn82] B. Shneiderman, G. Thomas, *An architecture for automatic relational database system conversion*, ACM Transactions on Database Systems, vol. 7, no. 2, pp. 235-257, 1982.

[Sim91] G. C. Simsion, *Creative data modelling*, in Proceedings of the International Conference on Entity-Relationship Approach, San Francisco, 1991.

[Som92] I. Sommerville, *Software engineering (fourth edition)*, Addison-Wesley, 1992.

[SQL92] *Database language SQL*, International Organization for Standardisation, technical document (ISO/IEC 9075), 1992.

[Ser99] *Transact-SQL Overview*, Microsoft Corporation, 1999.

[Swa76] E. B. Swanson, *The dimensions of maintenance*, In Proceedings of the International Conference on Software Engineering, pp. 492–497, San Francisco, August 1976.

[Syb00] *Adaptive Server Anywhere Reference*, Sybase Inc., 2000.

[Ulr97] W. M. Ulrich, I. S. Hayes, *The year 2000 software crisis : Challenge of the century*, Prentice-Hall, 1997.

[Uml99] *UML - OMG Unified Modeling Language Specification, Version 1.3 alpha R2*, OMG Inc., 1999.

[vBo92] P. van Bommel, T. P. van der Weide, *Towards Database Optimization by Evolution*, in Proceedings of the International Conference on Information Systems and Management of Data, pp. 273-287, Bangalore, July 1992.

[vBo93] P. van Bommel, *A Randomised Schema Mutator for Evolutionary Database Optimisation*, The Australian Computer Journal, vol. 25, no. 2, pp. 61-69, May 1993.

[vBo94] P. van Bommel , *Database Design Modifications based on Conceptual Modelling*, in Information Modelling and Knowledge Bases: Principles and Formal Techniques, IOS Press, pp. 275-286, Amsterdam, 1994.

[vdL93] R. van der Lans, *Introduction to SQL (second edition)*, Addison-Wesley, 1993.

[Ver97] J. Verelst, *Factors in conceptual requirements modeling influencing maintainability of information system: an empirical approach*, in Proceedings of the Doctoral Consortium of the IEEE International Symponium on Requirements Engineering, Annapolis, January 1997.

[Vzu91] H. Van Zuylen, *The REDO Handbook - A compendium of reverse engineering for Software Maintenance*, technical document, November 1991.

[Wag95] S. Wagner, L. Schmit, *Schema modification propagation for relational database applications*, mémoire de troisième maîtrise, Institut d'Informatique, FUNDP, Namur, 1995.

[Wed99] L. Wedemeijer, *Semantical change patterns in the conceptual schema*, in Proceedings of Entity-Relationship Workshop, Paris, 1999.

[Wei84] M. Weiser, *Program Slicing*, IEEE Transactions on Software Engineering, vol. 10, pp. 352-357, 1984.

[Wie95] G. Wiederhold, *Modeling and System Maintenance*, in Proceedings of the International Conference on Object-Oriented and Entity-Relationship Modeling, Springer-Verlag, Berlin, 1995.

[Zdo86] S. B. Zdonik, *Version Management in an Object-Oriented Database*, in Proceedings of the International Workshop on Advanced Programming Environments, LNCS vol. 244, pp. 405-422, Trondheim, June 1986.