HAL
open science

Gestion d'un évolution du schéma d'une base de données à objets: une approche par compromis

Boualem Benatallah

▶ To cite this version:

Boualem Benatallah. Gestion d'un évolution du schéma d'une base de données à objets: une approche par compromis. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1996. Français. NNT : . tel-00345357

HAL Id: tel-00345357
https://theses.hal.science/tel-00345357v1

Submitted on 9 Dec 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée par

BENATALLAH Boualem

Pour obtenir le titre de

Docteur de l'UNIVERSITÉ JOSEPH FOURIER GRENOBLE I

(arrétés ministriels du 5 juillet 1984 et du 30 mars 1992)

Spécialité

INFORMATIQUE

GESTION DE L'ÉVOLUTION DU SCHÉMA D'UNE BASE DE DONNÉES À OBJETS : une approche par compromis

Date de soutenance : 4 Mars 1996

Composition du jury :

Président :	Mr. Farid OUABDESSELAM
Rapporteurs :	Mme. Anne DOUCET Mr. Jacques LEMAITRE
Examinateurs :	Mr. Michel ADIBA Melle. Marie-Christine FAUVET

Thèse préparée au sein du laboratoire de Langages, Systèmes et Réseaux, LSR-IMAG

THESE

présentée par

BENATALLAH Boualem

Pour obtenir le titre de

Docteur de l'UNIVERSITÉ JOSEPH FOURIER GRENOBLE I

(arrêtés ministriels du 5 juillet 1984 et du 30 mars 1992)

Spécialité

INFORMATIQUE

GESTION DE L'ÉVOLUTION DU SCHÉMA D'UNE BASE DE DONNÉES À OBJETS : une approche par compromis

Date de soutenance : 4 Mars 1996

Composition du jury :

Président :	Mr. Farid OUABDESSELAM
Rapporteurs :	Mme. Anne DOUCET Mr. Jacques LEMAITRE
Examinateurs :	Mr. Michel ADIBA Melle. Marie-Christine FAUVET

Thèse préparée au sein du laboratoire de Langages, Systèmes et Réseaux, LSR-IMAG

A la mémoire de mes parents.

A ma famille.

Je leur dédie ce travail.

RÉSUME

Dans cette thèse, nous ne intéressons au problème de l'évolution des schémas pour les bases de données à objets. Nous considérons d'abord les solutions proposées pour la gestion de l'évolution de schéma de bases de données à objets. Nous proposons une classification des approches existantes. Pour chacune de ces approches nous décrivons son principe, les mécanismes d'évolution associés, ainsi que les produits et les prototypes qui l'implantent. Nous analysons ces travaux en soulignant les avantages et les inconvénients de chaque approche.

Nous présentons ensuite notre approche. D'une part, cette approche propose un cadre qui permet de combiner les fonctionnalités de la modification et du versionnement pour une meilleure gestion de l'évolution de schéma. D'autre part, elle offre à l'utilisateur un langage permettant de décrire les liens entre les différents états de la base de données afin de traduire le plus fidèlement possible les évolutions du monde réel.

Le versionnement de schéma évite la perte d'informations et assure que les anciens programmes d'applications continuent de fonctionner. Cependant, le nombre de versions peut devenir important ; ce qui rend complexe leur gestion. Notre approche permet de limiter le nombre de versions : (1) l'évolution d'un schéma est traduite par sa modification si l'évolution est non-soustractive (ne provoque pas la suppression de propriétés) ou si l'utilisateur le décide, (2) La technique utilisée pour adapter les instances au schéma après l'évolution, est basée sur la caractérisation de l'importance de l'existence en tant que telle d'une version d'objet. Ainsi, le nombre de versions est limité à celles qui sont fréquemment accédées par des programmes, (3) la possibilité donnée à l'administrateur de réorganiser la base de données lui permet de supprimer des versions historiques du schéma.

Mots-clés : Base de données, évolution de schéma, modification, version, pertinence d'une classe, conversion, émulation, schéma de correspondances

ABSTRACT

In this thesis, we are interested in the problem of schema evolution in object-oriented databases. First we consider proposed solutions. We propose a classification of existing schema evolution management approaches. For each approach, we describe its principle, associated evolution mechanisms, as well as prototype and commercial systems that implements it. We analyse these proposals to show their merits and limitations.

Then, we present our approach. Firstly, this approach combines *schema modification* and *schema versioning* approaches. Our aim is to effect a compromise between the functionality of these approaches for efficient management of schema evolution : (1) the schema change process combines *modification* and*versioning*, and (2) the database change (instances adaptation) process combines *conversion*, *versioning* and *emulation* (screening). Secondly, it provides a language for specifying the relationships between the different states of the database to capture accurately the semantic of changes to the schema.

Schema versioning mechanism avoids the loss of information and allows programs, acting on a database, to continue to function without alteration after change to the database schema. However, the number of versions may become considerable ; consequently, the management of versions is complex : (1) a schema evolution is translated to the modification of the schema if it is not *subtractive*. (i.e., involves information removal) or the user imposes "modification" mode, (2) the technique followed for supporting instances adaptation after a schema evolution is based on the characterisation of the importance of object version availability. The number of object versions is limited to those which are frequently accessed, (3) we have proposed an operation which allows the *immediate transformation of database*. The database administrator may launch this operation when he judge necessary (e.g., space optimisation). The database transformation delete classes and historical schema versions, which are not pertinent (or judged not pertinent by the database administrator) for the applications.

Key words : Database, schema evolution, modification, version, class pertinence level, conversion, screening, correspondences schema.

Je tiens à remercier

Mr Farid OUADBESSELAM, Professeur à l'université Joseph Fourier, qui me fait l'honneur de présider ce Jury.

Mme Anne DOUCET, Professeur à l'université de Paris 6, ainsi que Mr Jacques LEMAITRE, Professeur à l'université Toulon-Var, d'avoir bien voulu accepter de juger ce travail et de faire partie du jury de cette thèse. J'ai beaucoup apprécié les remarques et les critiques constructives qu'ils ont fait sur ce manuscrit.

Mr Michel ADIBA, Professeur à l'université Joseph Fourier, qui m'a accueilli dans son équipe et a dirigé cette thèse. Sa présence exigeante et interrogeante m'a permet d'évoluer dans mon travail. Ses critiques constructives et pertinentes m'ont été très bénéfiques pour faire aboutir ce travail. Je lui suis profondément reconnaissant.

Melle Marie-Christine FAUVET, Maître de Conférences à l'université Joseph Fourier, qui a co-dirigé cette thèse. L'intérêt extrême qu'elle a porté à mon travail, ses nombreuses critiques constructives, les discussions tout le long de ce travail, ses encouragements incessants, sa rigueur, sa patience, et ses relectures soigneuses du document m'ont été très bénéfiques. Qu'elle trouve ici ma profonde gratitude.

Mr Yves CHARAMELLA, Professeur à l'université Joseph Fourier et directeur du CLIPS, ainsi que Mr Jacques MOSSIERE, Professeur à l'INPG et directeur du LSR, pour l'accueil au sein du LGI et du LSR.

Je remercie également Mr Pierre-Claude SCHOLL, pour l'intérêt qu'il a porté à mon travail, les discussions fructueuses avec lui m'ont été très bénéfiques. Mr Christian ESCULIER, pour l'intérêt qu'il a porté à mon travail et sa lecture soigneuse du document. Son point de vue et ses nombreuses critiques constructives sur ce manuscrit m'ont beaucoup apporté. Melle Christine COLLET, pour sa gentillesse, sa disponibilité et son aide.

Je remercie spécialement Mr Zahir TARI, Senior Lecturer à RMIT (Melbourne, Australie), pour l'intérêt qu'il a porté à mon travail. Je lui suis très reconnaissant, pour son soutien tant scientifique que moral, de m'avoir aidé à franchir des étapes importantes.

Je remercie Mr Nouredine BELKHATIR pour son aide, Mr. Mohamed NACER. Mme Liliane DIAGICOMO et Mme Martine PERNICE, pour leur gentillesses.

Je remercie les membres du STORM, en particulier Jean-Pierre, Monique, Claudia, Jean-François, Celso, Javam, Hervet, Agnes, Françoise et Theiry.

Je voudrai enfin remercier tout ceux qui, de prés ou de loin, m'ont apporté leur soutien moral. Je remercie mes ami(e)s, en particulier, Ali, Kader et Moussa (Saida), les frères Diffallah (Ziama), Maria (Tak..Tak), Christine, Giannoula (Greeka), Ahmed (Cheick du thé), El Hadj (El Monchard), Mourad (Da..Azul), Rachid.

Table des matières

Introduction **1**

I. Le problème de l'évolution du schéma d'une base de données à objets . **9**
I.1. Bases de données à objets 9
I.1.1. Domaines d'applications 10
I.1.2. Concepts 11
I.2. Schéma d'une base de données 13
I.2.1. Rôle du schéma d'une base de données 14
I.2.2. Besoins en évolution de schéma 18
I.2.3. Problèmes posés par l'évolution de schéma 21
I.2.4. Conclusion 23
I.3. Approche objet et évolution de schéma 24
I.3.1. Sous-classe 25
I.3.2. Classe paramètrée 26
I.3.3. Type union 27
I.3.4. Conclusion 28

II. État de l'art **29**
II.1. Introduction 29
II.2. Critères de classification des approches 30
II.3. Approches fondées sur la modification 32
II.3.1. Changement du schéma 32
II.3.2. Adaptation des instances 35
II.4. Approches fondées sur les versions 38
II.4.1. Changement du schéma 39
II.4.2. Adaptation des instances 41
II.5. Approches fondées sur les vues 45
II.5.1. Mécanisme de vue 45
II.5.2. Vue & évolution de schéma 45
II.6. Travaux similaires dans d'autres domaines 48
II.6.1. Éditeurs syntaxiques 48
II.6.2. Édition structurée de documents 49
II.6.3. Représentation de connaissances par objets 50
II.6.4. Langages de programmation à objets 50
II.7. Conclusion 52
II.7.1. Tableaux récapitulatifs 52

II.7.2. Synthèse 54

III. Le modèle de versions : concepts et évolution de schéma 57
III.1. Introduction 57
III.2. Modèle de versions 58
III.2.1. Concepts généraux 58
III.2.2. Schéma - Version de schéma 59
III.2.3. Objet - Version d'objet 64
III.2.4. Liaison entre une version d'objet et programme 67
III.3. Un modèle formel d'une base de données 67
III.3.1. Classes, types et opérations 67
III.3.2. Schéma et instance 69
III.3.3. Cohérence structurelle de la base de données 72
III.4. Le problème de l'évolution de schéma 76
III.4.1. Le problème de la cohérence de structure d'un schéma 77
III.4.2. Le problème de la cohérence d'une instance vis-à-vis 78
III.4.3 Réorganisation d'une base de données 80
III.5. Conclusion et perspectives 84

IV. Un compromis : modification et versionnement du schéma 87
IV.1. Introduction 87
IV.2. Processus de changement du schéma 88
IV.2.1. Catégories des opérations sur le schéma 89
IV.2.2. Modification & version 91
IV.3. Adaptation des instances 91
IV.3.1. Terminologie 92
IV.3.2. Niveaux de pertinence d'une classe 94
IV.3.3. Technique d'adaptation des instances 98
IV.4. Réorganisation de la base de données 105
IV.4.1. Suppression d'une classe ou d'une version du schéma 106
IV.4.2. Opération de réorganisation 112
IV.5. Conclusion et perspectives 116

V. Langage de description d'une évolution de schéma 119
V.1. Introduction 120
V.2. Exemples 122
V.3. Description de la sémantique de l'évolution 130
V.3.1. Principe de base 131

V.3.2. Descripteur d'un lien de correspondance 134
V.4. Schéma de correspondances 143
V.4.1 Graphe de dépendances 144
V.4.2. Correction 146
V.5. Cohérence d'une base de données vis-à-vis d'un schéma de correspondances. 150
V.5.1. Le problème 150
V.5.2. Maintien de la cohérence 152
V.6. Conclusion et perspectives 154

VI. Mise en œuvre : le prototype DB_Evolution 157
VI.1. Introduction 157
VI.2. Architecture générale 157
VI.3. Scénario d'une commande d'évolution 160
IV.3.1. Changement du schéma 160
VI.3.2. Réorganisation de la base de données 162
VI.4. Le module noyau d'évolution 164
VI.4.1. Schéma du noyau d'évolution 164
VI.4.2. Utilisation 166
VI.5. Conclusion 166

Conclusion et perspectives 167

Bibliographie 171

Annexes 183

RÉSUME

Dans cette thèse, nous ne intéressons au problème de l'évolution des schémas pour les bases de données à objets. Nous considérons d'abord les solutions proposées pour la gestion de l'évolution de schéma de bases de données à objets. Nous proposons une classification des approches existantes. Pour chacune de ces approches nous décrivons son principe, les mécanismes d'évolution associés, ainsi que les produits et les prototypes qui l'implantent. Nous analysons ces travaux en soulignant les avantages et les inconvénients de chaque approche.

Nous présentons ensuite notre approche. D'une part, cette approche propose un cadre qui permet de combiner les fonctionnalités de la modification et du versionnement pour une meilleure gestion de l'évolution de schéma. D'autre part, elle offre à l'utilisateur un langage permettant de décrire les liens entre les différents états de la base de données afin de traduire le plus fidèlement possible les évolutions du monde réel.

Le versionnement de schéma évite la perte d'informations et assure que les anciens programmes d'applications continuent de fonctionner. Cependant, le nombre de versions peut devenir important ; ce qui rend complexe leur gestion. Notre approche permet de limiter le nombre de versions : (1) l'évolution d'un schéma est traduite par sa modification si l'évolution est non-soustractive (ne provoque pas la suppression de propriétés) ou si l'utilisateur le décide, (2) La technique utilisée pour adapter les instances au schéma après l'évolution, est basée sur la caractérisation de l'importance de l'existence en tant que telle d'une version d'objet. Ainsi, le nombre de versions est limité à celles qui sont fréquemment accédées par des programmes, (3) la possibilité donnée à l'administrateur de réorganiser la base de données lui permet de supprimer des versions historiques du schéma.

Mots-clés : Base de données, évolution de schéma, modification, version, pertinence d'une classe, conversion, émulation, schéma de correspondances

Introduction

L'intégration des bases de données et des langages de programmation à objets a permis l'émergence d'une nouvelle génération de systèmes de gestion de bases de données : les SGBD à objets. Un des objectifs de ces systèmes est de répondre aux besoins de nouveaux domaines d'applications (Bureautique, CAO, géographie, génie logiciel, documentation, etc.). Ces applications avancées sont larges, coopératives et nécessitent en particulier, la manipulation de structures de données complexes et extensibles, le stockage d'un grand nombre d'objets volumineux et parfois distribués. Un point essentiel qui caractérise les applications avancées est la nature dynamique, aussi bien des données modélisées que de leurs structures.

Dans le contexte d'une base de données à objets, une application est composée du schéma de la base (la librairie de classes, les contraintes ou règles de gestion), des objets et des programmes. La spécification et la conception de chacun de ces niveaux d'abstraction est le résultat du processus de développement de l'application. Un tel processus met en jeu de nombreux intervenants (administrateurs, concepteurs, programmeurs, usagers) et une quantité importante d'informations plus ou moins structurées (cahier des charges, documents de spécifications, schéma de la base, etc.).

En général, le thème*évolution* [CGS95, DT87, Mei95] dans le contexte des bases de données couvre les aspects suivants :

• Interopérabilité entre SGBD [Ron94, MBP95] : le but est de permettre la manipulation de données se trouvant dans des bases de données développées indépendamment les unes des autres. On trouve ici, les travaux sur l'intégration des bases autonomes (systèmes fédérées ou multibases) et la migration des applications développées en utilisant un système vers un autre (par exemple, du relationnel à l'objet).

• Évolution d'une base de données [ST94, Rod94] : cela couvre les travaux sur la mise à jour de données/objets, la multi-instanciation des objets (rôles, points de vue), la migration des objets (par exemple, d'une classe vers ses sous classes) et finalement la mise à jour du schéma d'une base de données qui est le sujet de cette thèse.

Problématique

Le schéma d'une base de données est rarement stable. Les outils proposés aux développeurs doivent donc permettre en particulier de supporter l'évolution du schéma de la base de données.

Celui-ci peut subir des changements aussi bien dans la phase de conception de l'application que dans sa phase d'exploitation [APL91] :

• Pendant la phase de conception, le schéma est souvent incomplet. Il est nécessaire de le faire évoluer tout au long du processus de sa conception. Cette évolution se traduit alors par l'affinement de la définition du schéma, la remise en cause de certains choix antérieurs, l'expérimentation de plusieurs choix de conception, ou encore l'intégration de nouvelles contraintes.

• Pendant la phase d'exploitation où l'application est opérationnelle, une évolution traduit souvent une modification plus ou moins importante de la réalité considérée. Il peut s'agir, par exemple, d'une re-formulation des besoins des utilisateurs, de la correction d'une erreur remettant en cause le schéma, ou encore de l'amélioration des performances de l'application.

Une évolution du schéma risque d'introduire des incompatibilités des données et/ou des programmes vis-à-vis du schéma. Elle doit être donc prise en compte en perturbant le moins possible le travail des usagers des applications et en privilégiant la pérennité des données. Dans un contexte à objets, rares sont les systèmes qui offrent une solution générale à ce problème [Bra93, Ben94, BK87, FMZ95a, Odb95]. Cela est dû à la complexité du maintien de la cohérence suite à la propagation des effets de l'évolution. Dans les travaux existants, l'évolution du schéma est étudiée en analysant les conséquences qu'elle engendre à chacun des niveaux des applications : schéma, données et programmes.

• Niveau schéma : à la suite du changement d'une partie du schéma, des modifications peuvent être propagées sur le reste du schéma afin de préserver sa cohérence. Par exemple, l'ajout d'un attribut att dans la classe c provoque son ajout dans toutes les sous-classes de c où aucun attribut de nom att n'existe.

• Niveau données : la cohérence d'une base est spécifiée au moyen de contraintes d'intégrité. Certaines de ces contraintes sont inhérentes au modèle de données ; elles portent sur la correspondance structurelle et sémantique entre les données de la base et le schéma. Une évolution du schéma doit être propagée sur les données de la base afin que celles-ci respectent toujours les contraintes de correspondance structurelle et sémantique entre les données et le schéma. Par exemple, à la suite de l'ajout de l'attribut att dans la classe c, les représentations des objets associés à c et aux classes qui héritent att de c, doivent être modifiées. La modification consiste à ajouter l'attribut att à la représentation de chacun de ces objets et à initialiser sa valeur.

• Niveau programmes : à la suite d'une évolution du schéma, certains programmes d'applications qui utilisent les entités modifiées peuvent être incompatibles avec le schéma. D'où la nécessité de

modifier ces programmes pour les rendre compatibles avec le schéma. Par exemple, supposons que l'opération m est définie dans la classe c (c'est-à-dire, m n'est pas héritée dans c). Donc, à la suite de la suppression de l'opération m dans la définition de c, les programmes qui font référence à m doivent être modifiés.

De nombreuses études ont été menées afin de prendre en compte le problème de l'évolution du schéma [Ben94, Rod92b]. Il existe deux familles importantes d'approches pour gérer l'évolution de schéma : les approches fondées sur la modification [BK87, FMZ95a], les approches fondées sur les versions [Cla92, Odb95, ED95]. Les solutions apportées sont encore partielles et limitatives :

• La modification peut introduire la perte d'informations et/ou l'incompatibilité des programmes qui utilisent l'ancien schéma. Par exemple, la suppression de l'attribut att de la classe c est traduite par la suppression de la représentation de att dans tous les objets concernés (c'est-à-dire, les objets associés à c et aux classes qui héritent att de c). Les programmes et les opérations qui font référence à l'attribut att ne sont plus compatibles avec le schéma. Par conséquent, ils doivent être modifiés ou tout au moins re-compilés. Ce qui, dans certains cas, est inacceptable (par exemple, dans les applications larges et/ou distribuées).

• Une approche fondée sur les versions permet d'éviter les inconvénients de la modification, mais conduit à d'autres problèmes. Particulièrement, le nombre de versions peut être important, dans la mesure où les applications à objets sont de nature évolutive. Du point de vue de l'utilisateur, la navigation dans la hiérarchie de versions devient de plus en plus compliquée. Du point de vue du système, cela impose un coût de gestion supplémentaire dû aux coût considérable de stockage des versions et au coût de maintenance des relations qui les lient.

• Dans les systèmes existants, l'approche de réorganisation de la base de données à la suite de l'évolution de son schéma est systématique. La réorganisation toujours automatique, génère un état de la base qui est cohérent, mais qui, parfois, ne reflète pas l'évolution du monde réel. Par exemple, supposons que la classe c contient l'attribut att_1. L'évolution du schéma consiste à remplacer l'attribut att_1 par l'attribut att_2 qui a une relation avec att_1 (par exemple, pour chaque objet de la classe c, qui est créé avant l'évolution, la valeur associée à l'attribut att_2 est fonction de celle qui est associée à l'attribut att_1). En l'absence de la description de la relation entre les attributs att_1 et att_2, le mécanisme de réorganisation de la base de données, introduit des valeurs par défaut :
- dans une approche qui ne gère pas les versions d'objets, att_1 est remplacé par att_2 dans la représentation de tous les objets concernés. Ensuite, la valeur de att_2 dans ces objets est initialisée par défaut.

- dans une approche qui gère les versions d'objets, pour chaque objet concerné, une nouvelle version est créée. La représentation de cette version contient l'attribut att_2 dont la valeur est initialisée par défaut.

Dans les deux cas, le nouvel état de la base est bien cohérent avec le schéma, mais il ne reflète pas forcément l'évolution du monde réel.

Dans cette thèse, nous développons une nouvelle approche pour la gestion de l'évolution du schéma. Notre objectif peut se décomposer de la manière suivante :

• Trouver un compromis entre les approches fondées sur la modification et celles fondées sur les versions. Ces approches offrent des fonctionnalités complémentaires. Nous voulons combiner ces fonctionnalités pour une meilleure gestion de l'évolution du schéma.

• Offrir à l'utilisateur la possibilité de spécifier les liens entre les différents états de la base de données afin de traduire le plus fidèlement possible les évolutions du monde réel.

Contributions

La contribution essentielle de cette thèse est résumée par les points suivants :

1. Définition d'un modèle de versions [Ben92, BF94]

Notre approche est basée sur les versions. Un schéma peut exister en plusieurs versions. Un objet peut être associé à plusieurs classes ; chacune dans une version différente du schéma. La représentation et le comportement d'un objet sous une classe constitue la version de l'objet sous cette classe. Au moment de sa compilation, un programme est associé à la version la plus récente du schéma, sous laquelle il est ensuite exécuté, jusqu'à une éventuelle re-compilation.

Nous proposons un modèle formel de description des versions d'un schéma et de son instance. Nous considérons pour cela un ensemble réduit des concepts de l'approche à objet mais cependant suffisant dans notre contexte. Ces concepts sont : l'identité d'objet, la relation d'héritage et la relation de référence (composition). Nous étudions le problème de l'évolution du schéma en utilisant ce modèle formel.

2. Proposition d'une approche combinant la modification et les versions pour la gestion de l'évolution du schéma [Ben95a, Ben95b, BF95]

• Le processus du changement de schéma combine la modification et le versionnement du schéma. Une évolution du schéma est traduite par la dérivation d'une nouvelle version du schéma seulement si cela est nécessaire (tracer l'évolution, rendre une modification réversible, assurer la pérennité des objets, assurer la compatibilité de programmes, etc.). Autrement, elle est traduite par la modification du schéma.

• La technique d'adaptation des instances combine la conversion, l'émulation et le versionnement d'objets. Quand un programme ou une requête fait référence à un objet o de la classe c, le système vérifie si l'objet o possède une version sous la classe c. Si oui, l'objet est utilisé par l'application sans aucune transformation. Sinon, une nouvelle version de l'objet o dont la valeur est conforme à la structure de la classe c doit être générée. Cette nouvelle version est *stockée* seulement si son existence en tant que telle est importante, par exemple, pour améliorer les performances ou assurer la pérennité, etc. Autrement, elle est *calculée,* c'est-à-dire qu'elle ne persiste pas.

Pour décider si une version d'objet doit être stockée ou calculée, nous introduisons la notion de *niveaux de pertinence* d'une classe. Une classe peut être *pertinente* ou *obsolète*. Elle est pertinente si elle est associée à la version la plus récente du schéma ou si elle est utilisée dans un nombre de programmes jugé suffisamment élevé par un utilisateur expert (par exemple, l'administrateur de la base de données). Autrement, elle est *obsolète*. Ainsi, la nouvelle version de l'objet o sous la classe c est stockée seulement si c est pertinente.

• Nous proposons une opération qui permet la réorganisation immédiate de la base de données. Lorsque la taille de la base de données est préjudiciable pour les performances du système, l'utilisateur peut choisir de la réorganiser. Le mécanisme associé à la réorganisation supprime les classes et les versions du schéma qui ne sont plus pertinentes (ou jugées non pertinentes) pour les applications. Par exemple, supprimer dans la base de données, les classes qui ne sont pas associées à la version la plus récente du schéma et qui ne sont utilisées par aucune application.

A la suite de la suppression d'une classe, ses instances (qui sont des versions d'objets) peuvent être supprimées ou converties. Une version d'objet est convertie si : (1) l'objet est membre d'autres classes et (2) cette version est la seule version stockée dans l'extension de la classe à supprimer. Autrement, cette version d'objet est supprimée.

Cette approche nous permet de limiter le nombre de versions à celles qui sont nécessaires et par conséquent, de réduire les coûts de stockage des versions et de maintenance des relations entre les versions.

3. Définition d'un langage de description de la sémantique d'une évolution du schéma

Une évolution de schéma dépend souvent de l'application et la sémantique associée peut varier selon le contexte. Les domaines du génie logiciel et des méthodologies à objets sont à l'origine des opérations d'évolution de schéma dites de haut niveau comme par exemple, la fusion de plusieurs classes en une seule, ou l'éclatement d'une classe en plusieurs autres. La réorganisation convenable de la base de données (c'est-à-dire, celle qui reflète le plus possible l'évolution du monde réel) à la suite d'une évolution de haut niveau, demande une analyse très fine des relations entre les classes du schéma avant et après l'évolution. A une ou plusieurs classes avant l'évolution peuvent correspondre une ou plusieurs classes après l'évolution. Ce sont les deux raisons principales pour lesquelles on constate qu'il est préférable, dans certaines situations, que l'utilisateur (par exemple, l'administrateur) doive intervenir lors de la réorganisation de la base de données à la suite de l'évolution de son schéma. Cependant, aucun système à l'heure actuelle ne propose une telle réorganisation.

Nous offrons à un usager la possibilité de décrire une sémantique particulière de l'évolution du schéma. Pour cela, nous définissons un langage qui permet de décrire des relations entre l'état de la base de données avant et après l'évolution. Ces relations sont exploitées par le système lors de la réorganisation de la base de données.

Organisation de la thèse

La thèse est organisée comme suit. Les chapitres I et II analysent la problématique de l'évolution du schéma dans le contexte des bases de données à objets et proposent un état des travaux existants. Les chapitres III, IV et V décrivent l'approche que nous proposons pour la gestion de l'évolution de schéma. La réalisation de certains aspects de notre approche est décrite dans le chapitre VI.

• Le **chapitre I** présente d'une part le rôle du schéma d'une base de données, et d'autre part les besoins et les problèmes posés par l'évolution du schéma.

• Le **chapitre II** présente un état de l'art sur l'évolution du schéma d'une base de données à objets. Nous proposons une classification des approches selon qu'elles sont fondées sur la modification,

sur les versions ou sur les vues. Pour chaque catégorie, nous décrivons son principe, les mécanismes d'évolution associés, ses avantages et ses inconvénients, les produits et les prototypes qui l'implantent. Les travaux similaires dans d'autres domaines sont ensuite présentés.

• Le **chapitre III** présente le modèle de versions à la base de notre approche. Nous discutons de la notion de cohérence structurelle d'une base de données et proposons une formalisation du problème de l'évolution de schéma.

• Le **chapitre IV** décrit notre approche pour la gestion de l'évolution du schéma. Nous montrons les modes possibles de l'évolution du schéma : évolution par *modification* et évolution par *versionnement* du schéma. Nous présentons la technique d'adaptation des instances à la suite de l'évolution de leur schéma. Nous étudions le problème de l'augmentation du nombre de versions et présentons l'opération que nous offrons à un utilisateur pour réorganiser la base de données en supprimant les informations qui ne sont plus pertinentes (ou jugées non pertinentes) pour les applications.

• Le **chapitre V** est consacré à la description de la sémantique d'une évolution du schéma. Nous proposons de décrire des relations entre l'état de la base de données avant et après l'évolution. Nous présentons un langage de description de ces relations ainsi que les règles d'utilisation de ce langage afin de ne pas spécifier de relations contradictoires. Nous discutons, ensuite de la cohérence d'une base de données vis-à-vis des relations décrites par les utilisateurs.

• Le **chapitre VI** est consacré aux aspects de la réalisation des mécanismes d'évolution proposés dans notre approche. Nous décrivons *DB_Evolution*, un prototype qui met en œuvre ces mécanismes. Nous présentons le schéma général de fonctionnement et les modules du système d'évolution.

La conclusion résume les aspects abordés dans la thèse en mettant l'accent sur les apports du travail effectué et propose des perspectives.

Chapitre I

LE PROBLÈME DE L'ÉVOLUTION DU SCHÉMA D'UNE BASE DE DONNÉES À OBJETS

Ce chapitre présente le problème de l'évolution de schéma dans un environnement de bases de données à objets. Après une introduction, en section I.1, de l'approche objet pour les bases de données, nous présentons en section I.2 le rôle du schéma d'une base de données, les besoins en évolution de schéma et les problèmes. La section I.3 aborde la capacité de l'approche objet à supporter l'évolution de schéma.

I.1. Bases de données à objets

L'objectif principal d'un système de gestion de bases de données (SGBD) est la gestion fiable de grandes quantités de données persistantes et partagées [DA82, Ull88]. Plusieurs étapes ont caractérisé l'évolution des SGBD. Les années 60 ont vue l'émergence des SGBD de la première génération (réseaux, hiérarchiques) [COD71]. En 1970, Codd proposa le modèle relationnel de données [Cod70, Ora]. Ce dernier est à la base de la deuxième génération des SGBD (relationnels). Récemment, les besoins des applications avancées (Bureautique, CAO, géographie, génie logiciel, documentation, etc.) ont montré que les SGBD relationnels sont insuffisants par leur pauvreté sémantique et structurelle [Ken79]. La première forme normale est trop restrictive et les langages de requêtes sont insuffisants pour exprimer un calcul général. La troisième génération des SGBD (relationnels étendus, à objets) est le résultat de différents efforts :

• Dans le souci de préserver les acquis du modèle relationnel (simplicité, déclarativité, produits commerciaux), certains ont préféré étendre le modèle relationnel pour satisfaire les besoins en modélisation des applications avancées et la puissance d'expression. *POSTGRES* [RS87, SK91] est un exemple de système relationnel étendu. Dans *POSTGRES*, le modèle relationnel est étendu par l'ajout des types abstraits, de l'héritage d'attributs ou de procédures, des relations emboîtées (NF2), etc.

• L'intégration des bases de données et des langages de programmation à objets a permis l'émergence des SGBD à objets [AC93, BCG*88, BDK92, Jos91]. Ces derniers se prêtent bien à la description des objets complexes et à l'intégration des traitements et des données ; ceci grâce à l'intégration des concepts structurels sur les objets complexes et les modèles sémantiques des bases

de données avec les structures de contrôle des langages de programmation. Dans le but d'améliorer la déclarativité et la puissance des SGBD à objets, des travaux sur l'intégration de la programmation logique et la programmation à objets ont conduit vers les SGBD à objets déductifs. L'approche objet pour les bases de données constitue une technologie prometteuse pour le développement des applications avancées [Kim90, KL89].

Dans un premier temps (paragraphe I.1.1), nous rappelons brièvement les caractéristiques des domaines d'applications qui ont été utilisées comme argument pour l'introduction de l'approche objet. Dans un second temps (paragraphe I.1.2), nous présentons les concepts essentiels de l'approche objet.

I.1.1. Domaines d'applications

Dans les années 80, en plus des applications classiques (par exemple, les applications de gestion), les SGBD ont été utilisés pour développer des applications avancées. Ces dernières sont toujours multi-utilisateurs et manipulent des données qui ont une durée de vie longue, et qui sont parfois distribuées [CK86, ZM90].

Des exemples de domaines d'applications typiques incluent les applications de conception (Computer Aided Design, VLSI, etc.), les applications de génie logiciel (Computer Aided Software Engineering, Software Engineering Environnements), les applications de recherche d'informations (bases de données géographiques, bases de documents, réseaux de télécommunications, etc.), etc [Dal91, DL85, Gan94].

Les caractéristiques de ces applications sont variées. Parmi ces caractéristiques nous citons les suivantes [Bra93, Kim89, Mun93, Nac94, Odb95, Pal89, Tay95] :

• **Complexité des données :** les données manipulées sont liées entre elles par différents types de relations : composition, dépendance, représentation (c'est-à-dire, point de vue ou perspective), généralisation / spécialisation, etc. Par exemple, une voiture est composée de roues, un module C++ inclut un autre module C++ (dépendance), une voiture peut être observée suivant plusieurs points de vue (point de vue client, commercial, etc.), une voiture électrique est une spécialisation d'une voiture, etc.

• **Evolutivité** : une des caractéristiques fondamentales des applications avancées est la nature dynamique des données modélisées et de leur schéma.

Le changement des données ou des relations entre les données est dû à l'évolution de l'application ou aux besoins des utilisateurs. Par exemple, un logiciel a une durée de vie longue et

peut être sujet à des changements profonds au cours de sa vie. En CAO, les utilisateurs ont souvent besoin d'expérimenter plusieurs versions d'un objet avant de choisir celle satisfaisant à leurs besoins.

Le changement du schéma est dû à l'évolution des besoins fonctionnels (par exemple, l'affinement de la modélisation) et non-fonctionnels (par exemple, l'amélioration des performances) de l'application [Ari91, CM92, KK87, Lib92, LM89, McL88, Rod91, SD91].

I.1.2. Concepts

L'approche objet pour les bases de données est le résultat de l'intégration des bases de données et les langages de programmation à objet [ABD*94]. En 1993, le groupe *ODMG* (Object Database Management Group) de l'*OMG* (Object Management Group) proposa un standard pour les SGBD à objets. Le modèle d'objets de ce standard est l'*ODMG* [Cat93]. L'objectif principal est de supporter l'interopérabilité des systèmes.

Dans cette section, nous dégageons les principaux concepts de l'approche objet. Nous nous nous contentons ici de présenter les concepts qui sont pertinents vis-à-vis de notre travail. Notre présentation s'appuie sur [ABD*89, Sch94, Ste89, Wal93] :

• **Fonctionnalités d'un SGBD** : les objets *persistent* après l'exécution du processus de leur création. Un SGBD offre la possibilité du *partage* des objets. Le mécanisme de contrôle de la *concurrence* assure la cohérence des objets stockés dans la base. Il est associé à la notion de *transaction*, qui garantit que la manipulation de la base résulte en un état cohérent [DA82]. Un SGBD est associé à un *langage de requête* (par exemple, le langage *OQL* pour l'*ODMG* [Cat93]). Ce langage est déclaratif, indépendant des applications. Il offre la puissance d'accès associatif (c'est-à-dire, le résultat d'une requête est un ensemble d'objets). Certains SGBD offrent aussi la fonctionnalité de gestion de *versions* [BB88, BH89, OT95].

• **Identité d'objet** : un objet a une identité qui est réalisée par un identificateur immuable fourni par le système. Cette identité reste la même pendant toute la durée de vie de l'objet. Elle est utilisée pour référencer l'objet indépendamment de sa valeur : Deux objets ayant la même valeur sont distinguables par leur identité. La valeur de l'objet peut évoluer de façon autonome. Par exemple, un professeur peut changer d'université mais ne change pas d'identité.

• **Classe / type** : une classe est une abstraction qui permet d'exprimer des propriétés des valeurs et des comportements (les opérations applicables) d'un ensemble d'objets. Donc, une classe est associée d'une part à des propriétés qui décrivent les caractéristiques structurelles et

comportementales des objets, et d'autre part à un ensemble d'objets. Cet ensemble est l'extension de la classe. Une propriété p peut être appliquée à un objet o de la classe c seulement si p est définie ou héritée dans c. Dans certains systèmes, le mot type désigne le niveau intentionnel (c'est-à-dire, la représentation et le comportement) et le mot classe désigne l'extension . La terminologie est variable d'un système à un autre.

• **Encapsulation** : un objet contient de l'information et répond à une liste de messages. L'information contenue dans l'objet est cachée aux autres objets, l'objet est dit *encapsulé*. Le seul moyen pour accéder à l'objet est de lui envoyer un message qui est dans la liste des messages auxquels l'objet peut répondre : son *interface* (une liste de propriétés). L'origine de l'encapsulation est le concept de *type abstrait* en génie logiciel, où la spécification de l'objet (information publique) est séparée de sa réalisation. En bases de données, la motivation pour l'encapsulation est la *protection* de l'information.

• **Héritage** : la relation d'héritage exprime le partage des propriétés entre deux classes. Si la classe c_1 hérite de la classe c_2 (c_1 est une sous-classe de c_2), alors les propriétés de c_2 sont aussi des propriétés de c_1. Elle définit une relation d'*ordre* partiel sur l'ensemble des classes du schéma [CW85]. Deux conséquences en résultent : (1) les objets d'une sous-classe sont compatibles avec les objets de la classe, on dit alors que les classes sont organisées en hiérarchie selon leur *spécification* ; (2) l'extension d'une sous-classe est un sous ensemble de l'extension de la classe, on dit alors que les classes sont organisées selon leur *extension* .

• **Surcharge** : une fonction (ou une opération) a un nom et une réalisation. Un nom est *surchargé* lorsqu'il est associé à plusieurs réalisations. Autrement dit, il y a surcharge si deux fonctions peuvent avoir le même nom dans le même contexte. L'exécution de l'envoi d'un message à un objet correspond à l'occurrence d'un nom. Le problème d'un modèle supportant la surcharge est de décider à quelle opération réfère l'occurrence du nom surchargé. Dans la littérature, on dit aussi résoudre la surcharge. En général, deux modes de liaison d'un nom à une réalisation sont considérés : (1) la *liaison statique* où un nom surchargé est lié à une réalisation au moment de la compilation, c'est-à-dire qu'elle ne dépend que du texte du programme, (2) la *liaison dynamique* où un nom surchargé est lié à une réalisation au moment de l'exécution de l'appel de ce nom, c'est-à-dire elle dépend aussi des données manipulées par le programme. La liaison dynamique est souvent combinée avec la *re-définition* des opérations dans les sous-classes.

• **Polymorphisme** : une *variable est polymorphe* lorsqu'on peut lui associer des objets de différentes classes. On dit qu'une *opération est polymorphe* si elle possède des arguments polymorphes [Car89, Con91, CW85].

Notons que l'approche à objet donne une certaine satisfaction vis-à-vis de la première caractéristique des applications avancées (Cf. Section I.1.1), dans le sens où elle offre des concepts permettant de capturer la sémantique des relations souvent complexes entre les données [Bra93].

En ce qui concerne la deuxième caractéristique (c'est-à-dire, l'évolutivité), nous consacrons la suite de ce chapitre à l'évolution du schéma. L'évolution des objets comme un besoin des applications est un autre sujet de recherche et de développement en bases de données [ABG*93, Agr90, Ara90, Ara92, FD91, Fau89, Fau92, FS95, Gan94, GJ94, MMW94, Pal89, Sci94, Su91, TOC93]. Nous ne le traitons pas ici.

I.2. Schéma d'une base de données

Une base de données est un ensemble structuré de données. Ces données sont une représentation de l'image informatique d'une partie du *monde réel*. Une application de bases de données est un ensemble de programmes (logiciels) qui réalisent des services d'un système d'information. Ces programmes manipulent les données d'une ou plusieurs bases, au travers d'un SGBD [Sch94].

Une bonne organisation des données d'une base est essentielle pour permettre leur manipulation (c'est-à-dire, recherche ou mise à jour) et pour assurer leur cohérence. Par exemple, les documents de la bibliothèque de l'IMAG sont rangés dans des répertoires selon des règles strictes. Les documents sont arrangés selon leurs catégories (c'est-à-dire, ouvrages, revues ou actes de conférences). Chaque catégorie est rangée par spécialité, etc. La structure organisationnelle des données d'une base est appelée le *schéma* de la base [Mon93].

Des avantages certains découlent de la notion de schéma d'une base de données, dont les plus importants sont l'organisation des données et la description séparée de la représentation physique d'une donnée et de sa structure. Des problèmes sont posés par l'évolution du schéma. La plupart de ces problèmes viennent du rôle important du schéma. Dans cette section nous abordons les questions suivantes :

- Quel est le rôle du schéma d'une base de données ?
- Pourquoi est-il indispensable de supporter l'évolution de schéma ?
- Quels sont les problèmes posés par l'évolution de schéma ?
- Les concepts de l'approche à objet sont-ils suffisants pour supporter l'évolution de schéma ?

I.2.1. Rôle du schéma d'une base de données

Le monde réel est constitué d'éléments ("une chose", "un événement", "une idée", etc.). Nous reprenons la terminologie utilisée dans [Odb95, Sch94] pour parler du monde réel. Ainsi, nous utilisons le terme *entité* pour désigner un élément. Une entité peut être *concrète*, dans ce cas elle a une réalité physique ou morale ("une personne", "une université", etc.). Elle peut être aussi *abstraite*, dans ce cas elle est utilisée pour caractériser des entités concrètes ("un chiffre", "une couleur", etc.). Pour simplifier le monde réel, on regroupe les entités de même nature dans un ensemble d'entités. Par exemple, on regroupe les entités "Adiba", "Giraudin", "Scholl" dans l'ensemble d'entités "Professeur".

En pratique, une partie seulement du monde réel représentant un intérêt particulier est modélisée. Par exemple, on s'intéresse au service du personnel de l'université de Grenoble. On parle alors de l'univers du discours (Cf. Figure I.1).

Une base de données est la matérialisation informatique des informations de l'univers du discours, c'est-à-dire, les entités, les concepts et leurs associations. Ces informations sont alors accessibles par les programmes d'applications. Le schéma de la base de données spécifie les données qui pourront être matérialisées et leurs associations. Il est défini par le concepteur de la base, en utilisant les concepts du modèle de données (en objet : classe, héritage, etc.). Autrement dit, il représente une interprétation particulière (celle du concepteur), exprimée en termes de concepts d'un modèle de données.

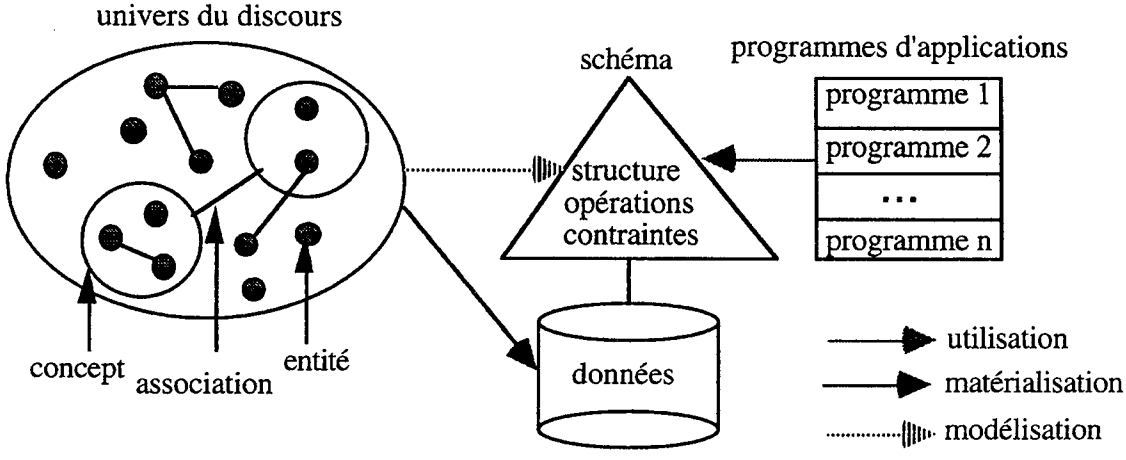


FIG. I.1 : rôle du schéma

Nous résumons le rôle du schéma dans les points suivants :

• **Description et organisation des données** : le schéma décrit les caractéristiques et les liens que les données peuvent avoir. Des structures de données (types pré-définis, types construits, etc.) sont utilisées à cet effet.

• **Restriction des erreurs d'exécution :** le schéma est l'unité de communication entre les programmes d'applications et une base de données. Autrement dit, les programmes manipulent les données d'une base au travers de son schéma. Ainsi, dans un programme, les expressions sont typées, d'où le problème de la *compatibilité* (ou sûreté de typage) d'un programme avec le schéma. La vérification de la compatibilité d'un programme avec le schéma est statique (c'est-à-dire, au moment de la compilation) afin d'éviter toute erreur au moment de l'exécution. De nombreuses études ont été faites sur ce sujet [CLZ91, Con91, RW92, Wal93].

• **Protection de données** : l'encapsulation est un mécanisme offrant un premier niveau de protection des données. Certains modèles proposent en plus le concept de vue permettant la restriction de la visibilité de la base de données [Run92, SDA94].

• **Intégrité des données** : il est également possible de spécifier dans le schéma des contraintes sur les données de la base. Le maintien de ces contraintes assure l'intégrité de la base de données au cours de son évolution [BDP93, BDS95].

Dans la suite de cette section, nous détaillons les notions qui sont liées à notre étude. Nous discutons de la cohérence d'une base de données à objets et de la compatibilité d'un programme d'application.

Définitions

Nous rappelons qu'une base de données à objets consiste en un schéma et une instance de ce schéma. En résumé, le schéma est un ensemble de classes, structuré par les relations d'héritage et de référence. Une classe consiste en une interface et une réalisation. L'interface contient les propriétés publiques. La réalisation contient les propriétés privées et le code des opérations. L'instance est un ensemble d'objets des classes du schéma. Les programmes d'application manipulent les objets de la base au travers du schéma (Cf. Figure I.1). Dans un programme, un objet ne peut être manipulé que via l'interface définie dans sa classe.

I.2.1.1 Cohérence du schéma

Deux notions sont introduites pour définir la cohérence d'un schéma : la cohérence de structure et la cohérence de comportement.

• ***Cohérence de structure*** [BK87, Del92, DZ91, PO95, Zic92] : cette notion concerne la partie statique du schéma. Elle est spécifiée par un ensemble d'invariants. En général, un schéma satisfait la cohérence de structure si la hiérarchie de classes est un graphe orienté, connexe et sans cycle, si les noms des propriétés (attributs ou opérations) d'une classe sont distincts, si les classes héritent de toutes les propriétés de leurs super-classes et si les attributs (opérations) vérifient les règles de compatibilité de types (signatures).

Exemple 1. Considérons les schémas des figures I.2.a et I.2.b. Le schéma de la figure I.2.a satisfait tout les invariants du schéma ; il est donc structurellement cohérent. Dans le schéma de la I.2.b., la classe Etudiant est isolée (c'est-à-dire, elle n'a pas de super-classes). Par conséquent, ce schéma n'est pas structurellement cohérent (le graphe d'héritage n'est pas connexe).

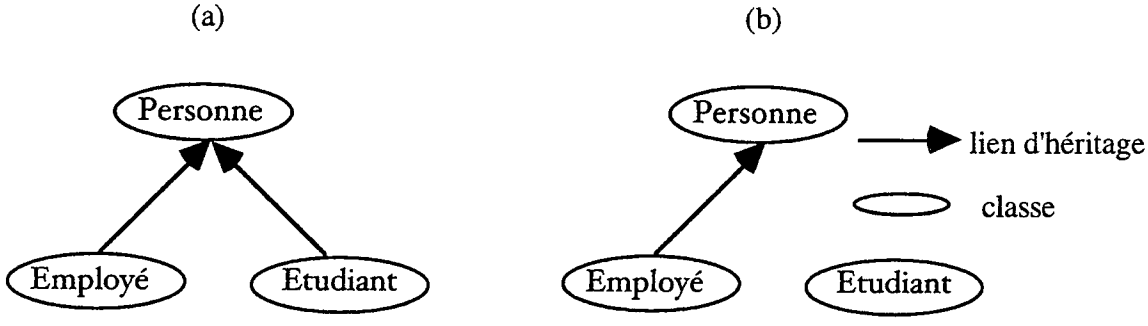


FIG. I.2 : cohérence de structure

• ***Cohérence de comportement*** [AKW90, CCL*93, CLZ91, HTY89, LT94, MNS94] : cette notion concerne la partie dynamique du schéma. En général, un schéma satisfait la cohérence de comportement si le code des opérations est compatible avec leur signatures (définitions) et si l'exécution du code d'une opération ne génère pas d'erreurs. Les résultats sont le plus souvent obtenus de manière ad-hoc. En général, on considère les contraintes suivantes :

- **Fermeture de types :** le type d'un attribut, d'un paramètre formel ou d'une variable locale référencée dans une opération doit être un type du schéma (un type de base, une classe, etc.).

- **Fermeture de références :** un objet nommé dans le code source d'une opération doit être soit un paramètre formel, soit un attribut soit une variable locale.

- **Existence des opérations** : les opérations appliquées à un objet nommé dans le code source d'une opération doivent être des opérations définies ou héritées dans la classe de l'objet nommé.

- **Compatibilité des appels** : l'appel d'une opération (c'est-à-dire, instancier les arguments de l'opération par des valeurs et l'appliquer à un objet) doit être compatible avec sa signature. C'est-à-dire, les types de valeurs des arguments sont des sous-types de ceux des paramètres formels associés.

Exemple 2. Reprenons le schéma donné dans la figure I.2.a. Supposons que la classe Employé contient l'attribut LeSalaire_mensuel (le salaire mensuel d'un employé) et l'opération LeSalaire_Annuel (le salaire annuel d'un employé). La signature et le code source de l'opération LeSalaire_Annuel sont donnés dans la figure I.3 (nous utilisons la syntaxe du langage *O2C* [OT94]). Le schéma satisfait la cohérence de comportement car les contraintes décrites ci-dessus sont vérifiées sur l'opération LeSalaire_Annuel.

```
method LeSalaire_Annuel : integer

method body LeSalaire_Annuel in class Employé {
return ((self->LeSalaire_mensuel)*12);
}
```

FIG. I.3 : opération LeSalaire_Annuel

I.2.1.2. cohérence de l'instance

L'instance d'une base de données est cohérente vis-à-vis de son schéma si la représentation de chaque objet de cette instance est conforme à la structure de sa classe [TK91, VH91]. Autrement dit, la valeur de chaque attribut dans la représentation d'un objet de l'instance, doit être un élément du domaine de l'attribut dans la définition de la classe de l'objet.

Exemple 3. Reprenons le schéma donné dans la figure I.2.a. Supposons que la classe Employé contient l'attribut LeCode qui est une chaîne de caractères (un code associe à un employé, nous ignorons ici la signification du code). I et I' sont deux instances du schéma (Cf. Figures I.4). Dans la figure I.4, l'expression Value(o) est la valeur de l'objet o (nous utilisons la syntaxe du modèle *O2* [OT94]). L'instance I est incohérente vis-à-vis du schéma car la valeur de l'attribut LeCode dans l'objet e2 est un entier, alors que le domaine de cet attribut dans la définition de la classe Employé est l'ensemble de chaînes de caractères. L'instance I' est cohérente vis-à-vis du schéma.

I={e_1, e_2} et I'={e_3} oĚ, e_1, e_2 et e_3 sont des objets de Employé,
/* Nous nous contentons ici de donner la valeur de l'attribut LeCode */
Value(e_1)=tuple(..., LeCode : "EUP", ...),
Value(e_2)=tuple(..., LeCode : 230, ...),
Value(e_3)=tuple(..., LeCode : "ESC", ...).

FIG I.4 : cohérence d'une instance

I.2.1.3. Compatibilité de programmes

Dans un programme d'application, la partie visible du schéma de la base est l'interface des classes (c'est-à-dire, les opérations et les attributs publics). La compatibilité d'un programme vis-à-vis du schéma est définie par analogie avec la cohérence d'une opération du schéma (Cf. Section I.2.1.1, cohérence de comportement). Ainsi, l'exécution du code d'un programme compatible vis-à-vis du schéma ne génère pas d'erreurs. Dans la définition des contraintes à satisfaire :

- On considère un programme au lieu d'une opération,
- On considère les interfaces des classes du schéma et les variables locales et globales du programme,
- Les objets nommés sont les objets associés aux classes.

I.2.2. Besoins en évolution de schéma

En général la taxonomie des opérations d'évolution de schéma, adoptée dans la majorité des systèmes est la suivante [BK87, Mon93] : changer le nom, ajouter, supprimer ou modifier une propriété dans une classe, ajouter ou modifier le code d'une opération, changer le nom d'une classe, ajouter ou supprimer une classe, ajouter ou supprimer un lien d'héritage, changer la position d'une classe dans la hiérarchie, diviser une classe en plusieurs autres, fusionner plusieurs classes en une seule, transférer des propriétés d'une classe dans une autre, etc. Les motivations pour supporter de telles évolutions sont diverses et se manifestent à différents niveaux de l'application :

- **Univers du discours** :

1) il est souvent *large* en volume : plus l'univers du discours est large en volume, plus il est difficile de concevoir le schéma correspondant en une seule fois. La conception s'effectue de manière incrémentale. L'univers du discours n'est alors modélisé qu'à la fin d'un long processus de conception.

2) ses éléments sont souvent *complexes* et difficiles à comprendre. D'où la nécessité de faire des révisions, par exemple pour compléter la conception ou corriger des erreurs.

3) l'univers du discours évolue. Une évolution peut venir de nouveaux besoins (par exemple, ajout d'un service dans l'université) ou d'un changement plus radical de la structure (par exemple, la restructuration d'un ensemble de laboratoires de recherche). Ces changements peuvent être motivés par la recherche d'une meilleure qualité, de meilleures performances, etc.

• **Environnement de la base de données** : le schéma peut évoluer pour prendre en compte les besoins de l'environnement de la base. Par exemple, restructurer le schéma pour améliorer sa qualité, pour améliorer les performances des applications, changer le schéma pour prendre en compte de nouvelles procédures d'administration de la base de données (protection des données, ajout d'index, etc.).

Remarquons que l'évolution de schéma concerne tout le cycle de vie d'une base de données (conception, exploitation). Notons au passage qu'une expérience à une échelle industrielle de quantification de l'évolution de schéma (fréquence et conséquences sur l'environnement en utilisant un système relationnel) sur une large application de gestion des hôpitaux [Sjo93a] confirme l'importance de l'évolution de schéma. L'outil développé mesure le changement du schéma durant la conception et l'exploitation de la base de données pendant une durée de 18 mois. Parmi les résultats de cette étude nous citons :

• Tous les schémas de relations ont été modifiés.
• Le nombre de schémas de relations a augmenté de 23 à 55.
• Le nombre de champs (attributs) a augmenté de 178 à 666.

Le problème de l'évolution de schéma est un problème classique des bases de données, auquel une attention particulière est accordée dans un contexte à objets. Avant de donner les raisons de ce que nous venons de dire, prenons à titre d'exemple le modèle relationnel [DA82, Cod70, MS90, Ora, Rod92a].

Le schéma d'une base de données relationnelle utilise peu de concepts ; relations et attributs décrits sur des domaines atomiques. Ainsi, le nombre de changements du schéma autorisé est limité (ajout/suppression d'un champ dans un schéma de relation, modification du type d'un attribut dans un schéma de relation, ajout/suppression d'un schéma de relation, etc.). Le changement d'une relation n'affecte que celle-ci ou peu d'autres si le changement concerne une clé étrangère.

Dans un contexte à objet le schéma est complexe [ZM90] : la structure de données est riche, les classes sont fortement liées entre elles. Relativement aux bases de données classiques, le problème de l'évolution de schéma en bases de données à objets présente une importante activité de recherche et de développement [ALP91, BF95, BK87, Bra92, Cas91, CCK*94, Cla92, Del92, ED95, FMZ95a, HY95, LH90, LM89, LT94, Mon83, NR89b, Odb94a, PS87, Rod92b, ST94, TK91, Zdo90]. Cela est dû particulièrement aux raisons suivantes :

• La nature évolutive des applications avancées.

• La structuration des classes en une hiérarchie est reconnue comme une tâche difficile [GTC*90]. En particulier, en génie logiciel où on a beaucoup de classes et peu d'objets, l'évolution des classes devient un phénomène naturel [Nac94].

• Classiquement, les parties statique et dynamique des objets sont séparées. Dans une base de données à objets le comportement fait partie du schéma (on parle des opérations décrites dans les classes), il est donc partagé par les applications. Dans ce contexte, une évolution du comportement se traduit donc par une évolution du schéma, plutôt que par une évolution de l'application [SF93]. Partant du fait que les procédures de manipulation (c'est-à-dire, le comportement) des données sont moins stables que leurs structures, on considère que l'évolution de schéma est plus fréquente dans les bases de données à objets que dans les bases de données classiques.

En conclusion, nous pouvons dire qu'il devient indispensable de supporter l'évolution de schéma, si on veut offrir une technologie (ici un SGBD) solide pour le développement des applications avancées.

Comme nous l'avons vu, les applications avancées sont complexes (larges, coopératives, objets volumineux et parfois distribués, etc.). Par conséquent, le coût de développement d'une base de données et des programmes des applications associées est énorme. Il devient irréaliste de reconstruire complètement la base de données (c'est-à-dire, le schéma et les données) à la suite d'une évolution du schéma car cela risque de perturber le fonctionnement des applications pendant plusieurs heures, voir plusieurs jours. Autrement dit, la base de données devient non exploitable pendant le temps de sa reconstruction.

Dans [ASM93], on caractérise la puissance d'un système à objets (un SGBD ou un langage de programmation à objets persistants) par sa capacité à supporter la conception incrémentale, le développement incrémental et le changement incrémental.

I.2.3. Problèmes posés par l'évolution de schéma

Dans cette section, nous étudions les conséquences que peut avoir une évolution du schéma sur l'environnement de la base de données. L'hypothèse de départ est que l'instance de la base est cohérente et les programmes d'applications sont compatible vis-à-vis du schéma de la base.

I.2.3.1. Niveau schéma

Le changement d'une partie du schéma peut engendrer des changements sur les autres parties du schéma. La propagation du changement au niveau du schéma est nécessaire pour assurer sa cohérence au cours de son évolution.

Exemple 4 (cohérence de structure). Reprenons le schéma de la figure I.2.a. Supposons que l'évolution du schéma consiste à ajouter l'attribut LeGenre dans la classe Personne. Nous supposons qu'il n'existe pas d'attributs qui ont le même nom que cet attribut. A la suite de cette évolution, le schéma n'est plus cohérent car l'invariant qui exige que "les classes héritent de toutes les propriétés de leurs super-classes" est violé (Cf. Section I.2.1.1, cohérence de structure) : l'attribut LeGenre n'existe pas dans les classes Employé et Enseignant. Donc, l'ajout de cet attribut doit être propagé aux classes Employé et Enseignant pour préserver l'invariant.

Exemple 5 (cohérence de comportement). Reprenons le schéma donné dans l'exemple 2. Supposons que l'évolution du schéma consiste à supprimer l'attribut LeSalaire_mensuel de la classe Employé. A la suite de cette évolution, le schéma n'est plus cohérent car la contrainte de "fermeture de références" est violée (Cf. Section I.2.1.1, cohérence de comportement) : le code de l'opération LeSalaire_Annuel référence l'attribut LeSalaire-mensuel qui n'existe plus. Donc, il doit être modifié pour préserver la contrainte.

I.2.3.2. Niveau instance

Dans ce paragraphe, nous supposons qu'après l'évolution, le schéma est cohérent. A la suite d'un changement du schéma, l'instance peut devenir incohérente vis-à-vis de son schéma. Le problème ici est que les représentations de certains objets créés avant l'évolution peuvent devenir non conformes à la structure de leurs classes après l'évolution. Une évolution de schéma doit donc être propagée aux instances du schéma pour assurer le maintien de leur cohérence.

Exemple 6. Reprenons le schéma donné dans l'exemple 3. Supposons que l'évolution du schéma consiste à changer le type de l'attribut LeCode d'une chaîne de caractères à un entier (c'est-à-dire,

maintenant le code d'un employé est un entier). La représentation d'un ancien objet de Employé (créé avant l'évolution) n'est pas conforme avec la nouvelle structure de Employé car l'ancien type de l'attribut LeCode (chaîne de caractères) n'est pas compatible avec son nouveau type (entier).

I.2.3.3. Niveau programmes d'applications

L'étude des conséquences d'une évolution du schéma sur les programmes dépend des solutions proposées pour les problèmes posés dans les deux paragraphes précédents. Nous reviendrons sur les solutions dans le chapitre II. Ici, nous supposons qu'après l'évolution du schéma, cohabitent dans la même base de données, le schéma, l'instance et les programmes d'avant l'évolution, avec le schéma, l'instance et les programmes d'après l'évolution. Le problème se pose alors quand un ancien objet est accédé par un nouveau programme ou un nouvel objet est accédé par un ancien programme [BF95, Mon93, Zdo90, Cla92].

Exemple 7 (un ancien objet et un nouveau programme). Reprenons l'exemple 4, où l'évolution de schéma consiste à ajouter l'attribut LeGenre dans la classe Employé. Supposons que l'objet o1 est créé avant l'évolution et l'objet o2 est créé après l'évolution (Cf. Figure I.5). Supposons que la requête suivante est écrite après l'évolution (nous utilisons la syntaxe du langage *O2SQL* [OT94]) :

Requête : select tuple (nom : x.LeNom, genre : x.LeGenre)
from x in employés /* employés contient tous les objets de la classe Employé */
Résultat : {(Rosine, féminin), **(Paul, erreur :"LeGenre n'existe pas")**}

L'exécution de cette requête génère une erreur car l'attribut LeGenre n'existe pas pour l'objet o1.

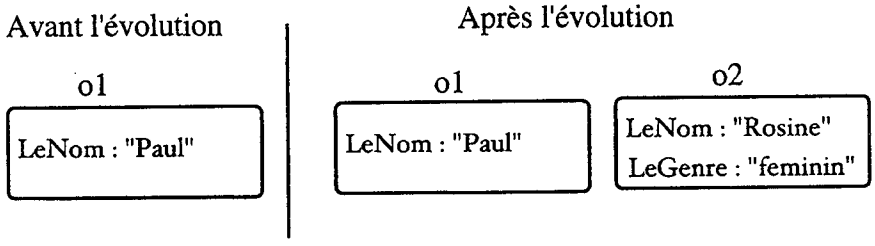


FIG. I.5 : un ancien objet et un nouveau programme.

Exemple 8 (un nouvel objet et un ancien programme). Reprenons l'exemple 5, où l'évolution de schéma consiste à supprimer l'attribut LeSalaire_mensuel dans la classe Employé. Supposons que l'objet o1 est créé avant l'évolution et l'objet o2 est créé après l'évolution (Cf. Figure I.6). Supposons que la requête suivante est écrite avant l'évolution :

Requête : select tuple (nom : x.LeNom, salaire : x.LeSalaire_mensuel)
from x in employés /* employés contient tous les objets de la classe Employé */
Résultat : {(Paul, 8000), **(Rosine, erreur :"LeSalaire_mensuel n'existe pas")**}

L'exécution de cette requête génère une erreur car l'attribut LeSalaire-mensuel n'existe pas pour l'objet o2.

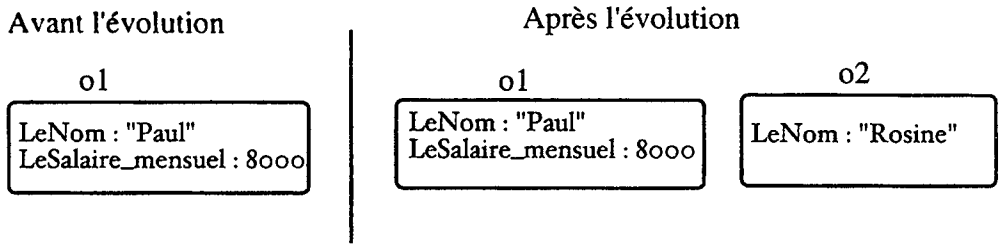


FIG. I.6 : un nouvel objet et un ancien programme.

Ici, la question cruciale est : est-il est préférable d'adapter les programmes aux objets, c'est-à-dire, changer les programmes pour qu'ils soient compatibles avec la nouvelle définition du schéma, ou d'adapter les objets aux programmes, c'est-à-dire, simuler ou gérer les versions d'objets sous les définitions du schéma avant et après l'évolution, ou encore de trouver un compromis entre les deux ? Notons que ce problème est toujours mal compris malgré les travaux dans ce sens [ABB*84, Ben95a, BF95, Bra92, Cla92, FMZ95a, ED95, Ler93, Liu94, Odb94a, Zdo86a, Zdo90]. Nous proposons dans le chapitre 4, une solution basée sur un compromis.

Par exemple, considérons une base de données distribuée [KF91]. Le changement du schéma sur un site, peut avoir des effets sur des programmes dans d'autres sites. Dans la mesure où certains sites peuvent être temporairement inaccessibles (par exemple à cause d'une panne), il est préférable de choisir l'adaptation des objets aux programmes comme solution [Cla93, Liu94].

I.2.4. Conclusion

Le changement du schéma consiste à :

• Ajouter des informations dans le schéma, comme l'ajout d'une propriété dans une classe, l'ajout d'une nouvelle classe ou l'ajout d'un lien d'héritage. Dans ce cas, les anciens programmes sont compatibles avec le schéma après l'évolution car l'interface d'une classe avant l'évolution, est compatible avec son interface après l'évolution.

• Supprimer des informations dans le schéma, comme la suppression d'une propriété dans une classe, la suppression d'une classe ou la suppression d'un lien d'héritage. Par conséquent, des propriétés peuvent être supprimées dans l'interface de certaines classes après l'évolution. Ce qui provoque l'incompatibilité des programmes qui font référence à des propriétés supprimées, avec le schéma après l'évolution.

• Modifier des informations dans le schéma, comme la modification de la signature d'une propriété dans une classe, la modification du code d'une opération, le changement du nom d'une propriété ou d'une classe, le transfert d'une propriété d'une classe dans une autre, le changement de la position d'une classe dans la hiérarchie, la fusion de plusieurs classes en une seule ou la division d'une classe en plusieurs autres. Comme dans le cas précédent, certains anciens programmes peuvent devenir incompatibles avec le schéma après l'évolution (par exemple, un programme qui fait référence à une propriété qui a changé de nom).

Supporter l'évolution de schéma dans un SGBD à objets consiste à permettre le changement de la structure et du comportement de ses classes sans perturber le fonctionnement des applications. Autrement dit, assurer l'accessibilité de l'instance à la suite de l'évolution de son schéma sans pénaliser les utilisateurs de la base.

D'une part, les mécanismes introduits ne doivent pas être la cause d'une dégradation non acceptable des performances du système. D'autre part, ces mécanismes doivent assurer un certain degré de pérennité des informations qui existaient avant l'évolution. On parle ici de l'équilibre "coût / bénéfice".

I.3. Approche objet et évolution de schéma

Une des motivations fondamentales de l'approche à objet est d'anticiper l'évolution [ASM93], on parle aussi de la caractéristique d'*extensibilité* de l'approche objet (programmation incrémentale, programmation par évolution, etc.). Cela revient à dire que, certaines évolutions du schéma n'ont pas d'effets sur l'environnement de la base car ils sont, en quelque sorte, prévus par le modèle.

Dans cette section nous discutons de la capacité de l'approche objet à supporter l'évolution de schéma, on dit aussi absorber l'évolution de schéma. Autrement dit, nous voulons répondre à la question suivante : les concepts qu'offre l'approche objet sont-ils suffisants pour supporter l'évolution de schéma ou est-il nécessaire d'offrir des mécanismes spéciaux pour cela ? Nous présentons les concepts essentiels qui contribuent à supporter l'évolution de schéma. Notre

présentation s'appuie sur [ASM93, Bra93, Cas91, CS89, Dea91, MZ93, NB88, Osb89, Rou91, SN88, WZ88]. Certains de ces concepts sont plutôt issus des langages de programmation et adaptés aux bases de données à objets.

I.3.1. Sous-classe

Dans une sous-classe, de nouvelles propriétés peuvent être définies, ou peuvent redéfinir (rendre plus spécifique) des propriétés définies dans une des super-classes.

Parfois, l'ajout d'une sous-classe n'a pas d'effet sur l'environnement de la base car : (1) le schéma n'est pas affecté puisque cette classe est une feuille dans le schéma. (2) Son extension est vide au moment de sa création. (3) Les objets existants (objets des super-classes) restent accessibles au travers de leurs classes, par les programmes. Les nouveaux objets (objets de la nouvelle classe) sont accessibles par les anciens programmes car un objet d'une classe est aussi un objet de ses super-classes.

Théoriquement, on peut traduire l'ajout ou la spécialisation d'une propriété définie dans une classe du schéma par l'ajout d'une nouvelle sous-classe qui contient la nouvelle propriété ou la nouvelle définition de la propriété. Cette solution peut dans certains cas poser problème car elle provoque l'inflation du nombre de classes du schéma.

Notons au passage que plusieurs tentatives [ABD*94, Bor88, Cas92], en particulier, dans le domaine des langages de programmation (en étendant la notion de sous-typage), ont essayé d'étendre le concept d'héritage afin de permettre la modélisation de situations plus générales qu'une simple spécialisation. A titre d'exemple, nous citons les travaux sur l'introduction du mécanisme d'exception de [Bor88, BW85]. Dans la définition d'une classe, la clause **excuses** de spécification d'exceptions (Cf. Figure I.7) peut être utilisée. Le type de l'attribut Diplôme dans Etudiant-Etranger (cette classe décrit les étudiants étranger) n'est pas compatible (c'est-à-dire, ce n'est pas un sous-type) avec son type dans Etudiant. La clause **excuses** spécifie qu'il s'agit là d'une exception. Ainsi, le système de type sous-jacent accepte cette incompatibilité.

Exemple :

```
class Etudiant tuple (
        ........
        LeDiplôme : set ("maĒtrise", "magistŹre", "ingénieur", "dea", "doctorat") ,
        )
end;
```

```
class Etudiant-Etranger inherit Etudiant tuple (
        ........
    LeDiplôme : set("bachelor", "master", "PhD") excuses on Etudiant,
      )
end;
```

FIG. I.7 : expression d'une exception.

En conclusion, l'ajout d'une sous-classe n'est pas suffisant car la majorité des évolutions ne se traduisent pas par l'ajout de sous-classes. Il est souvent nécessaire de modifier directement les propriétés des classes (ajout/suppression, modification de la définition) ou de modifier la hiérarchie de classes (ajout/suppression d'un lien d'héritage, etc.).

I.3.2. Classe paramètrée

Une classe (ou un type) paramètrée est une classe *générique* [Mey88, Car89]. Sa définition est exprimée en fonction d'un paramètre qui peut prendre ses valeurs dans un ensemble de classes. Le constructeur *Template* du C++ [Bra93] est un exemple d'une classe paramètrée. Donc, un modèle de données (ou un système de type en langages de programmation) qui supporte des classes paramètrées offre aux usagers la possibilité de définir des nouveaux constructeurs.

Une classe paramètrée anticipe certaines évolutions du schéma dans le sens où elle servira dans le futur pour la définition d'autres classes. Ces classes ont une certaine similarité. Supposons que Pile <T> est une classe qui est définie en fonction du paramètre T. Cette classe générique englobe toutes les classes qui définissent les piles (pile d'entiers, pile de chaînes de caractères, etc.). L'instanciation de cette classe en utilisant le type pré-défini int comme valeur du paramètre T, produit la classe des piles d'entiers : Pile <int>.

Nous invitons le lecteur à lire [ASM93] pour une bonne introduction sur l'*anticipation* de l'évolution de schéma. Notons au passage que les auteurs présentent un constructeur qui peut anticiper certaines évolutions importantes du schéma, comme l'ajout de nouvelles propriétés dans une classe. Malheureusement, ce constructeur présente des inconvénients majeurs, sur lesquels nous reviendrons dans la fin de ce paragraphe.

Le constructeur en question est le suivant (nous utilisons la syntaxe des auteurs, any est la racine du graphe d'héritage) :

type ProtectionChangement <PartieFixe> **is record** [fixe : PartieFixe, extra : any]

Par exemple, la classe Etudiant peut être déclarée par :
type Etudiant **is** ProtectionChangement < **record** [LeNom : string, ..., LaFiliŹre : string]>

Initialement, la valeur associée à l'attribut extra dans tous les objets de la classe Etudiant, est la valeur null (valeur par défaut). La structure invariante (c'est-à-dire, l'attribut fixe) permet d'assurer la stabilité des références de types et des références d'objets. Le changement concerne seulement l'attribut extra. Ainsi, les programmes sont aussi stables à la suite d'une évolution du schéma (le type any est l'union de tous les types).

Supposons que l'évolution du schéma consiste à ajouter l'attribut LeResponsable de type chaîne de caractères dans la classe Etudiant. Ainsi, on définit la nouvelle structure pour prendre en compte cette évolution :
type ExtraEtudiant **is record** [LeResponsable : string]

A la suite de cette évolution, les anciens programmes ne sont pas affectés. Les instances peuvent être modifiées de manière incrémentale, au moment de leur utilisation.

L'inconvénient majeur de cette solution est qu'elle réduit considérablement : (1) le rôle du schéma, dans le sens où la structure que peuvent avoir les données est plus ou moins libre. Ce n'est qu'au moment de l'exécution que le lien entre Etudiant et ExtraEtudiant est établi, (2) Le confort lors de la programmation qui est une des motivations du typage strict [Con91].

I.3.2. Type union

Certains systèmes offrent la possibilité de faire de l'union de types [Mor79]. Une valeur qui est conforme à un des types de l'union est conforme au type union. Un exemple de type union est le type de la classe Object. En général, à une valeur est associée l'information sur son type spécifique pour faire des traitements (par exemple, vérification de type) sur la projection du type union comme s'il s'agit du type original de la valeur.

Il devient possible de créer des programmes corrects (c'est-à-dire, leur compilation n'échoue pas) qui manipulent des données dont les types ne sont pas encore définis (ou sont en cours de définition).

Le type union permet d'anticiper certaines évolutions dans le sens où le changement d'une partie du type (donc du schéma) ne concerne pas les autres parties du type.

I.3.4. Conclusion

Nous pouvons dire que nous n'avons pas pu montrer que les concepts de l'approche objet facilitent l'évolution de schéma. Nous avons montré qu'elle n'est pas suffisante. Ils ne permettent pas de supporter toutes les évolutions possibles, par exemple, la modification du type d'un attribut dans une classe tel que, le nouveau type n'est pas un sous-type de l'ancien. De plus, il est parfois souhaitable de matérialiser les effets de l'évolution, par exemple, pour corriger une erreur.

Certains travaux sur les systèmes de types (modèles de données) visent à améliorer la capacité de l'approche objet à supporter l'évolution de type (schéma). Par exemple, définir de nouveaux concepts ou de nouvelles règles de typage (type matching rules). Les contraintes principales à satisfaire dans ce cadre sont : (1) la préservation des avantages du typage fort (Cf. Section I.2.1) dans le sens de la description et de la protection des données et dans celui du confort de la programmation, (2) Les concepts introduits ne doivent pas augmenter la complexité de l'écriture et le coût d'exécution des programmes.

Chapitre II

ÉTAT DE L'ART

Ce chapitre présente les approches pour la gestion de l'évolution de schéma de bases de données à objets. Nous proposons une classification des approches existantes. Pour chacune de ces approches nous décrivons son principe, les mécanismes d'évolution associés, ses avantages et ses inconvénients, ainsi que les produits et les prototypes qui l'implantent. Nous identifions et décrivons les travaux similaires dans d'autres domaines.

II.1. Introduction

Comme nous l'avons vu dans le chapitre I, le problème de l'évolution de schéma constitue l'une des voies importantes de recherche et de développement dans le domaine des bases de données à objets. Ce problème est considéré dans de nombreux systèmes [ALP91, Ari91, BF95, BK87, Bra92, Cas91, Cla92, Del92, FMZ95a, EO93, HY95, LH90, LM89, Mon93, NR89b, Odb94b, PS87, RR94, ST94, TK91, Zdo90]. Diverses approches sont proposées pour gérer l'évolution de schéma. Dans ce chapitre nous présentons une synthèse des travaux existants. Pour cela nous identifions des critères de classification, puis nous présentons les catégories d'approches issues de cette classification.

Tout d'abord, nous rappelons brièvement certaines considérations communes aux approches d'évolution de schéma (notre présentation s'appuie sur [Ben94, PO95, Rod94]) :

• Il est commode que le langage d'expression d'évolution de schéma soit fourni en termes d'une algèbre sur le schéma (taxonomie d'opérations). Ainsi, certaines propriétés du langage d'expression peuvent être vérifiées comme la complétude (c'est-à-dire, la capacité du langage à exprimer toutes les évolutions possibles d'un schéma) et la minimalité (c'est-à-dire, qu'une opération du langage ne peut pas être remplacée par une combinaison d'autres opérations).

• Il est souhaitable de minimiser l'intervention des utilisateurs pour réaliser la propagation de l'évolution de schéma au niveau des objets. Les détails sur la réalisation dépendent beaucoup plus de l'environnement opérationnel que des besoins des utilisateurs. Cependant, il est reconnu que dans certaines situations, il est préférable que la propagation soit guidée par l'administrateur de la base.

• Il est souhaitable que l'évolution de schéma soit le plus symétrique possible. Une évolution est symétrique lorsque les objets d'avant l'évolution sont accessibles au travers du nouveau schéma, et inversement, que les objets d'après l'évolution sont accessibles au travers de l'ancien schéma. Par conséquent, un certain degré de stabilité des programmes d'applications est maintenu. Il n'est alors pas nécessaire de re-compiler les anciens programmes à la suite d'une évolution de schéma. La réversibilité du changement est assurée, c'est-à-dire qu'il est toujours possible de revenir en arrière si le changement ne convient pas.

Ce chapitre est organisé comme suit. La section II.2 présente les critères de la classification des approches que nous proposons. Les sections II.3, II.4 et II.5 présentent par catégories, les approches considérées. La section II.6 présente les travaux similaires dans d'autres domaines.

II.2. Critères de classification des approches

En général, la terminologie utilisée pour parler de ce nous avons appelé jusqu'à maintenant *évolution de schéma* est variée (évolution de schéma, mise à jour de schéma, modification de schéma, versionnement du schéma, etc.) ; il en découle une confusion d'usage. Dans cette section nous dégageons les définitions des termes essentiels utilisés pour parler de l'évolution de schéma. A partir de ces définitions, nous proposons une classification des approches suivies. Ces définitions sont cohérentes avec la majorité des travaux sur l'évolution de schéma.

Avant de donner les définitions nous rappelons les pré-requis nécessaires. Les utilisateurs utilisent une base de données à partir du schéma de cette base. Quand le schéma évolue, deux cas sont possibles : (1) les utilisateurs utilisent la base à partir d'une seule définition du schéma, (2) les utilisateurs utilisent la base à partir plusieurs définitions du schéma.

Par exemple, dans la figure II.1, la base de données est constituée du schéma S0 et de l'instance I0. P0 est un ensemble de programmes d'applications construits autour de la base de données. Ces programmes manipulent les données de I0 au travers du schéma S0. Supposons que l'évolution de schéma consiste à changer S0 en S1. Dans le premier cas S1 remplace S0 qui est perdu. Dans le deuxième cas, S0 et S1 coexistent en même temps.

En conséquence, un objet de la base peut avoir une ou plusieurs représentations. Nous parlons ici des représentations des objets sous différentes définitions du schéma. Dans la suite, sous une définition particulière du schéma, nous considérons une seule représentation par objet. L'inverse (c'est-à-dire, plusieurs représentations d'un objet sous une définition particulière) est plutôt pertinent pour l'évolution des objets dans un schéma constant (sujet traité par ailleurs).

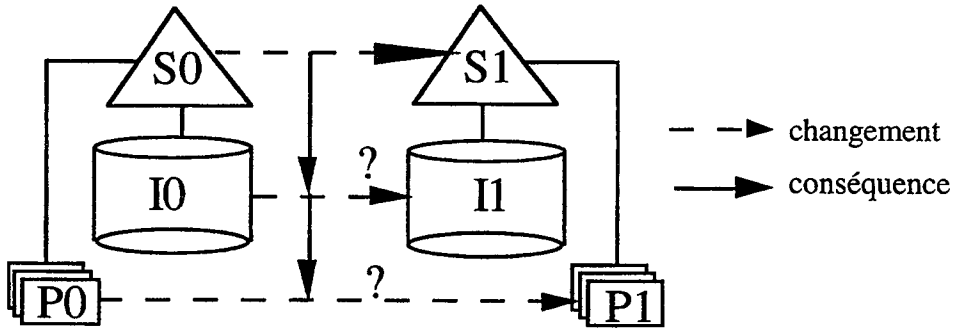


FIG. II.1 : : évolution de schéma.

Certains systèmes supportent une seule définition du schéma mais offrent aux utilisateurs de définir des vues (schémas externes) sur le schéma [BM93, KK90, Run92, SDA94]. Ainsi, un objet de la base possède une seule représentation *réelle* (ou physique) dans la base. Au travers d'une vue sur le schéma, l'utilisateur accède à une représentation *virtuelle* de l'objet. La représentation virtuelle d'un objet est dérivée à partir de sa représentation réelle. Ainsi, une évolution de schéma peut être traduite par la création d'une vue sur ce schéma. Par exemple, dans la figure II.1, S1 est une vue sur S0.

Nous distinguons deux niveaux : fonctionnalité d'évolution de schéma et mécanisme d'évolution de schéma.

Fonctionnalité d'évolution de schéma

A ce niveau, étant donné un système, nous utilisons le terme *évolution* de schéma pour dire que le système permet l'expression des changements du schéma d'une base de données et assure leur exécution. Il supporte alors la fonctionnalité de l'évolution du schéma.

Mécanisme d'évolution de schéma

A ce niveau, étant donné un système qui supporte la fonctionnalité d'évolution de schéma, la question est de savoir comment le système réalise cette fonctionnalité. Nous utilisons le terme *modification* de schéma pour dire que le système autorise le changement du schéma d'une base de données et traduit ce changement par la mise à jour du schéma et de l'instance. Nous utilisons le terme *versionnement* du schéma pour dire que le système autorise l'existence de plusieurs versions du schéma. Les objets de la base sont accessibles à partir des versions du schéma.

En se basant sur ces définitions, les approches suivies pour prendre en compte l'évolution de schéma peuvent être classées en trois catégories :

• Les approches fondées sur la *modification* du schéma où il y a toujours un seul schéma et une seule représentation par objet. Cette catégorie est discutée dans la section II.3.

• Les approches fondées sur les *versions* du schéma ou le schéma peut avoir plusieurs versions et les objets peuvent avoir plusieurs représentations (ou versions). Cette catégorie est discutée dans la section II.4.

• Les approches fondées sur les *vues* où une évolution de schéma est traduite par la création d'une vue sur le schéma. Cette catégorie est discutée dans la section II.5.

Dans la suite, l'*adaptation des instances* désigne le processus qui assure, en la modifiant, la vérification et la maintenance de la cohérence d'une instance d'une base de données vis-à-vis de son schéma.

II.3. Approches fondées sur la modification

Dans cette catégorie, le schéma de la base de données possède une seule définition partagée par tous les utilisateurs. Un objet possède une seule représentation. Une évolution du schéma est traduite par la modification de ce schéma [Bar91, BK87, Bou95, FMZ95a, HY95, Ita93, LH90, NR89b, OT94, OD93, PS87, SGD93, TK91, Tre91, Ver92].

II.3.1. Changement du schéma

Le langage d'expression de l'évolution (un ensemble de primitives) dépend du modèle de données. Ainsi, les primitives proposées, les aspects liés à la complétude et parfois la minimalité du langage diffèrent d'un système à un autre. Dans cette section nous discutons de la propagation du changement pour préserver sa cohérence : cohérence de structure et cohérence de comportement.

Cohérence de structure

La cohérence de structure du schéma est définie à l'aide des *invariants* de schéma [BK87, DZ91, PO95] (Cf. Chapitre I). Par exemple, l'invariant "hiérarchie de classes" exige que le schéma soit un graphe orienté, connexe et sans cycle. Des règles sont proposées pour maintenir la cohérence de structure. Une règle vérifie des conditions et exécute des actions qui propagent l'évolution au niveau du schéma. Par exemple, une règle consiste à rejeter une évolution du schéma si elle entraîne la production d'une classe isolée (c'est-à-dire, elle n'a pas de super-classes). On peut dire qu'une évolution de schéma consiste en une transaction appliquée à un schéma cohérent (les invariants sont vérifiés) et qui produit un schéma cohérent.

Cohérence de comportement

La cohérence de comportement du schéma (c'est-à-dire, les opérations) est définie en termes de contraintes à satisfaire par les opérations du schéma afin que leurs exécutions ne génèrent pas d'erreurs [CLZ91, LT94, Mor92, MNS94] (Cf. Chapitre I).

Pour maintenir la cohérence de comportement, plusieurs approches ont été considérées [Ben94]. Parmi ces approches nous citons : les approches basées sur les *dépendances inter-propriétés* et celles basées sur l'*analyse statique de types* .

Dans une approche basée sur les dépendances inter-propriétés, le système gère des méta-informations sur les propriétés du schéma (attributs, opérations, classes, etc.) [Bar91, Ben92, BF94, Mor92, Zic92]. Ces informations sont représentées dans ce qu'il est habituel de nommer un méta-schéma et dont le schéma est décrit par des méta-classes. Ainsi, les propriétés (attributs et opérations) des méta-classes contiennent les informations nécessaires sur le comportement du schéma. Pour une opération donnée, on gère ses dépendances (les attributs/ opérations/ classes utilisés, les opérations qui l'utilisent, etc.) et son statut (exécutable, non-exécutable).

Les dépendances des opérations sont gérées à l'aide d'un graphe orienté de dépendances. Les noeuds sont les propriétés du schéma et les arcs "dépend de" indiquent les dépendances entre les propriétés.

A la suite d'une évolution de schéma, une opération du schéma peut ne plus respecter les contraintes de cohérence de comportement. Par exemple, à la suite de la suppression d'un attribut, l'exécution des opérations qui l'utilisent peut générer des erreurs car la contrainte de fermeture de références n'est pas vérifiée. Ces opérations sont déterminées (opérations non-exécutables) en utilisant le graphe de dépendances. Elles seront re-compilées ou tout au moins signalées à l'utilisateur.

Une approche intéressante, basée sur l'analyse statique de types, proposée pour des systèmes qui supportent le polymorphisme (sémantique d'inclusion) et la liaison dynamique avec condition de covariance, est présentée dans [CCL*93, CLZ91]. Notons que pour cette catégorie de systèmes, la détection statique des erreurs qui peuvent apparaître à l'exécution est un problème indécidable.

Exemple : extrait de [CCL*93]

```
class Employé tuple (
        LaFonction : string,
```

```
        LeChef : Employé,
        LOrganisation : Organisation;)
method
        substituer : Employé, /* remplace un employé par un autre employé ou un manager */
        remplacer_par (e : Employé) : boolean, /* évalue la condition de remplacement d'un
                                                  employé par un autre */
end;
method body remplacer_par in class Employé {
        return ((self.LOrganisation=e.LOrganisation) and (self.LaFonction=e.LaFonction));
 }

class Manager inherit Employe tuple (
        LeDept : Departement /*est une classe du schéma*/
)
method
        remplacer_par(m: Manager) : boolean, (*évalue la condition de remplacement d'un
                                                  manager par un autre*)
        remplacer_quand_vacant (e : Employé) : boolean
end;
method body remplacer_par in class Manager {
        return((self.LOrganisation=m.LOrganisation) and (self.LeDept=m.LeDept));
}
method body emplacer_quand_vacant class Manager {
/* cherche un substituant possible, cette méthode peut être définie par exemple dans la classe Organisation */
        b : boolean, x : employé;
        x := e.substituer();
        b := x.remplacer_par(e);
.       .........}
```

Lors de l'exécution de la méthode **remplacer_quand_vacant**, si x est un Manager et e un Employe, alors le message **x.remplacer_par(e)** génère une erreur : l'attribut **LeDept** n'est pas défini dans Employé.

Cette approche utilise la notion d'ensemble de types *(typeset)*. L'ensemble de types d'un attribut (ou d'un paramètre formel d'une méthode) est l'ensemble de tous les types des valeurs (ou des objets) qui peuvent lui être substitués au moment de l'exécution. Il est calculé au moment de la

compilation des méthodes. L'analyse des ensembles de types associés à un schéma permet la détection statique des erreurs qui peuvent apparaître à l'exécution.

Dans la suite nous décrivons l'algorithme associé à cette approche. On note TS_i l'ensemble de types de l'attribut x_i du type t_i.

Définitions

- Le type d'un attribut x_i du type t_i est appelé le type statique de x_i.
- Un TS_i est majoré par le type t, si pour tout $t_j \in TS_i$, t_j est un sous-type de t.
- L'ensemble $TD = \{(x_i, TS_i)\}$ de tous les couples (x_i, TS_i) est appelé le descripteur de t_i. Δ

Les auteurs de cette approche démontrent qu'un schéma satisfait la cohérence de comportement s'il vérifie les conditions suivantes :

- Chaque ensemble de types doit être majoré par le type statique de l'attribut associé.
- Si une méthode m est appliquée à un attribut a (a.m()), alors m doit être définie dans le type statique de a.
- Les ensembles de types associés aux paramètres effectifs d'une méthode doivent être majorés par les types statiques de ses paramètres formels.

La vérification de la cohérence de comportement se fait en analysant toutes les méthodes du schéma. L'analyse d'une méthode consiste à examiner les effets de chaque instruction de son code sur les descripteurs du schéma. On note `tdf`(I, T) la fonction, qui à une instruction I et un descripteur T associe le descripteur après l'exécution de I. La sémantique de `tdf` est présentée dans [CLZ91]. L'algorithme de vérification utilise la fonction `tdf` pour vérifier la satisfaction des conditions de cohérence de comportement par les méthodes. Il n'est pas nécessaire de tenir compte de tous les attributs mais seulement de ceux qui sont référencés dans les méthodes. Notons qu'il est possible, que l'exécution d'une méthode ne génère pas d'erreurs même si elle ne satisfait pas une des trois conditions décrites ci-dessus. Dans ce sens, l'algorithme est pessimiste.

II.3.2. Adaptation des instances

L'évolution de schéma d'une base de données peut introduire une incohérence de l'instance vis-à-vis de son schéma (Cf. Chapitre I).

Par exemple, supposons que le schéma d'une base de données contienne les classes Employé et Enseignant, elle même une sous-classe de Employé (Cf. Figure II.2). La structure de la classe Employé contient l'attribut LeChef dont le type est Employé. L'évolution du schéma consiste à changer le type de l'attribut LeChef de Employé à Enseignant (c'est-à-dire, maintenant le chef d'un employé est un enseignant). La représentation d'un ancien objet de Employé créé avant l'évolution, dont la valeur associée à l'attribut LeChef est un objet de la classe Employé, n'est pas conforme avec la nouvelle structure de Employé (Employé n'est pas une sous-classe de Enseignant).

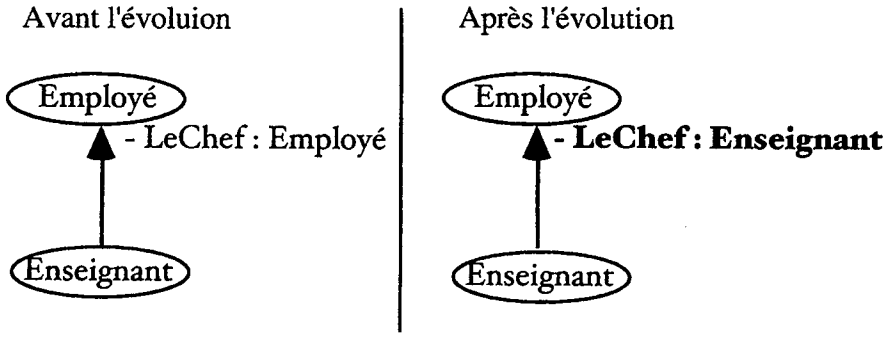


FIG. II.2 : changement du type de l'attribut LeChef dans la classe Employé

Pour maintenir la cohérence d'une instance vis-à-vis de son schéma, on utilise la *conversion* [FMZ94a, FMZ94b, Tre91] comme technique d'adaptation, afin que la représentation des objets de l'instance soit conforme avec sa structure. Par exemple, à la suite du changement du type de l'attribut LeChef, la valeur associée à l'attribut LeChef dans les anciens objets de Employé est mise à jour. Une adaptation triviale consiste à associer une valeur par défaut (nil) à la valeur de l'attribut LeChef.

La conversion peut être *physique* ou *logique*. La conversion physique restaure la cohérence d'une instance par la mise à jour des représentations (physique) des objets. Dans la conversion logique, les représentations des objets ne sont pas mises à jour mais plutôt interprétées (transformées) dans un format conforme aux classes au moment de leur utilisation. Le terme *conversion* est introduit pour désigner la conversion physique et le terme *émulation* pour désigner la conversion logique.

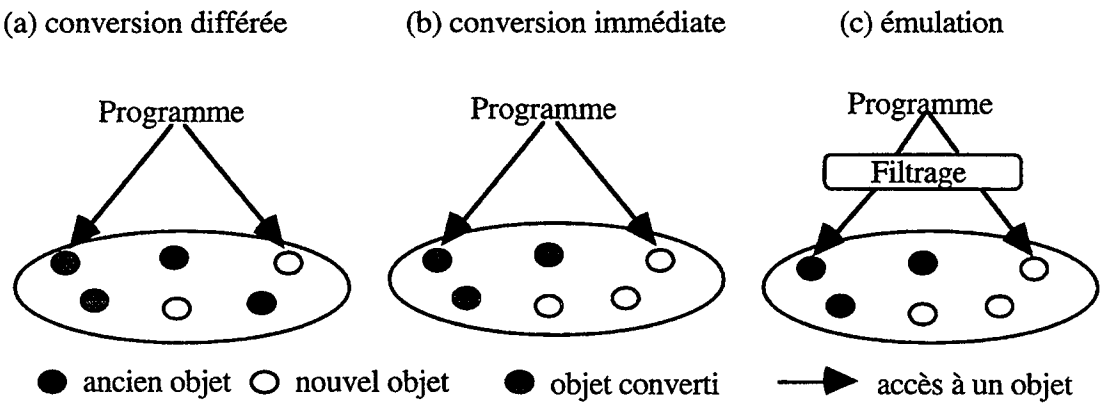


FIG. II.3 : conversion physique et conversion logique

a) Conversion

La conversion peut être manuelle (activée par l'utilisateur) ou automatique. Elle peut introduire une *perte d'informations* (suppression ou modification). Par exemple, la suppression de l'attribut LeChef de la définition de la classe Employé provoque la suppression de la valeur associée à cet attribut dans tous les anciens objets de Employé.

La perte d'informations peut engendrer l'*incompatibilité des anciens programmes* qui utilisent la base de données au travers de la partie modifiée du schéma. Ces programmes devront être modifiés. Par exemple, supposons que LeSalaire-mensuel soit un attribut de la classe Employé et Salaire_Annuel (obj_e : Employé) est un programme qui calcule le salaire annuel de l'objet obj_e de la classe Employé, en fonction de son salaire mensuel. L'évolution du schéma consiste à supprimer l'attribut LeSalaire-mensuel. La conversion provoque la suppression de l'attribut LeSalaire-mensuel dans tous les objets de la classe Employé. Puis l'exécution du programme Salaire_Annuel (o) où o est un objet de Employé, génère une erreur car l'attribut LeSalaire-mensuel n'existe plus.

La conversion peut être *immédiate* ou *différée*.

Conversion immédiate

Dans la conversion immédiate (Cf. Figure II.3.b), la modification du schéma provoque la mise à jour de tous les objets qui ont une représentation non conforme avec la nouvelle structure de leurs classes. Ainsi, la cohérence d'une instance vis-à-vis de son schéma est toujours garantie. Le coût de la conversion est fonction du nombre d'objets à modifier. Ce coût devient prohibitif dans le cadre d'applications qui manipulent un grand nombre d'objets volumineux et complexes. Par conséquent, la durée pendant laquelle la base de données n'est pas accessible peut être longue.

Cette technique est proposée dans le système *GemStone* [PS87, BMO*89]. Une extension intéressante est proposée dans le prototype *OTGen* [LH90] où on génère systématiquement un ensemble de *règles de conversion*, que l'utilisateur peut éventuellement modifier. Dans le système *ObjectStore* [OD93], des *fonctions de transformation* peuvent être associées à une classe qui vient d'être modifiée. Celles-ci sont utilisées par la mise à jour des objets de la classe. Une technique similaire est proposée pour le prototype *COCOON* [Tre91, ST94].

Conversion différée

Dans la conversion différée (Cf. Figure II.3.a), la mise à jour n'est effectuée sur un objet que lorsque celui-ci est accédé. Elle remédie à l'inconvénient de la conversion immédiate : un objet est mis à jour seulement si nécessaire. Ici, il est nécessaire de gérer l'historique des mises à jour différées. Ainsi, l'accès aux objets dans les applications où l'évolution du schéma est fréquente, induit une dégradation des performances du système.

Cette technique est proposée par Tan et Katayama [TK91]. Elle est utilisée aussi dans les systèmes *Itasca* [Ita93] et *Versant* [Ver92]. Une extension intéressante est proposée dans le système *O2*, dans le cadre du projet *GOODSTEP* [FMZ95a]. A une classe modifiée sont associées des *fonctions de conversion* ou de *migration* des objets de la classe. De plus, l'administrateur de la base peut forcer la conversion immédiate.

b) Émulation (screening)

Dans l'émulation (Cf. Figure II.3.c), la modification du schéma ne provoque pas la mise à jour des objets. Un mécanisme de *filtrage* est proposé. Par exemple, il permet de transformer un attribut dans le format demandé au moment où il est accédé. Cette approche évite les inconvénients de la conversion. Ici, les inconvénients sont la complexité du mécanisme de filtrage et la dégradation des performances, particulièrement, quand les objets affectés sont fréquemment accédés. Elle est proposée dans le système *Orion* [BK87].

II.4. Approches fondées sur les versions

Dans cette catégorie, le schéma (respectivement, un objet) de la base de données peut avoir plusieurs versions. Un objet créé sous une version du schéma peut être accessible via une autre version du schéma. Une évolution de schéma est traduite par la dérivation d'une nouvelle version du schéma [ALP91, BB88, Ben92, BM93, Cla93, ED95, KC88, MS93, Nac94, Odb95, Zdo90].

La motivation principale pour l'utilisation du versionnement pour l'évolution du schéma est d'autoriser les programmes d'applications à accéder aux objets de la base à partir de plusieurs versions du schéma. Ainsi, les anciens programmes sont exécutables après l'évolution du schéma

car l'ancienne version du schéma existe toujours, on parle alors de la *transparence* de l'évolution de schéma [Cla92]. Cela est particulièrement important quand le nombre de programmes affectés est grand et/ou il est impossible d'accéder aux textes (codes sources) de ces programmes pour les modifier.

Deux types de versions sont distingués : les versions *historiques* et les versions *parallèles* [ALP91, CJK91, KC88]. Dans l'approche avec versions historiques chaque version de schéma est conservée avec les données qui lui sont associées. Ceci permet de constituer les archives de la base. Ces versions, stockées dans des espaces séparés, sont toujours indépendantes. Les anciennes versions ne sont accessibles qu'en consultation. Toute mise à jour est toujours effectuée sur la base de données courante (c'est-à-dire, la plus récente). Cette approche consiste à gérer autant de versions de la base qu'il existe de versions du schéma.

Dans l'approche avec versions parallèles les différentes versions de schéma de la base de données résident dans un espace commun et opèrent sur le même ensemble de données. Toutes ces versions coexistent et les mêmes opérations leur sont applicables. Elles sont accessibles en consultation et en mise à jour.

II.4.1. Changement du schéma

La granularité de l'unité de l'évolution, c'est-à-dire l'élément sur lequel s'applique l'évolution, diffère d'un système à un autre. Trois types de granularité sont considérés : classe, schéma et contexte (par exemple, une partie du schéma).

Versions de classes [Ben92, BF94, BF95, Cla92, ED95, Mon93, Nac94, Zdo86a, Zdo86b]

Une évolution apportée à une classe crée une nouvelle version de cette classe et de ses sous-classes ainsi que de toutes les classes qui lui sont liées par agrégation. Une version du schéma global est ensuite créée de façon virtuelle en exploitant les relations entre les différentes classes, il s'agit alors d'une *configuration* [Est88, Mun93, Sci94]. Pour représenter le schéma d'une base de données à un instant donné, l'utilisateur doit choisir une version particulière pour chacune des classes définies à cet instant et établir des liens entre ces différentes versions. Par ailleurs, la redéfinition d'une classe peut générer un nombre important de versions de ses sous-classes. Enfin, tout changement effectué sur une classe non racine de la hiérarchie nécessite de gérer les liens entre la version qui en résulte et les versions des classes situées en amont dans la hiérarchie.

Versions de schéma [BM93, KC88, Odb94b, Odb95]

Une évolution de schéma d'une base de données provoque la dérivation d'une nouvelle version du schéma complet de la base. A chaque version correspond un état de la totalité de la base à un instant donné. Il n'est donc pas possible de développer en parallèle des versions partielles du schéma, par exemple pour tester les conséquences de la transformation d'une partie de la structure de la base sur les données. Pour réaliser ce type d'expérience, une version du schéma entier doit être dérivée. Par conséquent un nombre important de schémas peut être généré, notamment lorsque les modifications qui sont effectuées ne concernent qu'une infime partie du schéma.

Notons que l'approche versions de schéma est conceptuellement puissante. D'une part, il est préférable pour un utilisateur de voir le schéma entier après son évolution [Odb94a]. D'autre part, elle est satisfaisante pour supporter les évolutions de schéma dites *multiples,* c'est-à-dire, les évolutions qui affectent plusieurs classes en même temps, comme la division d'une classe en plusieurs autres, la fusion de plusieurs classes en une seule, le changement de la position d'une classe dans la hiérarchie, etc.

Par exemple, supposons que l'évolution du schéma consiste à remplacer la classe Enseignant par les deux classes Professeur et Assistant (Cf. Figure II.4). Avec la modification du schéma, la classe Enseignant est supprimée et les classes Professeur et Assistant sont créées. Par conséquent, les programmes qui utilisent les objets de la classe Enseignant ne sont plus exécutables. Il n'est pas possible de traduire cette évolution par la dérivation d'une nouvelle *version de classe,* car la classe Enseignant n'existe plus après l'évolution. Par contre, avec les *versions de schéma* où l'ancienne version du schéma contient la classe Enseignant et la nouvelle version du schéma contient les classes Professeur et Assistant, ces problèmes ne se posent pas.

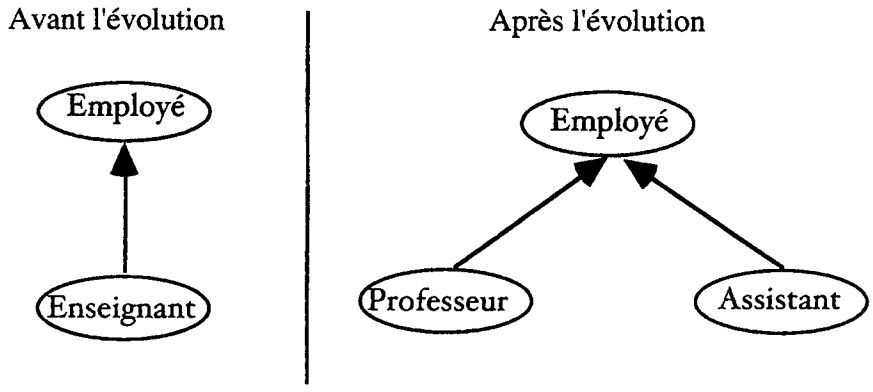


FIG. II.4 : remplacement de Enseignant par Professeur et Assistant

Versions de contextes [AFL93, ALP91]

Un contexte permet, comme une vue, d'introduire un niveau d'abstraction entre le schéma de la base de données et les utilisateurs. Dans un contexte sont regroupés des éléments du schéma (par exemple, un ensemble de classes pour en masquer d'autres). Un modèle de versions de contextes est considéré dans *Farandole* [ALP91]. A partir d'une version de contexte donné, il est possible d'en dériver plusieurs autres qui correspondent chacune à des changements de schéma. Elles opèrent toutes sur la même collection d'objets. Cette approche remédie à certains inconvénients des précédentes dans la mesure ou la granularité de l'unité de l'évolution est variable, mais conduit à d'autres problèmes. Particulièrement, le versionnement du schéma devient très difficile.

II.4.2. Adaptation des instances

Pour maintenir la cohérence d'une instance vis-à-vis de son schéma, l'*émulation* ou le *versionnement des objets* [Ben94] sont utilisés comme techniques d'adaptation. Dans la suite de cette section les évolutions sont considérées au niveau de la classe. Lorsque nous raisonnons sur des évolutions aux niveaux schéma ou contexte, nous le précisons explicitement.

a) Émulation [ABB*84, Zdo86a]

L'émulation (Cf. Figure II.6.a) est proposée dans le prototype *Encore* [SZ86, Zdo86a, Zdo86b]. Un objet est associé à une version de la classe sous laquelle il a été créé. On ne restructure pas la représentation physique d'un objet après sa création. Le système doit garantir l'utilisation par les programmes, des objets créés avant et après l'évolution. Il fait cela par la simulation de l'interface de la nouvelle version de la classe au dessus des objets de l'ancienne version de la classe, ou vice-versa. Une technique similaire est utilisée dans le prototype *Avance* [BB88].

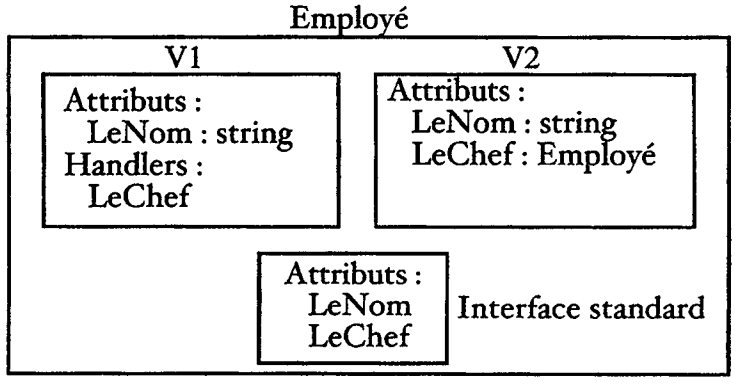


FIG. II.5 : la classe Employé

De manière générale, le mécanisme d'émulation est basé sur l'association, à chaque classe, d'une interface standard (*Version Set Interface*) définie par l'union des interfaces de ses versions de

classes. Quand la classe possède une seule version, l'interface standard de la classe est la même que celle de la version de classe. Quand une nouvelle version de classe est créée, l'interface standard de la classe est étendue par les attributs ajoutés. Par exemple, la création de la version V2 de la classe Employé, provoque l'ajout de l'attribut LeChef à l'interface standard (Cf. Figure II.5).

Seule l'interface standard est visible par les programmes. Puisque les attributs ne sont jamais supprimés, un programme trouve toujours la référence à un attribut, même si celui-ci n'existe pas pour des objets associés à certaines versions de la classe. Pour cela, dans une version de classe, l'utilisateur doit ajouter une procédure d'exception (*handler*) pour chaque attribut qui est défini dans l'interface standard et non défini dans cette version de classe. Cette procédure fournit une valeur (par défaut) de l'attribut quand celui-ci n'existe pas pour l'objet accédé sous cette version. Par exemple, une procédure d'exception qui fournit la valeur par défaut nil, peut être ajoutée pour l'attribut LeChef dans la version V1 de la classe Employé (Cf. Figure II.5).

Dans le système *Encore*, deux types de procédures d'exception sont considérées, selon que l'accès demandé est en lecture ou en écriture :

- En lecture (*Read-Handler)* : la procédure fournit une valeur par défaut, dés lors qu'un attribut non-défini est référencé dans un programme.

- En écriture (*Write-Handler)* : la procédure fournit une valeur booléenne. Cette valeur est "vrai" lorsque l'accès en écriture est possible, elle est "faux" sinon. L'accès en écriture est possible lorsque la valeur de l'attribut est la même que la valeur par défaut donnée par la procédure d'exception en lecture.

Cette technique assure la compatibilité des programmes vis-à-vis du schéma. Les anciens objets sont accessibles par les nouveaux programmes et inversement. Par contre, la définition des procédures d'exception nécessite un effort supplémentaire (qui est parfois considérable) de la part de l'utilisateur : à la suite de la dérivation d'une nouvelle version de classe, il ajoute des procédures d'exception dans certaines versions de cette classe afin d'assurer la comptabilité entre les objets et les versions de la classe. Le coût de l'émulation est fonction du nombre d'occurrences d'incompatibilités entre les objets et les programmes. Plus ce nombre est important, plus les performances du système se dégradent. Enfin, la valeur d'un attribut ajouté (respectivement, supprimé) dans un ancien (respectivement, nouvel) objet est toujours par défaut parce qu'il est impossible de restructurer la valeur d'un objet après sa création.

Pseudo-émulation

Dans [HZ90, Zdo90] on a proposé une technique dans laquelle un objet est représenté par des points de vues multiples. Cette technique remédie à l'inconvénient de la perte d'informations introduit par le mécanisme d'émulation de *Encore*. Ici, quand une nouvelle version de classe est créée, si celle-ci contient un nouvel attribut alors l'attribut est ajouté à la valeur de l'objet. Ainsi, la valeur d'un objet contient tous les attributs définis dans l'interface standard. Un point de vue est alors la valeur de l'objet sous une version particulière de la classe. Une technique similaire est proposée pour le système *Adele* [Nac94].

Odberg [Odb95] a proposé une technique basée sur les versions de schéma dans laquelle un objet est représenté par plusieurs *rôles* (ou points de vues). Ici, un objet est lié à une classe par l'intermédiaire d'un rôle. Comme dans la technique proposée dans [Zdo90], la valeur d'un objet contient tous les attributs définis dans toutes les classes de l'objet.

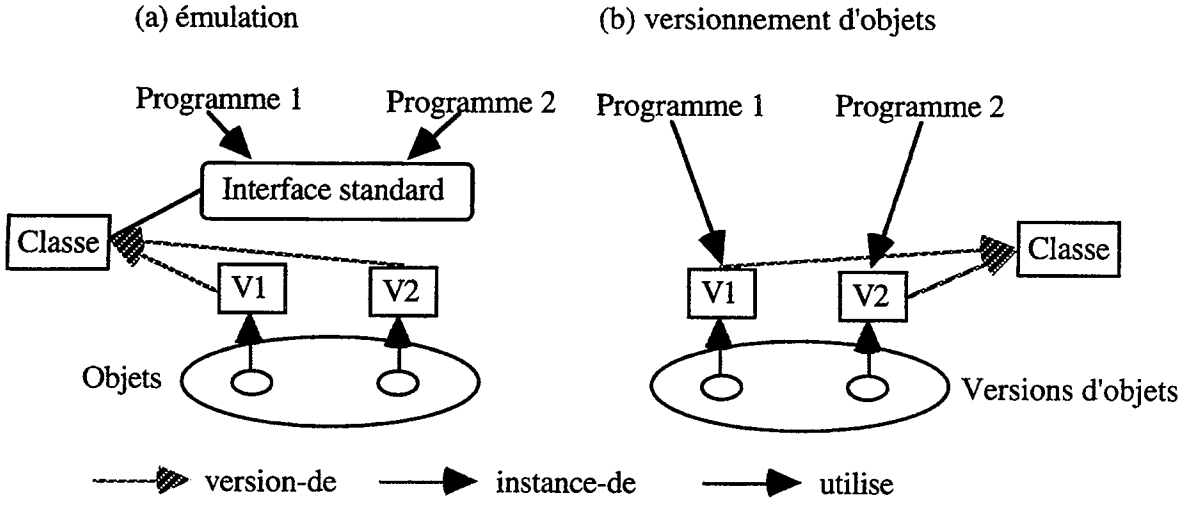


FIG. II.6 : émulation et versionnement d'objets

b) Versionnement des objets [Ben92, Cla92, ED95]

Un objet peut avoir plusieurs versions (Cf. Figure II.6.b). Une version d'objet est associée à une version de classe (une classe dans une approche version de schéma). L'évolution d'une classe se traduit par la dérivation d'une nouvelle version de la classe et la dérivation d'une nouvelle version de chacun de ses objets. Par exemple, l'évolution de schéma qui consiste à changer le type de l'attribut LeChef de la classe Employé de Employé à Enseignant (Cf. Figure II.2) provoque :

• La dérivation d'une nouvelle version de la classe Employé. Le type de l'attribut LeChef dans cette version de classe est Enseignant.

• La dérivation d'une nouvelle version de chacun des objets de l'ancienne version de la classe Employé. Les nouvelles versions d'objets sont associées à la nouvelle version de la classe Employé. La valeur de l'attribut LeChef dans une nouvelle version est une valeur du domaine du nouveau type de l'attribut (par exemple, l'objet nil).

Avec cette technique la perte d'informations est évitée. Par contre, le nombre de versions d'objets peut être important, dans la mesure où l'évolutivité est une caractéristique importante des applications à objets [BF95, Ben95a, Ben95b]. Ceci constitue l'obstacle majeur à l'utilisation d'une approche fondée sur les versions dans un système industriel car les coûts de stockage des versions et ceux de la maintenance des relations qui les lient augmentent avec le nombre de versions.

Le versionnement d'objets peut être *manuel* ou *automatique*, *immédiat* ou *différé*.

Le système *Objectivity* [Ob93] utilise le versionnement manuel des objets. Le changement du schéma ne se propage pas automatiquement sur les instances du schéma. L'utilisateur doit développer un programme qui dérive les nouvelles versions d'objets à partir des anciennes versions d'objets. Il doit spécifier le mode (*immédiat* ou *différé*) de transformation de la base de données.

Clamen [Cla92, Cla93] a proposé une technique de *versionnement différée* des objets. Après l'évolution du schéma et lorsqu'un accès à un objet est effectué sous une version de classe, une nouvelle version (*facet*) de l'objet, qui est conforme à la définition de la version de classe, est générée. L'espace de stockage est réduit par la possibilité du partage des attributs (par exemple, les attributs communs) entre les versions d'objets. La mise à jour de la valeur d'un attribut dans une version d'objet provoque, lorsqu'il y a une dépendance entre les attributs la mise à jour différée des attributs dans d'autres versions de l'objet.

Pseudo-versionnement

La technique proposée dans le prototype *CLOSQL* [MS93, Mon93] combine les versions de classes et la conversion différée des objets. L'évolution d'une classe est traduite par la dérivation d'une nouvelle version de cette classe. A un instant donné, un objet est associé à une seule version de classe. Lorsqu'un accès à un objet est demandé sous une autre version de classe, l'objet est converti pour être conforme avec la définition de cette version de classe. La conversion utilise les deux opérations *update* et *backdate* : l'opération *update* permet de convertir un objet d'une classe version de classe soit v_i en un objet d'une version v_{i+1} de la même classe, l'opération *backdate* est la fonction inverse de *update*. Ces opérations sont définies par l'utilisateur à la suite du changement du schéma, l'opération *update* est définie dans v_i, l'opération *backdate* est définie dans v_{i+1}. Cette approche évite la prolifération du nombre de versions d'objets.

II.5. Approches fondées sur les vues

II.5.1. Mécanisme de vue

Le mécanisme de *vue* [BM93, DST95, Run92, SDA94] est largement utilisé en bases de données pour accroître la flexibilité des modèles de données. Une vue sur un schéma est utilisée afin d'adapter le schéma à des besoins spécifiques d'un utilisateur (ou un groupe d'utilisateurs), de protéger les données, etc. Dans le contexte des bases de données à objets, une vue[2.1] (schéma virtuel ou schéma externe) est définie par une abstraction sur le schéma réel de la base [SDA94]. L'instance d'un schéma virtuel est calculée à partir de celle du schéma réel. Le mécanisme de vue offre aussi les fonctionnalités pour masquer des classes, ajouter des classes virtuelles, ajouter des attributs virtuels, masquer des attributs, sélectionner un sous-ensemble de l'extension d'une classe, etc.

Par exemple, le schéma de la figure II.7.a représente le schéma réel de la base et le schéma de la figure II.2.6.b représente une vue sur ce schéma dans laquelle la classe Enseignant est cachée.

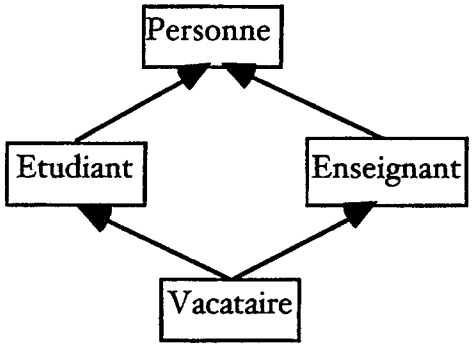


FIG. II.7.a : schéma réel

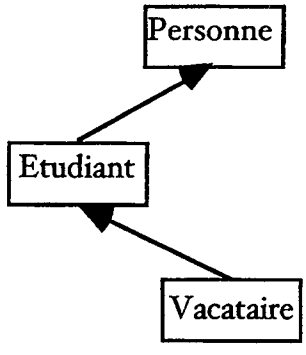


FIG. II.7.b : une vue

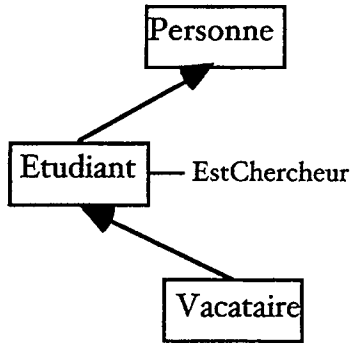


FIG. II.7.c : une vue

II.5.2. Vue & évolution de schéma

Certains travaux de recherche (voir, par exemple [Ber92, BFK95, Bra92, Liu94, RR94, TS93a, TS93b]) suggèrent l'utilisation du mécanisme de vue pour gérer l'évolution de schéma. La motivation principale de ces travaux est de supporter la transparence de l'évolution de schéma. L'idée est de traduire une évolution de schéma par la création d'une vue qui reflète la sémantique de cette évolution. Par exemple, supposons que la classe Etudiant dans la figure II.7.b contient l'attribut LeCycle : la valeur de cet attribut dans un objet est une chaîne de caractères indiquant le

[2.1] Dans cette section, nous utilisons indifféremment vue ou schéma virtuel.

cycle des études (par exemple, "1ème cycle"). L'utilisateur de cette vue décide d'ajouter l'attribut EstChercheur à la définition de la classe Etudiant tel que : la valeur de cet attribut est égal à vrai si celle de l'attribut LeCycle est égal à "3ème cycle" ; faux sinon. Cette évolution est alors traduite par la création de la vue de la figure II.7.c dans laquelle EstChercheur est un attribut *virtuel* de la classe Etudiant.

Une approche de transformation d'un schéma, qui réorganise le nouveau schéma en un ensemble de vues est proposée dans [TS93a, TS93b]. L'évolution de schéma est étudiée en utilisant un modèle formel basé sur la notion d'*interprétations d'un schéma* (en anglais *information capacity*): l'ensemble de tous les états possibles d'une base de données [MIR94]. Les auteurs de cette approche montrent que seules les évolutions qui ne nécessitent pas l'ajout des informations dans la base peuvent être traduites par un ensemble de vues (on appelle ces évolutions *information capacity reducing, information capacity preserving*).

Dans [Ber92] on propose un langage de définition de vues pour l'évolution de schéma. Des attributs et des méthodes peuvent être ajoutés dans la vue qui correspond au nouveau schéma. Une vue peut être dérivée à partir des classes du schéma réel et/ou des vues existantes. Les vues sont structurées en une hiérarchie qui est différente de la hiérarchie d'héritage. Ainsi, la vue et les classes dont elle est dérivée ne sont pas directement reliées. Par conséquent, les effets du changement d'une classe sur ses sous-classes ne sont pas pris en compte.

Dans l'approche *TSE* (Transparent Schema Evolution) [RR94], l'utilisateur peut définir une vue sur le schéma en utilisant le langage *MultiView* [Run92]. Une évolution est déclenchée par l'utilisateur d'une vue et elle est traduite par la dérivation d'une nouvelle vue qui remplace l'ancienne. Dans l'exemple précédent, la vue de la figure II.7.c remplace celle de la figure II.7.b. Puis les classes virtuelles sont intégrées dans le schéma global de la base en utilisant un algorithme de classification.

Les questions qui se posent dans ce cadre sont :

- Est-il est possible de traduire n'importe quelle évolution par des vues ?
- Le mécanisme de vue est-il suffisant pour supporter l'évolution de schéma ?

La réponse à la première question dépend de la puissance du langage de définition de vues utilisé. Il semble que les auteurs de cette approche donnent plutôt une réponse favorable.

Par contre la réponse à la deuxième question est décisive. Nous rappelons que :

• Le mécanisme de vue est concerné par la dérivation de nouvelles abstractions, à partir d'autres, et la propagation de la mise à jour d'une instance virtuelle à l'instance réelle.
• Le mécanisme d'évolution de schéma est concerné par le changement du schéma et la propagation des changements aux instances du schéma.

A partir de ces faits, nous pouvons dégager les constations suivantes :

• Le mécanisme de vue est insuffisant pour supporter les évolutions additives du schéma (c'est-à-dire, les évolutions qui nécessitent la réorganisation de la base par l'ajout d'informations ; dans [TS93a, TS94b] on appelle ces évolutions *information capacity augmenting*) comme l'ajout d'un attribut à une classe.

Les abstractions ajoutées par les vues (par exemple, un attribut virtuel, une classe virtuelle, etc.) sont dérivées à partir d'autres qui existent déjà, alors que les informations ajoutées par une évolution du schéma sont en général nouvelles et la plupart du temps indépendantes des informations existantes.

Notons que parfois, une évolution de schéma est de nature corrective, c'est-à-dire nécessite la mise à jour de la base de données ; donc sa traduction par des vues ne reflète pas le sens de l'évolution (par exemple, il est nécessaire de supprimer des informations qui sont obsolètes ou introduites par erreur).

• Les vues sont souvent structurées en une hiérarchie indépendante de la hiérarchie d'héritage du schéma réel. La position d'une classe virtuelle dans le schéma global, qui contient les classes réelles et virtuelles, dépend du processus de son intégration (on parle de l'algorithme d'intégration d'une classe virtuelle dans le schéma global). Par conséquent, la propagation des effets du changement d'une classe sur ses sous-classes n'est pas toujours considérée (parfois impossible, parfois difficile).

Simulation de l'évolution de schéma

Dans le cadre du projet *GOODSTEP* [BFK95], le concepteur peut définir un schéma virtuel en utilisant le langage *VDL* (View Definition Langage) au lieu de déclencher une séquence d'opérations de modification du schéma. Il teste si ce schéma virtuel reflète les modifications. Si oui, alors il *matérialise* le schéma virtuel en un schéma réel. La matérialisation consiste à traduire la définition du schéma virtuel en une séquence de commandes : les opérations de modification du schéma et les fonctions de conversion/migration d'objets (le langage *O2C* [OT94] est utilisé pour

écrire le code de ces fonctions). Cette séquence est appliquée sur le schéma réel comme s'il s'agit d'une modification du schéma.

Conclusion

Nous pouvons dire que le mécanisme de vue est une solution partielle au problème de l'évolution de schéma. Il peut offrir des fonctionnalités importantes, comme la transparence et la simulation. La puissance d'un mécanisme d'évolution de schéma peut donc être augmentée en lui associant la notion de vue.

Par exemple, la fusion de deux classes en une seule est une évolution qui est mieux gérée en exploitant les fonctionnalités d'un mécanisme de vue. Le problème de cette évolution est l'initialisation de l'extension de la nouvelle classe. Il existe souvent une relation de correspondance entre les objets de la nouvelle classe et ceux des anciennes classes. Par exemple, l'extension de la nouvelle classe contient certains objets qui sont construits par la fusion de deux objets chacun appartenant à l'extension d'une ancienne classe. Un mécanisme de vue offre la possibilité de dériver l'extension d'une classe (virtuelle) à partir d'autres, donc il permet de capturer les relations en question. Dans le cas de cette évolution, l'extension de la nouvelle classe peut être l'union des extensions des deux anciennes classes, une jointure sur les extensions des deux anciennes classes, etc.

Parmi les travaux sur l'intégration des mécanismes de vue et d'évolution de schéma nous citons Tresch et Scholl [TS93a, TS93b], Bratsberg [Bra92, Bra93] et Liu [Liu94].

II.6. Travaux similaires dans d'autres domaines

Cette section identifie les travaux similaires à l'évolution du schéma en bases de données dans d'autres domaines, et introduit brièvement les solutions proposées.

II.6.1. Éditeurs syntaxiques

Un éditeur syntaxique appartient à une classe d'environnements de programmation, qui est celui des environnements structurés, comme *Gandalf* [GKL86], *Centaur* [BCD*87] et *Synthesizer Generator* [RT84]. Il associe aux programmes (on parle ici des textes de programmes) des arbres syntaxiques (abstraits). Les arbres syntaxiques sont décrits par une description formelle ou une *grammaire*. Un arbre syntaxique est une donnée persistante. L'analogie avec les bases de données est :

• une grammaire dans un éditeur syntaxique représente ce qui est un schéma dans une base de données.

• un arbre syntaxique dans un éditeur syntaxique représente ce qui est une valeur (ou un objet) dans une base de données.

Dans le domaine des éditeurs syntaxiques, le besoin en évolution des grammaires est très fréquent [GKL86]. Par exemple, le changement de la production "X::=ab" en "X::=abc", on parle de l'ajout du composant c à la production X. Quand on change une grammaire (ajouter/supprimer une production, ajouter/supprimer un composant, etc.), les arbres syntaxiques peuvent devenir non conformes avec la nouvelle grammaire. Ainsi, l'exécution des programmes donne souvent des résultats erronés.

Parmi les travaux dans ce domaine, nous citons la solution proposée pour le système *Gandalf* [GKL86] où l'outil *TransformGen* est développé. Dans TransformGen, le programmeur peut changer la grammaire. A partir de la grammaire et du changement associé, cet outil génère systématiquement la nouvelle grammaire et un ensemble de règles de transformation des arbres syntaxiques. Ces règles permettent de transformer des arbres syntaxiques qui sont conformes à la grammaire avant l'évolution, en des arbres syntaxiques qui sont conformes à la grammaire après l'évolution. Lorsque les règles de transformation ne donnent pas satisfaction, c'est-à-dire que la transformation des arbres syntaxiques ne génère pas les résultats attendus, le programmeur peut modifier ces règles pour les adapter à ses besoins.

II.6.2. Édition structurée de documents

Dans un système d'édition structurée de documents, comme *Grif* [Akp93] et *Rita* [CMS*91], les documents sont conformes à une description logique qui est basée sur les *grammaires hors contexte* . On parle de schémas de structure. Ceux-ci décrivent la structure et les relations entre les documents. Par exemple, le schéma de structure Article décrit une classe de documents qui sont des articles. Il contient des éléments de description (ou attributs) comme Auteur, Contenu, etc.

Deux types d'évolutions sont identifiés : les évolutions statiques et les évolutions dynamiques. Une évolution statique concerne la structure logique, comme l'ajout/suppression d'une classe de documents, l'ajout/suppression d'un élément à une classe de documents ou la réorganisation de la composition d'une classe de documents. Une évolution dynamique a lieu en cours d'édition comme l'opération copier/coller.

Deux problèmes en découlent : (1) à la suite du schéma de structure, certains documents peuvent être non conformes à leur schéma de structure. Par conséquent, il devient impossible

d'éditer ces documents. (2) Il est impossible de restructurer les documents en cours d'édition ou d'utiliser l'opération couper/coller.

Parmi les travaux dans ce domaine, nous citons la solution proposée pour le système *Grif* [Akp93]. Un ensemble de relations structurelles entre types (par exemple, sous-typage, compatibilité et équivalence) est défini pour traiter l'évolution. En plus de la prise en compte des évolutions statiques, une approche grammaticale est utilisée pour les évolutions dynamiques. A un schéma correspond une grammaire algébrique. Des règles de transformation d'un schéma de structure en une grammaire algébrique sont définies. Ainsi, un document est considéré comme un mot de la grammaire.

II.6.3. Représentation de connaissances par objets

La représentation des connaissances par objets est le résultat de l'intégration des réseaux sémantiques, des frames et du modèle à objets [MNC*89]. Ce domaine se caractérise par le nombre élevé des classes. Par exemple, dans les bases de connaissances pour la biologie et la médecine, plus de 3000 classes sont en général recensées. L'ajout manuel d'une classe devient difficile. La recherche de la position de la nouvelle classe dans la hiérarchie par l'utilisateur n'est ni facile (l'utilisateur ne connaît pas toutes les classes) ni pratique (le temps de recherche peut être important). Pour cette raison le processus de recherche de la position adéquate d'une nouvelle classe a été (semi) automatisé. Il s'agit alors de la classification automatique ou assistée.

La classification est un mécanisme d'inférence particulier, basé essentiellement sur l'héritage et la comparaison de définitions des classes. Parmi les travaux dans ce domaine, nous citons la solution proposée pour le modèle *Tropes* [Cap95]. La solution consiste à déterminer, au fur et à mesure de sa description, la position de la nouvelle classe dans la hiérarchie, en se basant sur une relation de sous-typage sur les attributs. Une évolution de la hiérarchie est propagée au niveau de la hiérarchie et au niveau des instances.

II.6.4. Langages de programmation à objets

Le problème de l'évolution de type en langages de programmation à objets persistants [CCK*94] se pose de la même manière que l'évolution de schéma en bases de données à objets. Dans ce domaine, des travaux sur la restructuration de la hiérarchie des classes sont considérés [Cas91, LWX93, Opd92].

Étant donnés une hiérarchie de classes et un ensemble d'assertions (par exemple, une assertion consiste à dire qu'une méthode n'utilise que les méthodes de sa classe), la restructuration consiste à transformer la hiérarchie de classes en une autre qui satisfait les assertions de l'ensemble, en utilisant des règles de transformation [LWX93, JF88]. En bases de données relationnelles, la restructuration est utilisée pour *normaliser* des schémas de relations [DA82].

En programmation à objets, les programmeurs développent des applications en utilisant des classes existantes (bibliothèques de classes). Ils ont souvent besoin d'adapter des classes selon leurs besoins spécifiques car les hiérarchies de classes existantes ne sont pas toujours satisfaisantes [JF88].

La motivation principale pour la restructuration d'une hiérarchie de classes est de remédier à ce problème, c'est-à-dire, de restructurer pour produire une hiérarchie de classes plus *réutilisable* et facilement *modifiable* (sa modification est peu coûteuse) ; on parle de la qualité d'une hiérarchie de classes [Mey88]. Dans [JF88, LHR88], on propose des règles pour guider le concepteur afin de concevoir des classes plus générales, abstraites, réutilisables et modulaires (modularité des propriétés des classes).

Les travaux sur la restructuration concernent la conception et la re-conception d'une hiérarchie de classes, le problème de la propagation des effets de la restructuration au niveau des instances des classes n'est pas posé. Parmi ces travaux, nous citons la loi de *Demeter* [Lie92, LH89a, LH89b, LX93], l'approche algorithmique pour l'évolution proposée par Casais [Cas91, Cas92, GTC*90] et les primitives de restructuration proposées par Opdyke [Opd92].

La loi de *Demeter* établit des propriétés que doivent satisfaire les méthodes dans une hiérarchie de classes. Le but est d'organiser et de réduire les dépendances entre les classes en se basant sur les dépendances de leurs méthodes. Par conséquent, certains principes de conception de bons logiciels sont renforcés (augmenter le degré de réutilisation et de modularité, faciliter la maintenance, etc.). Des règles de transformation sont proposées pour obtenir une hiérarchie de classes conforme à cette loi.

Casais propose des algorithmes pour restructurer une hiérarchie de classes à la suite d'une nouvelle classe (une feuille de la hiérarchie). La définition de la nouvelle classe peut affiner, redéfinir ou éliminer des propriétés héritées de ses super-classes. La restructuration est basée sur la structure des classes, la relation d'héritage et les dépendances entre les attributs. Les algorithmes analysent les adaptations apportées par la classe et restructurent la hiérarchie (par exemple, ajouter

des classes abstraites intermédiaires) en fournissant une nouvelle hiérarchie, qui vérifie les contraintes de spécialisation et factorise les propriétés.

Opdyke propose des primitives de restructuration (refactoring) d'une hiérarchie de classes comme la création d'une super-classe abstraite, la spécialisation, l'agrégation (échange de propriétés entre une classe agrégat et une classe composante, convertir une relation d'héritage en une agrégation, etc.). La restructuration préserve le comportement, c'est-à-dire les programmes sont syntaxiquement corrects après le restructuration.

II.7. Conclusion

II.7.1. Tableaux récapitulatifs

Les tableaux suivants donnent un résumé des différents systèmes qui offrent la fonctionnalité d'évolution du schéma. Les systèmes que nous considérons sont, selon le cas, des prototypes ou des produits commerciaux. Le choix a été imposé par l'absence ou la présence d'informations disponibles au travers de publications. Les tableaux 1, 2 et 3 résument les différentes opérations de changement du schéma. Le tableau 4 résume les approches de gestion de l'évolution de schéma.

Table 1 : Opérations de changement d'une classe

Propriété	Encore	GemsTone	Orion	OTGen	COCOON	Farandole	O2
ajout	Y	Y*	Y	Y	Y*	Y	Y
suppression	Y	Y*	Y	Y	Y*	Y	Y
renommage		Y*	Y	Y	Y*		Y
redéfinition de type	Y	Y*	Y**	Y	Y*	Y	Y*

Table 2 : Opérations de changement d'une hiérarchie de classes

Classe	Encore	GemsTone	Orion	OTGen	COCOON	Farandole	O2
ajout	Y	Y	Y	Y	Y	Y	Y
suppression	Y***	Y***	Y	Y	Y	Y	Y
renommage		Y	Y	Y	Y		Y

Table 3 : Opérations de changement des liens d'héritage

Lien	Encore	GemsTone	Orion	OTGen	COCOON	Farandole	O2
ajout	Y		Y	Y		Y	Y
suppression	Y		Y	Y		Y	Y
changement d'ordre	Y		Y				Y

Y : : le système offre l'opération

Y* : l'opération concerne un attribut

Y** : le type d'un attribut ne peut être que généralisé

Y*** : l'opération est possible seulement si l'extension de la classe est vide

Table 4 : Approche d'évolution de schéma

	Schéma	Instance	Evolution multiples
Adele3	versions de classes	émulation	
Avance	versions de classes	émulation	
COCOON	modification	conversion différée	
CLOSQL	versions de classes	conversion différée	
Encore	versions de classes	émulation	
Farandole	versions de contexte	émulation	
GemStone	modification	conversion immédiate	
Itasca	modification	conversion différée	
O2	modification	conversion*	partielle
ObjectStore	modification	conversion immédiate	
Objectivity	versions de classes	versionnement manuel	
Ontos	modification	conversion manuelle	
Orion	modification	émulation	
Versant	modification	conversion différée	
Bratsberg [Bra94]	versions de classes	versionnement *	partielle
Clamen [Cla92]	versions de classes	versionnement différée	
Katayama et Tan [KT91]	modification	conversion différée	
Odberg [Odb95]	versions de schéma	versionnement**	oui
Liu [Liu94]	versions de schéma	émulation*	oui
OTGen	modification	conversion diffrée	partielle
SHOOD [Bou95]	modification	conversion immédiate	

conversion* : conversion différée et immédiate

versionnement* : versionnement différé et immédiat

versionnement** : basé sur la notion de rôles d'un objet (mode immédiat et différé)

émulation* : les objets sont des vues sur des relations stockées dans la base

II.7.2. Synthèse

Dans ce chapitre, nous avons proposé une classification des approches de gestion de l'évolution du schéma. Nous avons montré que ces approches sont souvent complémentaires :

Approche fondée sur la modification

Une évolution du schéma s'exprime par la modification d'une classe ou d'un lien d'héritage. Les nouvelles définitions des classes remplacent les anciennes qui sont perdues. L'adaptation des instances est de type "conversion" : les objets de la base doivent être convertis pour avoir une instance valide vis-à-vis du nouveau schéma. Cela peut provoquer une perte d'informations. De plus, les programmes qui utilisent l'ancien schéma peuvent être incompatibles vis-à-vis du nouveau schéma.

Approche fondée sur les versions

Une évolution du schéma s'exprime par la dérivation de nouvelles versions de classes ou du schéma. Les classes et les objets vont pouvoir prendre plusieurs formes selon le stade de l'évolution de la base où ils interviennent. Les programmes qui étaient exécutables sous l'ancienne version du schéma restent toujours exécutables sous cette version car elle est conservée. L'inconvénient principal de cette catégorie est qu'à la complexité de la navigation dans le graphe d'héritage et de composition est ajoutée celle de la navigation dans une hiérarchie de versions.

Deux types de techniques d'adaptation des instances sont considérées :

• **Versionnement** : la dérivation d'une nouvelle version d'une classe provoque systématiquement la génération d'une nouvelle version de ses objets. Ainsi, le problème de la perte d'informations est résolu. L'inconvénient principal de cette approche est l'augmentation du nombre de versions. Cela peut avoir des conséquences sur le fonctionnement du système car les coûts de stockage des versions et de maintenance des relations qui les lient augmentent avec le nombre de versions.

• **Émulation** : l'émulation consiste à assurer l'accès aux objets, sans les modifier, sous toutes les versions de la classe. Ici, en plus de la dégradation des performances lors des accès aux objets, due à la complexité des mécanismes mis en jeu, la valeur associée à un attribut ajouté (respectivement, supprimé) dans un ancien (respectivement, nouvel) objet est constante (valeur par défaut). En effet, cette valeur est la même pour tous les objets, et ne peut pas être modifiée car elle n'existe pas dans la structure physique de l'objet.

Approche fondée sur les vues

L'idée est de traduire une évolution de schéma par la création d'une vue qui reflète la sémantique de cette évolution. Cette approche est suffisante pour gérer des évolutions de schéma qui ne nécessitent pas la réorganisation physique des informations de la base de données. Elle apporte des solutions satisfaisantes pour les évolutions de schéma dites multiples. Nous avons montré que la puissance d'un mécanisme d'évolution de schéma peut être augmenté en lui associant la notion de vue.

L'objectif principal que nous voulons atteindre est de trouver un compromis entre les fonctionnalités de ces approches. L'approche que nous proposons combine la modification et les versions d'une part et intègre d'autre part la puissance des langages déclaratifs (comme dans un mécanisme de vue) pour permettre la description de la sémantique d'une évolution de schéma.

Chapitre III

LE MODÈLE DE VERSIONS : concepts et évolution de schéma

Notre approche pour gérer l'évolution de schéma est basée sur les versions de schéma et d'objets. Cette approche sera présentée dans les chapitres IV et V. Ce chapitre présente le modèle de versions à la base de notre approche. Nous discutons de la notion de cohérence structurelle d'une base de données. Nous proposons une formalisation du problème de l'évolution de schéma.

III.1. Introduction

Dans la première partie de ce chapitre nous développons un modèle de versions qui offre les concepts de base pour une approche d'évolution de schéma fondée sur les versions (Cf. Section III.2). La réalité modélisée consiste en un ensemble d'entités (objets, schéma) qui évoluent dans le temps. Chaque entité est associée à un ensemble de versions qui reflètent son évolution. Une entité n'est manipulée que par l'intermédiaire de ses versions.

Dans la deuxième partie de ce chapitre (Cf. Section III.3) nous développons un modèle formel d'une base de données. Comme nous l'avons vu dans le chapitre I, le problème de l'évolution de schéma est lié à la notion de cohérence d'une base de données. Ce modèle permet la description de la structure du schéma d'une base de données et ses instances. Notre but est de formaliser pour mieux comprendre et expliquer le problème de l'évolution de schéma. Nous discutons de la notion de la cohérence structurelle d'une base de données dans le cadre de ce modèle. C'est-à-dire, la notion de la cohérence de structure de schéma et la cohérence d'une instance vis-à-vis de son schéma.

Dans la troisième partie de ce chapitre (Cf. Section III.4) nous proposons, en utilisant le modèle formel, une formalisation du problème de l'évolution de schéma. Nous donnons l'énoncé du problème et discutons ses solutions possibles. Dans cette partie, étant donnée une base de données et une évolution de schéma, les problèmes sont de vérifier la cohérence de structure du schéma résultant, vérifier la cohérence de l'instance vis-à-vis du schéma résultant (un schéma cohérent) et identifier une séquence de mises à jour sur l'instance afin de restaurer sa cohérence.

III.2. Modèle de versions

Le modèle de versions que nous proposons est basé sur la notion d'objet dont les principaux concepts sont décrits dans [AC93, AK89, Cat93].

III.2.1. Concepts généraux

III.2.1.1. Entité - Version d'entité

Nous utilisons le terme *entité* indifféremment pour désigner un schéma ou un objet de la base de données.

Durant son évolution, une entité passe par des étapes clés. A chacune de ces étapes est associée une *version* de l'entité. Ainsi, une version caractérise un état de l'entité que le système ou l'utilisateur juge important de conserver (par exemple, pour assurer la pérennité des données, améliorer les performances ou tracer l'évolution de l'entité). Ainsi, à une entité est associée un ensemble de versions (Cf. Figure III.1).

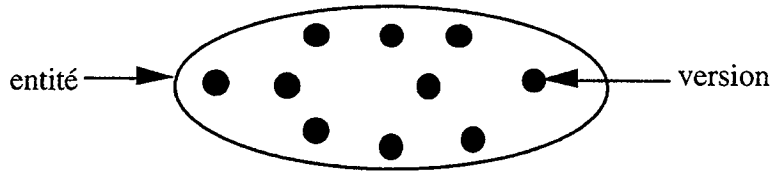


FIG. III.1 : version, entité

Au niveau du système, une entité consiste en un couple (ent_id, vers) tel que, ent_id est l'identificateur de l'entité et vers est l'ensemble des identificateurs des versions de l'entité. L'identificateur d'une version est un couple (ent_id, numéro) tel que, ent_id est l'identificateur de l'entité et numéro est le numéro que le système associe à la version au moment de sa création. La fonction `VersSet`(ent) associe à l'entité ent l'ensemble de ses versions.

III.2.1.2. Processus d'évolution - Dérivation

Le processus d'évolution est une séquence d'actions qui tend à construire le premier état de l'entité ou à transformer une entité, d'un certain état vers un autre. Par exemple, pour un schéma le nouvel état peut mettre en évidence de nouvelles fonctionnalités du monde réel.

La *première version* d'une entité est créée au moment de la création de l'entité. Une nouvelle version d'entité est *dérivée* à partir d'une autre. Dans une première étape, la nouvelle version est générée par copie ; puis dans une deuxième étape, cette copie est modifiée.

Dans la figure III.2, l'ensemble des versions d'une entité est structuré en une séquence. Un lien entre deux versions symbolise la dérivation. La version v_0 est la première version de l'entité (racine de la séquence de dérivation). La version v_1 est dérivée à partir de v_0. Nous notons que le lien de dérivation introduit un ordre total sur les versions d'une entité.

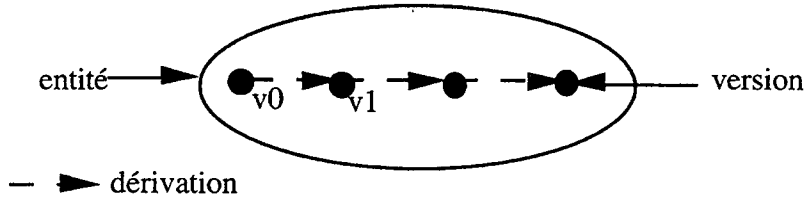


FIG. III.2 : lien de dérivation

III.2.2. Schéma - Version de schéma

Une entité *schéma* est associée à un ensemble de versions de schéma, structuré en une séquence. La première version du schéma est la version *racine*. Les autres versions s'il en existe, peuvent contenir des classes déduites d'autres versions du même schéma. La dernière version du schéma (la plus récente) est la version *courante* du schéma, toutes les autres versions du schéma sont des versions *historiques*. Une version de schéma consiste en un triplet (sid, numéro, valeur) tel que, (sid, numéro) est l'identificateur de la version et valeur est la valeur de la version. La valeur d'une version du schéma est un ensemble de classes reliées par les relations d'héritage et de référence [BM93].

III.2.2.1 Exemple

La figure III.3 montre deux versions ("GestionUniv", 0) et ("GestionUniv", 1) du schéma "GestionUniv". La version ("GestionUniv", 0) est la première version du schéma. La version ("GestionUniv", 1) est dérivée à partir de ("GestionUniv", 0). Dans une version du schéma un lien entre deux classes symbolise l'héritage. La définition de la classe Personne dans ("GestionUniv", 1) est la même que dans ("GestionUniv", 0). La définition de la classe Employé dans ("GestionUniv", 1) est déduite à partir de celle de la classe Employé dans ("GestionUniv", 0) par l'ajout de l'attribut LeService. La classe Etudiant est définie seulement dans ("GestionUniv", 0).

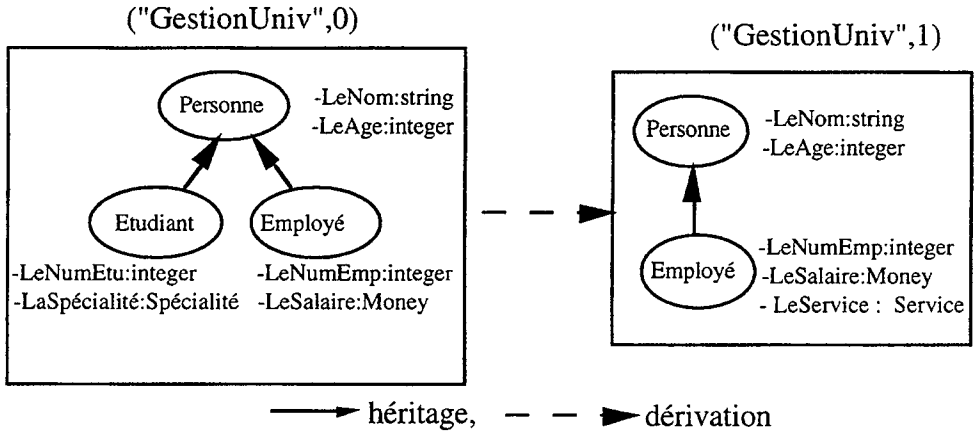


FIG. III.3 : versions d'un schéma

III.2.2.2. Caractérisation des classes

Dans une version de schéma, une classe peut être :

• *importée* : c'est-à-dire définie dans une précédente version du schéma. Ainsi, une classe peut être partagée par plusieurs versions du schéma.

• *dérivée* : une classe dont la définition est déduite à partir d'une autre définie dans la version précédente du schéma. Cette classe a tout d'abord été copiée dans la version courante du schéma, à partir de la version précédente. Puis elle a été modifiée. Elle a le même nom que la classe dont elle est déduite.

• *locale* : une classe qui est créée dans cette version du schéma.

Dans l'exemple (Cf. Figure III.4) toutes les classes définies dans la version ("GestionUniv", 0) du schéma sont locales à cette version (il ne peut pas en être autrement). La classe Personne de ("GestionUniv", 1) est importée de ("GestionUniv", 0). La classe Employé de ("GestionUniv", 1) est dérivée à partir de la classe Employé de ("GestionUniv", 0). La figure III.5 montre le même schéma que celui décrit par la figure 3.4, mais donne une vision qui reflète plus la caractérisation des classes. La classe Personne est partagée par les deux versions du schéma, un lien entre deux classes dans différentes versions du schéma symbolise la dérivation.

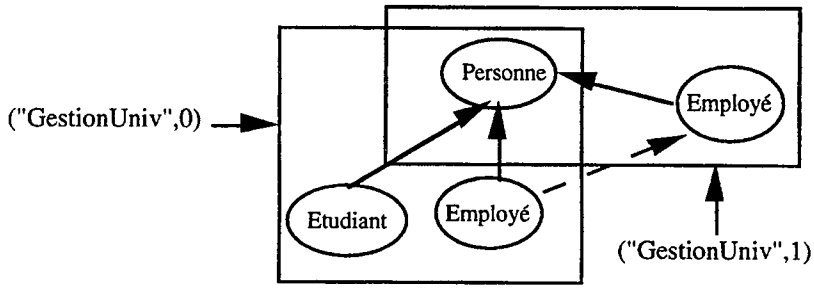


FIG. III.4 : caractérisation des classes

Notons que nous ne considérons toujours, à un moment donné, qu'un seul schéma. Dans la suite, une classe est identifiée par son nom et le numéro de la version où elle est définie, c'est-à-dire par un couple (c, n) où c est le nom de la classe et n est le numéro de la version du schéma. Par exemple, (Employé, 0) est la classe de la version 0 du schéma de la base de données.

III.2.2.3. Primitives de base

Dans ce qui suit, nous dénotons par S l'ensemble des noms de schémas, par C celui de noms des classes et par N l'ensemble des entiers naturels. Nous introduisons tout d'abord deux fonctions nécessaires à la manipulation des versions :

`SchVerClasses` : $S \times N \rightarrow 2^{C \times N}$
{`SchVerClasses`(s, n) est l'ensemble de classes de la version numéro n du schéma s.}

`Current` : $S \rightarrow S \times N$
{`Current`(s) est la version courante (la plus récente) du schéma s.}

Nous proposons deux opérations de base associées au concept de version de schéma : l'opération de création de la première version du schéma et l'opération de dérivation d'une version à partir d'une autre.

schema_version_creation (<nom_schéma>)
<spécifications_de_classes>

• Cette opération crée la première version du schéma nom-schéma. Elle est déclenchée par la définition d'un schéma effectuée par l'utilisateur. L'identificateur de cette version est initialisé par le couple (nom_schéma, 0). La valeur de cette version est la hiérarchie des classes définies dans la clause <spécifications_de_classes>. Toutes les classes sont locales.

schema_version_derivation (<nom_schéma>)
<changement_du_schéma>

• Cette opération dérive une nouvelle version du schéma nom-schéma à partir de la version courante du schéma en utilisant la séquence d'opérations de modification du schéma spécifiée par la clause <changement_du_schéma> (Cf. Annexe A).

Soit (s, n) l'identifiant de la version courante du schéma s. (s, n+1) est l'identifiant de la nouvelle version, qui devient la version courante. (s, n) devient une version historique.

La valeur de la nouvelle version est déduite en suivant les étapes suivantes : 1) une copie de la version précédente est effectuée. 2) cette copie est modifiée en utilisant la séquence des opérations définies dans la clause <changement_du_schéma>. Les classes qui ne sont pas affectées par le changement sont des classes *importées*, les classes affectées par le changement sont des classes *dérivées* et les nouvelles classes sont des classes *locales*.

Exemple. Nous reprenons ci-dessus l'exemple donné dans la figure III.3 :

```
schema_version_creation ("GestionUniv")
class Personne tuple (
        LeNom : string,
        LAge : integer)
end;

class Etudiant :inherit Personne tuple (
        LeNumEtu : integer,
        LaSpécialité : Spécialité)
end;
```

```
class Employé inherit  Personne tuple(
        LeNumEmp : integer,
        LeSalaire; : Money)
end;

/* Les classes Spécialité et  Money sont définies par ailleurs  */
```

FIG. III.5 : schéma "GestionUniv"

La version ("GestionUniv", 1) de la figure III.6 est obtenue en appliquant l'opération de dérivation d'une nouvelle version du schéma sur le schéma "GestionUniv" (la version courante est ("GestionUniv", 0)) :

```
schema_version_derivation ("GestionUniv")
Add_class (Enseignant, interface Enseignant : Employé {.........});
Del_Edge (Etudiant, Personne);
Add_Edge (Etudiant, Employé);
```

La hiérarchie de classes associée à la nouvelle version ("GestionUninv", 1) est déduite de celle associée à ("GestionUniv", 0) comme suit :
1) l'ajout de la classe Enseignant comme une sous-classe de la classe Employé.
2) la suppression du lien d'héritage entre les classes Etudiant et Personne et l'ajout d'un lien d'héritage entre les classes Etudiant et Employé.

Les classes Personne et Employé sont importées, la classe Enseignant est locale à la nouvelle version du schéma et Etudiant est dérivée car elle est affectée par la modification.

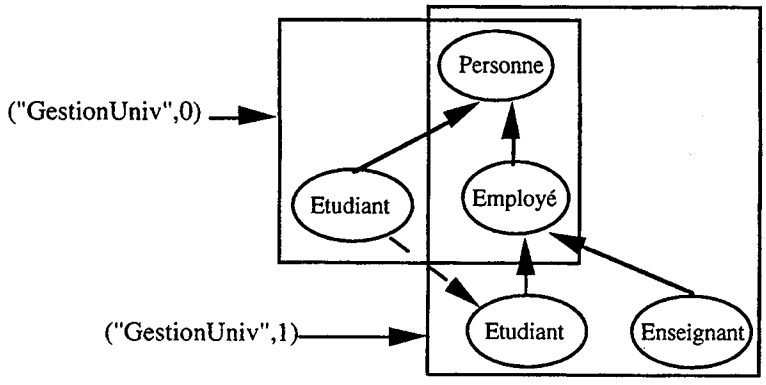


FIG. III.6 : dérivation d'une version du schéma

III.2.3. Objet - Version d'objet

Une entité objet est associée à un ensemble de versions d'objet. Sous une version du schéma, une entité objet est associée à une seule classe. Cette même entité peut être associée à une autre classe dans une autre version du schéma. La représentation et le comportement d'une entité objet sous une classe constitue la version de l'objet sous cette classe.

Une version d'objet est un triplet (oid, numéro, valeur) tel que, oid est l'identificateur de l'objet, numéro est un entier naturel et valeur la valeur de la version d'objet. La version d'objet est identifiée par le couple (oid, numéro). Puisque dans notre modèle, un objet a au plus une version sous une classe, numéro est celui de la classe correspondante. La valeur est un élément du domaine d'un type atomique (par exemple, entier) ou un élément du domaine d'un type construit (par exemple, une valeur ensemble ou une valeur n-uplet). Une version d'objet est une instance d'une classe.

III.2.3.1. Exemple

Reprenons le schéma "GestionUniv" (Cf. Figure III.6). Joseph est un objet, qui admet deux versions, une dans la classe Etudiant de ("GestionUniv", 0) et une autre dans la classe Etudiant ("GestionUniv", 1). Autrement dit, (Joseph, 0) est une instance de (Etudiant, 0) et (Joseph, 1) est une instance de (Etudiant, 1).

III.2.3.2. Primitives de base

Dans la suite, O dénote les identificateurs d'objets et VAL dénote l'ensemble des valeurs possibles pour les versions d'objets. Nous introduisons tout d'abord trois fonctions nécessaires à la manipulation des versions :

`ObjVerValue` : $O \times N \rightarrow VAL$
{`ObjVerValue`(o, n) est la valeur de la version numéro n de l'objet o.}

`Objects` : $C \times N \rightarrow 2^O$
{`Objects`(c, n) est l'ensemble d'objets associé à la classe (c, n). Elle définit une partition sur l'ensemble des objets.}

Ext : $C \times N \rightarrow 2^{O \times N}$

{Ext(c, n) est l'extension de la classe (c, n). C'est-à-dire, l'ensemble de versions d'objets associé à (c, n). Elle définit une partition sur l'ensemble des versions d'objets.}

Nous proposons quatre opérations associées au concept de version d'objet : création de la première version d'objet, suppression d'une version d'objet, dérivation d'une nouvelle version d'objet et conversion d'une version d'objet. Notons que la définition de ces opérations contient la clause <initialisation> sur laquelle nous revenons dans le chapitre 5.

object_version_creation (<ident_objet>, <classe>)
[<initialisation>]

• Cette opération crée la première version de l'objet identifié par ident_objet. Elle est déclenchée par la création d'un objet en utilisant la primitive *create* (primitive pré-définie) dans un programme. L'identificateur de cette version est le couple (ident_objet, n) où n est le numéro de la classe sous laquelle l'objet est créé. Cette classe est identifiée par classe. La valeur de cette version est initialisée en utilisant les paramètres de la primitive create. La clause <initialisation> (si elle spécifiée) contient des valeurs qui sont utilisées pour initialiser certains attributs.

object_version_deletion (<ident_version>)
[<initialisation>]

• Cette opération supprime la version de l'objet identifiée par ident_version. Cette opération déclenche la mise à jour des versions qui référencent la version supprimée. Cette mise à jour remplace chaque référence à cette version par la valeur nil ou par une version d'objet dont le type est un sous-type de la version supprimée. Cette dernière est spécifiée dans la clause optionnelle <initialisation>.

object_version_derivation (<ident_objet>, <classe1>, <classe2>)
[<initialisation>]

• Les classes classe1 et classe2 sont associées à des versions adjacentes[3.1] du schéma. Cette opération crée une nouvelle version de l'objet identifié par ident_objet. Cette version est une instance de la classe classe2 et elle est dérivée à partir de la version de l'objet sous la classe

[3.1] Dans le chapitre IV, nous considérons des versions non adjacentes du schéma.

classe1. Elle est générée par le mécanisme de propagation associé à l'évolution du schéma (Cf. Chapitre IV). L'identificateur de cette version est le couple (ident_objet, n), où n est le numéro de la classe classe2. La valeur de la version est conforme avec la structure de la classe classe2. Elle est déduite à partir de la version de l'objet sous la classe classe1 en utilisant le mécanisme de transformations par défaut [Ben92, BF94, LH90, FMZ95a] (Cf. Annexe C).

Pour plus de clarté, nous en rappelons ici le principe. Il est basé sur la comparaison des structures des deux classes. La valeur associée à un attribut qui est défini seulement dans classe2 est initialisée par la valeur par défaut du type de l'attribut (0 pour un entier, nil pour une classe, etc.). La valeur associée à un attribut présent dans les deux classes classe1 et classe2 est déduite selon la relation qui existe entre les types de l'attribut dans les deux classes. Par exemple, si les types de l'attribut sont des classes, alors la valeur de l'attribut est maintenue seulement si le type de l'attribut dans classe1 est un sous-type de son type dans classe2. Autrement, elle est initialisée par la valeur nil.

Opération non primitive

object_version_conversion (<ident_objet>, <classe1>, <classe2>) [<initialisation>]

• Les classes classe1 et classe2 sont associées à des versions adjacentes du schéma. Cette opération provoque la migration (conversion) de la version d'objet de la classe classe1 vers la classe classe2. Cette opération a la même sémantique que la dérivation. Cependant, après la génération de la version de l'objet sous la classe classe2, la version de l'objet sous la classe classe1 est supprimée en utilisant l'opération de suppression d'une version d'objet décrite plus haut.

Exemple. Reprenons l'exemple III.2.3.1 et supposons que l'objet Joseph a évolué de la manière suivante : il est créé dans la classe (Etudiant, 0) ; puis quand la classe (Etudiant, 1) est dérivée, une nouvelle version de Joseph est dérivée à partir sa première version.

• La création de l'objet déclenche la création de sa première version :

object_version_creation (Joseph, (Etudiant, 0));

• La dérivation de la version de Joseph sous la classe (Etudiant, 1) de sa version sous (Etudiant, 0) utilise la primitive :

object_version_derivation (Joseph, (Etudiant, 0), (Etudiant, 1));

III.2.4. Liaison entre une version d'objet et programme

La gestion des versions est transparente à l'utilisateur. Que ce soit dans un programme ou dans une requête, l'usager ne manipule que des objets ou des classes, jamais des versions. Le mécanisme que nous introduisons ici est utilisé par le système pour établir la correspondance entre l'entité à laquelle l'usager veut accéder et la version qui lui correspond à un moment donné. La liaison est faite de la manière suivante :

Au moment de sa compilation, un programme est associé à la version courante du schéma, sous laquelle il est ensuite exécuté jusqu'à une éventuelle re-compilation. La re-compilation d'un programme entraîne la ré-association de ce programme à la version courante du schéma, qui a pu changé depuis la compilation précédente. L'expression `LinkPrSch(p)` (liaison entre un programme et un schéma) est la version du schéma associée au programme p. C'est la version sous laquelle p a été compilé.

La référence à un objet dans un programme est une référence à une version de cet objet. C'est la version de l'objet sous une classe de la version du schéma associée au programme. L'expression `LinkObjVer(o, p)` est la version de l'objet o qui correspond à la référence de o dans le programme p, c'est-à-dire :

$$\texttt{LinkObjVer}(o, p) \in \texttt{Ext}(c, n) \text{ tel que,}$$
$$(c, n) \in \texttt{SchVerClasses}(\texttt{LinkPrSch}(p))$$

III.3. Un modèle formel d'une base de données

Dans la section précédente nous avons défini les notions de schéma et d'objet. Une base de données consiste en un schéma et une instance qui est un ensemble d'objets. Nous discutons ici de la cohérence d'une base de données, c'est-à-dire de son schéma et son instance, qui est un point essentiel du problème de l'évolution de schéma.

III.3.1. Classes, types et opérations

Nous donnons ici un modèle de description d'une version de schéma et de son instance. Notre but est d'en avoir une description formelle, pour mieux comprendre et mieux expliquer les aspects liés au problème de l'évolution de schéma. Dans ce modèle, nous considérons un ensemble

réduit des concepts de l'approche à objet mais qui suffisent cependant à notre étude. Ces concepts sont l'*identité d'objet*, la relation *d'héritage* et la relation de *référence* [BDK92, Cat93, Kos91].

Nous supposons l'existence des ensembles finis et disjoints suivants :

- l'ensemble D des valeurs atomiques. D est l'union des domaines atomiques : integer, string, boolean, etc.,
- L'ensemble A des noms d'attributs,
- L'ensemble OP des noms d'opérations,
- L'ensemble O des identificateurs d'objets,
- L'ensemble C des noms de classes,
- L'ensemble N des entiers naturels,
- L'ensemble T des types,
- L'ensemble VAL des valeurs des versions d'objets.

Définition 1 (valeur). La valeur d'une version d'objet peut être :

- **de base** : si $v \in D \cup O$, alors $v \in VAL$. *nil* $\in VAL$ (*nil* est un symbole spécial).
- **une valeur n-uplet** : si $v_i \in VAL$ (v_i est une valeur) et $a_i \in A$ (a_i est un attribut) où ($i \in [1..m]$ et $a_i \neq a_j$ pour $i \neq j$), alors $[a_1 : v_1, a_2 : v_2, ..., a_m : v_m] \in VAL$.
- **une valeur ensemble** : si $v_i \in VAL$ (v_i est une valeur) où ($i \in [1..m]$), alors $\{v_1, v_2, ..., v_m\} \in VAL$.

Définition 2 (type). Un type peut être :

- **atomique** : integer, string, boolean, etc.
- **un identificateur de classe** : si $t \in C \times N$, alors $t \in T$.
- **un type n-uplet** : si $t_i \in T$ (t_i est un type) et $a_i \in A$ (a_i est un attribut) où ($i \in [1..m]$ et $a_i \neq a_j$), alors $[a_1 : t_1, ..., a_n : t_n] \in T$.
- **un type ensemble** : si $t \in T$, alors $\{t\} \in T$.

L'expression $\partial(c, n)$ détermine le type de la classe (c, n). L'expression dom(t) détermine le domaine (l'ensemble de valeurs) du type t. Il est important de remarquer qu'une classe peut avoir différents types, chacun dans une version différente du schéma.

Dans une version de schéma, les classes sont structurées en une hiérarchie de classes selon une relation d'ordre partiel $<_h$, appelée relation d'héritage [BDK92]. $<_t$ est une relation d'ordre partiel sur T, définie par :

- $<_t$ est pré-définie sur les éléments de D,

- $\forall (c, n), (c', n') \in C \times N, (c, n) <_h (c', n') \Rightarrow \partial(c, n) <_t \partial(c', n')$.
- $\forall t_i, t'_i \in T, a_i \in A, i \in [1..n+p]$ et $p \geq 0$,
  $t_i <_t t'_i \Rightarrow [a_1 : t_1, \ldots, a_n : t_n, a_{n+1} : t_{n+1}, \ldots, a_{n+k} : t_{n+k}] <_t [a_1 : t'_1, \ldots, a_n : t'_n]$
- $\forall t, t \in T, t <_t t' \Rightarrow \{t\} <_t \{t'\}$.

Dans un modèle à objets, les opérations (méthodes) sont définies dans les classes. Elles déterminent le comportement des objets des classes auxquelles elles sont associées. A une opération est associée une signature et un code. La partie visible d'une opération est sa signature. La signature d'une opération est une expression de la forme :

$op : (c, n) \times t_1 \times t_2 \times \ldots \times t_n \rightarrow t$

tel que, $op \in OP$ (nom de l'opération) et $(c, n) \in C \times N$ est la classe où l'opération est définie et $t_1, \ldots, t_n, t \in T$.

Le code (par exemple, un programme en C++) d'une opération est sa réalisation. Une opération définie dans une classe peut être redéfinie dans une sous-classe de cette classe, en lui associant un autre code ou/et une autre *signature compatible* avec la première. Nous considérons qu'une opération consiste en un couple (op, s) tel que, op est le nom de l'opération et s est sa signature.

Définition 3 (compatibilité de signatures). Soient s et s' deux signatures, telles que :
$s = op : (c, n) \times t_1 \times t_2 \times \ldots \times t_n \rightarrow t$ et $s' = op : (c', n) \times t'_1 \times t'_2 \times \ldots \times t'_n \rightarrow t'$
s' est compatible avec s si et seulement si : $t' <_t t$ et $\forall i \in [1..n], t'_i <_t t_i$.

III.3.2 Schéma et instance

L'ensemble des classes et des types d'une version du schéma d'une base de données est structuré en un graphe dont les noeuds sont les classes et les arcs sont les liens de référence entre les classes.

Un lien de référence entre une classe c et un type t, est caractérisé par un nom d'attribut att, indiquant que le domaine de att dans c est t. A partir des liens de référence entre classes et types sont déduits les liens entre versions d'objets et valeurs.

Un lien de référence entre classe et type est simple ou multivalué selon le constructeur utilisé pour définir le domaine de l'attribut :

• **Lien de référence simple** : un lien de référence simple d'une classe (c, n) à un type t par l'intermédiaire d'un attribut att est décrit par l'arc orienté $(c, n) \xrightarrow{att} t$.

Si une version d'un objet (o, n) définie dans la classe (c, n) a une valeur n-uplet [..., att : o', ...] telle que, o' ∈ dom(t) alors le lien de référence de (o, n) à o' est décrit par l'arc orienté $(o, n) \xrightarrow{att} o'$.

Exemple :

Le lien de référence entre la classe (Département, n) et la classe (Employé, n) par l'intermédiaire de l'attribut LeDirecteur est décrit par :

$$(\text{Département}, n) \xrightarrow{\text{LeDirecteur}} (\text{Employé}, n)$$

Le lien de référence $(d1, n) \xrightarrow{\text{LeDirecteur}} (e1, n)$ signifie que la version d'objet (d1, n) de la classe (Département, n) est associée à la version d'objet (e_1, n) de la classe (Employé, n) par l'intermédiaire de l'attribut LeDirecteur.

• **Lien de référence multivalué** : un lien de référence multivalué d'une classe (c, n) à un type t par l'intermédiaire d'un attribut att est décrit par l'arc orienté $(c, n) \xrightarrow{att\{\}} t$.

Si une version d'objet de (c, n) a une valeur n-uplet [..., att : $\{o'_1, ..., o'_m\}$,...] telle que, o'1, ..., o'm ∈ dom(t), alors il existe un lien de référence de o à chaque o'_i (i ∈ [i..n]), décrit par l'arc orienté $(o, n) \xrightarrow{att} o'_i$.

Exemple :

Le lien de référence entre la classe (Enseignant, n) et (Cours, n) par l'intermédiaire de l'attribut LesCours est décrit par $(\text{Enseignant}, n) \xrightarrow{\text{LesCours}\{\}} (\text{Cours}, n)$.

Nous associons à un schéma un graphe orienté dont les noeuds sont les classes de ce schéma et les arcs les liens de références entre les classes.

• **Graphe d'une version du schéma :** à une version du schéma S, est associé un graphe orienté S (CS, ES), tel que :

- $CS \subseteq C \times N$ est l'ensemble des noeuds associés aux classes de S,
- $ES \subseteq (C \times N) \times A \times T$ est l'ensemble des arcs orientés associés aux liens de S.

Ainsi, une version d'un schéma (S, n) est associée à un quadruplet, (CS_n, ES_n, OP_n, $<_h$), tel que : (CS_n, ES_n) est le graphe associé à (S, n), OP_n est l'ensemble des opérations des classes de (S, n) et $<_h$ est la relation d'héritage sur CS_n. Le graphe associé à une version (S, n) est la restriction du graphe associé à S sur n. C'est-à-dire, CS_n est l'ensemble des noeuds associées aux classes de (S, n) et ES_n est l'ensemble des arcs orientés associés aux liens de (S, n).

Exemple. La figure III.7 montre le graphe associé à une partie du schéma qui décrit la base de données de la gestion de l'université. Les noeuds correspondent aux classes Objet (la racine de toutes les hiérarchies de classes), Département et Employé. Les arcs correspondent aux liens d'héritage entre la classe Objet et les classes Département, Employé d'une part et le lien de référence entre la classe Département et Employé par l'intermédiaire de l'attribut LeDirecteur, d'autre part.

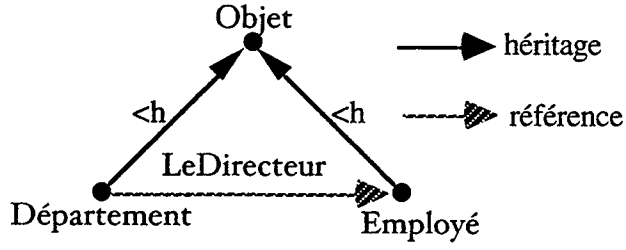


FIG. III.7 : graphe associé à une version du schéma

Nous associons à un ensemble d'objets, un graphe orienté dont les noeuds sont les versions des objets de cet ensemble et les arcs sont les liens de référence entre les versions. Les identificateurs d'objets dénotent les entités (objets) du monde réel et le graphe d'objets décrit la manière dont les versions de ces objets sont liées. L'expression `LinksOf`(o, n) détermine l'ensemble des liens de référence de la version n de l'objet o.

- **Graphe d'objets :** à un ensemble d'objets I est associé un graphe I (OI, IE), tel que :
  - $OI \subseteq (O \times N) \cup D$ est l'ensemble des noeuds associés aux objets de I,
  - $EI \subseteq (O \times N) \times A \times VAL$ est l'ensemble des arcs associés aux liens de références de I.

Ainsi, une instance I d'un schéma S est associée à un triplet (OI, EI, `Classes`) tel que : (OI, EI) est le graphe associé à I et `Classes` est une application (une partition sur C×N) définie par :

`Classes` : $OI \rightarrow 2^{C \times N}$

{`Classes`(o) est l'ensemble des classes sous lesquelles l'objet o possède une version. Ces classes doivent être définies dans des versions différentes du schéma. C'est-à-dire :
$\forall (c, n), (c, m) \in$ `Classes`(o), $n \neq m$}.

Exemple, la figure III.8 montre un graphe d'objets dont le schéma est décrit dans la figure III.7. Les noeuds associés aux versions d'objets de la même classe sont entourés par un ovale étiqueté par le nom de la classe à laquelle ils appartiennent.

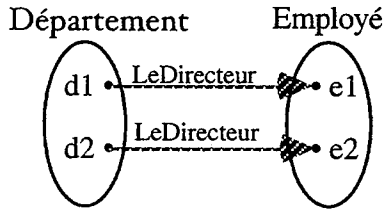


FIG. III.8 : graphe d'objets

III.3.3. Cohérence structurelle de la base de données

Une base de données (S, I) est structurellement cohérente si :
1) chaque version de S satisfait les invariants de schéma.
2) la valeur de chaque version d'objet de I est conforme à la structure de la classe dont elle est instance.

Nous détaillons ces deux points ici.

III.3.3.1. Cohérence du schéma

Les invariants que nous considérons sont les suivants [Ben92, BK87, Del92, Zic92] :

• **Hiérarchie de classes** : la hiérarchie de classes est un graphe connexe, orienté et sans cycle. La racine de la hiérarchie est la classe Objet.

• **Noms distincts** : les noms des classes et des propriétés (attributs ou opérations) d'une classe sont distincts. Les noms des classes sont distincts deux à deux dans une version du schéma. Les noms des propriétés sont distincts deux à deux dans une classe.

• **Héritage complet** : une classe hérite de toutes les propriétés de ses super-classes. Une propriété héritée peut être redéfinie dans la classe. Nous ne considérons pas les conflits liés à l'héritage multiple.

• **Compatibilité des types d'attributs :** si dans une classe, soit c, un attribut att redéfinit un autre attribut att' d'une super-classe de c alors le type de att doit être un sous-type de att'.

• **Compatibilité des signatures des opérations (covariance) :** si dans une classe, soit c, une opération op redéfinit une autre opération op' alors la signature de op doit être compatible avec celle de op'.

Dans la suite, la notation, $\psi(S, n)$, signifie que la version (S, n) du schéma est cohérente (c'est-à-dire, elle satisfait les invariants du schéma). La notation, $\psi+(S)$, signifie que le schéma S est cohérent (c'est-à-dire, $\forall i \in [1..n]$, $\psi(S,i)$).

Exemple. Dans le but de clarifier la notion de cohérence d'une version de schéma (ou d'un schéma), nous donnons comme exemple le schéma S (Cf. Figure III.9). (S, i) et (S, i+1) sont deux versions de S. La version (S, i) satisfait tous les invariants du schéma. La version (S, i+1) viole l'invariant de compatibilité de types car le type de l'attribut LeChef dans la classe Enseignant (c'est-à-dire, Employé) n'est pas un sous-type du type de l'attribut LeChef dans la classe Employé (c'est-à-dire, Enseignant).

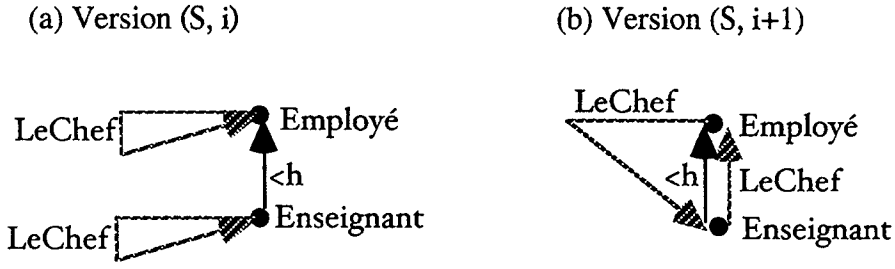


FIG. III.9 : versions de schéma (a) cohérente, (b) incohérente.

III.3.3.2 Cohérence d'une instance

Informellement, une instance I, d'une base de données (S, I) est dite cohérente vis-à-vis de son schéma S si pour chaque objet o de I, sa version sous toute classe (c, n) de S est conforme avec la structure de (c, n). C'est-à-dire :

• Le type de la version de o sous (c, n) est un sous-type du type de (c, n). Dans la suite, nous allons définir la cohérence d'une instance vis-à-vis de son schéma en utilisant le modèle formel introduit précédemment.

Dans un premier temps, nous définissons la conformité de la version d'un objet vis-à-vis de la structure d'une classe. Puis, nous définissons la cohérence d'une instance d'une base de données vis-à-vis de son schéma.

Cohérence d'une version d'objet. Une version d'objet (o, n) est conforme avec la structure de (c, n), si et seulement si :

1) $\exists a \in A,\ ((c,n) \xrightarrow{a} t) \in SE_n \Leftrightarrow \exists((o,n) \xrightarrow{a} o') \in OI,\ o' \in dom(t') \wedge t' <_t t$

Cela signifie que s'il existe dans le schéma, un lien de référence simple entre une classe (c, n), et un type t, alors chaque version d'objet de (c, n) doit avoir exactement un lien vers une valeur du domaine de t.

2) $\exists a \in A,\ ((c,n) \xrightarrow{a\{\}} t) \in SE_n \Leftrightarrow card\{(o \xrightarrow{a} o') \in OI / o' \in dom(t') \wedge t' <_t t\} \geq 1$

Cela signifie que, s'il existe dans le schéma un lien de référence multivalué entre une classe (c, n), et un type t, alors chaque version d'objet de (c, n) doit avoir au moins un lien vers une valeur du domaine de t.

Dans la suite, la notation $\delta(o, c, n)$, signifie que la version de l'objet o sous la classe (c, n) est conforme avec la structure de (c, n).

Cohérence d'une instance. Soit (S, I) une base de données, posons S (CS, ES) et I (OI, EI) leurs graphes associés. I est une instance cohérente vis-à-vis de S, si et seulement si :

1) $\exists(o, n) \in OI \times N \Rightarrow \exists c \in C,\ ((c, n) \in CS_n) \wedge ((c, n) \in Classes(o))$
2) $\forall o \in OI,\ (c, n) \in Classes(o) \Rightarrow \delta(o, c, n)$

Cela signifie qu'une instance I est cohérente vis-à-vis de son schéma S si chacun de ses objets a une version conforme avec chaque classe dont il est instance. Dans la suite, la notation $\delta(I, S)$, signifie que l'instance I est cohérente vis-à-vis du schéma S.

Exemple. Dans le but d'illustrer la notion de cohérence d'une instance vis-à-vis de son schéma, nous donnons comme exemple le schéma S qui a deux versions (S, 0) (Cf. Figure III.10.a) et (S, 1) (Cf. Figure III.10.b), et une instance I (Cf. Figure III.11) :

OI ={d, e0, e1},

`Classes`(d)={(Département, 0), (Département, 1)},

`Classes`(e0)=`Classes`(e1)={(Employé, 0), (Employé, 1)}.

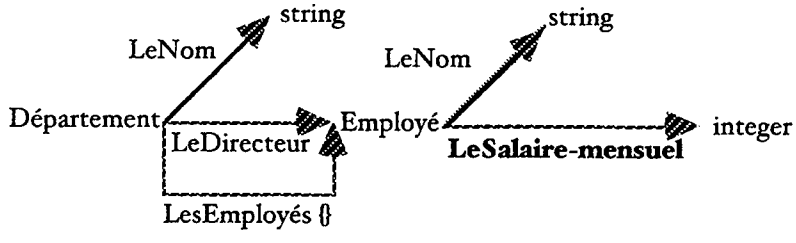
L'instance I est cohérente vis-à-vis de S car :

1) chaque version de l'ensemble {(d, 0), (d, 1), (e0, 0), (e0, 1), (e1, 0), (e1, 1)} est associée à une classe du schéma. (d, 0) est associée à (Département, 0), (d, 1) est associée à (Département, 1), (e0, 0) et (e1, 0) sont associées à (Employé, 0) et (e0, 1) et (e1, 1) sont associées à (Employé, 1).
2) la valeur de chaque version d'un objet de OI est conforme avec la structure de sa classe.

Nous montrons ici à titre d'exemple la conformité de la valeur (d,0) avec la structure de (Département, 0). C'est-à-dire, nous montrons que δ(d, Département, 0) en supposant δ(e0, Employé, 0) et δ(e1, Département, 0) :

• **Lien de référence simple :** au lien (Département, 0) $\xrightarrow{\text{LeDirecteur}}$ (Employé, 0) de ES_0 correspond le lien (d, 0) $\xrightarrow{\text{LeDirecteur}}$ (e1 ,0) de EI.
Au lien (Département, 0) $\xrightarrow{\text{LeNom}}$ string correspond le lien (d, 0) $\xrightarrow{\text{LeNom}}$ "maths" de EI tel que, "maths" ∈ dom(string).
• **Lien de référence multivalué** : au lien (Département, 0) $\xrightarrow{\text{LesEmployés}\{\}}$ (Employé, 0) de ES_0 correspond les deux liens (d, 0) $\xrightarrow{\text{LesEmployés}}$ (e0, 0) et (d, 0) $\xrightarrow{\text{LesEmployés}}$ (e1,0) de EI.

(a) version 0 du schéma S



(b) version 1 du schéma S

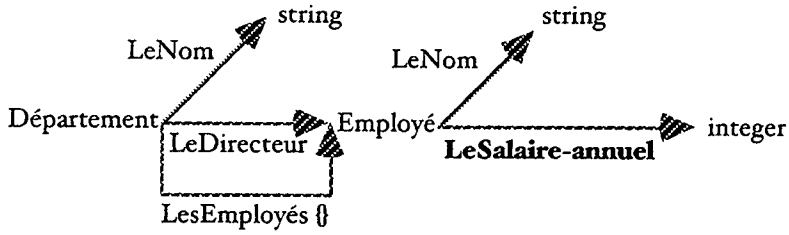


FIG. III.10 : le schéma S

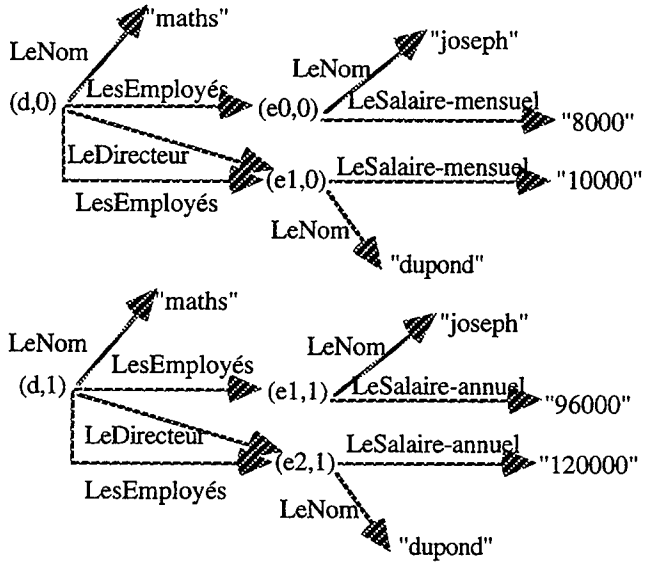


FIG. III.11 : l'instance I

III.4. Le problème de l'évolution de schéma

Nous considérons une base de données et supposons que son schéma est cohérent. Étant donnée une évolution de schéma, c'est-à-dire une séquence de primitives de changement du schéma, les questions liées à cette évolution sont :

- le schéma résultant est-il cohérent ?
- si oui, l'instance associée est-elle cohérente ?

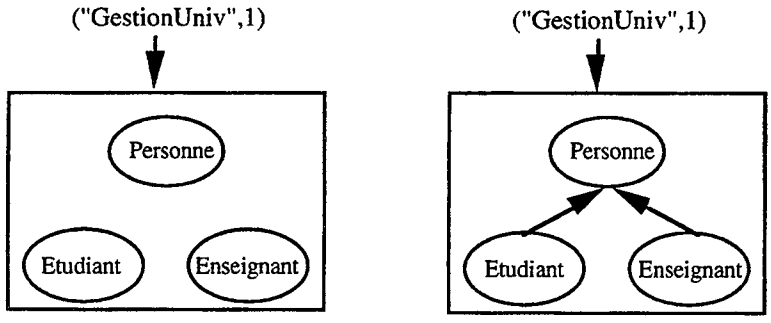
De nombreuses études ont abordé le problème lié à la cohérence de structure [Ben92, BK87, Bou95, Del92, DZ91, LH90, NR89a, PS87, SGD93, Tre91, Zic92]. La multiplicité des solutions proposées réside dans la diversité des modèles de données utilisés et de la nature des applications. Nous revenons sur ce sujet au travers d'un exemple (section III.4.1).

Par contre, le problème lié à la cohérence d'une instance vis-à-vis de son schéma est encore ouvert, malgré les efforts faits dans le domaine [BF95, Cla92, FMZ95a, ED95, HY95, Ler93, Mon93, NR89b, TK91, ST94]. Nous détaillons cet aspect dans la section III.4.2 en nous reposant sur le formalisme introduit jusqu'ici, puis nous proposons un modèle de solution à ce problème dans la section III.4.3.

III.4.1. Le problème de la cohérence de structure d'un schéma

Considérons la version 1 du schéma "GestionUniv" (Cf. Figure III.6). L'évolution du schéma consiste à supprimer la classe Employé. Après cette évolution nous obtenons une version du schéma qui contient des noeuds isolés (Cf. Figure III.12.a). Le schéma est incohérent car l'invariant de "la hiérarchie de classes" est violé. Différentes solutions sont proposées pour résoudre ce problème. Parmi ces solutions nous citons :

• la connexion des sous-classes (c'est-à-dire, Enseignant et Etudiant) de la classe supprimée (c'est-à-dire, Employé) à sa super-classe (c'est-à-dire, Personne) (Cf. Figure III.13.b).



(a) : version incohérente (b) : version cohérente

FIG. III.12 : deux versions du schéma "GestionUniv"

En général des règles sont proposées pour préserver les invariants du schéma. Par exemple dans *Orion* [BK87], la règle suivante est proposée pour préserver l'invariant de "la hiérarchie de classes" à la suite de la suppression d'un lien d'héritage :

• Si une classe c_1 est supprimée de la liste des super-classes de la classe c_2 et si c_1 était la seule super-classe de c_2 (Cf. Figure III.13.a), alors c_2 est rendue une sous-classe immédiate de chacune des super-classes de c_1 (Cf. Figure III.13.b). L'ordre des nouvelles super-classes de c_2 est le même que celui des super-classes de c_1. Dans le cas où la classe Objet est la seule super-classe de c_2, la suppression du lien qui relie Objet à c_2 est rejetée.

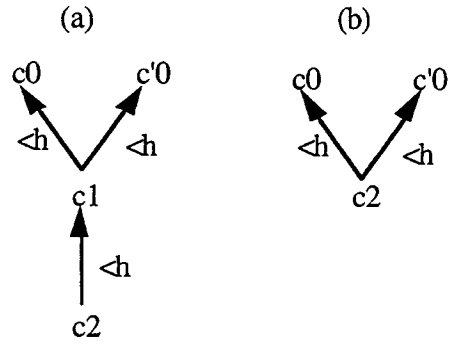


FIG. III.13 : suppression d'un lien d'héritage

III.4.2. Le problème de la cohérence d'une instance vis-à-vis de son schéma

Dans cette partie, nous supposons qu'une évolution de schéma produit un schéma cohérent. Étant donnée une base de données (S, I), telle que $\delta(I, S) \wedge \psi+(S)$ et une évolution de schéma U sur S, telle que $S'=U(S) \wedge \psi+(S')$. La question que nous abordons est la suivante : I est-elle cohérente vis-à-vis de S' (c'est-à-dire, $\delta(I, S')$) ?

Une incohérence de I vis-à-vis de S', peut être due à l'existence de certaines versions d'objets de I qui ne sont pas conformes avec la structure de certaines de leurs classes dans S', ou à l'existence de certaines versions d'objets qui ne sont associées à aucune classe de S'. Dans le premier cas, il s'agit de références mal typées et dans le second cas, il s'agit de références folles :

1. **Références mal typées** : $(\exists o \in OI, (c, n) \in \texttt{Classes}(o)) \wedge (\neg\delta(o, n, c))$.
2. **Références folles** : $(\exists(o, n) \in OI \times N) \wedge \neg(\exists(c, n) \in CS, (c, n) \in \texttt{Classes}(o))$.

Exemples. Pour illustrer l'incohérence de l'instance d'une base de données à la suite de l'évolution de son schéma, nous considérons certaines évolutions du schéma S de la figure III.10. Nous supposons que ces opérations sont traduites par la modification de la version courante du schéma (c'est-à-dire, (S, 1)).

1. `Add_Attribute`(Employé, LesHeures, integer)

• Cette opération se traduit par l'ajout du lien de référence $((\text{Employé}, 1) \xrightarrow{\text{LesHeures}} \text{integer})$ dans l'ensemble des liens de référence associés à la version (S, 1) (c'est-à-dire, ES_1). Soit e un objet de I tel que, (Employé, 1) $\in$ `Classes`(e). La base est incohérente car il n'existe pas de lien

((e, 1) $\xrightarrow{\text{LesHeures}}$?) dans l'ensemble des liens de référence associés à (e, 1) (c'est-à-dire, `LinksOf`(e, 1)) .

2. `Del_Attribute`(Département, LeDirecteur)

• Cette opération se traduit par la suppression de ((Département,1) $\xrightarrow{\text{LeDirecteur}}$ (Employé,1)) dans l'ensemble ES_1. Soit d un objet de I tel que, (Département,1) ∈ Classes(d) et (d,1) $\xrightarrow{\text{LeDirecteur}}$ (e,1) est un lien de `LinksOf`(d, 1). La base de données est incohérente car il n'existe pas de lien ((Département,1) $\xrightarrow{\text{LeDirecteur}}$?) dans ES_1.

3. `Update_Attribute`(Département, LeDirecteur, Enseignant)

• Supposons que la classe (Enseignant, 1) est une sous-classe de (Employé, 1). Cette opération se traduit par le remplacement du lien ((Département,1) $\xrightarrow{\text{LeDirecteur}}$ (Employé,1)) par le lien ((Département,1) $\xrightarrow{\text{LeDirecteur}}$ (Enseignant,1)) dans ES_1. Soit d un objet de I, tel que (Département, 1) ∈ `Classes`(d) et ((d,1) $\xrightarrow{\text{LeDirecteur}}$ (e,1)) est un lien de `LinksOf`(d, 1) tel que, (Employé, 1) ∈ `Classes`(e). La base de données est incohérente car (Employé, 1) n'est pas une sous-classe de (Enseignant, 1).

4. `Del_Class`((Enseignant, 1))

• Cette opération se traduit par la suppression des propriétés de (Enseignant, 1) dans la version (S, 1). Soit e un objet de I tel que, (Enseignant, 1) ∈ `Classes`(e) et ((d,1) $\xrightarrow{\text{LeDirecteur}}$ (e,1)) est un lien de `LinksOf`(d, 1). La base de données est incohérente car (Enseignant, 1) n'est pas une classe de l'ensemble des classes de (S, 1) (c'est-à-dire, CS_1).

Les instances d'un schéma doivent être cohérentes vis-à-vis du schéma après son évolution. Ainsi, un énoncé possible du problème est le suivant :

• Étant donnée une base de données (S, I) telle que, $\delta(I, S) \wedge \psi+(S)$ et une opération d'évolution de schéma U sur S, telle que $S'=U(S) \wedge \psi+(S')$. Peut-on trouver I', déduite de I, telle que, $\delta(I', S')$?

Déterminer I' satisfaisant cette condition revient à déterminer une séquence de mises à jour sur l'instance I de la base de données. Classiquement une telle séquence est appelée *réorganisation*

[3.2] [ST94]. Dans la section suivante, nous proposons un langage d'expression d'une telle réorganisation.

III.4.3 Réorganisation d'une base de données

Une réorganisation $\Re$ est une séquence de mises à jour qui modifient un état cohérent de l'instance de la base de données en un autre état cohérent de la base.

Les primitives de mises à jour que nous proposons s'appliquent à un état de l'instance de la base de données. Prises indépendamment les unes des autres, elles peuvent dans certains cas, générer un nouvel état incohérent. Ainsi, une réorganisation peut être vue comme une transaction.

Une primitive de mise à jour a un paramètre d'entrée et un paramètre de sortie. Le paramètre d'entrée est l'état de la base de données avant la mise à jour et le paramètre de sortie est l'état de la base de données après la mise à jour. L'état d'une base de données soit bd avant et après la mise à jour est respectivement représenté par bd et bd'. Nous nous contentons ici, de donner la description des éléments différents de l'état de la base avant et après la mise à jour.

III.4.3.1. Primitives de base

m1. Ajouter un nouvel identificateur d'objet : `InsObjOid`

Cette opération ajoute un nouvel oid à l'ensemble des objets associés à une classe. Elle est définie comme suit :

`InsObjOid` : $C \times N \rightarrow (C \times N) \times O$

{Soit ((c, n)',o)=`InsObjOid`(c, n) tel que :

$(o \notin \texttt{Objects}(c, n)) \wedge (\texttt{Objects}(c, n)' = \texttt{Objects}(c, n) \cup \{o\})$

}

m2. Ajouter un identificateur d'une nouvelle version d'objet : `InsVerOid`

Cette opération ajoute un nouvel v-oid à l'extension d'une classe. Elle est définie comme suit :

`InsVerOid` : $(C \times N) \times O \rightarrow C \times N$

{Soit (c, n)'=`InsVerOid`((c, n), o) tel que :

[3.2] En théorie, la réorganisation a lieu à schéma constant mais ici c'est à schéma non constant.

$((o, n) \notin \mathtt{Ext}(c, n)) \wedge (\mathtt{Ext}(c, n)' = \mathtt{Ext}(c, n) \cup \{(o, n)\})$

}

m3. Supprimer un identificateur d'une version d'objet : `DelVerOid`

Cette opération supprime un v-oid de l'extension d'une classe. Elle est définie comme suit :

$\mathtt{DelVerOid} : (C \times N) \times O \rightarrow C \times N$

{Soit $(c, n)' = \mathtt{DelVerOid}((c, n), o)$ tel que :

$((o, n) \in \mathtt{Ext}(c, n)) \wedge (\mathtt{Ext}(c, n)' = \mathtt{Ext}(c, n) - \{(o, n)\})$

}

m4. Supprimer un identificateur d'objet : `DelObjOid`

Cette opération supprime un oid de l'ensemble des objets associés à une classe. Elle est définie comme suit :

$\mathtt{DelObjOid} : (C \times N) \times O \rightarrow C \times N$

{Soit $(c, n)' = \mathtt{DelObjOid}((c, n), o)$ tel que :

$(o \in \mathtt{Objects}(c, n)) \wedge (\mathtt{Objects}(c, n)' = \mathtt{Objects}(c, n) - \{o\})$

}

m5. Ajouter un nouveau lien de référence : `InsRefLink`

Cette opération ajoute un nouveau lien de référence à l'ensemble des liens de références d'une version d'objet. Dans la suite, L dénote l'ensemble des liens de référence des versions d'objets. `InsRefLink` est définie comme suit :

$\mathtt{InsRefLink} : P(L) \times L \rightarrow P(L)$

{Soit $\mathtt{LinksOf}(o, n)' = \mathtt{InsRefLink}(\mathtt{LinksOf}(o, n), l)$ tel que :

$((l \notin \mathtt{LinksOf}(o, n)) \wedge (\mathtt{LinksOf}(o, n)' = \mathtt{LinksOf}(o, n) \cup \{l\})) \wedge$

$((l = (o, n) \xrightarrow{a} av) \wedge (\exists (c, n) \xrightarrow{a} t \in ES_n, (c, n) \in \mathtt{Cl}(o) \wedge av \in \mathtt{dom}(t)))$

$\vee((l = (o, n) \xrightarrow{a} av) \wedge (\exists (c, n) \xrightarrow{a\{\}} t \in ES_n, (c, n) \in \mathtt{Cl}(o) \wedge av \in \mathtt{dom}(t)))$

}

m6. Supprimer un lien de référence : `DelRefLink`

Cette opération supprime un lien de référence de l'ensemble des liens de références d'une version d'objet. Elle est définie par :

$\mathtt{DelRefLink} : P(L) \times L \rightarrow P(L)$

{Soit $\mathtt{LinksOf}(o, n)' = \mathtt{DelRefLink}(\mathtt{LinksOf}(o, n), l)$ tel que :

$(l \in \mathtt{LinksOf}(o, n)) \wedge (\mathtt{LinksOf}(o, n)' = \mathtt{LinksOf}(o, n) - \{l\})$

}

m7. Modifier un lien de référence : `ModRefLink`

La modification d'un lien de référence consiste à remplacer la valeur associée à l'attribut par une autre valeur (spécifiée dans l'opération). Elle est définie par :

`ModRefLink` : $L \times VAL \rightarrow L$

{Soit l'= `ModRefLink`(l, nv) tel que :

$((l = (o, n) \xrightarrow{a} av) \wedge (l' = (o, n) \xrightarrow{a} nv)) \vee ((l = (o, n) \xrightarrow{a} av) \wedge (l' = (o, n) \xrightarrow{a} nv))$

$\wedge$ (`LinksOf`(o, n)' = `LinksOf`(o, n) − {l} + {l'})

}

m8. Migrer un identificateur d'objet : `MigObjOid`

Cette opération provoque la migration de l'identificateur d'objet d'une classe vers une autre. Elle est définie par :

`MigObjOid` : $(C \times N) \times (C \times N) \times O \rightarrow (C \times N) \times (C \times N)$

{Soit ((c1,n)',(c2,m)')= `MigObjOid`((c1,n),(c2,m),o) tel que :

(o $\in$ `Objects`(c1, n) $\wedge$ o $\notin$ `Objects`(c2, m)) $\wedge$

`Objects`(c1, n)' = `Objects`(c1, n) - {o} $\wedge$ `Objects`(c2, m)' = `Objects`(c2, m) $\cup$ {o}

}

III.4.3.2. Composition des primitives

La syntaxe que nous adoptons pour décrire la réorganisation de l'instance d'une base de données, soit $\Re$, est la suivante :

$\Re$= [**let** $x_1;x_2;...;x_k$ **in**] $<t_1;...;t_n>$

L'expression entre crochets ([let...in]) est optionnelle; elle permet d'introduire des constructions intermédiaires. x_i est une expression qui a la forme suivante : x=exp telle que, x est une variable et exp est une expression, t_i est un terme du langage Γ, récursivement défini, comme suit [BDP93, BDS95] :

- **un terme de base** : si m $\in$ {m1,..., m8}, alors m $\in$ Γ.
- **un terme construit** :

- **séquence** : $\forall t_1, t_2 \in \Gamma$, $\{t_1;t_2\} \in \Gamma$
- **alternative** : $\forall t_1, t_2 \in \Gamma$, if (eb) then t_1 else $t_2 \in \Gamma$, où eb est une expression booléenne.

- **itération** : $\forall t \in \Gamma$, for (el $\in$ ens) $t \in \Gamma$, où el $\in O \cup L$(c'est-à-dire, un oid ou un lien de référence) et ens $\in 2^O \cup 2^L$ (c'est-à-dire, un ensemble d'oids ou un ensemble de liens).

Exemples. Nous reprenons le schéma "GestionUniv" (Cf. Figure III.10). Nous donnons quelques exemples d'évolutions auxquelles nous associons une réorganisation possible, qui transforme l'instance de la base de données en une instance cohérente.

1. Add_Attribute(Employé, LesHeures, integer)

• Cette évolution se traduit par l'ajout d'une valeur par défaut (ici 0) associée à l'attribut LesHeures dans toutes les versions d'objets de la classe (Employé, 1).

$\Re$=<for(o$\in$ Objects(Employé,1))
InsRefLink(LinksOf(o,1), ((o,1)$\xrightarrow{\text{LesHeures}}$0))>

2. Update_Attribute(Département, LeDirecteur, Enseignant)

• Cette évolution se traduit par la modification de la valeur associée à l'attribut LeDirecteur dans toutes les versions d'objets de la classe (Département, 1). La nouvelle valeur de l'attribut LeDirecteur dans ces versions est par défaut (ici nil).

$\Re$ = let x={(o,1)$\xrightarrow{\text{LeDirecteur}}$o' $\in$ EI / o $\in$ Objects((Département,1)}
in <for (l $\in$ x) ModRefLink(l, nil)>

3. Del_Attribute(Département, LeDirecteur)

• Cette évolution se traduit par la suppression de la valeur associée à l'attribut LeDirecteur dans toutes les versions d'objets de la classe (Département, 1).

$\Re$ =let x=<{(o,1)$\xrightarrow{\text{LeDirecteur}}$o' $\in$ EI / o $\in$ Objects(Département,1)}
in <for (l $\in$ x)DelRefLink(EI, l)>

4. Del_Class((Enseignant, 1))

• Nous donnons deux réorganisations possibles. En utilisant la réorganisation $\Re$, cette évolution se traduit par la suppression de toutes les versions d'objets de la classe (Enseignant, 1) et la mise à jour des versions d'objets qui référencent ces versions.

```
ℜ = let ℜ1=<for (o∈Objects(Enseignant,1))
        {for (l ∈ LinksOf(o,1)) DelRefLink(LinksOf(o, l), l);
        DelVersOid((Enseignant,1), o);
        DelOid((Enseignant,1), o);
        }
        ℜ2= let x={(o,m) --a--> (o',l) ∈ EI / o' ∈ Objects(Enseignant,1)}
             in <for (l ∈ x)ModRefLink(l,nil)>
in <ℜ1;ℜ2>
```

En utilisant la réorganisation $\Re'$, cette évolution se traduit par la migration des objets de la classe (Enseignant, 1) vers la classe (Employé, 1).

```
ℜ'=<for (o∈Objets (Enseignant, 1))
{MigOid((Enseignant, 1), (Employé, 1), o);
InsVerOid((Employé, 1), o); }>
```

III.5. Conclusion et perspectives

Nous avons fourni un modèle de versions de bases de données à objets. Un schéma ou un objet peut avoir plusieurs versions. Les primitives de manipulation de versions de schéma et d'objets sont définies. Ces primitives seront utilisées par le mécanisme d'évolution de schéma (Cf. Chapitres IV et V). Nous avons étudié le problème de cohérence structurelle d'une base de données en utilisant le modèle formel de description d'une base de données et avons proposé une formalisation du problème de l'évolution de schéma.

Les perspectives du travail présenté dans ce chapitre sont :

• **Cohérence de comportement** : le problème de cohérence du comportement de schéma n'est pas discuté. Nous envisageons l'extension du modèle formel pour prendre en compte les concepts liés aux opérations comme les dépendances entre les opérations et les dépendances entre les

opérations et les attributs. Ainsi, il sera possible de dégager une caractérisation de la cohérence du comportement de schéma.

• **Complétude du langage d'expression de la réorganisation** : nous avons proposé huit primitives de base pour la mise à jour de l'instance d'une base de données et avons adopté un langage pour exprimer sa réorganisation. Nous envisageons d'étudier la complétude du langage d'expression de la réorganisation. Autrement dit, étant donnée une base de données et une évolution du schéma ; nous devrions pouvoir démontrer que la réorganisation de la base en une instance quelconque, cohérente vis-à-vis du schéma résultant pourrait être exprimée dans ce langage. Cette étude permettra d'étudier la puissance des solutions proposées.

Chapitre IV

UN COMPROMIS : MODIFICATION ET VERSIONNEMENT DU SCHÉMA

Ce chapitre présente une nouvelle approche pour gérer l'évolution de schéma. Le processus du changement du schéma combine la *modification* et le *versionnement* du schéma. La technique d'adaptation des instances combine la *conversion*, l'*émulation* et le *versionnement* des objets. Nous offrons à l'utilisateur la possibilité de supprimer les informations de la base de données qui ne sont plus pertinentes (ou jugées non pertinentes) pour les applications. Notre approche effectue un compromis entre les fonctionnalités des approches existantes pour une meilleure gestion de l'évolution du schéma. Elle prend en compte la pérennité des informations, la fiabilité des programmes et les performances du système.

IV.1. Introduction

Nous développons dans ce chapitre notre approche pour la gestion de l'évolution de schéma. Cette approche combine les techniques existantes. En particulier, notre but est de trouver un compromis entre :

1) la modification et le versionnement du schéma quand le schéma est changé,

2) la conversion, l'émulation et le versionnement pour adapter les instances après le changement de leur schéma.

Nous identifions les opérations d'évolution de schéma pour lesquelles il est préférable de dériver une nouvelle version du schéma. La technique utilisée pour adapter les instances au schéma après l'évolution, est basée sur la caractérisation de l'importance de l'existence en tant que telle d'une version d'objet. Lorsqu'un accès à un objet est demandé sous une classe dont l'extension ne contient pas de version de cet objet, une nouvelle version de l'objet est générée seulement si la classe est fréquemment utilisée. Autrement, la technique d'émulation est utilisée. Les seules versions d'objets stockées sont celles qui sont fréquemment accédées par des programmes. Notre solution effectue un compromis entre les fonctionnalités proposées dans les autres approches de manière à remédier à leurs principaux inconvénients.

Nous proposons une opération de réorganisation immédiate de la base de données. Son utilisation provoque la suppression des classes et des versions historiques qui ne sont plus pertinentes pour les applications. Elle est déclenchée par un utilisateur quand il le juge nécessaire.

Les questions auxquelles nous voulons répondre dans ce chapitre sont :

• Comment déterminer si l'évolution du schéma doit être traduite par la modification du schéma ou la dérivation d'une nouvelle version du schéma ?

• Après l'évolution et lorsqu'un accès à un objet est demandé sous une classe, dont l'extension ne contient pas de version de cet objet, comment déterminer s'il est préférable de dériver une nouvelle version d'objet ou de simuler son comportement sous cette classe ?

• Comment déterminer si une classe ou une version du schéma n'est plus pertinente pour les applications ?

Pour répondre à la première question, nous considérons deux critères : (1) les effets de l'évolution sur l'environnement (c'est-à-dire, les données de la base et les programmes qui sont associés au schéma), (2) les caractéristiques des applications. Par exemple, dans le cas des applications orientées conception, les utilisateurs préfèrent garder la trace des évolutions du schéma.

Pour répondre à la deuxième et à la troisième question, nous considérons l'importance de l'existence d'une version d'objet, d'une classe ou d'une version du schéma dans la base de données (par exemple, pour assurer la pérennité des données ou augmenter les performances).

Ce chapitre est organisé comme suit. Dans un premier temps (section IV.2), nous présentons le processus lié au changement du schéma. Dans un second temps (section IV.3), nous présentons notre technique d'adaptation des instances. Dans la section IV.4, nous présentons l'opération de réorganisation immédiate de la base de données.

IV.2. Processus de changement du schéma

Pour changer le schéma, l'utilisateur utilise une ou plusieurs opérations de la taxonomie que nous détaillons dans l'annexe A. L'évolution est appliquée sur la version courante du schéma. Le système accepte cette évolution si elle satisfait la cohérence de structure. Si l'évolution est acceptée, elle est traduite par une *modification* du schéma ou par la *dérivation d'une nouvelle version* du schéma. La décision d'utiliser la modification ou le versionnement du schéma est basée sur l'intention de l'utilisateur d'une part et les effets de l'évolution sur les données de la base et les programmes associés d'autre part.

Une évolution de schéma est étudiée en analysant ses conséquences sur l'environnement (la base de données et ses applications). Elle peut se traduire par la suppression ou la modification des informations (c'est-à-dire, données ou opérations) et certains programmes peuvent devenir incompatibles vis-à-vis du schéma après l'évolution.

Dans un premier temps (paragraphe IV.2.1), nous présentons une typologie des opérations d'évolution de schéma. Dans un second temps (paragraphe IV.2.2), nous présentons les règles utilisées pour traduire une évolution par le versionnement ou la modification du schéma.

IV.2.1. Catégories des opérations sur le schéma

Une évolution de schéma peut être *soustractive* ou *non-soustractive*. Une évolution est soustractive si elle introduit une perte d'information.

D'après sa définition, une évolution soustractive peut introduire la suppression ou la modification :
1) des valeurs de certains objets. Il s'agit alors d'une suppression ou d'une modification de certains attributs.
2) des interfaces de certains objets. Il s'agit dans ce cas de la suppression ou de la modification de certaines opérations.

Pour savoir si une évolution de schéma est soustractive ou non, le système compare les définitions du schéma avant et après l'évolution.

Notation : dans ce paragraphe lorsque nous utilisons le terme *schéma,* c'est pour désigner la *version courante* du schéma. Lorsque t désigne un terme avant l'évolution, t' désigne le même terme après l'évolution. Une propriété désigne un attribut ou une opération.

Une propriété p est définie dans une classe c. La propriété p est héritée dans chacune des sous-classes de c où p n'est pas redéfinie. Ainsi, une propriété est associée à un ensemble de classes. L'expression `C_Set(p)` est l'ensemble des classes du schéma auxquelles est associée la propriété p. L'expression `Sign(p, c)` est la signature de la propriété p dans la classe c. Si p est un attribut, alors `Sign(p, c)` est un type sinon, `Sign(p, c)` est la signature.

Définition (évolution soustractive). Étant donné un schéma s d'une base de données. Soient C(s) l'ensemble de toutes ses classes et Prop(s) l'ensemble de toutes les propriétés de ses classes. Une évolution est soustractive si une des deux conditions suivantes est vérifiée :
1) $\exists p_i \in \mathtt{Prop}(s)$, $C_Set(p_i) - C_Set(p'_i) \neq \{\}$.
2) $\exists (p_i, c_j) \in \mathtt{Prop}(s) \times C(s)$, $\mathtt{sign}(p'_i, c'_j)$ n'est pas compatible avec $\mathtt{sign}(p_i, c_j)$. Notons que pour un attribut, "n'est pas compatible" signifie "n'est pas un sous-type".Δ

La première condition signifie que l'évolution provoque la suppression de certaines propriétés. La suppression d'une propriété, d'une classe ou d'un lien d'héritage en sont des exemples.

La deuxième condition signifie que l'évolution provoque le remplacement des définitions de certaines propriétés par d'autres définitions qui ne sont pas compatibles avec les anciennes. La modification du type d'un attribut en un type qui n'est pas sous-type de l'ancien en est un exemple.

Si aucune des deux conditions n'est vérifiée, alors l'évolution est non-soustractive. Dans ce cas, les définitions des classes après l'évolution sont compatibles (au sens du sous-typage) avec leur définition avant l'évolution. Par conséquent, la manipulation d'un objet créé après l'évolution par un programme qui existait avant l'évolution ne génère pas d'erreur et la conversion d'un objet créé avant l'évolution pour être conforme à la définition de sa classe après l'évolution n'introduit pas de perte d'information.

Exemple. LeSalaire-mensuel est un attribut de la classe Personne et Salaire_Annuel() est un programme qui fait référence à cet attribut (Cf. Figure IV.1). L'évolution du schéma consiste à remplacer l'attribut LeSalaire-mensuel par l'attribut LeSalaire-annuel. L'objet E de la classe Personne est créé après l'évolution du schéma. L'accès à l'objet E dans le programme Salaire_Annuel() (c'est-à-dire, Salaire_Annuel(E)) produit une erreur car l'attribut LeSalaire-mensuel n'existe pas pour l'objet E. Par conséquent, cette évolution est soustractive.

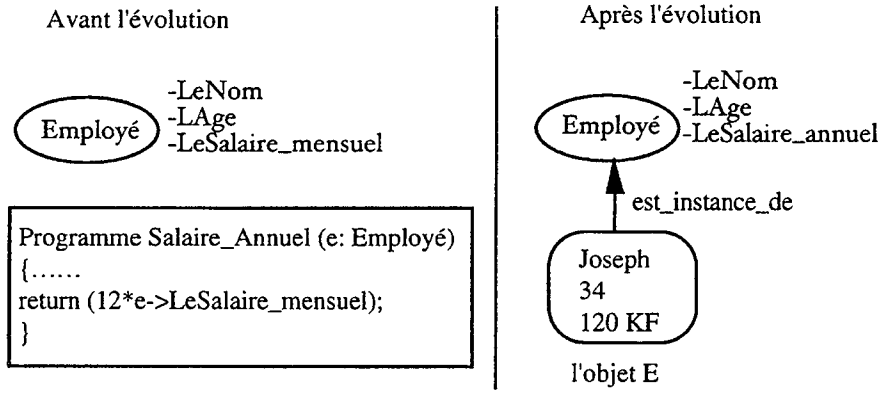


FIG. IV.1 : évolution soustractive

IV.2.2. Modification & version

La décision d'utiliser le versionnement ou la modification pour traduire une évolution du schéma est basée sur les caractéristiques de l'évolution (soustractive ou non) et les caractéristiques des applications.

Notre approche combine la modification et le versionnement pour gérer l'évolution de schéma, en utilisant les règles suivantes :

Règle MD (Mode par Défaut) : tout évolution soustractive provoque la dérivation d'une nouvelle version du schéma. Tout autre évolution se traduit par la modification du schéma.

Règle MI (Mode imposé) : le mode d'évolution du schéma peut être imposé par l'utilisateur ("modification" ou "versionnement").

La règle MD est une conséquence directe de la typologie des opérations présentées dans le paragraphe précédent. Ainsi, une nouvelle version est systématiquement générée seulement pour assurer la pérennité des données et/ou la compatibilité des programmes.

Nous citons à titre d'exemple, qu'une expérience à l'échelle industrielle, effectuée pendant 18 mois pour mesurer l'évolution de schéma de bases de données relationnelles (fréquence des opérations et conséquences sur l'environnement) [Sjo93a, Sjo93b], montre que, les opérations non-soustractives sont plus fréquentes que les opérations soustractives.

Dans la mesure où nous considérons que les évolutions soustractives ont une faible fréquence, cette règle ne pose pas le problème de la prolifération du nombre de versions.

La règle MI est proposée pour prendre en compte les intentions de l'utilisateur. Par conséquent elle permet de considérer les caractéristiques des applications et la nature de l'évolution. Par exemple, une évolution du schéma peut traduire la correction d'erreurs et doit donc provoquer la modification du schéma.

IV.3. Adaptation des instances

Nous avons vu dans le chapitre 3, que l'évolution du schéma d'une base de données peut introduire une incohérence de l'instance vis-à-vis du schéma. Dans cette partie, étant donnés un schéma et une instance, les questions sont :
1) l'instance est-elle cohérente vis-à-vis du schéma ?
2) sinon, comment la réorganiser pour la rendre cohérente ?

Nous rappelons que l'adaptation des instances est le terme utilisé pour désigner le processus de vérification de la cohérence et de réorganisation de l'instance d'un schéma [Cla92, BF95, FMZ95a].

Dans cette section, nous développons une technique d'adaptation des instances qui combine la conversion, l'émulation et le versionnement [Ben95b]. Quand un objet o est accédé, dans un programme ou une requête, sous la classe (c, n), le système vérifie si l'objet o possède une version sous la classe (c, n). Si oui, l'objet est utilisé par l'application sans aucune transformation. Si non, une nouvelle version de l'objet o dont la valeur est conforme à la structure de la classe (c, n) doit être générée.

Cette nouvelle version est *stockée* seulement si sa conservation en tant que telle est importante, par exemple, pour améliorer les performances ou assurer la pérennité, etc.. Autrement, elle est *calculée,* c'est-à-dire qu'elle ne persiste pas.

Pour décider si une version d'objet doit être stockée ou calculée, nous avons introduit la notion de *niveaux de pertinence* d'une classe. Une classe peut être *pertinente* ou *obsolète*. Elle est pertinente si elle est associée à la version courante du schéma ou si elle est utilisée dans un nombre de programmes jugé suffisamment élevé par un utilisateur expert (par exemple, l'administrateur de la base de données). Autrement, elle est *obsolète*. Ainsi, la nouvelle version de l'objet o sous la classe (c, n) est stockée seulement si (c, n) est pertinente.

Cette section est organisée comme suit. Dans la section IV.3.1, nous rappelons brièvement la terminologie qui sera utilisée dans la suite de ce chapitre. Dans la section IV.3.2, nous décrivons les concepts introduits pour définir les niveaux de pertinence d'une classe. Dans la section IV.3.3, nous détaillons notre technique d'adaptation.

IV.3.1. Terminologie

Dans cette section, nous rappelons les termes utilisés dans notre technique d'adaptation des instances. Nous montrons la conversion d'objets, le versionnement d'objets et l'émulation. Nous donnons tout d'abord l'exemple que nous utilisons pour illustrer ces techniques.

Exemple. Nous supposons que la définition de la classe Personne dans la version (s, 0) du schéma s contient les attributs LeNom et LAge. per est un objet qui est créé sous la classe (Personne, 0). La version de l'objet per sous la classe (Personne, 0) est une version stockée.

L'évolution du schéma consiste à supprimer l'attribut LAge de la définition de la classe Personne. Cette évolution est traduite par la dérivation d'une version (s, 1) du schéma s. La définition de la classe Personne dans la version (s, 1) contient l'attribut LeNom. L'objet per n'a pas de version sous la classe (Personne, 1).

Dans la suite nous illustrons les techniques considérées quand un accès à l'objet per est demandé sous la classe (Personne, 1).

Conversion (Cf. Figure IV.2.b)

La version de l'objet per sous la classe (Personne, 0) est convertie pour être conforme avec la classe (Personne, 1). L'objet per n'a plus de version sous (Personne, 0). L'inconvénient principal de cette technique est la perte d'informations [Mon93].

Émulation (Cf. Figure IV.2.c)

La version (per, 1) sous la classe (Personne, 1) est calculée à partir de la version (per, 0) sous la classe (Personne, 0). En fait, seule la version (per, 0) existe physiquement. Cette technique présente deux inconvénients [Zdo86a] :

- L'accès à un objet peut introduire une dégradation des performances du système,
- La valeur d'un attribut ajouté (respectivement supprimé) dans un objet créé avant l'évolution du schéma (respectivement créé après l'évolution du schéma) est toujours donnée par défaut.

Versionnement (Cf. Figure IV.2.d)

La version (per, 1) sous la classe (Personne, 1) est dérivée à partir de la version (per, 0) sous la classe (Personne, 0). L'avantage est que l'accès à une version d'objet se fait sans coût supplémentaire. Cependant, le nombre de versions d'objets peut être important car les applications à objets ont une nature évolutive [Ben95a, HK94, LW94]. L'augmentation du nombre de versions

peut être un obstacle car les coûts de stockage et de maintenance des relations entre les versions augmentent avec le nombre de versions. Pour le détail sur le problème de la maintenance des relations entre les versions, nous invitons le lecteur à lire les travaux présentés dans [BM88, Cla92, TOC93].

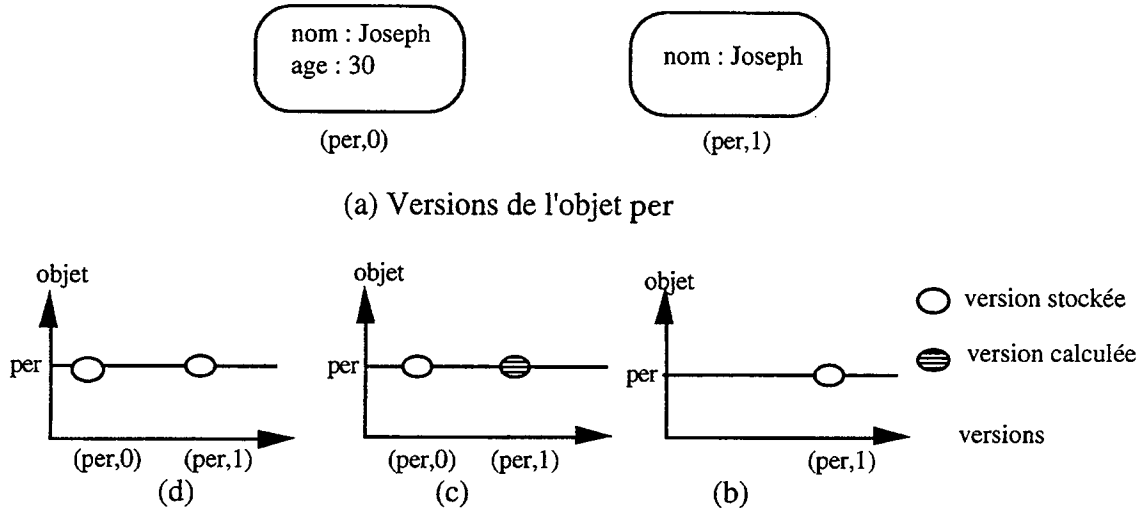


FIG. IV.2 : version stockée et version calculée

IV.3.2. Niveaux de pertinence d'une classe

Nous rappelons que, dans la technique d'adaptation que nous proposons, quand une nouvelle version d'objet est dérivée, celle-ci est conservée seulement si le système la juge importante (par exemple, pour assurer la pérennité des données, améliorer les performances, tracer l'évolution de l'objet). Le système utilise le niveau de pertinence de la classe à laquelle est associée la version d'objet pour décider de la conserver ou non.

Dans cette section nous introduisons la notion de niveaux de pertinence d'une classe. Avant de définir les niveaux de pertinence d'une classe, nous donnons les notions sur lesquelles s'appuient les définitions :

• **Version courante du schéma** : le schéma d'une base de données est partagé par toutes les applications de cette base. Dans une approche fondée sur les versions, les applications peuvent avoir des visions différentes sur le schéma, puisqu'il en existe plusieurs versions. Pour un changement du schéma reflétant une évolution du monde réel, nous considérons que la version courante du schéma est la version qui reflète le mieux le monde réel courant. Dans une approche fondée sur la modification, cette version est la seule vision instantanée du schéma.

• **Nombre de programmes qui utilisent une classe :** le nombre de programmes qui utilisent une classe constitue une mesure quantitative importante de sa pertinence. À une classe est associé un ensemble de programmes. Leur nombre varie au cours du temps car les programmes évoluent à leur tour. Les programmes faisant référence à une classe supprimée doivent être modifiés pour être compatibles avec la version courante du schéma afin de lui être re-associés. Si nous supposons que l'effort de modification est le même pour tous les programmes, alors le coût total de la modification augmente avec le nombre de programmes à modifier. Ainsi, plus le nombre de programmes associés à une classe est élevé moins on a intérêt à la supprimer. Pour simplifier, dans la suite, nous retenons cette hypothèse.

Définition (*poids* d'une classe). Le poids d'une classe est une valeur réelle comprise entre 0 et 1. Il mesure le degré d'importance de la conservation des instances de cette classe (des versions d'objets). Δ

La fonction `C_Weight` associe à une classe son poids. La définition de cette fonction prend en compte le statut (courante ou historique) des versions du schéma auxquelles est associée la classe et le nombre de programmes associés à la classe. Nous identifions dans la section IV.5 d'autres paramètres pour déterminer le poids d'une classe.

Le poids d'une classe définie dans la version courante du schéma est 1 (poids maximal). Pour une classe qui n'est pas associée à la version courante du schéma, le poids est égal au rapport du nombre de programmes qui l'utilisent par le nombre de tous les programmes qui utilisent le schéma.

Le *contexte* d'un programme est l'ensemble des ressources (paramètres formels, variables globales et locales, attributs et méthodes) qu'il utilise. L'expression `C_Context(p)` est l'ensemble des classes utilisées dans le code du programme p. Cet ensemble est constitué des classes utilisées comme types des attributs, des paramètres formels et des variables.

Soit `Use(p)` l'ensemble de programmes appelés par le programme p et `C_Ref(p)` l'ensemble de classes référencées (directement ou indirectement) par les classes de `C_Context(p)`. `Use(p)` est déterminé en utilisant les *liens d'appel* entre les programmes. C_Ref(p) est déterminé en utilisant les *liens de référence* entre les classes (Cf. Chapitre III, pour les liens de référence).

L'expression `ClassesOfProgram`(p) est l'ensemble de toutes les classes dont des objets peuvent être accédés par le programme p. Cet ensemble est déterminé au moment de la compilation de p en utilisant l'expression suivante :

$$\texttt{ClassesOfProgram}(p) = \texttt{C_Context}(p) \cup \texttt{C_Ref}(p) \bigcup_{p_i \in \texttt{Use}(p)} \texttt{ClassesOfProgram}(p_i)$$

Exemple. Dans la figure IV.3, le programme p_1 utilise la classe c_1 et appelle les programmes p_2 et p_3. La classe c_1 référence directement la classe c_2 (par exemple, par l'intermédiaire d'un attribut) et indirectement la classe c_3 (par l'intermédiaire de la classe c_2). Le programme p_2 utilise la classe c_4. Le programme p_4 appelle le programme p_5 qui utilise la classe c_5. Par conséquent, `ClassesOfProgram`$(p_1) = \{c_1\} \cup \{c_2, c_3\} \cup \{c_4, c_5\}$.

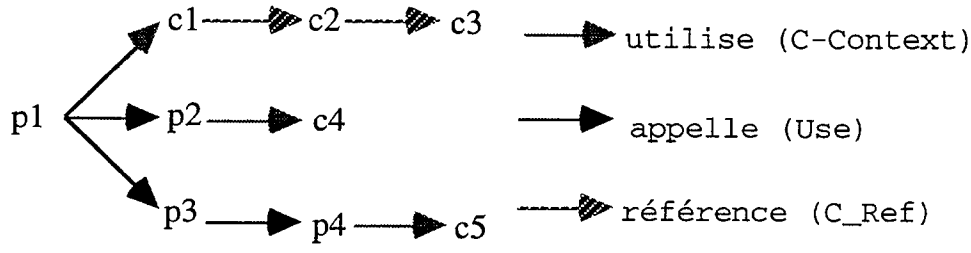


FIG. IV.3 : graphe de dépendance

Soit s le schéma d'une base de données. La fonction `C_Weight` est définie par :

$C_Weight : C \times N \rightarrow [0,1]$

{`C_Weight`(c, n) est le poids de la classe (c, n),

$$\texttt{C_Weight}(c, n) = \begin{cases} 1 \text{ si } (c, n) \in \texttt{SchVerClasses}(\texttt{Current}(s)) \\ \dfrac{\texttt{card}\{p_i \in P / (c, n) \in \texttt{ClassesOfProgram}(p_i)\}}{\texttt{AllPr}(s)} \text{ sinon} \end{cases}$$

où :

`AllPr` (s) est le nombre de tous les programmes qui utilisent s. Nous supposons que `AllPr` (s)≠0, P est l'ensemble de tous les programmes,

`SchVerClasses`(s, m) est l'ensemble des classes de la version (s, m) du schéma s}

La valeur 1 de `C_Weight`(c, n) représente la situation où la conservation des instances de la classe (c, n) est d'une grande importance (la classe (c, n) est associée à la version courante du schéma ou alors elle est utilisée dans tous les programmes), la valeur 0 représente la situation où la conservation des instances de la classe (c, n) est inutile (la classe (c, n) n'est pas associée à la version courante du schéma et elle n'est utilisée dans aucun programme). Les valeurs intermédiaires

représentent les situations où la conservation des instances de la classe (c, n) est plus ou moins importante.

Remarque (généralisation) : Dans le cas où l'effort de modification n'est pas le même pour tous les programmes (par exemple, la modification d'un programme est moins coûteuse que celle d'un autre), le poids d'une classe peut être calculé de la manière suivante :

$$\mathtt{C_Weight}^{g}(c, n) = \begin{cases} 1 \text{ si } (c, n) \in \mathtt{SchVerClasses}(\mathtt{Current}(s)) \\ \dfrac{\sum_{p_i \in PC} \mathtt{Effort}(p_i)}{\sum_{p_k \in PS} \mathtt{Effort}(p_k)} \text{ sinon} \end{cases}$$

avec :

$\mathtt{Effort}(p_i)$ est une valeur réelle strictement positive qui représente l'effort de modification du programme p_i, c'est-à-dire l'effort que doit fournir le programmeur pour modifier p_i. Plus cette valeur est élevée plus la modification du programme est coûteuse. Cette valeur est fixée, pour tous les programmes, par l'administrateur de la base.

PC est l'ensemble des programmes qui utilisent la classe (c, n), c'est-à-dire :

$PC=\{p_i \in P / (c, n) \in \mathtt{ClassesOfProgram}(p_i)\}$.

PS est l'ensemble des programmes qui utilisent le schéma s. Δ

Avec cette définition, pour une classe qui n'est pas associée à la version courante du schéma, le poids est égal au rapport de l'effort de modification des programmes qui l'utilisent par l'effort de modification de tous les programmes qui utilisent le schéma. Dans le cas où l'effort de modification est le même pour tous les programmes, les deux définitions du poids d'une classe sont équivalentes, c'est-à-dire :

$\forall (c, n) \in C \times N,\ \mathtt{C_Weight}^{g}(c, n) == \mathtt{C_Weight}(c, n)$.

Définition (*niveaux de pertinence* d'une classe). Étant donnée une base de données, le niveau de pertinence d'une classe caractérise l'importance de la conservation de ses instances vis-à-vis des applications de cette base de données. Nous considérons deux niveaux de pertinence. Une classe peut être :

• ***Pertinente*** : la conservation des instances d'une classe pertinente est d'une grande importance, par exemple, une classe utilisée dans un nombre de programmes suffisamment élevé.

• ***Obsolète*** : la conservation des instances n'a aucune importance, par exemple, une classe qui n'est pas associée à la version courante du schéma de la base et qui n'est utilisée dans aucun programme. Δ

Une classe est pertinente si son poids est supérieur à un seuil (seuil d'*obsolescence*) fixé à l'avance, par l'administrateur de la base de données par exemple. Dans le cas contraire, elle est obsolète.

Plus formellement, soit `PL` la fonction qui à une classe associe son niveau de pertinence, soit OT (OT ∈ R et OT ∈ [0, 1]) le seuil d'obsolescence, `PL` est telle que :

$$PL{:}C \times N \rightarrow \{\text{"Pertinente"},\text{"Obsolete"}\}$$

$$PL(c,\ n) = \begin{cases} \text{"Pertinente" si } \texttt{C_Weight}(c,\ n) > OT, \\ \text{"Obsolete" sinon} \end{cases}$$

IV.3.3. Technique d'adaptation des instances

La technique que nous proposons pour adapter les instances est basée sur les niveaux de pertinence d'une classe. Quand un objet o est accédé sous une classe (c, n) dont l'extension ne contient pas de version de cet objet, alors une nouvelle version stockée de o est générée seulement si (c, n) est pertinente. Autrement (c'est-à-dire, (c, n) est obsolète), une version calculée est générée. Ainsi, seules les versions dont la conservation est importante pour les applications de la base de données sont stockées.

Le processus de génération d'une nouvelle version (stockée ou calculée) de o sous la classe (c, n) consiste en une séquence de primitives de base (dérivation ou conversion d'une version d'objet) sur les versions de cet objet. La première primitive est appliquée sur une version stockée de l'objet, que nous appelons la version *origine* de la génération.

Dans la section IV.3.3.1, nous présentons les primitives utilisées par le processus de génération d'une nouvelle version d'objet : la fonction qui détermine la version origine d'un objet et l'opération qui génère une version d'un objet à partir de sa version origine. La section IV.3.3.2 présente l'algorithme qui réalise la technique d'adaptation des instances.

IV.3.3.1 Primitives

La fonction `OriginVersion`

Un objet peut avoir des versions stockées et des versions calculées. Au cours de sa vie, un objet possède toujours au moins une version stockée. Les versions stockées d'un objet peuvent être associées à des classes définies dans des versions du schéma qui ne sont pas successives.

Par exemple, dans la figure IV.4 , l'objet o a deux versions stockées. La version (o, 0) qui est une instance de la classe (c, 0) et la version (o, 3) qui est une instance de la classe (c, 3). Les classes (c, 0) et (c, 3) sont définies dans des versions non successives du schéma.

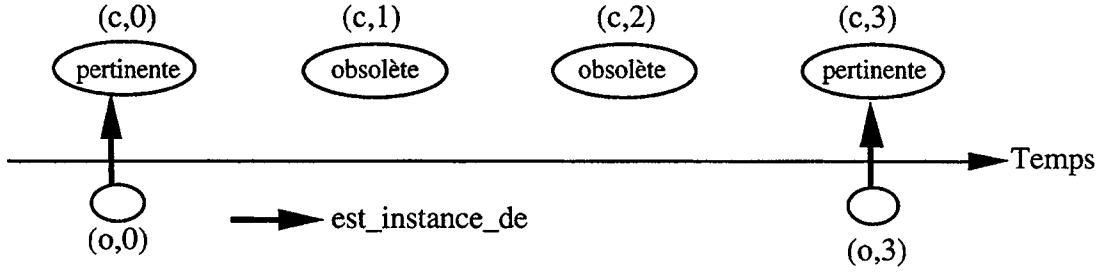


FIG. IV.4 : versions stockées de l'objet o.

La version origine pour générer la version d'un objet o sous la classe (c, n) est la version stockée de o, dont le numéro est le plus proche de n. Elle est déterminée par la fonction `OriginVersion` définie comme suit :

`OriginVersion` : $O \times (C \times N) \rightarrow O \times N$

{`OriginVersion`(o, (c, n)) est la version origine pour générer la version de l'objet o sous la classe (c, n), telle que :

`OriginVersion`(o, c, n) $\in$ {(o, i) $\in$ `ObjVers`(o) / $\forall$(o, j) $\in$ `ObjVers`(o), $|n - i| \leq |n - j|$}

et `ObjVers`(o) est l'ensemble des versions stockées de l'objet o}

La figure IV.5 illustre trois cas d'application de la fonction `OriginVersion`. La version origine pour générer la version de l'objet o sous la classe (c, n) est :

- (o, n+1) dans le cas présenté par la figure IV.5.a.
- (o, n-2) dans le cas présenté par la figure IV.5.b.
- (o, n-1) dans le cas présenté par la figure IV.5.c.

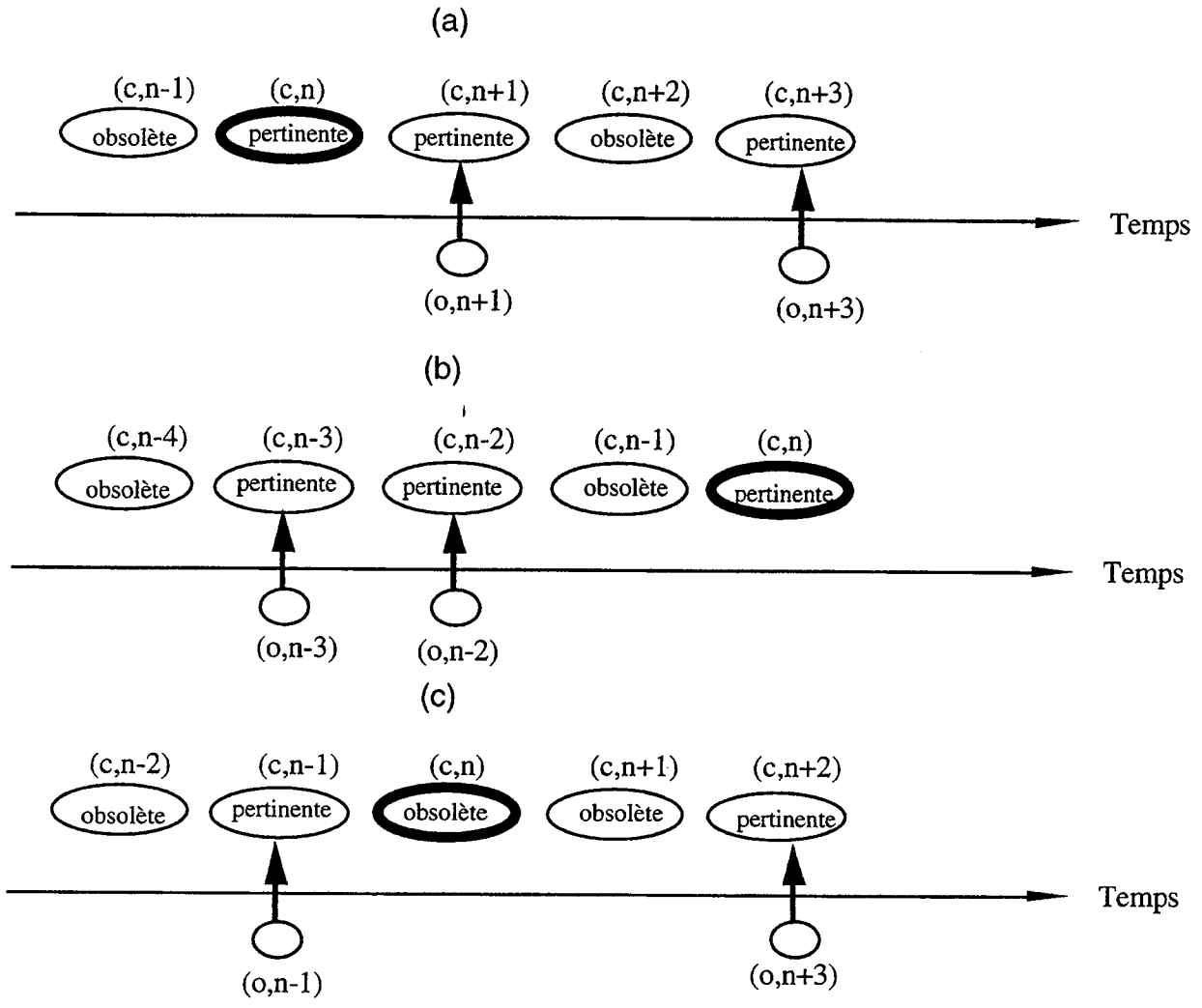


FIG. IV.5 : positions de la classe (c, n).

L'opération *object_version_gen*

Cette opération génère la version d'un objet quand celui-ci est accédé sous une classe dont l'extension ne contient pas de version de cet objet. Elle est définie comme suit :

object_version_gen (<ident_objet>,[<numéro_origine>], <classe> [, <statut>])

ident_objet est l'identificateur de l'objet et classe est l'identificateur de la classe. Le paramètre optionnel numéro_origine est le numéro de la version de ident_objet à partir de laquelle la nouvelle version doit être générée. Le paramètre optionnel statut prend ses valeurs dans l'ensemble {"stockée", "calculée"}. Cette opération provoque le stockage ou le calcul de la version

générée selon que la valeur du paramètre statut est "stockée" ou "calculée". Autrement, si la valeur de statut n'est pas fixée, alors la version générée dépend du niveau de pertinence de la classe classe. Elle est stockée si classe est pertinente et calculée sinon. L'utilisation du paramètre statut est parfois nécessaire. Par exemple, dans certaines situations la version générée doit être stockée quelque soit le niveau de pertinence de classe. Ces situations seront décrites plus loin (Cf. Section IV.4).

Cette opération déclenche une séquence de dérivation de versions (stockées ou calculées) de l'objet ident_objet. La première primitive de la séquence est la dérivation d'une version de l'objet ident_objet à partir de la version origine pour générer la version de ident_objet sous classe (c'est-à-dire, (ident_objet, numéro_origine) si la valeur de numéro_origine est fixée, OriginVersion(ident_objet, classe) sinon). La dernière primitive de la séquence dérive la version de l'objet ident_objet sous la classe classe. Dans la suite de ce paragraphe, (c_1, n_1) dénote la classe de la version OriginVersion(ident_objet, classe) et (c_2, n_2) dénote la classe classe.

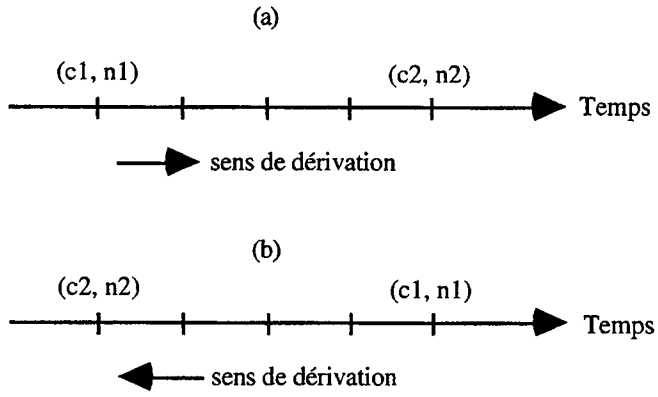


FIG. IV.6 : sens de dérivation des versions

Si la classe (c_2, n_2) est plus récente que la classe (c_1, n_1) (Cf. Figure IV.6.a, $n_2>n_1$), alors à la première étape, une version de l'objet ident_objet est dérivée sous la classe de ident_objet, qui est la *voisine droite* de la classe (c_1, n_1) dans la séquence des classes de ident_objet. A l'étape i, une version de l'objet est dérivée sous la classe de ident_objet, qui est la voisine droite de la classe de la version de ident_objet qui est dérivée à l'étape i-1.

Si la classe (c_2, n_2) est plus ancienne que la classe de la classe (c_1, n_1) (Cf. Figure IV.6.b, $n_2<n_1$), alors à la première étape, une version de l'objet ident_objet est dérivée sous la classe de ident_objet, qui est la *voisine gauche* de la classe (c_1, n_1) dans la séquence des classes de ident_objet. A l'étape i, une version de l'objet est dérivée sous la classe de ident_objet, qui est la voisine gauche de la classe de ident_objet qui est dérivée à l'étape i-1.

L'ensemble des classes d'un objet forme une séquence ordonnée selon les numéros des classes (Cf. Chapitre III). La voisine droite (respectivement gauche) d'une classe dans la séquence des classes d'un objet est déterminée par la fonction `NextClass` (respectivement `PreviousClass`). Les fonctions `NextClass` et `PreviousClass` sont définies par :

`NextClass` : $O \times (C \times N) \rightarrow C \times N$

{`NextClass`(o, (c, k)) est la voisine droite de la classe (c, k) dans la séquence des classes de o, telle que :

`NextClass`(o, (c, k)) $\in \{(c, i) \in$ `Classes`(o) / $\forall (c, j) \in$ `Classes`(o), $i > k \wedge i \leq j\}$

}

`PreviousClass` : $O \times (C \times N) \rightarrow C \times N$

{`PreviousClass`(o, (c, k)) est la voisine gauche de la classe (c, k) dans la séquence des classes de o, telle que :

`PreviousClass`(o, (c, k)) $\in \{(c, i) \in$ `Classes`(o) / $\forall (c, j) \in$ `Classes`(o), $i < k \wedge i \geq j\}$

}

Durant le processus de génération, des versions stockées ou calculées de l'objet ident_objet sont dérivées. Quand on veut dériver une version de ident_objet sous une classe, le niveau de pertinence de cette classe est déterminé en utilisant la fonction `PL` définie plus haut (Cf. Section IV.3.2). Si la classe est pertinente, alors une version stockée de l'objet ident_objet est dérivée. Sinon, une version calculée de l'objet o est dérivée.

Une nouvelle version de l'objet `ident_objet` est dérivée en utilisant la primitive `object_version_derivation` (Cf. Chapitre III). Cette primitive dérive une version stockée d'un objet, alors qu'ici une version d'objet peut être stockée ou calculée. Pour cette raison, cette primitive est redéfinie en ajoutant le paramètre statut qui prend ses valeurs dans l'ensemble {"stockée", "calculée"}. Ainsi, cette primitive provoque le stockage ou le calcul de la version dérivée selon que la valeur du paramètre statut est "stockée" ou "calculée" :

`object_version_derivation` (<ident_objet>, <classe1>, <classe2>, <statut>) [<initialisation>]

IV.3.2.2 Algorithme

Dans cette section, nous présentons l'algorithme qui réalise notre technique d'adaptation des instances. Avant de donner l'algorithme, nous rappelons que le principe fondamental d'une

technique d'adaptation des instances différée est de garder la trace de l'évolution du schéma. Ainsi, lorsqu'une évolution de schéma est traduite par une modification, le système génère une nouvelle version qui reflète la modification et pour des raisons d'implantation conserve l'ancienne version (Cf. Figure IV.7). Cette dernière est appelée *version invisible* car elle est transparente pour les applications.

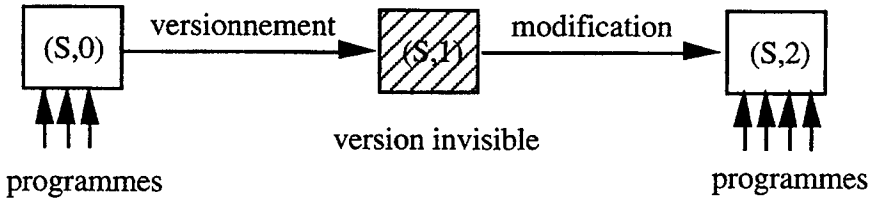


FIG. IV.7 : trace de l'évolution du schéma S

Par exemple, considérons le schéma s qui a évolué selon la trace donnée dans la figure IV.7 :

1) la première évolution de s est traduite par le versionnement de s, c'est-à-dire la dérivation de la version (s, 1) du schéma s,

2) la deuxième évolution est traduite par la modification du schéma s.

Dans le dernier cas, la version (s, 2) qui reflète la modification est dérivée à partir de la version (s, 1). La version (s, 1) n'est pas visible par les programmes des applications. Tous les programmes qui étaient associés à (s, 1) avant l'évolution, sont associés à (s, 2) après l'évolution. Le poids des classes qui sont propres à (s, 1) (c'est-à-dire, les classes qui ne sont pas importées) est égal à 0. Ces classes sont donc obsolètes. Dans cette partie, nous présentons l'algorithme qui réalise notre technique d'adaptation des instances.

L'algorithme *HybrideAdaptation* (Cf. Figure IV.8)

C'est l'algorithme utilisé par le système lorsqu'un objet o est accédé sous une classe (c, n) dans une application.

Lors de tout accès à un objet o sous une classe (c, n), l'algorithme vérifie si l'objet o possède une version stockée sous la classe (c, n). Sinon, une nouvelle version (stockée ou calculée) de l'objet o dont la valeur est conforme à la structure de la classe (c, n) doit être générée.

L'algorithme utilise l'expression `OriginVersion`(o, (c, n)) pour déterminer l'origine de génération de o sous la classe (c, n) et la primitive `object_version_gen`(o, (c, n)) pour générer la version de l'objet o sous la classe (c, n).

Après la génération de la version de l'objet o sous la classe (c, n), l'algorithme vérifie si le poids de la classe de l'origine de génération de o sous (c, n) est nul et si l'objet o possède au moins une autre version stockée qui est différente de l'origine de génération. Si oui, alors l'origine de génération est supprimée en utilisant la primitive `del_object_version` (o, r) telle que, (o, r) = `OriginVersion`(o, (c, n)).

En effet, le poids d'une classe est nul dans l'un des deux cas suivants :
1) la classe est propre (définie ou redéfinie) à une version du schéma qui est invisible par les applications. C'est-à-dire, une version du schéma sur laquelle une évolution du schéma est traduite par une modification.
2) la classe n'est pas associée à la version courante du schéma et elle n'est utilisée par aucun programme d'application.

La suppression de l'origine de génération signifie que l'algorithme utilise la *conversion différée* quand il n'est pas nécessaire de la conserver. Ainsi, l'origine de génération est conservée seulement pour préserver la propriété qui exige qu'un objet doit avoir au moins une version stockée.

HybrideAdaptation (o : O, (c, n) : C×N)
{o est un objet de la classe (c, n). Notons que dans cet algorithme l'expression `Classe`(o, i) est la classe numéro i de l'objet o.
1. Vérifier si l'objet o a une version stockée sous la classe (c, n) : (o, n) ∈ `Ext`(c, n) ?.
2. Si (o, n) ∈ `Ext`(c, n), alors l'objet o peut être utilisé par les applications sans aucune adaptation.
3. Sinon, générer une nouvelle version de o, dont la valeur est conforme à la structure de (c, n).
    3.1. Trouver la version origine pour générer la version de o sous (c, n).
        (o, r)=`OriginVersion`(o, (c, n)); */* *(o, r) est l'origine de génération* */*
    3.2. Générer la version de l'objet o sous la classe (c, n).
        `object_version_gen`(o, (c, n)).
    3.3. Si le poids de la classe à laquelle est associée la version (o, r) est nul et l'objet o a au moins une version stockée qui est différente de (o, r), alors Supprimer la version (o, r) :
        si (`card`(`ObjVers`(o))>1 et `C_Weight`(`Classe`(o, r))=0)
        alors `del_object_version`(o, r);
}

FIG. IV.8 : algorithme HybrideAdaptation

L'algorithme provoque la génération d'une version stockée d'un objet seulement si la classe associée est pertinente :

• Elle est définie dans la version courante du schéma.

• Elle est utilisée dans un nombre de programmes, jugé suffisamment élevé par un utilisateur expert de la base de données. Par conséquent, l'accès aux versions associées à cette classe est fréquent, dans la mesure où nous considérons que le taux d'accès aux objets par des programmes qui les référencent est le même. Ainsi, une version d'objet est conservée pour améliorer les performances du système.

L'algorithme génère une version d'un objet à partir de sa version stockée, qui est associée à la classe de l'objet qui est la plus proche, dans la séquence des classes de l'objet, parmi toutes les classes qui possèdent une version stockée de l'objet. Ainsi, le processus de génération déclenche un ensemble minimal d'opérations sur les versions de l'objet. Par conséquent, la redondance est évitée et le temps de génération est minimisé.

IV.4. Réorganisation de la base de données

L'algorithme présenté dans la section précédente pose les problèmes suivants :

• Le système garde la trace de l'évolution durant toute la vie de la base de données. En effet, même si certaines classes ou versions du schéma doivent être supprimées, parce que les évolutions associées sont traduites par des modifications, ces classes et ces versions du schéma sont seulement rendues invisibles pour les applications.

• Une fois créée, une version du schéma est conservée même si elle associée à un nombre très faible de programmes.

Ainsi, la taille de la base de données risque d'augmenter considérablement. La question est de remédier aux inconvénients de notre technique d'adaptation des instances [Ben95a].

Nous pensons que l'importance accordée à la conservation d'une version ou d'une classe n'est pas absolue et peut varier durant la vie de la base de données. En effet, il est préférable de caractériser l'importance de l'existence physique d'une version ou d'une classe, au cours de sa vie. Par exemple, il est important d'offrir à l'utilisateur la possibilité de supprimer des versions d'objets qui ne sont plus aux services des applications.

Exemple. Supposons que la classe c possède les caractéristiques suivantes :

1) c est propre à une version invisible du schéma.

2) chacun des objets associés à c possède une version stockée dans au moins une autre classe.

Il est inutile de conserver une telle classe car les applications n'y font plus référence. La première caractéristique signifie qu'aucun objet ne sera accédé par les applications sous cette classe. La seconde caractéristique signifie que les instances (versions stockées) de cette classe ne sont pas nécessaires pour dériver des versions d'objets sous d'autres classes.

Dans cette partie, nous proposons une opération de *réorganisation immédiate* de la base de données. Cette opération peut être déclenchée par l'administrateur de la base de données. Elle est traduite par la suppression physique de certaines classes et/ou de certaines versions historiques du schéma.

Dans un premier temps nous discutons de la suppression d'une classe ou d'une version du schéma (section IV.4.1). Puis, nous présentons l'opération de réorganisation d'une base de données (section IV.4.2).

IV.4.1. Suppression d'une classe ou d'une version du schéma

Dans le paragraphe IV.4.1.1, nous donnons les conditions qui doivent être vérifiées pour supprimer une classe et la primitive utilisée pour supprimer une classe. Dans le paragraphe IV.4.1.2, nous donnons les conditions qui doivent être vérifiées pour supprimer une version du schéma et la primitive utilisée pour supprimer une version du schéma.

IV.4.1.1. Suppression d'une classe

Conditions de suppression

Une classe peut être supprimée si elle satisfait les conditions suivantes :

• *Le poids de cette classe est nul.* C'est-à-dire, elle n'est ni associée à la version courante du schéma ni utilisée dans un programme d'application.

• *Chacune de ses sous-classes (lorsqu'il en existent) dans toutes les versions du schéma satisfait les conditions de suppression.*

La deuxième condition est imposée car la suppression d'une classe qui a des sous-classes ne vérifiant pas les conditions de suppression est une opération coûteuse. Elle provoque en effet, la réorganisation des sous-classes et de leurs instances pour préserver la cohérence de la base de données. Par conséquent, certains programmes devront être modifiés.

Primitive de suppression

L'opération utilisée pour supprimer une classe est définie par :

delete_class(<ident-classe>)

Cette opération supprime la classe qui est identifiée par ident-classe. La classe ident-classe doit être une feuille[4.1] dans toutes les versions du schéma qui lui sont associées et doit satisfaire les conditions de suppression.

Au moment de la suppression d'une classe, il est possible que son extension contienne des versions stockées. Parmi celles-ci, il peut y en avoir certaines qui seront utiles pour dériver des versions d'objets sous d'autres classes. Pour cela, la suppression d'une classe peut provoquer la suppression de certaines versions d'objets et/ou la conversion d'autres versions d'objets vers d'autres classes.

Étant donnée une classe (c, n). Soit o un objet dont une version est stockée sous la classe (c, n). La suppression de la classe (c, n) provoque la conversion de la version (o, n) vers une autre classe si les deux conditions suivantes sont vérifiées. Nous parlons des *conditions de conversion* de la version (o, n) :

• o est associé à d'autres classes qui sont différentes de (c, n) et qui ne vérifient pas les conditions de suppression. Cette condition signifie que l'objet o est susceptible d'être accédé sous une classe après la suppression de (c, n).

• o a une seule version stockée. Cette condition signifie que si (o, n) est supprimée après la suppression de (c, n), alors au moment de l'accès à o sous une autre classe soit (c, m), il n'existe pas d'origine de génération de la version de o sous (c, m).

Si une version (o, n) de la classe (c, n) doit être convertie (c'est-à-dire, elle satisfait les conditions de conversion), alors elle est convertie vers une classe de o que nous appelons la *classe d'accueil* de o. La classe d'accueil de o est la classe pertinente la plus proche de (c, n) si o est associé à des classes pertinentes. Sinon, c'est la classe la plus proche de (c, n).

Nous utilisons la fonction `ReceptionClass` pour déterminer la classe d'accueil d'un objet :

[4.1] Nous reviendrons sur la suppression d'une classe non feuille dans §IV.1.2.

ReceptionClass : $O \times (C \times N) \rightarrow C \times N$

{ReceptionClass(o, (c, n)) est la classe d'accueil pour convertir la version n de l'objet o.

$$\text{ReceptionClass}(o, (c, n)) \in \begin{cases} \{(c', i) \in PC / \forall (c'', j) \in PC, |n - i| \leq |n - j|\} \text{ si } PC \neq \{\} \\ \{(c', i) \in \text{Classes}(o) / \forall (c'', j) \in \text{Classes}(o), |n - i| \leq |n - j|\} \text{ sinon} \end{cases}$$

avec :

PC = {ci ∈ Classes(o) / PL(ci) ="Pertinente"} / * Les classes pertinentes de o * /

}

La conversion de la version (o, n) vers la classe ReceptionClass(o, (c, n)) consiste à :

• Générer une nouvelle version stockée de o sous la classe ReceptionClass(o, (c, n)) en utilisant la primitive object_version_gen (o, n, ReceptionClass(o, (c, n)), "stockée").

• Supprimer la version (o, n) en utilisant la primitive del_object_version(o, n).

L'algorithme de l'opération delete_class est donné dans la figure IV.9. Lors de la suppression d'une classe, pour chaque version d'objet de l'extension de cette classe, l'algorithme vérifie si elle satisfait les conditions de conversion. Si oui, la version d'objet est convertie vers la classe d'accueil de l'objet. Sinon, elle supprimée.

Algorithme DeleteClass ((c, n) : C×N))

{(c, n) vérifie les propriétés de suppression.

1. Convertir les versions d'objets de la classe (c, n) vers leur classes d'accueil.

Pour chaque version (o, n) ∈ Ext(c, n) faire.

1.1 Vérifier si (o, n) satisfait les conditions de conversion :

(card({ci ∈ Classes(o) / C - Weight(ci) ≠ 0}) > 1) ∧ (ObjVers(o) = {(o, n)})?

1.2. Si oui, générer une version stockée de l'objet o sous sa classe d'accueil.

1.2.1. Chercher la classe d'accueil de o

(c', r)=ReceptionClass(o, (c, n));

1.2.2. Générer la version stockée de l'objet o sous la classe (c', r). L'origine de génération est la version (o, n).

object_version_gen(o, n, (c', r) , "stockée")

1.3. Sinon, supprimer la version (o, n).

del_object_version(o, n).

2. Supprimer la définition de classe (c, n).

La structure et le comportement de la classe (c, n) sont supprimés. La classe (c, n) est supprimée de l'ensemble de classes associées aux versions du schéma qui la partagent.}

FIG.IV.9 : algorithme DeleteClass.

IV.4.1.2. Suppression d'une version du schéma

Pour savoir si une version du schéma peut être supprimée ou non nous utilisons son *poids*. Avant de donner les conditions de suppression, nous introduisons la définition du poids d'une version de schéma.

Définition (*poids* d'une version de schéma). Le poids d'une version du schéma est un entier naturel, qui mesure le degré d'importance de cette version du schéma vis-à-vis des applications de la base de données. Le poids d'une version historique du schéma est le nombre de programmes qui sont associés à cette version. Le poids de la version courante du schéma est fixé à l'*infini*.

Nous utilisons la fonction `S_Weight` pour déterminer le poids d'une version du schéma :

$S_Weight : S \times N \rightarrow N$

{`S_Weight`(s, n) est le poids de la version numéro n du schéma s,

$$\texttt{S_Weight}(s, n) = \begin{cases} \infty \text{ si } (s,n) = \texttt{Current}(s) \\ \texttt{card}\{p_i \in P / (s, n) = \texttt{LinkPrSch}(p_i)\} \text{ sinon} \end{cases}$$

où :

`LinkPrSch`(p) est la version du schéma à laquelle est associé le programme p (Cf. Chapitre 3)}. Δ

Remarque : Comme pour le poids d'une classe, dans le cas où l'effort de modification n'est pas le même pour tous les programmes, le poids d'une version de schéma peut être calculé en fonction de l'effort de modification des programmes qui lui sont associées.

Condition de suppression

• *Une version du schéma peut être supprimée si son poids est nul.*

Primitive de suppression

L'opération utilisée pour supprimer une version de schéma est définie par :

`delete_schema_version`(<ident-version-schéma>)

Cette opération supprime la version (du schéma) qui est identifiée par ident-version-schéma. La version ident-version-schéma doit vérifier la condition de suppression. Elle est traduite par la suppression des classes de la version ident-version-schéma qui vérifient les conditions de suppression.

Remarquons que cette opération provoque la suppression de toutes les classes qui sont propres (c'est-à-dire, elles ne sont pas associées à d'autres versions du schéma) à ident-version-schéma car une classe c propre à ident-version-schéma, vérifie les conditions de suppression d'une classe :

• *Le poids de c est nul.* Elle n'est, ni utilisée dans un programme, ni associée à la version courante du schéma. Elle n'est pas directement utilisée dans un programme parce que le nombre de programmes associés à ident-version-schéma est nul. Elle n'est pas indirectement utilisée dans un programme parce qu'elle est référencée seulement dans les classes qui sont propres à ident-version-schéma (Cf. Chapitre III).

• *Si c' est une sous-classe de c, alors c' est propre à ident-version-schéma.* L'importation d'une classe entraîne l'importation de ses super-classes (Cf. Chapitre III). Par conséquent, c' n'est pas importée dans ident-version-schéma car c n'est pas importée dans ident-version-schéma et c' n'est pas importée dans une autre version du schéma car c n'est pas importée dans d'autres versions du schéma.

Nous utilisons la fonction `ClassesToDel` pour déterminer les classes qui doivent être supprimées dans une version du schéma. Dans la définition de `ClassesToDel`, l'expression `S_Sub(c)` est l'ensemble des sous-classes directes de la classe c dans toutes les versions du schéma. Dans la figure IV.10, d est une classe associée aux versions (s, 1) et (s, 2) du schéma s et e est une classe associée à la version (s, 2) du schéma s; `S_Sub(c)={d, e}`.

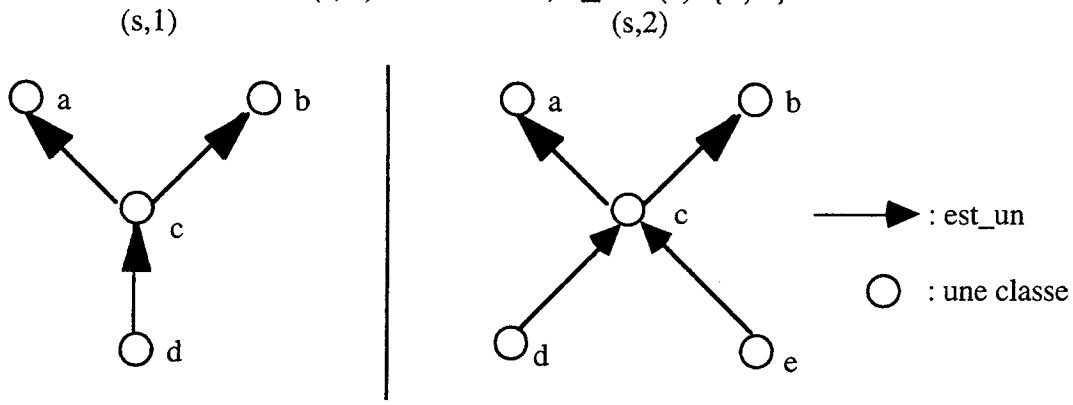


FIG. IV.10 : la position de la classe c dans le schéma s

La fonction `ClassesToDel` est définie comme suit :

`ClassesToDel` : $S \times N \rightarrow 2^{C \times N}$

{`ClassesToDel`(s, n) est l'ensemble des classes qui peuvent être supprimées dans la version n du schéma s.

`ClassesToDel`(s, n) = {c ∈ `SchVerClasses`(s, n) / ((`C_Weight`(c) = 0) ∧ ((`S_Sub`(c) = {})
∨ (∀c' ∈ `S_Sub`(c), c' vérifie les conditions de suppression))}.

}

L'algorithme de l'opération `delete_schema_version` est donné dans la figure IV.11. Pour chaque classe c de l'ensemble des classes qui peuvent être supprimées dans la version du schéma, le traitement suivant est effectué : si c est une feuille dans toutes versions du schéma qui la partagent, alors c est supprimée en utilisant la primitive `delete_class`(c). Sinon, la suppression de c provoque la suppression de ses sous-classes dans toutes les versions du schéma. Ainsi, la suppression d'une classe dans une version du schéma peut provoquer la suppression d'une classe d'une autre version du schéma.

Algorithme DeleteSchemaVersion ((s, n) : S×N))

{ (s, n) vérifie les propriétés de suppression.

1. Supprimer les classes de l'ensemble `ClassesToDel`(s, n).

   `DeleteClasses`(`ClassesToDel`(s, n)) telle que :

   `DeleteClasses`(EC : $2^{C \times N}$)

   /* *`DeleteClasses` supprime l'ensemble donné des classes, et pour chacune toutes ses sous-classes* */

   {Si EC ≠ { } alors

   soit c ∈ EC

   si c n'est pas une feuille (c'est-à-dire, `S_Sub`(c)≠{ }),

   alors `DeleteClasses`(`S_Sub`(c));

   `delete_class`(c);}

2. Supprimer la définition de (s, n).

   Supprimer la version (s, n) de l'ensemble des versions du schéma s.

}

FIG. IV.11 : algorithme DeleteSchemaVersion

IV.4.2. Opération de réorganisation

Dans cette section nous présentons l'opération de *réorganisation immédiate* de la base de données. Cette opération est déclenchée par un utilisateur. Elle supprime les informations (classes ou versions du schéma) qui ne sont plus pertinentes pour les applications :

• Dans le passé, la conservation de ces informations était jugée importante.

• Dans le présent, ces informations ne sont pas utilisées par les applications car la base de données et les applications ont évolué.

• Ces informations ne seront plus utilisées dans les futures applications.

Dans un premier temps, nous décrivons la primitive `reorganise_database` qui est utilisée par l'utilisateur pour déclencher la réorganisation de la base de données. Dans un second temps, nous présentons l'algorithme qui réalise la réorganisation.

***Primitive* `reorganise_database`**

```
reorganise_database
[np : <nombre-de-programmes>]
[nv : <nombre-de-versions-du-schéma>[,<ordre>]]
[cl : <suppression-classes>];
```

Tous les paramètres sont optionnels. A la suite de la réorganisation, les versions historiques du schéma qui sont associées à un nombre de programmes inférieur ou égal à nombre-de-programmes, sont supprimées. La valeur par défaut de ce paramètre est 0.

La valeur du paramètre nombre-de-versions-du-schéma fixe le nombre maximal des versions historiques du schéma que l'utilisateur décide de conserver. A la suite de la réorganisation, le nombre de versions historiques du schéma est inférieur ou égal à la valeur de ce paramètre. La valeur par défaut de ce paramètre est ∞. Le paramètre ordre prend sa valeur dans l'ensemble {"poids", "age"}. Si la valeur de ordre est égal à "age", alors il s'agit de conserver les versions les plus récentes du schéma. Sinon, les versions conservées sont celles dont le poids est plus élevé que les autres. Dans le dernier cas, s'il est nécessaire de choisir entre des versions parce qu'elles ont le même poids, alors les plus récentes sont conservées. La valeur par défaut de ce paramètre est "poids".

Le paramètre suppression-classes prend sa valeur dans l'ensemble {"version", "schéma"}. Si la valeur du paramètre suppression-classes est égal à "version", les classes sont supprimées à partir de la suppression des versions du schéma. Dans ce cas, il est possible que la réorganisation ne provoque pas la suppression de toutes les classes qui vérifient les conditions suppression. Une classe qui a les caractéristiques suivantes, n'est pas supprimée même si elle vérifie les conditions de suppression :

• Elle est associée à une version du schéma qui n'est pas supprimée par la réorganisation.

• Elle n'est pas supprimée à partir de la suppression d'une autre version du schéma. Par exemple, elle est une sous-classe directe de la classe objet (racine des hiérarchies d'héritages).

Si la valeur de suppression-classes est égal à "schéma", alors la réorganisation provoque la suppression de toutes les classes qui vérifient les conditions de suppression. La valeur par défaut de ce paramètre est "version".

Algorithme

L'algorithme de l'opération `reorganise_database` est donné dans la figure IV.12. Il consiste en quatre phases :

Phase 1 (recherche des versions du schéma qui vont être supprimées) : cette phase consiste en deux étapes. Dans la première étape, toutes les versions dont le poids est inférieur ou égal à nombre-de-programmes sont identifiées. Ces versions vont être supprimées.

Dans la deuxième étape, si le nombre de versions qui ne sont pas identifiées dans la première étape est supérieur à nombre-de-versions-du-schéma (nombre maximal de versions que l'utilisateur décide de conserver), alors les versions qui vont être supprimées en plus sont identifiées. Ces versions sont identifiées en se basant sur la valeur du paramètre ordre.

Phase 2 (notification) : les programmes qui sont associés aux versions qui vont être supprimées (c'est-à-dire, celles qui sont identifiées dans la phase précédentes), sont signalés à l'utilisateur. Ils devront être re-compilés afin d'être associés à la version courante du schéma.

Phase 3 (suppression de versions du schéma) : dans cette phase, les versions qui sont identifiées dans la première phase, sont supprimées selon l'ordre croissant de leurs numéros. Une version du schéma est supprimée en utilisant l'opération `delete_schema_version` (Cf. Paragraphe IV.4.1.2).

Phase 4 (suppression de classes) : cette phase a lieu seulement si la valeur du paramètre suppression-classes est égal à "schéma". Elle consiste à supprimer toutes les classes qui vérifient les conditions de suppression et qui ne sont pas supprimées dans la phase précédente. Ces classes sont supprimées en utilisant l'opération `delete_class` (Cf. paragraphe IV.4.1.1).

Algorithme ReorganiseDatabase
(nombre-de-programmes : N, nombre-de-versions-du-schéma : N, ordre : {"poids", "age"}, suppression-classes : {"schéma", "version"})

{ Dans l'algorithme l'expression `MostYoungs`(e, n) est l'ensemble des n versions les plus récentes parmi les versions de l'ensemble e (un ensemble de versions du schéma). `VerSet`, `Current` et `SchVerClasses` sont des primitives définies dans le chapitre III.

1. Déterminer les versions historiques du schéma qui vont être supprimées. Soit VS cet ensemble.
   VS = {s_i ∈ `VerSet`(s) / `S_Weight`(s_i) ≤ nombre – de – programmes};
   / s est le schéma de la base */*

   1.1. Si le nombre de versions historiques du schéma, qui ne sont pas dans VS est supérieur au nombre maximal de versions qui doivent être conservées (c'est-à-dire, `card`(`VerSet`(s) - VS) > nombre-de-versions-du-schéma + 1), alors compléter l'ensemble VS :
   Soit nr le nombre de versions qui vont être ajoutées à VS.
   nr = `card`(`VerSet`(s) - VS) - nombre-de-versions-du-schéma - 1;

   1.1.1. Si ordre="age", alors ajouter à VS l'ensemble des nr versions les plus récentes, parmi les versions historiques du schéma qui ne sont pas dans VS.
   VS=VS∪`MostYoungs`(nr, `VerSet`(s)-({`Current`(s)}∪VS));
   1.1.2. Sinon, ajouter à VS l'ensemble des nr versions qui ont les plus faibles poids, parmi les versions historiques du schéma qui ne sont pas dans VS. Soit S' cet ensemble.
   Initialisation : Sr=`VerSet`(s)-({`Current`(s)}∪VS), S'={};
   Tant que `card`(S') < nr faire
   Sa = {(s, i) ∈ Sr / ∀(s, j) ∈ Sr, `S_Weight`(s, i) ≤ `S_Weight`(s, j)};
   S'=S'∪ Sa;
   Sr=Sr-Sa;
   FinTanque

Si card(S')>nr, alors S'=S'-MostYoungs(card(S') - nr, Sa);
VS=VS∪S';

2. Notifier les programmes qui sont associés aux versions de l'ensemble VS. Ces programmes devront être associés à la version courante du schéma.

3. Supprimer les versions du schéma qui sont dans l'ensemble VS. Ici, nous supposons que VS est ordonné selon l'ordre croissant des numéros de versions.
Pour chaque version s_i dans l'ensemble VS faire
delete_schema_version(s_i);

4. Si suppression-classes = "schéma", alors supprimer toutes les classes qui vérifient les conditions de suppression. Ici, nous supposons que l'ensemble VerSet (s) est ordonné selon l'ordre croissant des numéros des versions.
Pour chaque s_i dans VerSet(s) faire

4.1. Déterminer les classes de s_i, qui peuvent être supprimées. Soit CS cet ensemble. CS=ClassesToDel(SchVerClasses(s_i));

4.2. Supprimer les classes qui peuvent être supprimées en conséquence.
DeleteClasses(CS) /* *cette primitive est définie dans §4.1.2* */

}

FIG. IV.12 : algorithme ReorganiseDatabase

L'utilisateur peut décider de la suppression des versions historiques du schéma s'il juge que ses versions sont associées à un nombre très faible de programmes. Il peut aussi fixer le nombre maximal des versions historiques du schéma.

IV.5. Conclusion et perspectives

Dans ce chapitre, nous avons proposé une nouvelle approche pour gérer l'évolution de schéma. L'idée est de trouver un compromis entre les fonctionnalités des approches existantes pour une meilleure gestion de l'évolution de schéma :

L'extension de la définition d'une évolution soustractive : la définition d'une évolution soustractive n'est pas très précise. Elle porte seulement sur le schéma (attributs, opérations et classes) en laissant de coté les programmes d'applications. Prenons le cas de la suppression d'une opération qui n'est pas utilisée (ni directement ni indirectement) par les programmes. Cette évolution est qualifiée de soustractive, même si elle ne provoque pas la perte d'informations ou l'incompatibilité des programmes. Nous voulons étendre la définition pour détecter de manière plus exacte les évolutions soustractives. Nous envisageons, la prise en compte des dépendances entre les programmes et les interfaces des classes pour affiner la définition.

• *Le processus du changement du schéma combine la modification et le versionnement du schéma.* L'évolution du schéma se traduit par la dérivation d'une nouvelle version seulement si l'évolution est soustractive. L'utilisateur a cependant la possibilité d'imposer le mode de l'évolution (évolution par dérivation d'une version ou par modification du schéma).

• *La technique d'adaptation des instances combine la conversion, l'émulation et le versionnement des objets.* L'idée est de limiter le nombre de versions d'objets à celles qui sont nécessaires et par conséquent, de réduire les coûts de stockage des versions et de maintenance des relations entre les versions. Cette technique utilise :

- Le **versionnement** dans le sens où l'algorithme génère une version stockée si elle est jugée importante.
- l'**émulation** dans le sens où l'algorithme génère une version calculée lorsqu'elle est rarement accédée.
- la **conversion** dans le sens où l'algorithme génère une version stockée ou calculée et supprime la version origine lorsqu'il détecte qu'elle n'est plus utilisée par des applications.

• *Nous avons proposé une opération qui permet la réorganisation immédiate de la base de données.* Lorsque la taille de la base de données est préjudiciable aux performances du système, l'administrateur peut choisir de réorganiser la base de données. La réorganisation supprime les classes et les versions du schéma qui ne sont plus pertinentes (ou jugées non pertinentes) pour les applications. Le choix des valeurs des paramètres de l'opération de réorganisation est à la charge de l'utilisateur. Ainsi, un utilisateur expert peut faire des choix différents selon les besoins des applications.

Les perspectives du travail présenté dans ce chapitre sont :

L'extension de la définition du poids d'une classe : la définition du poids d'une classe prend en compte le statut (courante ou historique) des versions du schéma auxquels est associée la classe et le nombre de programmes associés à la classe. L'extension de cette définition consiste d'une part à l'identification des paramètres qui affectent la pertinence d'une classe, comme la taille des objets, le nombre d'objets de l'extension d'une classe, le degré de référence des objets, la taille des clients, le rapport lecture/écriture des clients, etc. D'autre part la proposition de métriques (fonctions) d'évaluation de la pertinence d'une classe. Nous voulons proposer une fonction de quantification de chaque paramètre vis-à-vis d'une classe et une fonction globale pour mesurer la pertinence de la classe, où cette dernière est une fonction de pondération sur les précédentes.

Chapitre V

LANGAGE DE DESCRIPTION DE LA SÉMANTIQUE D'UNE ÉVOLUTION DE SCHÉMA

Ce chapitre traite le problème de la *réorganisation automatique* de l'instance d'une base de données à la suite de l'évolution de son schéma. En effet, nous proposons un langage de description des relations entre l'état de la base de données avant et après l'évolution.

Au moment du changement du schéma, l'utilisateur peut, à l'aide de ce langage, décrire des relations entre l'état de la base de données avant et après l'évolution. Ceci, en utilisant des *descripteurs de correspondances*. Ces relations décrivent la sémantique de l'évolution du schéma. Elles sont exploitées par le système lors de la réorganisation de la base de données.

Un descripteur de correspondances décrit une relation entre deux versions adjacentes du schéma, nous parlons d'un lien de correspondances. Il consiste en trois éléments :

1. L'identification de la *source* (un ensemble de classes définies dans une version du schéma) et la *cible* (une classe définie dans l'autre version du schéma) du lien de correspondance.

2. L'*expression de localisation* des objets sources qui sont en correspondances avec des objets cibles. Nous avons choisi la syntaxe du langage de requête*OQL* de l'*ODMG*.

3. L'*expression de dépendance* entre la valeur de la version d'un objet cible et les valeurs des versions des objets avec qui l'objet a un lien (une expression booléenne).

L'ensemble des liens de correspondances associés au schéma d'une base de données constitue son schéma de correspondances. Nous discutons de la *correction* d'un schéma de correspondances et de la *cohérence* d'une base de donnés vis-à-vis de son schéma de correspondances.

V.1. Introduction

Comme nous l'avons vu dans le chapitre 2 consacré à l'état de l'art, dans la quasi totalité des systèmes qui supportent l'évolution de schéma, la réorganisation est systématique. Cette réorganisation est basée sur la comparaison de la définition (structure) d'une classe avant et après l'évolution pour générer la séquence de mises à jour à effectuer sur la base de données. La notion de valeur par défaut associée à un type est utilisée, d'où l'appellation *réorganisation par défaut* [LH90]. Celle-ci est restrictive principalement pour les deux raisons suivantes :

• Une évolution de schéma dépend souvent de l'application et les intentions sous-jacentes peuvent varier selon le contexte [FMZ94a]. La réorganisation par défaut dépend du système et n'offre aucune possibilité de prendre en compte les spécificités d'un domaine d'application.

• Les besoins du domaine de génie logiciel et des méthodologies à objets sont à l'origine des opérations d'évolution de schéma dites de haut niveau (on dit aussi opérations multiples) comme par exemple, la fusion de plusieurs classes en une seule, ou l'éclatement d'une classe en plusieurs autres. En génie logiciel, ces opérations favorisent la réutilisation et l'intégration des modules existants pour en développer d'autres [BFK95, Bra93, Cas91, Opd92].

Cependant, une réorganisation *convenable* de la base de données (c'est-à-dire, celle qui reflète le plus possible l'évolution du monde réel) à la suite d'une évolution de haut niveau, demande une analyse très fine des relations entre les classes du schéma avant et après l'évolution [Ler93]. A une ou plusieurs classes avant l'évolution peut correspondre une ou plusieurs classes après l'évolution. La réorganisation par défaut est insuffisante pour capturer ces relations car elle est basée sur la comparaison de la structure de classes ayant les mêmes noms.

Nous avons vu dans le chapitre III, que plusieurs réorganisations pouvaient être associées à une évolution de schéma. La réorganisation par défaut en est une.

Étant donnée une base de données (S, I) cohérente, ce qui est dénoté par : $\delta(I, S) \wedge \psi+(S)$. Étant donnée une évolution U sur S (notée U(S)), telle que : $\psi+(U(S))$. La question abordée dans ce chapitre est de trouver une réorganisation $\Re$ qui tient compte des spécificités du domaine d'application, ou de l'application considérée, et telle que $\delta(\Re(I), U(S))$.

Une telle réorganisation peut être :

• **Manuelle :** l'utilisateur développe le code effectuant la réorganisation, comme toute autre procédure de l'application. Un niveau de complexité supplémentaire est alors introduit dans le développement des applications, dans la mesure où il est nécessaire de manipuler les définitions de classes définies dans des versions différentes du schéma. De plus, le développement ad hoc des procédures associées peut générer des erreurs.

• **Automatique :** le système génère la réorganisation. Même si cette voie paraît séduisante, nous pensons qu'elle n'est pas raisonnable pour les raisons suivantes :
1) les modèles de données à objets dont nous disposons sont plus ou moins pauvres pour représenter la complexité de la sémantique des applications. Par conséquent, il est difficile d'exprimer les relations qui existent entre les classes du schéma avant et après l'évolution.
2) une évolution de schéma dépend du domaine d'application ou de l'application.

• **Assistée :** à partir de la spécification des relations entre les classes du schéma avant et après l'évolution, il est possible de déduire une réorganisation de la base de données. Cela consiste à proposer des outils pour :
1) permettre la description des relations entre les classes du schéma avant et après l'évolution. Ces relations fournissent l'information que le système va utiliser pour réorganiser la base à la suite de l'évolution de schéma.
2) vérifier la correction des relations décrites et de la réorganisation associée.

La solution que nous proposons se place dans la dernière catégorie. Plus précisément, notre objectif peut se décomposer de la manière suivante :

• fournir à l'usager un langage de spécification des relations entre l'état de la base de données avant et après l'évolution.
• définir des règles d'utilisation de ce langage afin de ne pas spécifier de relations contradictoires.

L'ensemble des relations entre deux états de la base de données est structuré en *un schéma de correspondances.*

Nous présentons tout d'abord, des exemples (section V.2), afin d'illustrer le problème posé. Dans un second temps (section V.3), nous présentons le langage de description de la sémantique d'une évolution de schéma. La section V.4 discute de la correction d'un schéma de correspondances et présente la technique utilisée pour la vérification de la correction. La section V.5 discute de la cohérence d'une base de données vis-à-vis d'un schéma de correspondances et présente la technique utilisée pour la maintenance de la cohérence.

V.2. Exemples

Les exemples sont des exemples de réorganisation dans lesquelles on ne tient pas compte des liens entre l'ancien et le nouvel état de la base de données. Nous montrons que la prise en compte de ces liens permet de caractériser l'état de la base de données qui reflète le mieux possible l'évolution de schéma.

Nous utilisons la notation introduite dans le chapitre V.3 pour exprimer une réorganisation. Dans l'énoncé d'un exemple l'expression entre parenthèse indique le mode de l'évolution ("modification" ou "version"). Nous rappelons que S dénote le schéma de la base de données avant l'évolution, (S, I) dénote la base de données avant l'évolution du schéma, EI désigne l'ensemble des liens de références de I, Objects (c) est l'ensemble d'objets associés à la classe c.

1. Remplacement d'un ensemble d'attributs par un autre attribut (modification) :

Supposons que le schéma S contient la classe Enseignant définie par :

```
class Enseignant :inherit Employé tuple (
        LaSpécialité : Spécialité,
        LeGrade : string,
        LesHeuresTd : integer,
        LesHeuresCours : integer)
method
        LesVacations : integer,
        LaPrime_de_recherche : Money
end;
method body LesVacations in class  Enseignant {
        o2 integer LeMaximum = 100;
        if (self->LeGrade=="prof") LeMaximum=2
```

```
        else if (self->LeGrade=="mc") LeMaximum=15;
        return LeMaximum;}
method body LaPrime_de_recherches in class Enseignant {
        o2 Money  LaPrime=6.000 FF;
        if (self->LeGrade=="prof")  LaPrime=30.000 FF;
         else if (self->LeGrade=="mc") LaPrime=20.000 FF;
        return LaPrime;}
```

FIG. V.1 : définition de la classe Enseignant

L'évolution du schéma consiste à remplacer les attributs LesHeuresTd et LesHeuresCours par l'attribut LesHeures de type entier. En l'absence de la description de la relation entre les anciens attributs (LesHeuresTd et LesHeuresCours) et le nouveau (LesHeures), la réorganisation de la base de données proposée ci-dessous, introduit des valeurs par défaut. Le nouvel état de l'instance (I') est bien cohérent avec le nouveau schéma (S'), mais il ne reflète pas forcément l'évolution du monde réel.

$\Re$= let $\Re$1=let

$x = \{(o,0) \xrightarrow{a} o' \in EI \,/\, o \in \texttt{Objects}(\text{Enseignant},0) \wedge (a \in \{\text{LesHeuresTd}, \text{LesHeuresCours}\}\}$

in <for (l∈ x) DelRefLink(EI, l)>;

$\Re$2=<for(o∈ Objects(Enseignant, 0))

InsRefLink(EI, $(o,0) \xrightarrow{\text{LesHeures}} 0$)>

in <$\Re$1; $\Re$2>

Il peut exister une relation entre les anciens attributs et le nouveau. Par exemple, pour chaque objet de la classe Enseignant, la valeur associée à l'attribut LesHeures est la somme de celles qui sont associées aux attributs LesHeuresTd et LesHeuresCours. Cette relation est exprimée dans la réorganisation $\Re'$ décrite ci-dessous :

$\Re'$=let $\Re$3=

<for(o∈ Objects(Enseignant, o))

InsRefLink(EI, $(o, 0) \xrightarrow{\text{LesHeures}} (o, 0).\text{LesHeuresTd} + (o, 0).\text{LesHeuresCours}$)>

in <$\Re$3 ; $\Re$1>

2. Partition d'une classe (version) : Nous reprenons le schéma S dont la classe Enseignant est définie dans la figure V.1. Le code des méthodes associées à la classe Enseignant contient des conditions qui portent sur l'attribut LeGrade. L'évolution du schéma consiste à remplacer la classe Enseignant par les trois classes Professeur, Maître_de_conférences et Ater définies par :

```
class Professeur  inherit Employé tuple(
        LaSpécialité : Spécialité,
        LesHeuresTd : integer,
        LesHeuresCours : integer,
method
        LesVacations : integer,
        LaPrime_de_recherche : Money,
end;

method body LesVacations in class Professeur { return (200);}
method body LaPrime_de_recherche in class Professeur {return (30.000 FF);}

class Maître_de_conférence inherit Employé tuple(
        LaSpécialité : Spécialité,
        LesHeuresTd : integer,
        LesHeuresCours : integer)
method
        LesVacations : integer,
        LaPrime_de_recherche : Money,
end;

method body LesVacations in class Maître_de_conférences {return (150);}
method body LaPrime_de_recherche in class Maître_de_conférences {return (20.000 FF)

class Ater inherit Employé tuple(
        LaSpécialité:Spécialité ,
        LesHeuresTd : integer,
        LesHeuresCours : integer)
```

```
method
        LesVacations : integer,
        LaPrime_de_recherche : Money
end;

method body LesVacations in class Ater {return (100);}
method body LaPrime_de_recherche in class {return (6.000 FF);}
```

FIG. V.2 : définition des classes Professeur, Maître_de_conférences et Ater

Cette évolution se traduit par la dérivation d'une nouvelle version (S, 1) du schéma S dérivée à partir de (S, 0), puis la suppression dans (S, 1) de la classe Enseignant et de l'ajout dans (S, 1) des classes Professeur, Maîtres_de_conférences et Ater.

Plusieurs réorganisations de la base de données peuvent être considérées selon les points de vue suivants :

• **Oubli du passé** :

- les extensions des nouvelles classes sont vides.
- les objets de la classe Enseignant ne sont accessibles qu'à partir des programmes associés à (S, 0).
- les programmes associés à (S, 1) et à toutes les versions futures ne peuvent pas accéder aux objets de la classe Enseignant.

La réorganisation consiste à construire une nouvelle instance sans les objets de la classe Enseignant. C'est ce mécanisme qui est généralement appliqué dans une réorganisation automatique.

• **Mémoire du passé** : les extensions des nouvelles classes contiennent certains objets de la classe Enseignant.

La réorganisation $\Re$ décrite ci-dessous, prend en compte la relation entre la classe Enseignant et les nouvelles classes. Elle consiste à migrer les objets de Enseignant dont la valeur associée à l'attribut LeGrade est égale à "prof" vers la classe Professeur, ceux dont la valeur

associée à l'attribut LeGrade est égale à "mc" vers la classe Maîtres_de_conférences et ceux dont la valeur associée à l'attribut LeGrade est égale à "ater" vers la classe Ater.

```
ℜ =let ℜ1= let prof={o∈ Objects(Enseignant, 0)/o.LeGrade=="prof"};
in <for (o ∈ prof)
      InsVerOid((Professeur,1),o);>
      ℜ2= let mc={o∈ Objects(Enseignant, 0)/ o.LeGrade=="mc"};
      in <for(o∈ mc)
            InsVerOid((Maître_de_conférences, 1),o);>
      ℜ3 = let at={o∈ Objects(Enseignant, 0)/ o.LeGrade=="ater"};
      in <for(o∈ at)
            InsVerOid((ater,1), o);>
in <ℜ1; ℜ2; ℜ3;.>
```

3. Fusion de deux classes en une seule (version) : Nous supposons que le schéma S contient les classes Etudiant et Employé définies par :

```
class Employé :inherit Personne tuple(
        LeNumEmp : integer,
        LeService : Service,
        LeSalaire : Money)
method
        Augmenter_salaire (taux : integer)
end;

class Etudiant inherit Personne tuple(
        LeDépartement : Département,
        LesCours : set(Cours),
        LesHeures : integer)
end;
```

FIG. V.3.a : les classes Employé et Etudiant

L'évolution du schéma consiste en une fusion des classes Employé et Etudiant en une seule classe Etudiant-Employé définie par :

```
class Etudiant-Employé inherit Personne tuple (
        LeNumEmp : integer,
        LeService: Service,
        LeSalaire : Money,
        LeDépartement : Département,
        LesCours : set(Cours),
        LesHeures : integer)
method
        Augmenter_salaire (taux : integer)
end;
```

FIG. V.3.b : la classe Etudiant-Employé

Cette évolution se traduit par la dérivation d'une nouvelle version du schéma, notée (S, 2). A la suite de cette évolution, l'instance de la base peut avoir évolué selon les deux points de vue :

• **Oubli du passé** : l'extension de la nouvelle classe est vide. Dans ce cas, même les objets qui représentaient en réalité des étudiants qui sont aussi des employés ne sont pas visibles comme des objets de Etudiant-Employé dans tous les futurs programmes.

• **Mémoire du passé** : l'extension de la nouvelle classe contient certains objets qui sont construits par la fusion de deux objets appartenant respectivement à (Employé, 1) et (Etudiant, 1) et représentant en réalité des étudiants-employés. Le problème posé est la localisation de ces deux objets puisqu'ils ont des identificateurs différents dans les deux classes.

4. Division d'une classe en deux classes (modification) : Nous supposons que le schéma S contient la classe Cours définie par :

```
class Cours tuple (
        LaDescription : string,
        LeResponsable;: string,
        LAdresse : Adresse,
        LeTéléphone;: Phone,
        LeGrade : Grade)
end;
```

FIG. V.4 : la classe Cours avant la division

La classe Cours contient les informations sur les cours enseignés dans l'université et sur les professeurs responsables de ces cours. Par conséquent, les méthodes associées à cette classe manipulent à la fois des cours et des professeurs. De plus, les informations sur un seul professeur peuvent être dupliquées dans plusieurs cours (autant de fois que le professeur enseigne de cours). L'évolution du schéma consiste à diviser la classe Cours en deux classes Professeur et Cours (Cf. Figure V.5).

```
class Professeur tuple (
        LeNom : string ,
        LAdresse : Adresse,
        Téléphone : Phone,
        LeGrade : Grade,
        LesCours : set(Cours))
end;

class Cours tuple( /* la nouvelle définition de la classe Cours */
        LaDescription : string,
        LeResponsable : Professeur)
end;
```

FIG. V.5 : les classes Professeur et Cours

Selon la réorganisation choisie, des informations de la base de données peuvent être détruites. Par exemple, le calcul de la nouvelle extension de Cours à partir de son ancienne extension en utilisant la réorganisation $\Re$ décrite ci-dessous, génère un nouvel état de la base de données dans lequel les informations sur le responsable sont perdues :

$\Re$ =let

L1= $\{(o, 2) \xrightarrow{a} o' \in EI \,/\, o \in Objects(Cours) \wedge (a \in \{LeResponsable, LAdresse, Téléphone, LeGrade\}\}$;

L2 = $\{(o, 2) \xrightarrow{a} o' \in EI \,/\, o \in Objets(Cours) \wedge a == LeResponsable\}$;

```
in <{for (l ∈ L1) DelRefLink(EI,l);
      for (l ∈ L2) ModRefLink(l,nil);}>
```

Ici, le calcul de la nouvelle extension de Cours n'est possible qu'après celui de l'extension de Professeur. Cet exemple illustre l'importance de l'opération de séquence dans l'expression d'une réorganisation.

5. Échange de propriétés entre deux classes (modification) : Dans la conception des applications à objets, en plus de la modélisation par héritage, la modélisation par agrégation est utilisée [Opd92]. Dans la terminologie objet, un agrégat est un objet qui est composé d'autres objets appelés composants. Un composant peut être ou non partagé par plusieurs agrégats. Une évolution possible du schéma est l'échange des propriétés entre les agrégats et les composants. Par exemple, l'évolution du schéma consiste à déplacer l'attribut LaVille de la classe Adresse vers la classe Professeur :

```
class Adresse tuple(
        LaVille : Ville,
        LaRue : string,
        LeNuméro : int,
        LePays : Pays)
end;
```

FIG. V.6 : la classe Adresse avant l'évolution

```
class Professeur tuple(
        LeNom : string,
        LAddresse : Adresse,
        LeTéléphone : Phone,
        LeGrade : Grade,
        LesCours : set(Cours),
        LaVille : Ville )

class Adresse tuple(
        LaRue : string,
        LeNuméro : integer,
        LePays : Pays)
```

FIG. V.7 : les classes Professeur et Adresse après l'évolution

La traduction de cette évolution par la suppression de l'attribut LaVille de la classe Adresse et l'ajout de cet attribut à la classe Professeur (Cf. la réorganisation $\Re$) entraîne une perte d'informations (les valeurs associées à l'attribut LaVille).

$\Re$ = let L1 = $\{(o, 2) \xrightarrow{a} o' \in EI \,/\, o \in$ Objects(Adresse, 2) $\wedge$ a == LaVille$\}$
in <{for (l $\in$ L1) DelRefLink (EI, l);
for (o $\in$ Objects((Professeur, 2))
InsRefLink(EI, ((o, 2) $\xrightarrow{\text{LaVille}}$ nil)) ;}>

Il est peut-être important de restituer dans le nouvel état de la base de données (I') les valeurs qui étaient associées à l'attribut LaVille dans l'ancien état de la base de données (I). Par exemple, $\Re'$ est une réorganisation qui tient compte de cette contrainte :

$\Re'$ =let L1 = $\{(o, 2) \xrightarrow{a} o' \in EI \,/\, o \in$ Objects((Adresse, 2)) $\wedge$ a == LaVille$\}$
in <{for (o $\in$ Objects(Professeur, 2))
InsRefLink(EI, ((o, 2) $\xrightarrow{\text{LaVille}}$ (o, 2).LAdresse.LaVille));
for (l $\in$ L1) DelRefLink (EI, l);}>

Les exemples présentés ici illustrent des situations où plusieurs réorganisations sont possibles à la suite d'une évolution de schéma. Nous allons montrer dans la suite qu'en se basant sur la spécification de la relation entre les classes avant et après l'évolution, il est possible de choisir une réorganisation parmi toutes celles disponibles.

V.3. Description de la sémantique de l'évolution de schéma

Notre approche est basée sur la description des liens de correspondances entre l'état de la base de données avant l'évolution du schéma et celui après l'évolution. Ces liens constituent l'information que le système va utiliser pour réorganiser la base de données. Dans cette section, nous présentons le processus de description d'un lien de correspondance. La section V.3.1 présente le principe de base et introduit le vocabulaire utilisé. La section V.3.2 présente la construction que nous introduisons pour décrire les liens, nous parlons du *descripteur de correspondances*.

V.3.1. Principe de base

Le but est de décrire les liens qui existent entre les classes de deux versions adjacentes d'un schéma. Notre approche est basée sur la description d'un lien qui existe entre une classe (*cible*) définie dans l'une des versions du schéma et un ensemble de classes (*source*) définies dans l'autre version du schéma.

Exemple. Dans la figure V.5.8, LC_1 est un lien de correspondance entre la classe cible (c_1, m) et la classe source (c_1, n). Ce cas correspond à la description d'un lien de correspondance entre deux définitions de la même classe dans deux versions différentes du schéma. LC_2 est le lien de correspondance entre la classe cible (c_4, m) et la classe source (c_2, n). Ce cas correspond à la description d'une relation de correspondance entre deux classes différentes définies dans deux versions différentes du schéma. LC_3 est le lien de correspondance entre la classe cible (c_5, m) et les classes sources (c_2, n), (c_3, n), ..., (c_i, n). Ce cas correspond à la description d'une relation de correspondance entre une classe définie dans une version du schéma et plusieurs classes définies dans l'autre version du schéma.

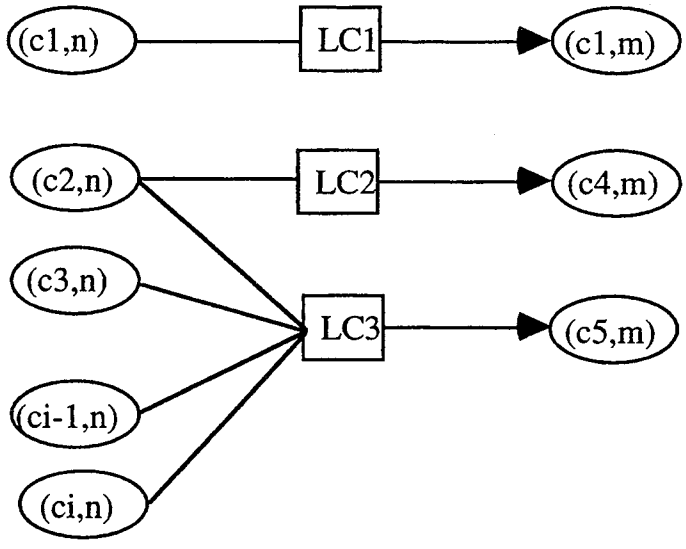


FIG. V.8 : liens de correspondances

• La description d'un lien entre un ensemble de classes définies dans une des versions du schéma et un ensemble de classes définies dans l'autre version du schéma est traduite par la description des liens entre le premier ensemble de classes et chacune des classes du deuxième ensemble.

• La description des liens entre des classes définies dans la même version d'un schéma est ignorée car c'est plutôt un problème d'intégrité qu'un problème d'évolution de schéma.

L'expression d'un lien de correspondance est composée de :

1. L'identification de la source et de la cible.

2. Une expression de *localisation* : c'est-à-dire, la localisation des objets des classes sources qui ont un lien avec un objet donné de la cible. Ainsi, un lien de correspondance entre l'objet de la classe cible et les objets de classes sources.

3. Une expression de *dépendance* : c'est-à-dire, la description des relations entre la valeur de la version d'un objet de la classe cible et les valeurs des versions des objets des classes sources, qui lui sont liés par un lien de correspondance.

Pour illustrer l'expression d'un lien de correspondance, nous considérons les deux versions (S, 0) et (S, 1) du schéma S. La version (S, 0) contient la classe ProgrammeC, dont les objets sont des programmes écrits en langage C et la classe ProgrammeC++, dont les objets sont des programmes écrits en C++. La version (S, 1) contient la classe Programme dont les objets sont des programmes. La classe Programme contient un attribut LeLangage dont la valeur est une chaîne de caractères par exemple, "C++". La définition des autres attributs est ignorée ici.

La cible et la source

La cible est la classe Programme et la source est l'ensemble {ProgrammeC, ProgrammeC++}.

Expression de localisation

Cette expression consiste à localiser les objets des classes de la source qui ont un lien de correspondance avec un objet donné de la classe cible. Elle constitue l'information sur la manière d'initialiser ou modifier l'extension de la classe cible. Dans cette partie seuls les identificateurs des objets sont mis en relation. A un identificateur d'un objet de la classe cible sont associés un ou plusieurs identificateurs d'objets de classes sources. Dans l'exemple, à chaque objet de la classe

ProgrammeC ou de la classe ProgrammeC++ correspond un objet de la classe Programme. Dans ce cas simple, l'expression de correspondance décrit le fait que les objets des classes ProgrammeC et ProgrammeC++ sont aussi des objets de la classe Programme (Cf. Figure V.9).

Expression de dépendance

Cette expression décrit la relation qui existe entre la valeur de la version d'un objet sous la classe cible et les valeurs des versions des objets qui lui sont liés par un lien de correspondance sous les classes sources. Elle constitue l'information sur la manière d'initialiser ou de modifier la valeur de la version d'un objet sous la classe cible. Dans l'exemple, la valeur associée à l'attribut LeLangage de la version d'un objet sous la classe Programme est égale au nom de la classe de l'objet de la classe source (ProgrammeC ou ProgrammeC++) qui est lié avec cet objet par lien de correspondance. La valeur de l'attribut LeLangage dans la version de l'objet c1 (Cf. Figure V.9) sous la classe Programme est égale à "ProgrammeC", qui est le nom de la classe source ProgrammeC (la classe de l'objet c1).

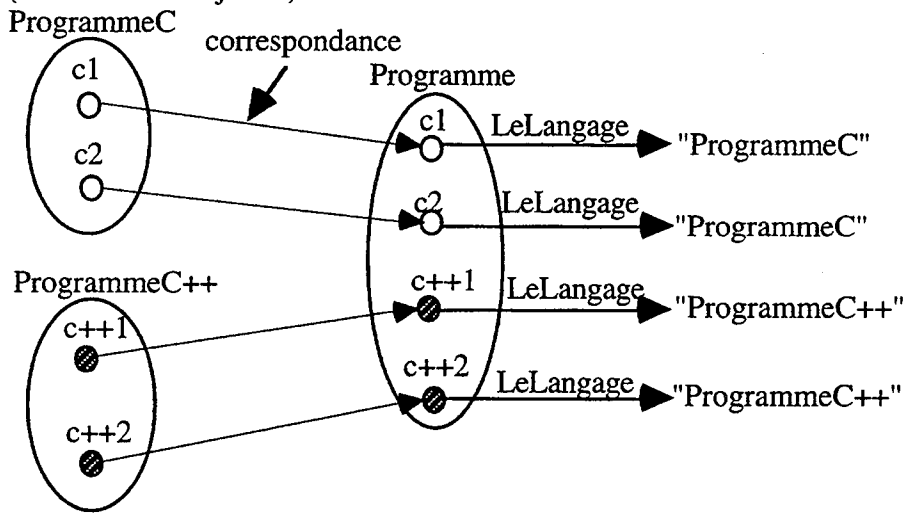


FIG. V.9 : un lien de correspondance

Le lien de correspondance décrit dans cet exemple, constitue l'information que le système va utiliser pour réorganiser la base de données à la suite de la fusion des deux classes ProgrammeC et ProgrammeC++ en la classe Programme.

A la suite de cette évolution, l'instance de la base de données évolue de la manière suivante :

1) l'extension de la classe Programme est initialisée par la dérivation d'une nouvelle version sous la classe Programme de chacun des objets des classes ProgrammeC et ProgrammeC++,
2) la valeur associée à l'attribut LeLangage dans une nouvelle version est initialisée par le nom de la classe de sa version racine.

V.3.2. Descripteur d'un lien de correspondance

Un lien de correspondance entre deux versions adjacentes d'un schéma exprime une relation entre les deux instances associées. Par exemple, à chaque objet d'une classe, qui est définie dans une version du schéma, sont associés deux objets, chacun d'une classe différente dans l'autre version du schéma.

Un lien de correspondance est spécifié par ce que nous appelons un *descripteur de correspondance*. Un descripteur de correspondance est spécifié en utilisant la grammaire[5.1] décrite dans la figure V.10. Les mots en gras sont les terminaux et les autres sont les non-terminaux :

```
source {<nom_classe>+} target <nom_classe>
[object_correspondencies
      localise <expression de selection d'objets sources>
      link <ident_objet_cible>]
[attribute_dependencies
      {(<nom_attribut><spécification_attribut>)+}]
<spécification_attribut>::= imported <attribut>|
                            derived [{<attribut>*}] with <corps_fonction>|
                            dependent {<attribut>+}
                            new with <corps_fonction>]
<attribut>::=<nom_attribut> [<cnom_classe>]
<nom_attribut> ::=string
<nom_classe> ::=string
<corps_fonction>::=string
<ident_objet_cible>::same_as_source [<nom_classe>] | new
```

FIG. V.10 : syntaxe d'un descripteur de correspondance

[5.1] L'étude de la complétude du langage sort du cadre de ce travail.

Exemple 1. Le descripteur décrit dans la figure V.11, spécifie le lien de correspondance associé à l'exemple décrit plus haut. Dans le descripteur, l'expression C_Name(c) est le nom de la classe c, l'expression Classes(o) est l'ensemble des classes auxquelles est associé l'objet o et l'expression Target_Link(o) associe à l'objet o de la classe cible l'identificateur d'objet de la classe source auquel il est lié par la clause **object-correspondencies**. Le mot-clé **self** désigne l'objet en cours d'utilisation.

```
source {ProgrammeC, ProgrammeC++} target Programme
object_correspondencies
      localise Objects(ProgrammeC) union Objects (ProgrammeC++)
      link same_as_source
attribute_dependencies
      LeLangage derived with C_Name (element(select c
                                 from c in {"ProgrammeC", "ProgrammeC++"}
                                 where c in Classes(Target_Link(self))
```

FIG. V.11 : un descripteur de correspondance

La clause **source** introduit l'ensemble des classes sources ({ProgrammeC, ProgrammeC++}). La clause **target** introduit la classe cible (Programme).

La clause **object_correspondencies** introduit l'expression de localisation. Le mot-clé **localise** introduit l'expression de sélection des identificateurs d'objets des classes ProgrammeC et ProgrammeC++, qui ont un lien avec des objets de la classe Programme. Ici, il s'agit de l'union des identificateurs des objets de ProgrammeC et des identificateurs des objets de ProgrammeC++. Le mot-clé **link** introduit l'expression qui établit la liaison entre un identificateur d'un objet donné de Programme et des identificateurs d'objets de l'ensemble Objects (ProgrammeC) union Objets(ProgrammeC++). Le mot-clé same_as_source précise qu'à chaque objet source (objet de ProgrammeC ou de ProgrammeC++) correspond un objet de Programme (c'est-à-dire, une correspondance un-à-un) qui a le même identificateur que son correspondant. Ainsi, un objet de ProgrammeC ou de ProgrammeC++ est aussi un objet de Programme.

La clause **attribute_dependencies** introduit l'expression de dépendance. Cette expression décrit le fait que la valeur de l'attribut LeLangage de la version d'un objet de la classe Programme, qui est un objet de ProgrammeC ou de ProgrammeC++, est égale au nom de la classe source de l'objet (c'est-à-dire, "ProgrammeC" ou "ProgrammeC++").

Le système utilise ce descripteur pour :

1) Initialiser l'extension de la classe Programme :

- Pour chaque objet de `Objects`(ProgrammeC) union `Objects`(ProgrammeC++), une nouvelle version sous la classe Programme est dérivée.
- La valeur de l'attribut LeLangage de la nouvelle version est initialisée par le nom de la classe source de l'objet.

Pour dériver une nouvelle version de l'objet o de la classe c (ProgrammeC ou ProgrammeC++), sous la classe Programme, le système utilise la primitive :

`Object_Version_Derivation`(c, Programme, o)
`with` LeLangage `C_Name`(c)

2) Mettre à jour l'extension de la classe Programme. Quand, une nouvelle version d'un objet o est créée sous la classe ProgrammeC ou ProgrammeC++, une autre version de o est créée sous la classe Programme. Cette dernière est dérivée à partir de la première en utilisant la primitive décrite ci-dessus.

Dans la suite de cette section, nous détaillons les clauses d'un descripteur de correspondances.

V.3.2.1. Clause source et cible

Les classes données dans la clause **source** sont toutes définies dans la même version du schéma. La classe de la clause **target** est définie dans une autre version du schéma. Les deux versions sont adjacentes.

V.3.2.2 Clause `object correspondencies`

La clause **`object_correspondencies`** introduit la correspondance entre les identificateurs des objets de la classe cible et les identificateurs des objets des classes de la source.

Le mot-clé **`localise`** introduit l'*expression de sélection* des identificateurs d'objets des classes sources qui seront mis en correspondance avec un identificateur d'objet de la classe cible. Cette expression est déclarative. Il s'agit d'une requête *OQL*, le langage de requête du standard *ODMG* [Cat93]. Le contexte de cette requête est composé des objets des classes sources. La requête associée au lien de correspondance de l'exemple précèdent est :

`localise` `Objects` (ProgrammeC) union `Objects`(ProgrammeC++)

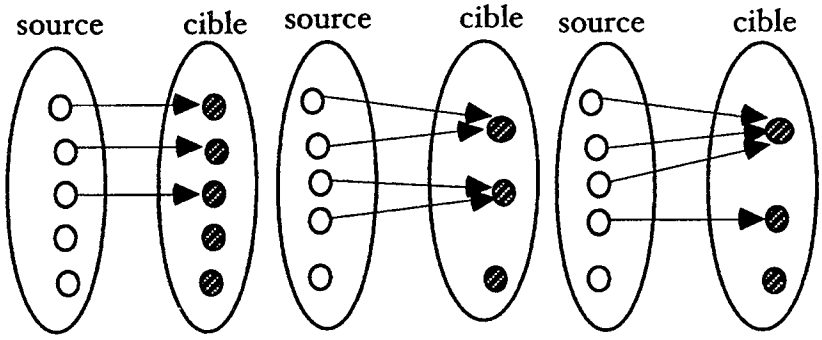


FIG. V.12 : correspondance entre objets

Le mot-clé **`link`** établit un lien entre un identificateur d'objet de la classe cible et des identificateurs d'objets des classes sources (Cf. Figure V.12). Cette liaison associe à l'identificateur de chaque objet de la classe cible des identificateurs d'objets des classes sources.

Cette expression est utilisée par le système pour :
- créer ou ajouter des identificateurs d'objets cibles,
- établir les liaisons entre ces identificateurs d'objets et des identificateurs d'objets sources.

Dans la suite, étant donné un descripteur de correspondance, l'expression `Target_Link(o)` est l'ensemble des identificateurs des objets avec qui l'objet o de la classe cible du descripteur est en correspondance.

Nous considérons deux types de liaison :

• *L'identificateur de l'objet cible est nouveau :* dans ce cas, le mot-clé **`link`** est suivi par **`new`**. Par exemple, si le résultat de l'expression de sélection est un ensemble de couples d'identificateurs d'objets, alors pour chaque couple, un nouvel identificateur de la classe cible est créé. Ainsi, une liaison est établie entre le couple d'objets sources et l'objet cible.

• *L'identificateur de l'objet cible est le même que l'identificateur d'un objet source :* dans ce cas le mot-clé **link** est suivi par **`same_as_source`** s'il s'agit d'une liaison entre un identificateur d'objet source et un identificateur d'objet cible (voir, l'exemple précédent). Dans le cas où la liaison existe entre plusieurs identificateurs d'objets chacun associé à une classe source différente (de celles auxquelles sont associés les autres identificateurs d'objets), le mot-clé **link** est suivi par **`same_as_source`** **nom_classe** où nom_classe est le nom de la classe source à laquelle est associé l'identificateur de l'objet cible.

Nous pouvons dire que, avant la définition de l'expression d'établissement des liens (c'est-à-dire, **`link`** <ident_objet_cible>), le descripteur n'a fait que localiser les identificateurs d'objets des classes sources qui seront mis en correspondance avec des identificateurs d'objets de la classe cible.

Exemple 2. Nous supposons que le schéma S a deux versions (S, 0) et (S, 1). La version (S, 0) contient les deux classes Etudiant et Employé. La version (S, 1) contient la classe Etudiant-Employé dont la définition est obtenue par la fusion des classes Etudiant et Employé. La sémantique que nous voulons associer à l'opération de fusion est la suivante : certains objets de la classe Etudiant-Employé sont construits par la fusion des couples d'objets des classes Etudiant et Employé qui ont le même nom.

Le lien de correspondance associé est spécifié par le descripteur de la figure V.13. Le résultat de l'expression de sélection est un ensemble de couples d'identificateurs. Ces couples sont formés par une jointure sur `Objects(Etudiant)` et `Objects(Employe)`. Deux identificateurs d'objets forment un couple si les objets ont le même nom. Pour chaque couple de l'ensemble fourni par l'expression de sélection, un nouvel identificateur de la classe Etudiant-Employé est créé.

```
source {Etudiant, Employé} target Etudiant-Employé
object_correspondencies
      localise select struct (id1 : x, id2 : y)
            from x in Objects(Etudiant), y in Objects(Employé)
            where x.LeNom=y.LeNom
      link new
```

FIG. V.13 : descripteur de la fusion des classes Employé et Etudiant

Le système utilise ce descripteur pour initialiser l'extension de la classe Etudiant-Employé. Pour chaque couple (x, y) de l'ensemble fourni par l'expression de sélection, un objet o de la classe Etudiant-Employé est créé. Le système matérialise la liaison entre (x, y) et o (c'est-à-dire, `Target_Link` (o)={x, y})

Exemple 3. Nous reprenons l'exemple 4 de la section V.2. Nous décrivons un lien de correspondances entre la classe Cours définie dans la version (S, 0) du schéma et la classe Professeur définie dans la version (S, 1) du schéma (Cf. Figure V.14). Dans cet exemple, nous ignorons l'expression de dépendance.

Le résultat de l'expression de localisation est une partition. Les ensembles de la partition contiennentt les identificateurs des objets de la classe Cours qui sont enseignés par un même professeur. Pour chaque ensemble ens de la partition, un objet o de la classe (Professeur, 1) est créé. Une liaison est établie entre ens et o (c'est-à-dire, `Link_Target`(o)=ens).

```
source { Cours } target Professeur
object_correspondencies
      localise select x.partition
            from x in group y in Objects (Cours)
            by (LeResponsable: y.LeResponsable)
      link new
```

FIG. V.14 : descripteur de correspondances entre (Cours, 0) et (Professeur, 1)

V.3.2.3. Clause `attribute dependencies`

La clause **`attribute_dependencies`** exprime la relation qui existe entre les attributs de la classe cible et ceux des classes sources. Pour un attribut donné, la spécification contient l'information sur la manière d'initialiser ou de modifier la valeur associée dans la version sous la classe cible d'un objet qui a un lien de correspondance avec des objets des classes sources. Ainsi, pour chaque objet de la classe cible, le système peut affecter à un ou plusieurs attributs une valeur calculée en fonction des valeurs des objets sources qui lui sont associés.

Dans un descripteur, un attribut est associé à une seule spécification. Un mot-clé précise la nature de la dépendance. Nous considérons quatre catégories de dépendance [Cla92] :

• ***Importé*** : un attribut importé est une copie d'un attribut défini dans une classe source. Il est introduit par le mot-clé **`imported`**, suivi du nom de l'attribut et du nom de sa classe (ce dernier peut être omis si il n'y a pas d'ambiguïté). Une ambiguïté est due à l'existence d'attributs de même nom dans des classes différentes de la source. La valeur associée à cet attribut dans la version sous la classe cible d'un objet est la même que celle associée à cet attribut dans la version de l'objet, avec qui il a une correspondance, sous la classe source.

• ***Dérivé*** : un attribut dérivé est un attribut dont la valeur est calculée en utilisant une fonction sur les objets des classes sources. Il est introduit par le mot-clé **`derived`**, éventuellement suivi par les noms des attributs des classes sources. Le mot-clé **`with`** introduit le corps de la fonction de calcul. L'attribut LeLangage de la classe Programme (Cf. Exemple 1 de la section V.3) est dérivé :

```
LeLangage derived with C_Name (element(select c
                               from c in {ProgrammeC, ProgrammeC++}
                               where c in Classes(Target_Link(self))))
```

FIG. V.15 : attribut dérivé

• ***Dépendant*** : un attribut dépendant est un attribut dont la valeur est affectée par la mise à jour des valeurs d'autres attributs définis dans des classes sources mais dont la valeur ne peut pas être calculée à partir de celles des attributs. Un attribut dépendant est introduit par le mot-clé **`dependent`**, suivi par la liste des attributs des classes cibles dont il est dépendant.

La spécification des attributs dépendants est importante dans le sens où le système peut détecter des incohérences d'un objet cible, liées à la mise à jour des objets sources. La maintenance de la cohérence de la base de données vis-à-vis des descripteurs de correspondance de la base sera discutée dans la section V.5.

Pour illustrer, nous reprenons l'exemple 1 de la section V.2. La classe (Enseignant, 0) contient les attributs LesHeuresTd et LesHeuresCours et la classe (Enseignant, 1) contient l'attribut LesHeures. Les attributs LesHeuresTd et HeuresCours sont dépendants de l'attribut LesHeures. Supposons qu'au moment de l'évolution qui consiste à remplacer les attributs LesHeuresTd et LesHeuresCours par l'attribut LesHeures, l'utilisateur n'ait pas spécifié comment calculer les valeurs de LesHeuresTd et LesHeursCours à partir de celle de LesHeures.

Supposons que la base de données contient l'objet e, tel que :

- A l'instant t_1, $(e,1) \xrightarrow{\text{LesHeures}} 120$, $(e,0) \xrightarrow{\text{LesHeuresCours}} 60$ et $(e,0) \xrightarrow{\text{LesHeuresTd}} 60$
- A l'instant t_2 ($t_2 > t_1$), la valeur de l'attribut LesHeures (e, 1) est modifiée pour devenir 100, c'est-à-dire $(e,1) \xrightarrow{\text{LesHeures}} 100$.

A l'instant t_2, le système ne provoque aucune action de mise à jour de (e, 0). Portant il est nécessaire de modifier la valeur de LesHeuresTd et/ou LesHeuresCours dans (e, 0) pour que la somme des valeurs de LesHeuresTd et de LesHeuresCours dans (e, 0) soit égal à la valeur de LesHeures dans (e, 1) ($100 \neq 60+60$).

Le descripteur décrit ci-dessous (Cf. Figure V.16), en déclarant LesHeuresTd et LesHeuresCours dépendants de LesHeures, permet au système de réagir à la suite de la mise de la valeur de LesHeures dans (e, 1). Par exemple, le système informe l'utilisateur afin qu'il modifie la valeur de l'attribut LesHeuresTd et/ou LesHeuresCours dans (e, 0).

```
source {(Enseignant, 1)} target (Enseignant, 0)
attribute_dependencies
      LesHeuresTd dependent {LesHeures}
      LesHeuresCours dependent {LesHeures}
```

FIG. V.16 : attributs dépendants

• ***Indépendant*** : un attribut indépendant est un attribut qui n'est ni importé, ni dérivé, ni dépendant. Par exemple, un nouvel attribut de la classe cible qui n'a aucune relation avec les attributs des classes sources. Il est introduit par le mot-clé **new**. Une fonction peut être associée à la spécification d'un attribut indépendant afin d'initialiser sa valeur au moment de la création d'un objet. Le corps de cette fonction est introduit par le mot-clé **with**.

V.3.2.4. Exemples

Le but ici est de montrer l'utilisation des descripteurs de correspondance pour décrire la sémantique d'une évolution du schéma. Dans les exemples, la spécification d'un attribut importé ou indépendant sera donnée seulement si nécessaire : (1) s'il y a une ambiguïté pour un attribut importé, (2) si une fonction est associée à un attribut indépendant.

1. Partition d'une classe (Cf. Section V.2, exemple 2) : les trois descripteurs suivants sont associés à cette évolution.

```
source Enseignant target Professeur
object_correspondencies
      localise  select x
            from x in Objects(Enseignant)
            where x.LeGrade=="prof"
      link  same_as_source

source Enseignant target Maître_de_conférences
object_correspondencies
      localise  select x
            from x in Objects(Enseignant)
            where x.LeGrade=="mc"
      link  same_as_source

source Enseignant target Ater
Object_correspondencies
      localise  select x
```

```
            from x in Objects(Enseignant)
            where x.LeGrade=="ater"
        link same_as_source
```

2. Échange de propriétés entre deux classes (Cf. Section V.2, exemple 5) : le mot-clé **old** désigne la définition de la classe avant l'évolution.

```
source {Professeur.old, Adresse.old} target Professeur
object_correspondencies
    localise Objects(Professeur.old)
    link same_as_source
attribute_dependencies
    LaVille derived
    with Target_Link(self).LAdresse.LaVille
```

V.4. Schéma de correspondances

L'ensemble de tous les liens de correspondance associés à un schéma constitue le *Schéma de Description de Correspondances (SDC).* Dans cette section nous discutons de la correction des descripteurs de correspondances d'un SDC, en particulier, celle des expressions de dépendances.

Les expressions de dépendances d'un SDC sont incorrectes si elles peuvent introduire des contradictions, dans le sens où la satisfaction d'une expression en viole une autre.

Pour vérifier la correction des expressions de dépendances d'un SDC, celui-ci est transformé en un graphe, appelé *graphe de dépendance* entre les attributs. Dans un premier temps (section V.4.1), nous donnons la définition d'un graphe de dépendances. Dans un second temps nous discutons de la correction des expressions de dépendances d'un SDC en étudiant les propriétés du graphe de dépendances associé. La correction des expressions de dépendance d'un SDC est basée sur l'analyse du graphe associé. Elle consiste à détecter les descripteurs en *conflit* et les descripteurs *bilatéraux* qui introduisent des contradictions, chaque fois qu'un nouveau descripteur de correspondance est ajouté.

V.4.1 Graphe de dépendances

A un attribut dérivé est associée une fonction, qui peut être définie par :
1) une expression sur un ensemble d'attributs des classes sources. Si a_i est un attribut qui est dérivé à partir d'un ensemble d'attributs $\{b_1, b_2,..., b_n\}$, alors la relation entre a_i et $b_1, b_2, ..., b_n$ est représentée en utilisant une fonction f_i telle que $a_i=f_i(b_1, b_2, ..., b_n)$. Par exemple, l'attribut LesHeures est dérivé à partir des attributs LesHeuresTd et LesHeuresCours en utilisant l'addition (opérateur défini sur l'ensemble des nombres réels) :

LesHeures=LesHeuresTd + LesHeuresCours

2) une expression sur un ensemble d'attributs des classes sources et/ou une expression globale sur des classes sources. Une expression globale sur des classes sources peut être une requête sur les objets du catalogue de la base de données (par exemple une classe). Si a_i est un attribut dérivé à partir d'un ensemble $\{b_1, b_2, ..., b_n\}$ d'attributs et un ensemble $\{c_1, c_2, ..., c_m\}$ de classes, alors la relation entre a_i et $b_1, ..., b_n, c_1, ..., c_m$ est représentée en utilisant une fonction $ai=fi(b_1, ..., b_n, c_1, ..., c_m)$. Par exemple, l'attribut LeLangage est dérivée en utilisant une requête sur les classes ProgrammeC et ProgrammeC++.

Dans la suite F dénote l'ensemble des fonctions de dérivation.

Définition. Un graphe de dépendances G est un triplet (NO, E, fon) tel que :

• NO est un ensemble de noeuds chacun associé à un ensemble d'attributs. Le nom d'un attribut est préfixé par le nom de la classe où il est défini.

• E est un ensemble d'arcs orientés $(u, v) \in NO \times NO$. Un arc (u, v) indique qu'au moins un des attributs de v est dérivé à partir de certains des attributs de u.

• derive est une fonction définie sur G, telle que :

derive : $NO \times NO \rightarrow F$

{derive(u, v) est l'ensemble des fonctions de dérivation des attributs de la classe v à partir des attributs de la classe u.}. Δ

Un SDC est transformé en un graphe de dépendances en suivant les règles suivantes :

1. Soit $a_i=f(b_1, b_2, ..., b_n)$ une fonction de dérivation. L'ensemble des attributs $\{b_1, ..., b_n\}$ est représenté dans un noeud, u, et l'ensemble $\{a_i\}$ dans un autre noeud, v. Un arc orienté de u vers v est créé.

2. Soit $a_i=f(b_1, ..., b_n, c_1, ..., c_m)$ une fonction de dérivation. L'ensemble $\{b_1, ..., b_n, (*, c_1), ..., (*, c_m)\}$ est représenté dans un noeud, u et l'ensemble $\{a_i\}$ dans un autre noeud, v. Un arc orienté de u vers v est créé. (*, c) est l'ensemble de tous les attributs de la classe c.

3. Soient $\{u_1, u_2, ..., u_p\} \subset NO$ un ensemble de noeuds de G, $v \in NO$ un noeud de G, tel que $v \neq u_i$ $(1 \leq i \leq p)$. Soient $a_{u1}, a_{u2}, ..., a_{up}, a_v$ les ensembles d'attributs respectivement associés à $u_1, u_2, ..., u_p, v$:

- Si $\bigcup_{1 \leq i \leq p} a_{ui} = a_v$, alors les noeuds $u_1, u_2, ..., u_p$ sont fusionnés dans le noeud v. Cette règle élimine les noeuds redondants ; c'est-à-dire les noeuds dont l'ensemble des attributs est inclus dans l'ensemble des attributs associés à un autre noeud du graphe.

Par exemple, le graphe de la figure V.17 est le graphe de dépendances qui représente le descripteur de correspondance donné dans l'exemple 1 de la section V.3.2.

```
f = C_Name (element(select c
            from c in {ProgrammeC, ProgrammeC++}
            where c in Classes(Target_Link(self))))
```

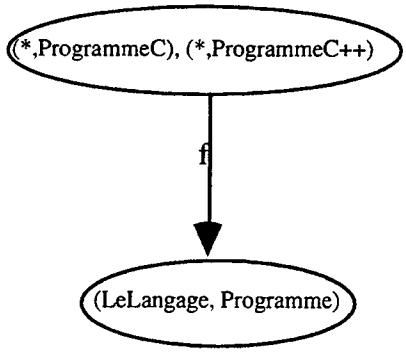


FIG. V.17 : graphe de dépendances

V.4.2. Correction

Dans cette section, nous identifions deux cas où des expressions de dépendance peuvent être contradictoires :

• la présence de descripteurs de correspondances qui ont la même cible (descripteurs en conflit). Ce cas est traité dans le paragraphe V.4.2.1.

• la présence de descripteurs de correspondances bilatéraux. Deux descripteurs sont bilatéraux si la cible de l'un des deux est une source de l'autre et inversement. Ce cas est traité dans le paragraphe V.4.2.2.

V.4.2.1. Descripteurs en conflit

Un conflit peut se produire lors de l'ajout d'un nouveau descripteur de correspondance qui a la même cible qu'un descripteur existant. Par exemple, supposons que nous avons le descripteur d_i où l'attribut a_i est dérivé en utilisant la fonction f_i. L'utilisateur ajoute le descripteur d_j qui a la même cible que d_i. Dans ce dernier l'attribut a_i est dérivé en utilisant f_j telle que f_j est différente de f_i.

Si o est un objet dont la version sous la classe cible est cohérente vis-à-vis de l'expression de dépendance de d_i (c'est-à-dire, la valeur de l'attribut a_i est calculée en utilisant la fonction f_i), alors la version de cet objet est incohérente vis-à-vis de l'expression de dépendance de d_j. Il est clair que la satisfaction de l'une des deux expressions de dépendances viole l'autre et on ne peut pas assurer la satisfaction de ces deux expressions en même temps.

En effet, pour éviter des expressions de dépendance susceptibles d'introduire des conflits, un graphe de dépendance doit vérifier la propriété suivante :

• ***une classe ne peut être une cible que de 0 ou un descripteur de correspondances*** :

`In_Node` **(u ∈ NO) ∈ {0, 1}** où l'expression `In_Node`(u ∈ NO) est le nombre d'arcs entrants du noeud u.

La vérification de cette propriété est assurée de la manière suivante. Au moment de la définition d'un nouveau descripteur, nous vérifions s'il en existe déjà un autre avec la même cible. Si oui, alors l'utilisateur est averti, et doit composer les deux descripteurs en un seul.

La composition des descripteurs est à la charge de l'utilisateur. Elle peut consister en un rejet de l'un des deux descripteurs, l'ajout d'un nouveau qui englobe les deux, etc. La solution par défaut consiste à rejeter l'ajout du nouveau descripteur.

V.4.2.2. Descripteurs bilatéraux

Un graphe de dépendance peut contenir des cycles dans la mesure où des descripteurs de correspondances bilatéraux peuvent être définis.

Par exemple, nous supposons que l'évolution de schéma qui consiste à remplacer l'attribut LeSalaire_mensuel par l'attribut LeSalaire_annuel dans la classe Employé est associée au descripteur de correspondance suivant :

```
(1) source {Employé.old} target Employé
      attribute_dependencies
            LeSalaire_annuel derived {LeSalaire_mensuel}
            with (self.LeSalaire_mensuel)*12
```

Soit e un objet de la classe Employé, tel que :

$(e.\mathtt{old} \xrightarrow{\text{LeSalaire_mensuel}} 5000)$ et $(e \xrightarrow{\text{LeSalaire_annuel}} 60000)$

{e.old dénote la version de e sous la classe Employé avant l'évolution, et e dénote la version de e sous la classe Employé après l'évolution}

Nous supposons maintenant que le descripteur suivant est ajouté :

```
(2) source {Employé} target Employé.old
      attribute_dependencies
            LeSalaire_mensuel derived {LeSalaire_annuel}
            with (self.LeSalaire_mensuel)/12
```

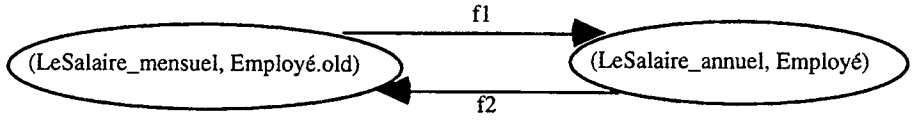


FIG. V.18 : cycle

Après l'ajout du descripteur (2), le graphe de dépendance (Cf. Figure V.18) contient un cycle. En fait, ce nouveau descripteur est défini pour décrire la relation de dépendance inverse de celle qui est décrite par le premier descripteur (1). Imaginons que l'expression de dépendances du descripteur (2) soit :

```
attributes_dependencies
        LeSalaire_mensuel derived {LeSalaire_mensuel}
        with self.LeSalaire_mensuel
```

Dans ce cas, l'existence du cycle introduit une contradiction dans le sens où la satisfaction d'une expression de dépendances provoque la violation d'une autre. Par exemple, la mise à jour de la version e.old pour satisfaire l'expression de dépendances du descripteur (2) provoque la violation de l'expression de dépendances du descripteur (1), la mise à jour de la version e pour satisfaire l'expression de dépendances du descripteur (1) provoque la violation de l'expression de dépendances du descripteur (2) et ainsi de suite.

Il est donc important de pouvoir caractériser les cycles qui introduisent des contradictions et ceux qui n'en introduisent pas. Nous appelons cycles *stables* [5.2] les cycles qui n'introduisent pas de contradictions. Ainsi, pour éviter des expressions de dépendances contradictoires, due aux descripteurs bilatéraux, un graphe de dépendance doit vérifier la propriété suivante :

- ***si le graphe contient un cycle alors ce dernier doit être stable.***

Nous nous intéressons ici à caractériser de tels cycles. Le cycle de la figure V.19, montre le cas où un attribut a_1 d'une classe c_1 est dérivé à partir d'un attribut a_2 de la classe c2 et inversement.

[5.2] Cette appellation est par analogie avec les cycles stables dans les travaux sur les multibases, présentées dans [KF91].

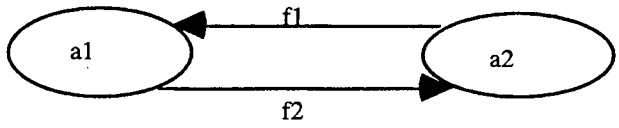


FIG. V.19 : cycle

Soient f_1, la fonction de dérivation de a_1 à partir de a_2 et f_2, la fonction de dérivation de l'attribut a_2 à partir de a_1. C'est-à-dire, $a_1=f_1(a_2)$ et $a_2=f_2(a_1)$. Dans ce cas particulier, le cycle est stable si f_1 est la fonction inverse de f_2 : $f_1(f_2(a))=a$.

Pour généraliser nous considérons le cycle de la figure V.20 :

- chaque attribut a_i ($1 \leq i \leq n$) est dérivé à partir des attributs $b_1, b_2, ..., b_m$. C'est-à-dire, $a_i=f_{ai}(b_1,...,b_m)$.
- chaque attribut b_i ($1 \leq i \leq m$) est dérivé à partir des attributs $a_1, a_2, ..., a_n$. C'est-à-dire, $b_i=f_{bi}(a_1,...,a_n)$.

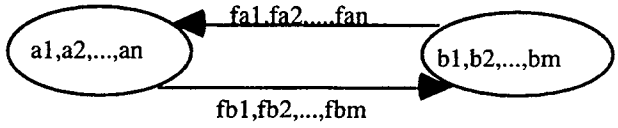


FIG. V.18 : stabilité d'un cycle

Le cycle est instable si la mise à jour d'un attribut a_i ($1 \leq i \leq n$) provoque la mise à jour des attributs $b_1, b_2, ..., b_m$, la mise à jour d'un attribut b_i (($1 \leq i \leq m$) provoque la mise à jour des attributs a1, a2, ..., an et ainsi de suite.

Le cycle est stable si le système $a_i=f_{ai}(b1, ..., b_m)$ ($1 \leq i \leq n$) est l'inverse du système $bi=f_{bi}(a_1,..., a_n)$ ($1 \leq i \leq m$). C'est-à-dire, la composition de l'ensemble des fonctions de dérivation $ai=fai(b1,..., b_m)$ ($1 \leq i \leq n$) et l'ensemble des fonction de dérivation $b_i=f_{bi}(a_1,..., a_n)$ ($1 \leq i \leq m$) est le système identité. La composition utilise la substitution des égaux par des égaux.

Exemple. Nous reprenons l'exemple 1 de la section V.2. Nous supposons que les attributs LesHeuresTd et LesHeuresCours sont dérivables à partir de l'attribut LesHeures et inversement. Les fonctions de dérivation associées sont :

f_1 : self.LesHeures = self.LesHeuresTD+self.LesHeuresCours

f_2 : self.LesHeuresTd = (self.LesHeures)/2

f_3 : self.LesHeuresCours = (self.LesHeures)/2

Les fonctions f_1 et f_2 définissent l'inverse de la fonction f_3. Par conséquent, le cycle est stable. Pour vérifier cette propriété nous utilisons la substitution des égaux par des égaux :

$$\texttt{self}.\text{LesHeures} \rightarrow \texttt{self}.\text{LesHeuresTd} + \texttt{self}.\text{LesHeuresCours}$$
$$\rightarrow \texttt{self}.\text{LesHeures} / 2 + \texttt{self}.\text{LesHeures} / 2 \rightarrow \texttt{self}.\text{LesHeures}$$

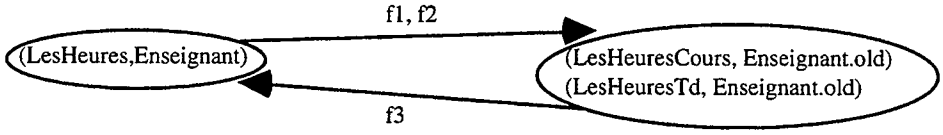


FIG. V.21 : cycle stable

V.5. Cohérence d'une base de données vis-à-vis d'un schéma de correspondances

Dans la section précédente, nous avons discuté de la correction des expressions de dépendance d'un schéma de description de correspondances. Dans cette section nous discutons de la cohérence d'une base de données vis-à-vis d'un schéma de description de correspondances.

Dans un premier temps (paragraphe V.5.1), nous donnons la définition de la cohérence d'une base de données vis-à-vis d'un schéma de description de correspondances et présentons comment cette notion de cohérence peut être affectée. Dans un second temps (paragraphe V.5.2), nous présentons la technique que nous utilisons pour assurer la maintenance de la cohérence d'une base de données vis-à-vis d'un schéma de description de correspondances.

V.5.1. Le problème

Définition

Une base de données est cohérente vis-à-vis d'un schéma de description de correspondances si pour chaque descripteur, la relation de dépendance est vérifiée entre chaque objet o de la classe cible et les objets qui ont un lien de correspondance avec o. Δ

Autrement dit, une base de données (S, I) est dite cohérente vis-à-vis du schéma de correspondances associé à S si pour chaque objet o de I, sa version sous une classe (c, n) qui est

une cible d'un descripteur de correspondance d, satisfait les relations décrites par les expressions de dépendances de d avec ses correspondants (c'est-à-dire, les versions d'objets qui sont en correspondance avec elle).

Utilisation d'un descripteur de correspondance

Dans cette partie, nous supposons que le schéma de correspondances est correct et I est cohérente vis-à-vis de S (c'est-à-dire, $\delta(S, I)$). Un descripteur de correspondance est utilisé pour initialiser l'extension de la classe cible associée ou pour propager la mise à jour des extensions des classes sources à l'extension de la classe cible. En ce qui concerne la propagation des mises à jour des extensions des classes sources, nous ne considérons maintenant que la propagation de la modification de la valeur d'une version. Nous reviendrons plus loin sur la propagation des opérations d'ajout ou de suppression des versions d'objets.

La création d'une classe cible provoque l'initialisation de son extension. L'initialisation consiste à créer des versions d'objets ou à dériver des nouvelles versions d'objets dans l'extension de la classe.

La création ou la dérivation d'une version d'objet sous la classe cible, utilise le descripteur associé. C'est-à-dire, si l'objet a un lien de correspondance, alors la liaison avec ses correspondants est établie. Cette information est stockée comme une valeur de l'attribut LesLiens (attribut pré-défini) de la version d'objet. La valeur de la version de l'objet est initialisée en utilisant l'expression de dépendance du descripteur. C'est-à-dire, si un attribut est déclaré dérivé ou indépendant mais associé à une fonction d'initialisation, alors la valeur de cet attribut dans la version d'objet est initialisée en utilisant la fonction associée.

Exemple. Étant donné le descripteur de correspondance suivant :

```
source {c'} target c
object_correspondencies
      localise Objects(c')
      link same_as_source
attribute_dependencies
      ai derived with fi
      bj new with fj
```

Pour dériver une nouvelle version de l'objet o sous la classe cible c, la primitive de dérivation est utilisée avec la clause initialisation. Ainsi, les valeurs des attributs a_i et b_i ne sont pas initialisées par le mécanisme de transformation par défaut, mais par l'utilisation du descripteur de correspondance.

```
Object_Version_Derivation (o, c', c)
with ai fi, bi fj ;
```

Mise à jour d'un objet sous une classe source

La mise à jour de la version d'un objet sous la classe source d'un descripteur de correspondance peut introduire l'incohérence de la version de l'objet de la classe cible, qui a une correspondance avec l'objet de la classe source. Par exemple, nous reprenons le descripteur (1) et l'objet e de la classe Employé, tel que :

```
attribute_dependencies
        LeSalaire_annuel derived {LeSalaire_mensuel}
        with (self.LeSalaire_mensuel) *12
```

$$(e.old \xrightarrow{LeSalaire_mensuel} 5000) \text{ et } (e \xrightarrow{LeSalaire_annuel} 60000)$$

Supposons que la valeur de l'attribut LeSalaire_mensuel de la version de e sous la classe Employé.old (c'est-à-dire, la classe source) est modifiée pour devenir 6000. La valeur de la version de l'objet e sous la classe Employé (c'est-à-dire, la classe cible) est incohérente vis-à-vis de l'expression de dépendance du descripteur (1) car 60000≠6000*12.

V.5.2. Maintien de la cohérence d'une base de données vis-à-vis de son schéma de correspondances

La maintenance de la cohérence de la base de données vis-à-vis du schéma de description de correspondances peut être immédiate ou différée. Dans le premier cas, la mise à jour d'une version d'objet sous la classe source entraîne la mise à jour de toutes les versions d'objets concernées (c'est-à-dire, celles qui sont incohérentes). Dans le deuxième cas, la mise à jour d'une version d'objet incohérente est différée à son utilisation dans un programme.

Notre approche est basée sur la maintenance différée. Avant de donner la technique que nous utilisons, nous donnons quelques définitions utiles.

Définitions

Nous supposons que le système utilise les primitives **Read** et **Write** pour lire ou écrire la valeur d'un attribut d'une version d'objet :

Read(att, oid, cid) retourne la valeur de l'attribut att de la version de l'objet identifié par oid sous la classe identifiée par cid . Nous supposons qu'au moment de la lecture d'un attribut, une valeur est déjà associée à cet attribut.

Write(att, oid, cid, nv) affecte la valeur nv à l'attribut att de la version de l'objet identifié par oid sous la classe identifiée par cid.

Etant donné un objet identifié par oid et un attribut dérivé att en utilisant la fonction f. L'expression f(LesLiens(oid)) est la valeur de la fonction f pour l'objet identifié par oid. C'est-à-dire, la valeur de l'attribut att dans la version correspondante de l'objet identifié par oid.

Technique utilisée

Quand un objet o est accédé sous une classe c, deux cas sont distingués :

• cas 1 (c est une source d'un descripteur d).

Soient o' l'objet de la classe cible c' du descripteur d, qui est en correspondance avec o. Quand un attribut att de la version de o est accédé en écriture sous la classe c, c'est-à-dire lorsque la primitive Write(att, o, c, nv) est utilisée, on vérifie s'il existe des attributs de la cible de d qui sont dépendants de l'attribut att. Si oui, la valeur des attributs de la version de o' peut être modifiée.

Nous utilisons le marquage des versions pour signaler que la valeur d'un attribut de la version de o' peut être modifié. Cette information est stockée comme une valeur de l'attribut LeMark (attribut pré-défini) de la version d'objet. Le domaine de l'attribut LeMark est l'ensemble $2^{A\times\{0,1\}}$ (A est l'ensemble des attributs). Quand une version d'objet est créée, la valeur de son

attribut LeMark est initialisée par l'ensemble $\{(a_1, 0), (a_2, 0), ..., (a_n, 0)\}$ où $a_1, a_2, ..., a_n$ sont les attributs de cette version.

Quand la classe cible a un attribut att' qui est dépendant de l'attribut att, la primitive `Write`(LeMark, o', c', nv) est utilisée pour signaler que la valeur de l'attribut att' de la version de o' peut être modifiée, tel que : nv = {...., (att', 1), ...}.

• **cas 2 (c est une cible d'un descripteur de correspondance).**

Quand un objet o est accédé en lecture sous une classe cible c, on vérifie si la version d'objet est cohérente vis-à-vis de l'expression de dépendance du descripteur. Sinon, cette version est modifiée pour être cohérente vis-à-vis de l'expression de dépendance.

La primitive `Read` est utilisée pour lire les attributs de la version d'objet o sous la classe c. Si att est un attribut de la classe c, alors les cas suivants sont possibles :

- **cas 21 (att est dérivé en utilisant la fonction f)** : si la relation de dépendance n'est pas vérifiée, c'est-à-dire `Read`(att, o, c) ≠ f(LesLiens(o)), alors la valeur de l'attribut att est modifiée en utilisant la primitive `Write`(att, o, c, f(LesLiens(o)), c'est-à-dire $(o,c) \xrightarrow{att} f(LesLiens(o))$.

- **cas 22 (att est dépendant)** : si la relation de dépendance n'est pas vérifiée, c'est-à-dire, l'attribut att de la version de o est marqué : $(o, c) \xrightarrow{LeMark} \{..., (att, 1), ...\}$, alors la valeur de att peut être modifiée. Cette valeur est modifiée en utilisant la primitive `Write`(att, o, c, nv) telle que, nv est une valeur proposée par l'utilisateur. Par défaut nv est la valeur *nil* .

V.6. Conclusion et perspectives

Dans ce chapitre, nous avons proposé une approche de description de la sémantique d'une évolution de schéma. L'usager dispose d'un langage de haut niveau pour décrire les liens de correspondances entre l'état de la base de données avant et après l'évolution. L'ensemble de liens associés à une base de données constitue son schéma de correspondances . Ainsi, la réorganisation de l'instance d'une base de donnés, en utilisant ces liens génère un état qui reflète le mieux possible l'évolution du schéma.

Nous avons discuté de la correction des expressions de dépendances d'un schéma de correspondances. La technique que nous proposons pour vérifier la correction évite les descripteurs qui ont la même cible (expressions de dépendances en conflit) et les descripteurs bilatéraux qui peuvent introduire des contradictions (cycles non stables). Nous avons discuté de la cohérence d'une base de données vis-à-vis de son schéma de correspondances et avons proposé une technique qui assure sa maintenance.

Les perspectives du travail présenté dans ce chapitre sont :

• l'extension de la technique de maintenance de la cohérence d'une base de données vis-à-vis de son schéma de correspondances. Dans ce chapitre, nous n'avons considéré que les mises à jour qui n'affectent pas la correspondance entre les identificateurs des objets. Nous envisageons la prise en compte de la propagation des autres mises à jour, comme la création, l'ajout et la suppression d'une version d'objet dans une classe source et la mise à jour de la valeur d'un attribut qui est utilisé dans une jointure dans une version d'objet dans une classe source. Cette extension sera basée sur l'exploitation des résultats des études sur le problème de la mise à jour des vues matérialisées [KKM91, Rou91, SR88].

• L'étude de la pertinence de la description des liens de correspondances entre deux versions du schéma qui ne sont pas adjacentes (importance de cette extension vs la difficulté qu'elle introduit). Cela permet par exemple de prendre en compte la réapparition dans une version k+i du schéma, d'un attribut qui est supprimé dans la version k du schéma.

Chapitre VI

MISE EN ŒUVRE : Le prototype *DB_Evolution*

Ce chapitre présente le prototype *DB_Evolution*, qui met en œuvre une partie des fonctionnalités de notre approche. Après une introduction de l'objectif et du principe suivi en section VI.1, la section VI.2 présente l'architecture générale du prototype. La section VI.3 présente le scénario d'une commande d'évolution. La section VI.4 présente le module de base du prototype.

VI.1. Introduction

Le prototype est une première expérimentation des aspects de cette approche. Nos objectifs sont :

• La validation des différents mécanismes d'évolution décrits dans les chapitres III, IV et V. Ainsi, nous démontrons que ces mécanismes sont réalisables.

• La construction d'un prototype qui doit servir de support à des tests et à des extensions futures des fonctionnalités de l'approche proposée.

Nous avons utilisé le SGBD O_2 [OT94] comme système cible.

VI.2. Architecture générale

Nous donnons ici, l'architecture générale du prototype (Cf. Figure VI.1) et son utilisation. Comme le montre la figure VI.1, le prototype *DB_Evolution* est composé de six modules : le *noyau d'évolution*, le *gestionnaire d'évolution*, le module de *vérification* de la cohérence de structure du schéma et de la correction des expressions de dépendances, le module de *propagation* d'une évolution du schéma, le module de *réorganisation* de la base de données et le module de l'*adaptation* des instances.

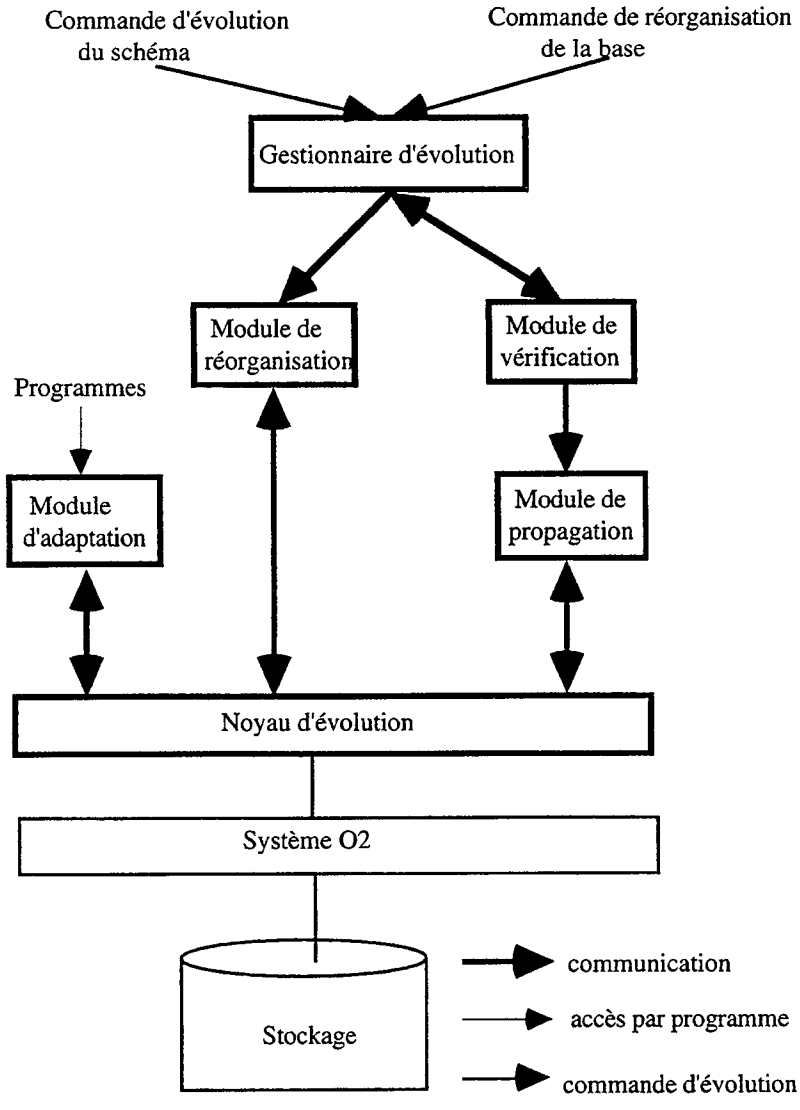


FIG. VI.1 : architecture générale

Le prototype peut être utilisé pour changer le schéma ou réorganiser la base de données. Dans la suite de cette section, nous décrivons brièvement les modules. Pour chacun, nous donnons son rôle et l'état d'avancement de son implantation.

• **Le noyau d'évolution** : c'est est une application au-dessus du système O_2. Il réalise le modèle de versions présenté dans le chapitre III. Ainsi, il offre les primitives de base pour manipuler les versions d'objets et les versions d'un schéma. Une version initiale complète est implantée.

• **Le gestionnaire d'évolution** : Il prend en entrée une commande d'évolution exprimée par un utilisateur. La commande peut être une évolution du schéma ou une réorganisation de la base de données. Il fait appel aux modules appropriés pour effectuer l'évolution demandée. Quand il s'agit

d'une réorganisation de la base, il fait appel au module de réorganisation. Quand il s'agit d'une évolution du schéma, il fait appel aux modules de vérification et de propagation. La partie implantée permet de traiter une évolution du schéma sans descripteurs de correspondances.

• **Le module de vérification** : Il prend en entrée une évolution du schéma. Il vérifie si l'évolution viole la cohérence de structure (c'est-à-dire, les invariants du schéma). Si oui, l'évolution est rejetée. Sinon, il vérifie si elle viole la correction du schéma de correspondances (c'est-à-dire, la présence de descripteurs en conflit ou un cycle instable). Si oui, l'évolution est rejetée. Sinon, il transmet l'opération d'évolution au module de propagation. La partie implantée concerne la vérification de la cohérence de structure.

• **Le module de propagation** : Il prend en entrée une évolution du schéma qui est acceptée par le module de vérification et propage ses effets. Au niveau du schéma, selon le mode de l'évolution, la propagation consiste à dériver une nouvelle version du schéma ou à modifier la version courante du schéma. Dans le cas de la modification, les programmes qui sont associés à la version courante doivent être re-compilés. Au niveau de l'instance, la propagation consiste à créer de nouveaux identificateurs d'objets ou à associer des identificateurs d'objets existants à d'autres classes.

Par exemple, à la suite de la dérivation de la classe (c, n) à partir de la classe (c, n-1), tous les objets de (c, n-1) sont associés à (c, n). L'initialisation de la classe Etudiant-Employé (Cf. Figure VI.2) entraîne la création de nouveaux identificateurs d'objets :

```
source {Etudiant, Employé} target Etudiant-Employé
object_correspondencies
        localise  select struct (id1 : x, id2 : y)
                  from x in Objects(Etudiant), y in Objects(Employé)
                  where x.LeNom=y.LeNom
        link  new
```

FIG. IV.2 : descripteur de correspondances

Notons que le module de propagation est utilisé aussi pour propager les effets d'une mise à jour des objets d'une classe, comme la création, l'ajout et la suppression d'un objet. Cet aspect est partiellement considéré dans la mesure où seules la création et l'ajout d'un objet sont propagées par le système. Par exemple, l'ajout d'un objet à la classe (c, n) provoque l'ajout de cet objet à la classe (c, n-1).

La partie implantée propage les effets de l'évolution au niveau du schéma et associe des identificateurs d'objets existants à d'autres classes.

• **Le module d'adaptation** : Il est utilisé pour initialiser ou modifier la valeur d'une version d'objet lors de son utilisation par un programme. Il prend en entrée un objet et une classe de l'objet. Il vérifie si l'objet possède une version sous la classe. Si oui, il vérifie si la valeur de cette version est cohérence vis-à-vis du schéma de correspondances et modifie la valeur si nécessaire. Sinon, une nouvelle version stockée ou calculée est générée (Cf. Chapitre IV). Une version initiale complète est implantée.

• **Le module de réorganisation** : il prend en entrée une commande de réorganisation. Il réorganise la base de données en supprimant les versions du schéma et les classes qui ne sont plus pertinentes pour les applications (Cf. Chapitre IV). Une version initiale complète est implantée.

VI.3. Scénario d'une commande d'évolution

Nous présentons dans cette section le scénario d'une commande d'évolution sur des exemples[6.1]. Le premier exemple montre comment le système traite un changement du schéma (Cf. Paragraphe IV.3.1). Le deuxième exemple montre comment le système traite une réorganisation de la base de données (Cf. Paragraphe IV.3.2).

IV.3.1. Changement du schéma

Supposons que le schéma S possède deux versions (S, 0) et (S, 1). La version (S, 0) contient la classe (Enseignant, 0). La version (S, 1) contient la classe (Employé, 1) telle que Objects((Employé, 1))={o1, o2, o3}. Le schéma de correspondances contient le descripteur suivant :

```
source {(Enseignant, 0)} target (Enseignant, 1)
attribute_dependencies
        LesHeures derived {LesHeuresTd, LesHeuresCours}
        with (self.LesHeuresTd + self.LesHeuresCours);
```

L'évolution du schéma consiste à remplacer l'attribut LeSalaire-mensuel de la classe Employé par l'attribut LeSalaire-annuel. Supposons que l'utilisateur spécifie cette évolution par (Cf. Annexe B pour la syntaxe):

6.1 l'exemple choisi sert à expliquer le scénario de la commande d'évolution, et non à justifier de l'évolution.

```
schema_evolution  S
    mode  : version                    /* le mode d'évolution, ici versionnement du schéma */
    operations          /* la séquence d'opérations de changement du schéma, ici deux opérations*/
        <Add_attribute(LeSalaire-annuel : integer, Employé);
        Del_attribute(LeSalaire-mensuel, Employé);>
    descriptors                /* les descripteurs de correspondances, ici un descripteur */
        <source {Employé.old} target Employé
        attribute_dependencies
            LeSalaire_annuel derived {LeSalaire_mensuel}
            with (self.LeSalaire_mensuel)*12;>
```

Cette évolution est appliquée sur la version (S, 1), qui est la version courante du schéma S. Pour réaliser ce changement, le gestionnaire d'évolution fait appel au module de vérification pour vérifier :

• **La cohérence de structure** : dans une première phase, il considère l'opération `Add_attribute`(LeSalaire-annuel : integer, Employé). L'ajout d'un attribut entraîne la vérification des invariants *"noms distincts"* et*"compatibilité de types d'attributs"* (Cf. Chapitre III). Ces invariants sont vérifiés car il n'existe pas un attribut qui a le même nom que LeSalaire-annuel. Dans une deuxième phase, il considère l'opération `Del_attribute`(LeSalaire-mensuel, Employé). Celle-ci ne viole aucun invariant. Donc, la cohérence de structure est vérifiée.

• **La correction du schéma de correspondances** : l'ajout du descripteur associé à l'évolution n'introduit pas la présence d'un cycle car sa cible n'est pas une source du descripteur existant (c'est-à-dire, la classe (Employé, 0)). Il n'est pas en conflit avec le descripteur existant car leurs cibles sont différentes. Donc, la correction du schéma de correspondances est vérifiée.

L'évolution est acceptée par le module de vérification. En effet, le gestionnaire d'évolution fait appel au module de propagation pour réaliser les effets aux niveaux :

• **Schéma** : cette évolution est traduite par la dérivation de la version (S, 2) du schéma S (Cf. Figure VI.3) car le mode "version" est spécifié par l'utilisateur. La version est dérivée en utilisant la primitive suivante (fourni par le noyau d'évolution) :

```
schema_version_derivation (S)
Add_attribute(LeSalaire-annuel : integer, Employé);
Del_attribute(LeSalaire-mensuel, Employé);
```

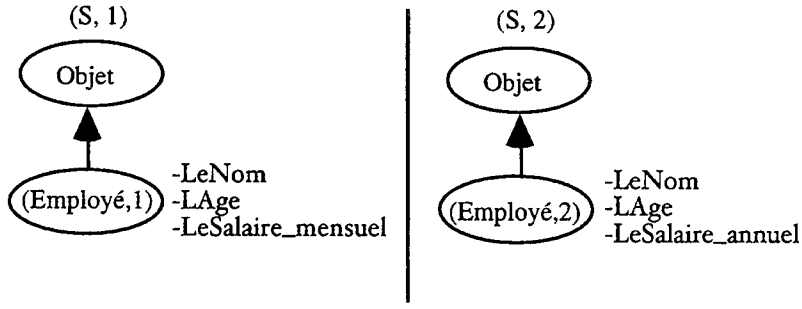


FIG. IV.3 : évolution du schéma

Le descripteur est ajouté au schéma de description de correspondances.

• **Instance** : les identificateurs d'objets de la classe (Employé, 1) sont associés à la classe (Employé, 2) ; c'est-à-dire `Objects` ((Employé, 2))={o_1, o_2, o_3}. Donc o_1, o_2 et o_3 sont des objets de (Employé, 1) et de (Employé, 2).

Les valeurs des versions de o_1, o_2 et o_3 sous la classe (Employé, 1) sont initialisées au moment de leur première utilisation, en utilisant le module d'adaptation. Supposons que l'objet o_1 est accédé sous (Employé, 2) et que (S, 2) est la version courante du schéma. Le module d'adaptation découvre que o_1 ne possède pas une version sous cette classe et par suite il dérive une nouvelle version de o_1 en utilisant la primitive suivante (fournit par le noyau d'évolution) :

```
Object_Version_Derivation (o1, (Employé, 1) , (Employé, 2))
with LeSalaire_annuel (self.LeSalaire_mensuel) *12;
```

La version dérivée est *stockée* car la classe (Employé, 2) est pertinente : elle est associée à la version courante du schéma.

VI.3.2. Réorganisation de la base de données

Supposons que le schéma possède quatre versions (S, 0), (S, 1), (S, 2) et (S, 3) (Cf. Figure VI.4). Les tableaux des figures VI.5 et VI.6 donnent le poids des versions du schéma et le poids des classes. Supposons que l'utilisateur décide de supprimer les versions historiques du

schéma qui ont un poids inférieur ou égal à 5, et les classes qui ne seront plus au service des applications (Cf. Chapitre IV). La réorganisation est décrite par la commande suivante :

```
reorganise_database np:5, cl : schéma;
```

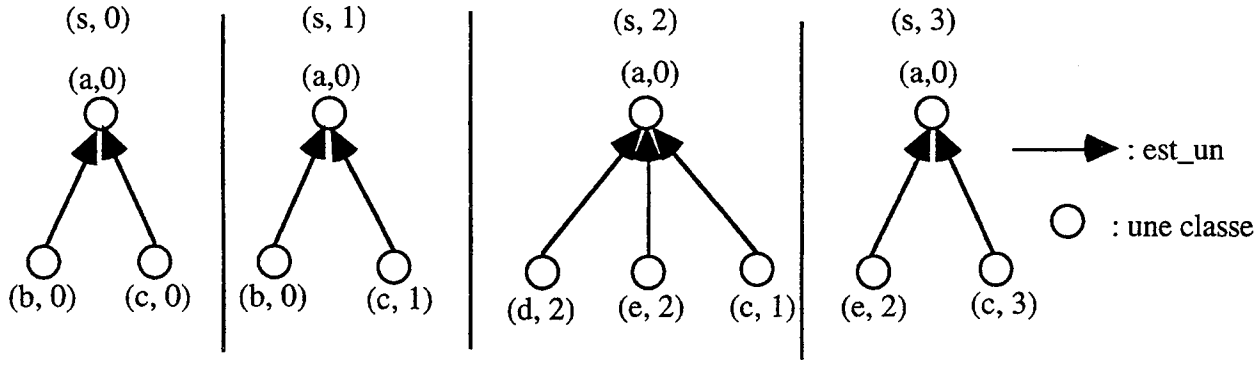


FIG. VI.4 : le schéma S avant la réorganisation

version de schéma	(S, 0)	(S, 1)	(S, 2)	(S, 3)
S_Poids	20	2	15	infini

FIG. VI.5 : poids des versions du schéma avant la réorganisation

classe	(a, 0)	(b, 0)	(c, 0)	(c, 1)	(d, 2)	(e, 2)	(c, 3)
C_Poids	1	0	0.5	0.6	0	1	1

FIG. VI.6 : poids des classes avant la réorganisation

Pour réaliser la réorganisation, le gestionnaire d'évolution fait appel au module de réorganisation. Dans une première étape, le module de réorganisation supprime la version (S, 1) car son poids est inférieur à 5 (Cf. Figure VI.5). Pour cela, il utilise la primitive `delete_schema_version` ((S, 0)). La suppression de (S, 1) provoque la suppression de la classe (b, 0) car elle satisfait les conditions de suppression (c'est-à-dire, son poids est nul et elle est feuille dans (S, 0) et (S, 1)). Pour cela, il utilise la primitive `delete_class`((b, 0)). Dans une deuxième étape, le module de réorganisation supprime la classe (d, 2) car la commande est déclenchée en fixant la valeur du paramètre `cl` à "schema". C'est-à-dire, toutes les classes qui vérifient les conditions de suppression sont supprimées (Cf. Figure VI.7).

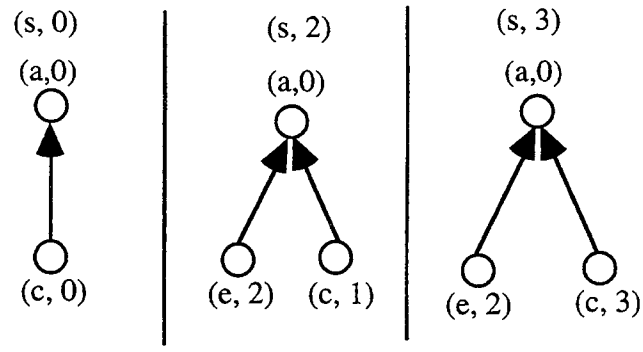


FIG. VI.7 : le schéma S après la réorganisation

VI.4. Le module noyau d'évolution

Le noyau d'évolution fournit les primitives pour gérer les versions d'un schéma et d'objets. Les versions sont des objets O_2. Le ***MétaCatalogue*** est le schéma O_2 de l'application qui réalise le noyau d'évolution. Les classes de ce schéma fournissent les services de gestion de l'évolution du schéma.

VI.4.1. Schéma du noyau d'évolution

Les primitives de gestion de versions, citées dans les trois chapitres précédents, sont réalisées en utilisant des méthodes du schéma du noyau d'évolution. La figure VI.8 présente une vue globale des classes du noyau d'évolution. Ce schéma est évidement extensible par de nouvelles classes ou l'affinement de classes existantes.

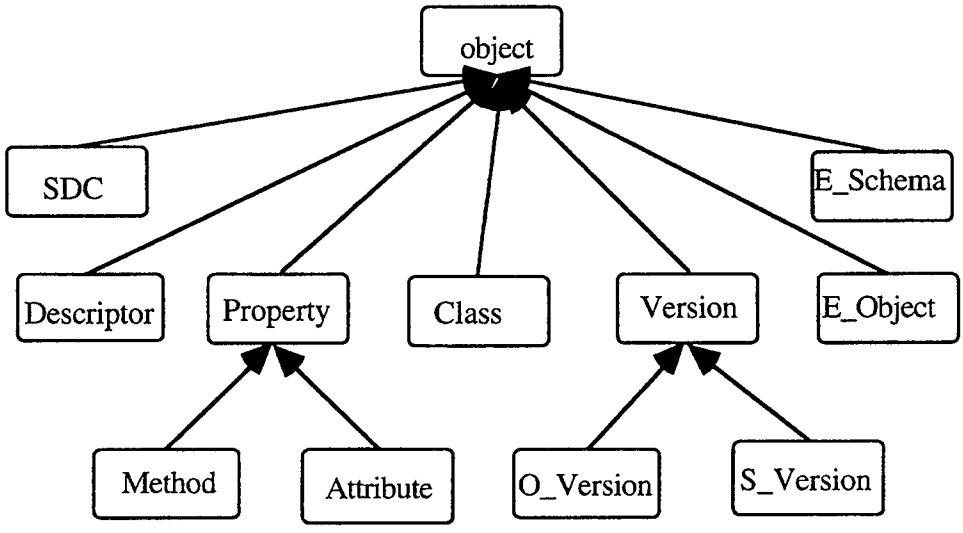


FIG. VI.8 : les classes du noyau d'évolution

Dans cette section, nous décrivons brièvement les classes principales :

• La classe **E_Schema** (entité schéma) met en œuvre des méthodes permettant la gestion d'un schéma. Par exemple, un objet de cette classe possède un attribut dont la valeur est le nom du schéma et un attribut dont la valeur est une référence à une collection d'objets de la classe **S_Version** (c'est-à-dire, les versions d'un schéma). Les méthodes de cette classe ont pour but de consulter et de mettre à jour toutes les informations d'un schéma. Par exemple, l'appel de la méthode **Current** sur un schéma existant fournit une référence à la version courante de ce schéma. Cette classe sert de point d'entrée à l'ensemble des versions d'un schéma, grâce à son unique objet nommé **Sch**.
• La classe **E_Object** (entité objet) met en œuvre des méthodes permettant la gestion d'un schéma. Par exemple, un objet de cette classe possède un attribut dont la valeur est une référence à une collection d'objets de la classe **O_Version** (c'est-à-dire, les versions d'un objet). Les méthodes de cette classe ont pour but de consulter et de mettre à jour toutes les informations d'un objet. Par exemple, l'appel de la méthode **Classes** sur un objet existant fournit une référence à une collection d'objets de la classe **Class** (c'est-à-dire, les classes de l'objet).

• La classe **Class** (classe) met en œuvre des méthodes permettant la gestion d'une classe. Par exemple, un objet de cette classe possède un attribut dont la valeur est une référence à une collection d'objets de la classe **S_Version** (c'est-à-dire, les versions du schéma auxquelles la classe est associée). Les méthodes de cette classe ont pour but de consulter et de mettre à jour toutes les informations d'une classe. Par exemple, l'appel de la méthode **PL** sur une classe existante fournit le niveau de pertinence de cette classe ("Obsolète" ou "Pertinente").

• La classe **S_Version** (version d'un schéma) met en œuvre des méthodes permettant la gestion d'une version de schéma. Par exemple, un objet de cette classe possède un attribut dont la valeur est une référence à une collection d'objets de la classe **Class** (c'est-à-dire, les classes de la version de schéma). Les méthodes de cette classe ont pour but de consulter et de mettre à jour toutes les informations d'une version de schéma. Par exemple, l'appel de la méthode **schema_version_derivation** (changement_du_schéma) sur la version courante d'un schéma permet de dériver une nouvelle version de ce schéma en utilisant la séquence d'opérations d'évolution du schéma spécifiée par le paramètre changement_du_schéma.

• La classe **O_Version** (version d'un objet) met en œuvre des méthodes permettant la gestion d'une version stockée d'un objet. Par exemple, un objet de cette classe possède un attribut dont la valeur est une référence à un objet de la classe **Class** (c'est-à-dire, la classe à laquelle est associée la version). Les méthodes de cette classe ont pour but de consulter et de mettre à jour toutes les informations d'une version d'objet. Par exemple, l'appel de la méthode **object_version_gen**

(classe) sur la version une version existante d'un objet permet de générer une nouvelle version de cet objet sous la classe identifiée par le paramètre classe.

VI.4.2. Utilisation

La fonction principale du noyau d'évolution est de fournir aux autres modules du prototype des primitives de manipulation des versions. Il s'agit des primitives de manipulation des instances des classes **S_Version** et **O_Version**. A titre d'exemple, nous citons la primitive **delete_schema_version** qui permet la suppression d'une version historique du schéma (Cf. Chapitre IV) et la primitive **objet_version_derivation** (Cf. Chapitre III) qui permet la dérivation d'une version d'un objet.

Ce module peut être utilisé directement par l'utilisateur, à l'aide du langage O2C ou à l'aide d'une interface graphique. L'utilisateur peut alors créer, dériver et consulter des versions.

VI.5. Conclusion

Le prototype *DB_Evolution* a pour but de valider les mécanismes d'évolution de schéma développés dans cette thèse, et de servir à des extensions futures des fonctionnalités de l'approche proposée. Le développement du prototype se poursuit par l'implantation du langage de description des relations correspondances, au dessus du système *O2*.

CONCLUSION ET PERSPECTIVES

Cette conclusion résume les aspects abordés dans cette thèse et situe notre contribution dans le domaine de l'évolution du schéma d'une base de données. Nous présentons ensuite les perspectives de notre travail.

VII.1. Bilan

L'étude présentée dans cette thèse se place dans le cadre des travaux de recherche sur le problème de l'évolution de schéma dans les SGBD à objets. L'évolution de schéma est d'une importance capitale pour les applications avancées. Le support de cette fonctionnalité est un facteur de puissance d'un SGBD à objets.

Nous avons, dans une première partie, analysé le problème, étudié et évalué les solutions proposées. Nous avons montré les insuffisances de ces solutions. Ensuite, nous avons relevé les constations, à notre point de vue, principales :

• Les approches proposées, en particulier, celles fondées sur la modification et celles fondées sur les versions, offrent des fonctionnalités complémentaires.

• En général, le mécanisme de réorganisation de la base à la suite de l'évolution de son schéma n'intègre pas le point de vue de l'utilisateur, porteur d'une certaine sémantique. Par conséquent, l'état de la base de données ne correspond pas quelques fois à l'évolution du monde réel, en particulier, une évolution qui est multiple.

Nous avons, dans la deuxième partie, développé une nouvelle approche pour la gestion de l'évolution de schéma. D'une part, cette approche propose un cadre qui permet de combiner les fonctionnalités de la modification et du versionnement pour une meilleure gestion de l'évolution de schéma. D'autre part, elle offre à l'utilisateur la possibilité de spécifier les liens entre les différents états de la base de données afin de traduire le plus fidèlement possible les évolutions du monde réel. Nous résumons les principaux apports de l'étude présentée dans cette thèse dans les points suivants :

• Nous avons défini un modèle de versions de schéma et d'objets. Nous avons proposé un modèle formel de description des versions d'un schéma et de son instance et avons étudié le problème de l'évolution du schéma en utilisant ce modèle formel.

• Le processus du changement de schéma combine la *modification* et le *versionnement* du schéma. Par défaut, l'évolution du schéma peut se traduire, soit par sa modification, soit par la génération d'une nouvelle version du schéma, selon que l'évolution est *soustractive* ou non *soustractive* . Le mode d'évolution ("modification" ou "versionnement") peut être imposé par l'utilisateur.

• La technique d'adaptation des instances combine le *versionnement*, l'*émulation* et la *conversion* des objets. Lorsqu'un accès à un objet est demandé sous une classe dont l'extension ne contient pas de version de cet objet, une nouvelle version *stockée* de l'objet est générée seulement si la classe est *pertinente*. Autrement, une version *calculée* de l'objet est générée. Après la génération de la nouvelle version, l'origine de génération (c'est-à-dire, la version à partir de laquelle est générée la nouvelle version) est supprimée lorsqu'elle n'est plus utilisée par des applications.

• Nous avons proposé une opération qui permet la réorganisation immédiate de la base de données. Le mécanisme associé à la réorganisation supprime les classes et les versions historiques du schéma qui ne sont plus pertinentes (ou jugées non pertinentes par l'administrateur de la base) pour les applications.

• Le langage d'expression d'une évolution de schéma permet de décrire des liens de correspondances entre l'état de la base de données avant et après l'évolution. L'ensemble de liens associés à une base de données constitue son schéma de correspondances. Ainsi, la réorganisation de l'instance d'une base de données, en utilisant ces liens génère un état qui reflète le mieux possible l'évolution du schéma.

Le versionnement de schéma évite la perte d'informations et assure que les anciens programmes d'applications continuent de fonctionner. Cependant, le nombre de versions peut devenir important ; ce qui rend complexe leur gestion. Notre approche permet de limiter le nombre de versions : (1) l'évolution d'un schéma est traduite par sa modification si l'évolution est non-soustractive ou si l'utilisateur le décide, (2) la technique d'adaptation des instances génère des versions calculées lorsque le versionnement des objets n'est pas nécessaire, (3) la possibilité donnée à l'administrateur de réorganiser la base de données lui permet de supprimer des versions historiques du schéma.

VII.2. Perspectives

Le sujet sur lequel porte cette thèse est très complexe ; il reste encore beaucoup à faire dans ce contexte. Nous avons proposé quelques perspectives associées à notre travail dans les chapitres III, IV et V.

Nous envisageons aussi de prolonger ce travail pour prendre en compte les aspects suivants :

Aspects d'utilisation

Dans le but d'évaluer les aspects liés à l'utilisation de notre approche, nous envisageons son utilisation dans des applications réelles et étudié : (1) quantitativement les effets d'une évolution de schéma (par exemple, le nombre de classes changées, le nombre de programmes qui doivent être modifiées, etc.), (2) la puissance et la simplicité du langage d'expression d'une évolution de schéma (par exemple, l'expérimentation de notre système en considérant différents niveaux d'utilisateurs et différents domaines d'applications).

Performances

La gestion de l'évolution de schéma peut avoir différents impacts sur les composants du système. Par conséquent, il devient nécessaire de considérer les problèmes de performance. Dans le cadre de notre approche, nous envisageons d'étudier : (1) les extensions au niveau des structures de stockage (par exemple, proposer des structures de stockage et des méthodes d'accès appropriées aux données dans le contexte d'une approche fondée sur les versions), (2) les impacts des mécanismes d'évolution (c'est-à-dire, la modification et le versionnement de schéma) sur les architectures d'un SGBD (client/serveur(s)), (3) la généralisation du *modèle d'évaluation de performances* de la conversion des objets, présenté dans [FMZ95b], aux autres techniques d'adaptation des instances (c'est-à-dire, l'émulation et le versionnement des objets). Un des résultats de l'étude présentée dans [FMZ95b] est que la *conversion différée* est préférable dans un système qui ne gère pas les extensions de classes, comme le SGBD O_2.

Évolution de méthodes

Les solutions apportées par des études faites sur la cohérence de comportement sont actuellement loin d'être complètes. Nous voulons analyser en profondeur l'interaction entre l'évolution du schéma et le concept de méthode dans un modèle à objets.

BIBLIOGRAPHIE

[ABB*84] Ahlsen M., Bjornerstedt B., Britts S., Hulten C., Soderlund L. *Making Type Changes Transparent.* SYSLAB, No. 22, University of Stockholm, 1984.

[ABD*89] Alkinston M., Bancilhon F., DeWitt D., Dittrich K., Maier D., Zdonik S. *The Object-Oriented Database System Manifesto.* Proc. of DOOD'89 Conf., Kyoto, Japan, Dec. 1989.

[ABD*94] Amiel E., Bellosta M., Dujardin E., Simon E. *Supporting Exceptions to Behavioral Schema Consistency to Ease Schema Evolution in OODBMS.* Proc. of the 20th VLDB Conf., pp. 108-119, Sep. 1994.

[ABG*93] Albano A., Bergamini R., Ghelli G., Orisini R. *An object data model with roles.* Proc. of the 18th VLDB Conf., Dublin, Ireland, pp. 9-51, 1993.

[AC93] Adiba M., Collet C. *Objets et bases de Données ; le SGBD O2.* Hermès, Jun. 1993.

[AFL93] Al-Jadir L., Flaquet G, Léonard M. *Context Versions in Object-Oriented Model.* Proc. of the 4th Int. DEXA Conf., pp. 24-35, LNCS Springer Verlag, Prague, Sep. 1993.

[Agr90] Agrawal R. *Object Versioning in Ode.* Proc. of the 16th VLDB Conf., Brisbane Australia, Aug. 1990.

[AK89] Abiteboul S., Kanellakis P. Object *Identity as a Query Language Primitive.* Proc. ACM SIGMOD, pp. 159-173, 1989.

[Akp93] Akpotsui E.*Transformation de types dans les systèmes d'édition de documents structurés.* Thèse de Doctorat, Institut National Polytechnique, Grenoble, Oct. 1993.

[AKW90] Abiteboul S., Kanellakis P., Waller E. *Method Schemas.* Proc. of ACM PODS'90.

[ALP91] Andany J., Léonard M. , Palisser C. *Management of schema evolution in database.* Proc. of the 17th VLDB Conf., Barcelona, 1991.

[Ara90] Arapis C. *Specifying Object Life-Cycles.* Université de Genève, faculté des Sciences, 1990.

[Ara92] Arapis C. *Dynamic Evolution of Object Behavior and Object Cooperation.* Université de Genève, faculté des Sciences, 1992.

[Ari91] Ariav G. Temporally *oriented data definitions : managing schema evolution in temporally oriented databases.* Data and Knoweldge Engineering, Vol. 6, pp. 451-467, 1991.

[ASM93] Atkinson M., Sjoberg D., Marrison R. *Managing Change in Persistent Object Systems.* JSSST Int. Symp. on Object Technologies for Advance Software, Kanazawa, Japan, Nov. 1993.

[Bar91] Barbedette G. *Schema modifications in the LispO2 persistent object-oriented language.* Proc. of the ECOOP'91 Conf., Geneva, Jul. 1991.

[BB88] Bjonrnerstedt A., Brigitts S. *AVANCE : An Object Management System.* Proc. of OOPSLA'88, pp. 206-221, San diego, 1988.

[BCD*87] Borras P., Clément D., Despeyroux T., Incerpi J., Lang B., Kahn G., Pascual V. *Centaur : the system.* Rapport de recherche INRIA, 777, Dec. 1987.

[BCG*88] Banerjee J., Chou H., Graza J., Kim W., Woelk D., Ballou N. *Integrating an object-oriented programming system with a database system.* ACM OOPSLA'88, 1988.

[BDK92] Bancilhon F., Delobel C. , Kanellakis P. *Building an Object-Oriented Database System; the Story of O2.* Morgan Kaufmann, 1992.

[BDP93] Benzaken V., Doucet A., Policella P-Y. *Définition et gestion de contraintes d'intégrité dans le langage Thémis.* In Proc. of the 9th PRC-BD3, Toulouse, France, Oct. 1993.

[BDS95] Benzaken V., Doucet A., Schaefer. *Integrity constraint checking optimisation based on abstract databases generation and program analysis.* Journal de l'Ingénierie des Systèmes d'information, vol. 1, No. 3, Mar. 1995.

[Ben92] Benatallah B. *Evolution de schéma dans les bases de données orientées-objet. Application au modèle Aristote.* Rapport Aristote N°NOT013, LGI-IMAG, Jun. 1992.

[Ben94] Benatallah B. *Evolution de schéma et systèmes à objets : une synthèse.* Congrès INFORSID'94, Aix-en-Provence, Mai 1994.

[Ben95a] Benatallah, B. *On Versions Control for Schema Evolution.* Rapport de recherche RR-947-I, LGI-IMAG, Grenoble, Jui. 1995.

[Ben95b] Benatallah B. *Towards an Hybrid Approach for Object-Oriented Schema Evolution Management..* Fourth Maghrebian Conf. on Software Engineering and Artificial Intelligence, Alger, Algeria, Apr. 1996.

[Ber92] Bertino E. *A view Mechanism for Object-Oriented Data Model.* Proc. of the Int. Conf. on Extending Database Technology, Vienna, Austria, March 1992.

[BF94] Benatallah B., Fauvet M. *Fiabilité des applications, pérennité et cohérence des informations d'une base de données autorisant l'évolution de schéma.* Third Maghrebian Conf. on Software Engineering and Artificial Intelligence, Rabat, Moroco, Apr. 1994.

[BF95] Benatallah B., Fauvet M. *Evolution de schéma & Adaptation des instances.* Congrès INFORSID'95, Grenoble, Mai 1995.

[BFK95] Brèche P., Ferrandina F., Kuklok M. *Simulation of Schema and Database Modifications using Views.* Proc. of DEXA'95 Conf., London, 1995.

[BH89] Bjonrnerstedt A., Hulten C. *Version Control in an Object-Oriented Architecture.* In Object-Oriented Concepts and databases, Ed. W. Kim, F. Lochovsky, Addison-Wesley, 1989.

[BK87] Banerjee J., Kim W. *Semantics and implementation of Schema Evolution in Object-Oriented Databases* . Proc. of the ACM SIGMOD Conf., pp. 311-323, San francisco, 1987.

[BM93] Byeon K., McLeod D. *Towards the Unification of Views and Versions for Object Databases.* JSST Int. Symp. on Object Technologies for Advanced Software, Kanzawa, Japan, Nov. 1993.

[BM88] Beech D., Mahbod B. *Generalized version control in an Object-Oriented database* . Proc. of the 4th Conf. on Data Engineering, Los Angeles, USA, Feb. 1988.

[BMO*89] Bretl R., Maier D., Otis A., Penny J., Schuchardt B., Stein J., Williams E., Williams M. *The GemStone Data Management System.* In Object-Oriented Concepts, Databases and Applications (Kim W., Lockovsky F. editors), Chapter 12, ACM Press, 1989.

[Bor88] Borgida A. *Modelling Class Hierarchies with Contradictions.* SIGMOD Record, Vol. 17, No. 3, ACM, Sep. 1988.

[Bou95] Bounaas F. *Gestion de l'évolution dans les bases de connaissances : une approche par les règles*. Thèse de Doctorat, Institut National Polytechnique, Grenoble, Oct.1995.

[Bra92] Bratsberg E. *Unified Class Evolution by Object-Oriented Views*. Proc. of the 11th Int. Conf. on the Entity-Relationship Approach, LNCS No. 645, Springer Verlag, Oct. 1992.

[Bra93] Bratsberg E. *Evolution and Integration of classes in Object-Oriented Databases*. PhD thesis, Norwegian Institute of Technology, Jun. 1993.

[BW85] Borgida A., Williamson K.E. *Accomodating Exceptions in Databases, and Refining the Schema by Learning from them*. Proc. of the 11th VLDB Conf., Stockholm, Sweden, Aug.1985.

[Cap95] Capponi C. *Identification et exploitation des types dans un modèle de représentation de connaissances*. Thèse de Doctorat, Université Joseph Fourier, Grenoble1, Oct. 1995.

[Car89] Cardelli L. *Typeful programming*. Technical Report, Digital Systems Corp. Research Center, Report 45, Palo Alto, CA 94301, USA, May 1989.

[Cas91] Casais E. *Managing Evolution in Object-Oriented Environnements : An Algorithmic Approach*. PhD Thesis, University of Geneva, 1991.

[Cas92] Casais E. *An Incremental Class Reorganisation Approach*. Proc. of the ECOOP'92 Conf., LNCS No. 615, Springer Verlag, Jul. 1992.

[Cat93] Cattel R. *The Object Database Standard : ODMG - 93 (1.1)*. Morgan Kaufmann, 1993.

[CCK*94] Connor R., Cutts Q., Kirby G., Morrison R. *Using Persistence Technology to Control Schema Evolution*. Proc. of the ACM SIGAP Conf., phoenix, Arizona, Mar.1994.

[CCL*93] Cattaneo F., Coen-Porisini A., Lavazza C., Zicari R. *Overview and Progress Report of the ESSE Project : Supporting Object-Oriented Database Schema Analysis and Evolution*. Proc. of TOOLS EUROPE' 93 Conf., Versailles, Mar. 1993.

[CGS95] Castro C., Grandi F., Scalas M. *On Schema Versioning in Temporal Databases*. Proc. of the 21th VLDB Conf., Zurich, Sep.1995.

[CJK91] Cellary W., Jomier G., Koszlajda T. *Formal Model of an Object-Oriented Database with Versioned Objects and Schema*. Prof of the DEXA'91 Conf., pp. 239-244, Aug. 1991.

[CK86] Chou H., Kim W. *A unifying Framework for Version Control in a CAD Environment*. Proc. of the 12th VLDB Conf., Kyoto, Aug. 1986.

[Cla92] Clamen S. *Type Evolution and Instance Adaptation*. Technical Report No. CMU-CS-92-133, School of Computer Science, Carnegie, Mellon University, Pittsburgh, PA 15213-3890. 1992.

[Cla93] Clamen S. *Schema Evolution and integration*. Distributed and Parallel Databases, Vol. 2(1), Jan. 1993.

[CLZ91] Coen-Porisini A., Lavazza C, Zicari R. *Verifying Behavioral Consistency of an Object-Oriented Database Schema*. Research Report No. 91-054, Politecnico di Milano, Nov. 1991.

[CM92] Chen I., McLeod D. *A unified Approach to Data and Meta-Data Modification for Data/Knoweldge Bases*. Theoretical Studies in Computer Science Journal, pp. 287-311, 1992, Academic Press, Inc.

[CMS*91] Cowan D., Mackie E., Smit V., Pianosi G. *Rita-An Editor and User Interface for Manipulating Structured Documents*. Electronic Publishing Origination Dissemination and Design, 4(3), pp. 125-150, Mar. 1991.

[Cod70] Codd E. *A relationnal model for large shared data banks*. Communication of ACM, 1970.

[COD71] *CODASYL Data Base Task Group*. ACM Report, 1971.

[Con91] Connor R. *Types and Polymorphism in Persistent Programming Systems*. PhD thesis, University of St Andrews, 1991.

[CS89] Christodoulakis I., Soupos P. *Adaptative Database Schema Evolution via ConstrainedRelationships*. Proc. of the Int. IEEE Workshop on Tools for IA : Architectures, Languages, and Algorithms, pp. 393-398, 1989.

[CW85] Cardelli L., Wegner P. *On understanding types, data abstraction and polymorphism*.ACM Computing Surveys, Vol. 17(4), 1985.

[DA82] Delobel C., Adiba M. *Bases de données et systèmes relationnels*. Dunod, 1982.

[Dal91] Dale S. *Object-Oriented Product Modelling for Structural Design*. PhD thesis, The Norwegian Institue of Technology, Feb. 1991.

[Dea91] Dearle A. *Environments : A Flexible binding mechanisms to support system evolution*. Proc. of the Twenty-Second Annual Hawai Int. Conf. on System Sciences, pp. 46-45, Jan. 1991.

[Del92] Delcourt C. *Evolution de schéma dans les bases de données orienté-objet*. Thèse de Doctorat, Université d'Orsay, Paris Sud, Novembre 1992.

[DL85] Dittrich K., Lorie R. *Version Support for Engineering Database Systems*. IBM Research Report No. R.J 4769 (50628), Jul. 1985.

[DST95] Delobel C., Sousa dos Santos C., Tallot. *Object Views on Relations*. Congrès INFORSID'95, Grenoble, Mai 1995.

[DT87] Dadam P., Teuhola J. *Managing Schema Versions in a Time-versioned Non-first-normal-form Relationnal Database*.Technical Report No. 87.01.001, IBM Heidelberg Scientific Center, Germany, Jan. 1987.

[DZ91] Delcourt C., Zicari R. *The design of an Integrity Consistency Checker (ICC) for an Object-Oriented Database System*. Proc. of the ECOOP'91 Conf., Geneva, Jul. 1991.

[ED95] Fontana E. , Dennebouy Y. *Schema Evolution by using Timestamped Versions and Lazy Strategy*. Proc. 11èmes Journées de Bases de Données Avancées, Nancy, Aou. 1995.

[EO93] Ewald C. and Orlowska M. *A procedural Approach to Schema Evolution*. Proc. of the 5th CAISE'93, LNCS, Springer Verlag, No. 685, Paris, 1993.

[Est88] Estublier J. *Configuration Management, the notion and the tools*. Proc. of the Int. Workshop on Software Version and Configuration Management, Grassau, Jan. 1988.

[Fau89] Fauvet M. *Définition et réalisation d'un modèle de versions d'objets*. 5èmes journées de Bases de Données Avancées, Genève, Sep. 1989.

[Fau92] Fauvet M. *Versions and Histories in Object Oriented Applications*. Proc. of the 7th Int. Conf. on Database (7SBBD), Porto-Alegre (Brazil), May 1992.

[FD91] Fauvet M., Ducreux F. *Définition et manipulation d'objets, de versions et d'historiques*. Actes du Congrès INFORSID'91, Paris, Jui. 1991.

[FMZ94a] Ferrandina F. , Meyer T., and Zicari R. *Implementing Lazy Database Updates for an Object Database System*. Proc. of the 20th VLDB Conf., Santiago, Chile, Sep. 1994.

[FMZ94b] Ferrandina F., Meyer T., Zicari R. *Correctness of Lazy Database Updates for Object Database*. POS'94, sep. 1994.

[FMZ95a] Ferrandina F., Meyer T., Zicari R. *Schema and Database Evolution in the O2 system*. Proc. of the 21th VLDB Conf., Zurich, Sep. 1995.

[FMZ95b] Ferrandina F., Meyer T., Zicari. *Measuring the Performance of Immediate and Deferred Updates in Object Database Systems*. In OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance, Austin, Texas, Oct. 1995.

[FS95] Fauvet M., Scholl P-C. *Temps et bases de données : concepts temporels pour la gestion de l'évolution des données*. Rapport de recherche RR-945-I, LGI-IMAG, Grenoble, 1995.

[Gan94] Gancarski S. *Versions et bases de données : modèle formel, supports de langage et d'interface-utilisateur*. Thèse de doctorat en science, Université Paris-Sud, Paris, Dec. 1994.

[GJ94] Gancarski S., Jomier G. *Un formalisme pour la gestion de versions d'entités dans leur contexte*. Proc. 10èmes Journées de Bases de Données Avancées, Clermont-Ferrand, Aou.1994.

[GKL86] Garlan D., Krueger C., Lerner B-S. *A Structural Approach to the Maintenance of Structure-Oriented Environments*. Proc. of ACM SIGSOFT / SIPGLAN Software Engineering Symp. on Practical Software Development Environments, Palo Alto, CA, USA, Dec. 1986.

[GTC*90] Gibbs S., Tsichritzis D., Casais E., Nierstrasz O., Pintado X. *Class Management for Software Communities*. Communications of the ACM, Vol. 33, No. 9, pp. 90-103, Sep. 1990.

[HK94] Hahn U., Klernner M. *Tracking the Evolution of Concepts in Dynamic worlds*. Proc. of the 5th DEXA'94 Conf., Athens, Greece, Sep. 1994.

[HTY89] Hull R., Tanaka K., Yoshikawa M. *Behavior Analysis of Object-Oriented Databases : Method Structure, Execution Trees, and Reachability*. Third Int. Conf. on Foundations of Data Organisation and Algorithms, Paris, Jun. 1989.

[HY95] Hong E., Yoo S. *A schema Evolution Mechanism*. Proc. of the 6th DEXA'95 Conf., London, Sep. 1995.

[HZ90] Heiler S., Zdonik S. *Object Views : Extending the Vision*. Proc. of the Int. Conf. on Data Engineering, pp. 86-93, Feb. 1990.

[Ita93] *Itasca Systems*, Inc. Itasca Systems technical report Number TM-92-001. OODBMS Feature Checklist. Rev.11, Dec. 1993.

[JF88] Johnson R, Foote B. *Designing Reusable Classes*. Journal of Object-Oriented Programming, Jul. 1988.

[Jos91] Joseph and al. *Object-Oriented Databases : Design and Implementation*. Proc. of the IEEE, Vol. 79, No. 1, Jan. 1991.

[KC88] Kim W., Chou H. *Versions of Schema for Object-Oriented Databases*. Proc. of the 14th VLDB Conf., Sep. 1988.

[Ken79] Kent W. *Limitations of record-based information models*. ACM TODS, 1979.

[KF91] Konomi S., Furukawa T. *Updating Duplicate values in Distributed Multi-database Systems*. Proc. of the 1st Int. IEEE Workshop On Interoperability in Multi-database syst., pp. 243-246, Apr. 1991.

[Kim89] Kim W. *Evolutionary Database Design*. Proc. of the Data Engineering Conf., pp. 618-624, Feb. 1989.

[Kim90] Kim W. *Research Directions in Object-Oriented Database Systems*. Proc. of the 9th ACM PODS, 1-15, 1990.

[KK87] Kim H., Korth H. *PSYCHO : a graphical language for supporting schema evolution in object-oriented databases*. Technical Report, University of Texas, TR-87-43, 1987.

[KK90] Kim H., Korth H. *Schema versions and views in Object-Oriented databases*. Proc.of Info Japan, Tokyo, Oct. 1990.

[KKM91] Kemper A., Kilger C., Moerkotte G. *Function Materialization in Object Bases*. Proc. of the ACM SIGMOD Int. Conf. Management of Data, pp. : 258-267, 1991.

[KL89] Kim W., Lochovsky. *Object-Oriented Concepts, Applications, and Databases*. Addison-Wesley, 1989.

[Kos91] Kosky A. *Modelling and Merging Database Schemas*. Technical Report, University of Pennsylvania, 1991.

[Ler93] Lerner B. *Extending the Notion of Type Conformance to Interfaces and Type Systems*. Position Paper for OOPSLA'93 Workshop on Supporting the Evolution of Class Definitions Jun. 1993.

[LH89a] Lieberherr K., Holland I. *Formulations and Benefits of the Law of Demeter*. SIGPLAN Notices, Vol.24, No.3, ACM, Mar. 1989.

[LH89b] Lieberherr K., Holland I. *Assuring Good Style for Object-Oriented Programming*. IEEE Software, Sep. 1989.

[LH90] Lerner B., Habermann A. *Beyond Schema Evolution to Database Re-Organisation*. Proc. of the ECOOP/OOPSLA'90, Ottawa, Oct. 1990.

[LHR88] Lieberherr K., Holland I., Riel A. *Object-Oriented Programming: an Objective Sense of Style*. SIGPLAN Notices, Vol.23, No.11, ACM, Nov. 1988.

[Lib92] Libourel T. *Introduction des relations pour exprimer l'évolutivité dans un système d'objets*. Thèse de doctorat, Université de Montpellier II, Mai 1992.

[Lie92] Lieberherr K. *Controlling the Evolution of Object-Oriented Applications*. Technical Report, North-eastern University, College of Computer Science, Boston, MA, Jan. 1992.

[Liu94] Liu C. *Database Schema Evolution through the Specification and Maintenance of Changes on Entities and Relationships*. PhD thesis, University of Pittsburgh, Apr. 1994.

[LM88] Li Q., McLeod M. *Object Flavor Evolution through Learning in an Object-Oriented Database System*. Proc. of the 2nd Int. Conf. on Expert Database Systems, pp. 241-256, Apr. 1988.

[LM89] Li Q., McLeod M. *Conceptual Database Evolution Through Learning*. In Object-Oriented Databases Applications (Gupta R., Horiwitz E. editors), Prentice-Hall, 1989.

[LT94] Li X., Tari Z. *Consistency Checking of Evolving Methods*. Proc. of the 5th DEXA'94 Conf., Athens, Greece, Sep. 1994.

[LX93] Lieberherr K., Xiao C. *Minimising Dependency on class Structures with Adaptive Programs*. Proc. of the Int. Symp. on Object Technologies for Advanced Software, Kanazawa, Japan, Springer Verlag, 1993

[LW94] Li C., Wang S. *Efficient Storage Structures for Temporal Object-Oriented Databases*. Proc. of the 5th DEXA'94 Conf., Athens, Greece, Sep. 1994.

[LWX93] Lieberherr K., Walter L, Xiao C. *Object-Extending Class Transformations*. Int. Journal of Formal Methods, 1993.

[MBP95] Milliner S., Bouguettaya A., Papazoglou M. *A Scalable Architecture for Autonomous Heterogeneous Database Interactions*. Proc. of the 21th VLDB Conf., Zurich,, Sep. 1995.

[McL88] McLeod D. *A Learning-Based Approach to Meta-Data Evolution in an Object-Oriented Database*. LNCS, No. 334, Springer Verlag, Advances in Object-Oriented Database Systems, 2nd Int. Workshop on Object-Oriented Database Systems, pp. 219-224, Sep. 1988.

[Mei95] Meier A. *Providing Database Migration Tools : A Practitioner's View*. Proc. of 21th VLDB Conf., Zurich, Sep. 1995.

[Mey88] Meyer B. *Object-Oriented Software Construction*. Prentice-Hall International, 1988.

[MIR94] Miller A. Ioannidis Y., Ranakrishnan. *The use of information capacity in schema integration and translation*. Proc. of the 19th VLDB Conf., Dublin, Ireland, 1993.

[MMW94] Mendelzon A., Milo T., Waller E. *Object Migration*. PODS'94.

[MNC*89] Masini G., Napoli A., Colnet D., Léonard D., Tomdre K. *Les langages à objets*. Inter Editions, 1989.

[MNS94] Morsi M., Navathe S., Shilling J. *On Behavioral Schema Evolution in Object-Oriented Databases*. Int. Conf. on Extending Data Base Technology, Cambridge, Mar. 1993.

[Mon93] Monk S. *A model for Schema Evolution in Object-Oriented Database systems*. PhD thesis, Lancaster University, Feb. 1993.

[Mor79] Morrison R. *On the developement of algol*. PhD thesis, Department of Computational Science, University of St. Andrews, 1979.

[Mor92] Morsi M. *Extensible Object-Oriented Database with Dynamic Schemas*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, Sep. 1992.

[MS90] McKenzie E., Snodgrass R. *Schema Evolution and Relational Algebra*. Information Systems Journal, Vol. 15, No. 2, pp. 207-232, 1990.

[MS93] Monk S., Sommerville I. *Schema Evolution in OODBs Using Class Versioning*. SIGMOD RECORD Vol. 22, No. 3, Sep. 1993

[Mun93] Munich B. *Versioning in a software engineering database-the change oriented way.* PhD thesis, University of Trondheim, Norway, 1993.

[MZ93] Moekotte G., Zachmann A. *Towards More Flexible Schema Management in Object-Bases.* Proc. Int.. Conf. on Data Engineering, pp. 174-181, Wien, Austria, IEEE, 1993.

[Nac94] Nacer M. *Un modèle de gestion et d'évolution de schéma pour les bases de données de Génie Logiciel.* Thèse de Doctorat, Institut National Polytechnique, Grenoble, Jui. 1994.

[NB88] Narayanaswamy K., Bapa Rao K. *An Incremental Mechanism for Schema Evolution.* Proc. of the 4th Int. Conf. on Data Engineering, pp. 294-301, Los Angles, CA, USA, IEEE, 1988.

[NR89a] Nguyen G., Rieu D. *Schema Evolution in Object-Oriented Databases.* Data and Knowledge Engineering, Vol. 4, No. 1, 1989.

[NR89b] Nguyen G., Rieu D. *Schema Change Propagation in Object-Oriented Databases.* Information Processing'89, North-Holland Publisher, pp. 815-820, San Francisco, USA, 1989.

[Ob93] Objectivity Inc. *Objectivity, User Manual.* Version 2.0, Mar. 1993.

[OD93] Object Design Inc. *ObjectStore User Guide.* Release 3.0, chapter 10, Dec. 1993.

[Odb94a] Odberg E. *Schema Modification Management for Object-Oriented Databases.* Proc. of the 6th CAISE'94 Conf., Jun. 1994.

[Odb94b] Odberg E. *MultiPerspectives : The classification Dimension of Schema Modification Management for Object-Oriented Databases.* Proc. of the Tools USA'94 Conf., Aug. 1994.

[Odb95] Odberg E. *Multiperspectives : Object Evolution and Schema Modification Management in Object-Oriented Databases.* PhD thesis, Norwegian Institute of Technology, Feb. 1995.

[Opd92] Opdyke W. *Refactoring Object-Oriented Frameworks.* PhD Thesis, University of Illinois at Urbana-Champaign, 1992.

[Ora] Oracle. *The ORACLE Database Administrator's Guide.* Oracle part No 3601.

[Osb89] Osborn S. *The Role of Polymorphism in Schema Evolution in an Object-Oriented Database* IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 3, pp. 310-317, Sep. 1989.

[OT94] O2 Technology. *The O2 User Manual.* Version 4.5, Nov. 1994.

[OT95] O2 Technology. *The O2 User Manual.* Version 4.6, Sep. 1995.

[Pal89] Palisser C. *Charly, un gestionnaire de versions pour la CAO en architecture.* Thèse de Doctorat de l'Université d'Aix-Marseille III, Nov. 1989.

[PO95] Peters R., Tamer Ozsu M. *Axiomatization of Dynamic Schema Evolution in ObjectBases.* Proc. 11th Int. Conf. on Data Engineering (ICDE'95), Taiwan, Mar. 1995.

[PS87] Penny D., Stein J. *Class Modification in the GemStone Object-Oriented DBMS.* Proc. of the ACM OOPSLA Conf., Sep. 1987.

[Rod91] Roddick J. *Dynamically Changing schemas within database models.* Australian Computer Journal, Vol. 23, No.3, pp. 105-109, 1991.

[Rod92a] Roddick J. *SQL/SE- A query language extension for database supporting schema evolution*. SIGMOD record, Vol. 21, No. 3, 1992.

[Rod92b] Roddick J. *Schema Evolution in Database Systems - An Annotated Bibliography*. Technical report, CIS-92-004, School of Computer and Information Science, University of South Australia, 1992.

[Rod94] Roddick J. *A survey on Schema Versioning Issues for Database Systems*. School of Computer and Information Science, University of South Australia, 1994.

[Ron94] Roncancio C. *Interopérabilité en SGBD : systèmes fédérés et systèmes multibases*. Technique et Science Informatiques, Vol.13, No. 3/1994, pp. 385-419, 1994.

[Rou91] Roussopoulos N. *An Incremental Access Method for ViewCache : Concept, Algorithms and Cost Analysis*. ACM Trans. Database Syst., Vol.16, No.3, pp. 535-563, 1991.

[RR94] Ra Y., Rundensteiner E. *A transparent Object-Oriented Schema Change Approach Using View Evolution*. Technical Report CSE-TR-211-94, Dept. of EECS, Univ. of Michigan, Feb. 1994.

[RS87] Rowe L., M. Stonebraker. *The POSTGRES Data Model*. Proc. of the 13th VLDB Conf., Brighton, pp.9-96, 1987.

[RT84] Reps T., Teitelbaum T. *The Synthesizer Generator*. Software Engineering Notes, 9(3), Mai 1984.

[Run92] Rundensteiner E. *MutiView : A Methodology for Supporting Multiple View Schemata in OODBs*. Proc. of the 18th VLDB Conf., pp. 187-198, Aug. 1992.

[RW92] Reed D., Wyant G. *How safe is C++*. Journal of Object-Oriented programming, May 1992.

[Sch93] Schiefer B. *Supporting Integration & Evolution with Object-Oriented Views*. FZI-Report 15/93, Jul. 1993.

[Sci94] Sciore E. *Versioning and configuration management in an object-oriented data model*. VLDB journal, 3(1):77-106, 1994.

[Sch94] Scholl P. *Modèles de données à objets : concepts et terminologie*. Note interne, Avr.1994.

[SR88] Srivastava J., Rotem D. *Analytical Modelling of Materialized View Maintenance*. Proc. ACM Symp. on Principles of Database Syst., pp. 126-134, 1988.

[SD91] Scharenberg M, Dunsmore H. *Evolution of classes and objects during object-oriented design and programming*. Journal of Object-Oriented Programming, pp. 30-34, Jan. 1991.

[SDA94] Souza dos Santos C., Delobel C., Abiteboul S. *Virtual Schemas and Bases*. Int. Conf. on Extending Data Base Technology, Cambridge, Mar. 1993.

[SF93] Segal M., Frieder O. *On-the-Fly Program Modification : Systems for Dynamic Updating*. IEEE Software, pp. 53-65, Mar. 1993.

[SGD93] Scherrer S., Geppert A., Dittrich K. *Schema Evolution in NO2*. Zurich University, Technical Report No. 93.12, Apr. 1993.

[Sjo93a] Sjoberg D. *Thesaurus-Based Methodologies and Tools for maintaining Persistent application Systems*. PhD thesis, University of Glasgow, Jun. 1993.

[Sjo93b] Sjoberg D. *Quantifying Schema Evolution*. Information and Software Technology Journal, Vol. 35, No. 1, pp. 35-44, Jan. 1993.

[SK91] M. Stonebraker, Kemnitz G. *The POSTGRESS Next-Generation Database Management System*. Communications of the ACM, 34(10), pp. 78-92, 1991.

[SN88] Schrefl R., Neuhold E. *Object Class definition by generalization using upward inheritance*. The 4th Int. Conf. on Data Engineering, pp. 4-13, IEEE Computer Society Press, 1988.

[ST94] Scholl M., Tresch M. *Evolution towards, in, and beyond Object Databases*. Proc. 3rd GI Workshop Information Systems and Artificial Intelligence, Humbury, Springer Verlag, LNCS 777, pp. 64-82, Mar. 1994.

[Ste89] Stein L. *Towards a Unified Method of Sharing in Object-Oriented Programming*. Proc. of the Workshop on Inheritance and hierarchies in Knowledge Representation and Programming Languages, Viagreggio, Feb. 1989.

[Su91] Su J. *Dynamic contraints and object migration*. Proc. of the 17th VLDB Conf., pp. 233-242, Barcelona, Spain, 1991.

[SZ86] Skarra A., Zdonik S. *The Management of Changing Types in an Object-Oriented Database* Proc. of the OOPSLA'86 Conf., Sep.1986.

[SZ89] Stein L., Zdonik S. *Clovers : The dynamic behavior of type and instances*. Technical report, No. CS-89-42, Brown Universiy, 1989.

[TOC93] Talen G., Oussalah O., Colinas M. *Versions of Simple and Composite Objects*. Proc. of the 17th VLDB Conf., Dublin, Ireland, Sep. 1993.

[Tay95] Tayar N. *Gestion des versions pour la construction incrémentale et partagée de bases de connaissances*. Thèse de Doctorat, Université Joseph Fourier, Grenoble1, Sep. 1995.

[TK91] Tan L., Katayama T. *Meta Operations for type Management in Object-Oriented Databases*. Proc. of the 1st Intl. Conf. on Deductive and Object-Oriented Databases (DOOD), Kyoto, Japan, 1991.

[Tre91]Tresch M. *A Framework for Schema Evolution by Meta Object Manipulation*. In Proc. 3rd Int. Workshop on Foundations of Models and Languages for Data and Objects, Aigen, Austria, Sep. 1991.

[TS92] Tresch M., Scholl M. *Meta object management and its application to database evolution* Proc. of the 11th Int., Entity-Relationship Approach Conf., pp. 299-321, 1992

[TS93a] Tresch M., Scholl M. *Schema transformation without database reorganization* ACM SIGMOD Record 22(1) : 21-27, Mar. 1993.

[TS93b] Tresch M., Scholl M. *Schema transformation processors for federated objectbases*. Proc. of the 3rd Int. Symp. on Database Systems for Advanced Applications (DAFAA), Daejon, Korea, Apr. 1993.

[Ull88] Ullman J. *Principles of database and knowledge-base systems*. Vol. 1, Computer Science Press, 1988.

[Ver92] Versant Object Technology. *Versant User Manual.* 1992.

[VH91] Ventore R., Heiler S. *Semantic Heterogeneity as a result of domain evolution.* SIGMOD Record, Vol. 20, No.4, pp. 16-20, 1991.

[Wal93] Waller E. *Méthodes et bases de données.* Thèse de Doctorat, Université de Paris-Sud, Centre d'Orsay, Jui. 1993.

[Wal91] Waller E. *Schema updates and consistency.* Proc. of the 2nd Int. Conf. on Deductive and Object-Oriented Databases, Munich, 1991.

[WZ88] Wegner P., Zdonik S. *Inheritance as an incremental modification mechanism or what like is and isn't like.* Proc. of the ECOOP, LNCS No. 322, pp. 55-77, Springer-Verlag, Oslo, Aug. 1988.

[Zdo86a] Zdonik S. *Maintaining consistency in a Database with changing types.* SIGPLAN Notices, Vol. 21, No. 10, pp. 120-127, Oct. 1986.

[Zdo86b] Zdonik S. *Version Management in an Object-Oriented Database.* Proc. of the IFP WG2.4 Int. Workshop on Advanced Programming Environments, Trondheim, Norway, Jun. 1986.

[Zdo90] Zdonik S. *Object-Oriented type evolution.* Advances in Database Programming languages, chapter 16, pp. 277-288. (Bancilhon F., Bunema P. editors), ACM press Nework, 1990.

[Zic92] Zicari R. *A Framework for Schema Updates in an Object-Oriented Database System.* Building an Object-Oriented Database System; the Story of O2, chapter 7, (Bancilhon F., Delobel C., Kanellakis P., editors), Morgan Kaufmann 1992.

[ZM90] Zdonik S., Maier D. *Fundamentals of Object-Oriented Databases.* In Readings in Object-Oriented Database Systems, pp. 1-32. The Morgan Kaufmann Series in Data Management Systems, 1990.

Annexes

A. Taxinomie

Nous donnons dans cette section l'ensemble des opérations du changements de schéma.

(1) Changement d'une classe

(1.1) `Add_Attribute` (c, a, t)
{Cette primitive ajoute l'attribut a dont le type est t à la définition de la classe c}

(1.2) `Del_Attribute` (c, a)
{Cette primitive supprime l'attribut a de la définition de la classe c}

(1.3) `Update_Attribute` (c, a, t)
{Cette primitive remplace dans la classe c, le type de l'attribut a par le type t}

(1.4) `Add_Operation` (c, op, s)
(1.5) `Del_Operation` (c, op)
(1.6) `Update_Operation` (c, op, s)

{Ce sont des instructions identiques aux précédentes, mais qui s'appliquent aux opérations. op est le nom de l'opération et s sa signature}

(2) Changement du graphe d'héritage

(2.1) `Add_Class` (c, def)
{Cette primitive ajoute la classe c dans le schéma. def est la définition de cette classe}

(2.2) `Add_Edge` (c1, c2)
{Cette primitive ajoute un lien d'héritage entre les classes c1 et c2. c1 devient une super-classe de c2}

(2.3) `Del_Edge (c1,c2)`
{Cette primitive supprime le lien d'héritage qui existe entre les classes c1 et c2. Après cette opération c1 n'est plus une super-classe de c2}

(2.4) `Del_Class(c)`

{Cette primitive supprime la classe c du schéma}

B. Syntaxe du langage d'évolution du schéma

<évolution_schéma> : : "**schema_evolution**"<nom_schéma> <spécifications>

<spécifications> : : [<mode_évolution>]<opérations>[<descripteurs>]

<mode_évolution> : : "**mode**" ":" <mode>

<opérations> : : "**operations**""<" <opération;><opération;>*">"

<descripteurs> : : "**descriptors**" "<"<descripteur;><descripteur;>*">"

<mode> : : "**version**" | "**modification**"

<opération> : : "**Add_Attribute** "("<nom_classe> "," <nom_attribut> "," <type>")"

| "**Del_Attribute**" "("<nom_classe> "," <nom_attribut>")"

| "**Update_Attribute**" "("<nom_classe> "," <nom_attribut> ","

<nouveau_type>")"

| "**Add_Operation**" "("<nom_classe> "," <nom_opération> "," <signature>")"

| "**Del_Operation**" "("<nom_classe> "," <nom_opération>")"

| "**Update_Operation**" "("<nom_classe> "," <nom_opération> ","

<nouvelle_signature>")"

| "**Add_Class**" "("<nom_classe> "," <définition_classe>")"

| "**Del_Class**" "("<nom_classe>")"

| "**Add_Edge**" "("<nom_classe> "," <nom_classe>")"

| "**Del_Edge**" "("<nom_classe> "," <nom_classe>")"

/ la clause <descripteur> est décrite dans le chapitre 5 */*

C. Transformation par défaut

Dans cette section, nous décrivons le mécanisme de*transformation par défaut* [LH90, Ben92, FMZ95a] qui est utilisé par les opérations de manipulation de versions des objets. Il est basé sur la comparaison des structures de deux classes. Etant données deux classes classe1, classe2 et un objet o. Le mécanisme de transformation par défaut est utilisé pour :

• Dériver une nouvelle version de o sous classe2 à partir de sa version sous classe1.
• Convertir la version de o sous classe1 vers classe2.

La valeur de la version de o sous la classe classe2 est initialisée comme suit :
• La valeur d'un attribut qui est défini seulement dans classe2 est initialisée par la valeur par défaut du type de l'attribut (Cf. Tableaux b.1 et b.2). Par exemple, supposons que l'attribut att : integer est défini seulement dans classe2. Dans ce cas, la valeur de att dans la version de o sous classe2 est égal à 0.

• La valeur associée à un attribut présent dans les deux classes classe1 et classe2 est déduite selon la relation qui existe entre les types de l'attribut dans les deux classes (Cf. Table b.3). Par exemple, supposons que : (1) le type de l'attribut att dans classe1 est tuple (no : integer, rue : string, ville : string) et le type de att dans classe2 est tuple(no : integer, rue : string), (2) la valeur de att dans la version de o sous classe1 est [no : 83, rue : "cité heureuse", ville : "Saida"]. Dans ce cas, la valeur de att sous classe2 est [no : 83, rue : "cité heureuse"].

Table b.1 : valeur par défaut d'un type atomique/classe. Pour simplifier, nous considérons seulement quelques types atomiques.

type	integer	real	boolean	char	string	nom_classe
valeur par défaut	0	0	false	" "	" "	nil

Table b.2 : valeur par défaut d'un type construit. Pour simplifier, nous considérons seulement les constructeurs ensemble et n-uplet. Dans la table, l'expression vpd(t) est la valeur par défaut du type t.

type	set(t)	tuple (a1: t1, a2 : t2, ..., an : tn)
valeur par défaut	{ }	[a1 : vpd(t1), a2 : vpd(t2), ..., an : vpd(tn)]

Table b.3 : transformation par défaut. La ligne de la table décrit le type de l'attribut dans la classe classe2 et la colonne décrit le type de cet attribut dans la classe1.

	integer	real	boolean	char	string	nom_classe	set	tuple
integer	id	id						
real		id						
boolean			id					
char				id				
string					id			
nom_classe						id (<t)		
set							tds	
tuple								tdt

• **id (identité)** : la valeur de l'attribut dans la version de o sous classe2 est la même que sa valeur dans la version de o sous classe1.

• **id(<t) (identité conditionnelle)** : si le type de l'attribut dans classe1 est un sous-type de son type dans classe2, alors la valeur de la l'attribut dans la version de o sous classe2 est la même que sa valeur dans la version de o sous classe1. Sinon, elle est initialisée par la valeur nil.

• **tds (transformation par défaut d'un type ensemble)** : la valeur de l'attribut dans la version de o dans classe2 est déduite récursivement en utilisant le mécanisme de transformation par défaut. La transformation dépend des types des constructeurs ensemble dans les deux classes.

• **tdt (transformation par défaut d'un type n-uplet)** : les valeurs des attributs du n-uplet dans la version de o sous classe2 sont déduites récursivement à partir du n-uplet dans la version de o dans classe1, en utilisant le mécanisme de transformation par défaut.

• **aucune indication** : si la case correspondante dans la table est vide, alors la valeur de l'attribut dans la version de o sous classe2 est la valeur par défaut associée à son type.