Université Henri Poincaré – Nancy I

Département de Formation Doctorale en Informatique **École Doctorale IAE + M**

Intégration, Interopérabilité et Evolution des Systèmes de Données : Application aux environnements logiciels

MÉMOIRE

présenté et soutenu publiquement le lundi 2 octobre 1995

pour l'obtention de l'

Habilitation à Diriger des Recherches
de l'Université Henri Poincaré – Nancy I
(Spécialité Informatique)

par

Nacer BOUDJLIDA

Composition du jury

Président :	Mr Jean-Pierre FINANCE
Rapporteurs :	Mme Marie-Christine HATON Mr Mokrane BOUZEGHOUB Mr Eric SIMON
Examinateurs :	Mr Jean-Jacques CHABRIER Mr Jean-Claude DERNIAME

Centre de Recherche en Informatique de Nancy

Remerciements

Je tiens à remercier, en premier lieu et à double titre Mr Jean-Pierre Finance, professeur, président de l'université Henri Poincaré Nancy 1 : il m'a fait l'honneur et l'amitié de présider le jury, d'une part, et d'autre part, il a été en partie, à l'origine de ma venue à Nancy.

Je remercie également, Mme Marie-Christine Haton, professeur à l'université Henri Poincaré Nancy 1 d'avoir accepté de juger ce travail, d'en rendre compte et de siéger dans son jury.

Je remercie profondément Mr Mokrane Bouzeghoub professeur à l'université de Versailles, qui a aussi accepté de juger ces travaux et de siéger dans son jury : ses nombreuses remarques et critiques m'ont aidé à en améliorer la présentation.

Mr Eric Simon, directeur de recherches à l'INRIA, a également été un lecteur et un rapporteur critique de ce mémoire. Des "ennuis mécaniques" l'ont empêché en dernière minute de prendre part au jury : je le remercie pour ses critiques, son compte rendu de mes travaux et . . . pour ses efforts (hélas vains) pour tenter d'être présent à leur présentation publique.

J'ai du mal à dissocier dans ma reconnaissance et mes remerciements, Mr Jean-Jacques Chabrier, professeur à l'université de Bourgogne et diresteur du Centre de Recherches en Informatique de Dijon, et Mr Jean-Claude Derniame, professeur à l'Institut Naational Polytechnique de Lorraine et directeur de l'équipe Génie Logiciel du Centre de Recherches en Informatique de Nancy où ces travaux se sont déroulés. L'un et l'autre m'ont, pendant toute la durée de ces travaux, apporté leur écoute, leurs conseils et, hors du domaine professionnel, leur amitié : je les en remercie très vivement.

J'associerai également Mme Marion Créhange, professeur à l'université Nancy 2, à la réussite de ces travaux : elle m'a accueilli dans son équipe de recherche lors de "mes premiers pas dans la recherche à Nancy". Je lui en suis profondément reconnaissant.

Je n'oublie pas que ces travaux ont avancé, notamment, grâce à la contribution de collègues de l'équipe génie logiciel et plus particulèrement grâce à Mrs Tahar Khammaci, maitre de conférences à l'université de Nantes, Olivier Perrin, maitre de conférences à l'université de Nancy 2 et Mourad Bouneffa, maitre de conférences à l'université du Littoral, dont j'ai eu le plaisir à diriger les travaux qui ont conduit à leurs thèses d'université.

Enfin, il y a Catherine, Jimmy, Titi, Jacqueline, Pascal, Marie-José, Fasia, Guy, Jeanine, Didier, Annie, Bernard, Françoise, "Caniche" et les autres, qui savent assez pourquoi je les remercie (N'est-ce-pas Catherine ?).

"Last but not least", comment pourrais-je ne pas associer ma famille, dont je sentais l'amour et le soutien de l'autre côté de la Méditerranée.

Il s'appelait Magid,
Tout le monde l'appelait Hocine.
Je l'appelais "Cheikh" *(mon vieux),*
Il était Mon Père.

Table des matières

Partie I Le décor : problématique et domaines d'application 1

Chapitre 1 Intégration et évolution de systèmes logiciels 3

1.1 Introduction : De l'univers du discours 3
1.2 Environnements de développement de logiciels et intégration 5
1.2.1 Notion d'environnement 5
1.2.2 Les dimensions de l'intégration 9
1.3 Evolution des environnements intégrés 11
1.3.1 Evolution de la collection d'outils d'un environnement 11
1.3.2 Evolution des modèles de procédés 11
1.3.3 Evolution des objets logiciels 12
1.3.4 Conclusion 13
1.4 Interopérabilité et polymorphisme 13
1.4.1 Interopérabilité orientée processus 13
1.4.2 Interopérabilité orientée objets 14
1.5 Evolution des programmes 14
1.6 Interopérabilité et évolution dans les bases de données 15
1.6.1 Fédération et répartition 16
1.6.2 SGBD extensibles 17
1.7 Interopérabilité et évolution dans les systèmes à bases de connaissances 21
1.7.1 Evolution des bases de connaissances 21
1.7.2 Interopérabilité dans les systèmes de raisonnement distribué 22
1.8 Structure et contenu du document 23

Partie II Environnements intégrés fondés sur les modèles de procédés 27

Chapitre 2 Environnements intégrés et modèles de procédés 29

2.1 Introduction 29

2.2 The ALF Concepts for Process Modeling . . . 31
2.2.1 Process Modeling Concepts . . . 32
2.2.2 Structuring Mechanisms and Component Relationships . . . 32
2.3 The Software Process Description Language . . . 33
2.3.1 Foundations of the MASP/DL . . . 33
2.3.2 Discussion . . . 37
2.4 The ALF system . . . 37
2.4.1 The User Interface Management System . . . 40
2.4.2 The Information System and the MASP Management Toolset . . . 40
2.4.3 The Piloting System Layer and the MASP Enaction Toolset . . . 42
2.4.4 The executive system layer . . . 43
2.5 Concluding remarks . . . 44

Partie III Evolution, Intégration et Interopérabilité Orientée Objets 47

Avant propos : Evolution et Intégration de schémas 49

Chapitre 3 GORM : un modèle d'abstraction de données 53
3.1 Caractéristiques du modèle . . . 53
3.2 Les types de base . . . 54
3.2.1 Définition d'un type de base : *DefTypBase* . . . 54
3.2.2 Instance d'un type de base . . . 55
3.3 La relation d'attribut . . . 55
3.3.1 Définition d'un type d'attribut : *DefTypAttribut* . . . 55
3.3.2 Instance d'un type d'attribut : *InstanceAttribut* . . . 56
3.4 La relation de composition . . . 56
3.4.1 Définition d'une composition : *DefComposition* . . . 57
3.4.2 Instance d'une composition : *InstanceComposition* . . . 58
3.5 La relation de sous-type . . . 58
3.5.1 Définition d'une relation de sous-type : *DefSousType* . . . 58
3.5.2 Instance d'une relation de sous-type : *InstanceSousType* . . . 59
3.6 Les types d'entité . . . 59
3.6.1 Définition d'un type d'entité : *DefTypEntité* . . . 59
3.6.2 Instance d'un type d'entité : *InstanceEntité* . . . 60
3.7 Les types d'association . . . 61
3.7.1 Définition d'un type d'association : *DefTypAssociation* . . . 61
3.7.2 Instance d'un type d'association : *InstanceAssociation* . . . 62

3.8 Le type schéma 63
3.8.1 Définition d'un type de schéma : *DefTypSchéma* 63
3.8.2 Instance d'un type de schéma : *InstanceSchéma* 63
3.8.3 Instance d'une base de données 64
3.9 Types et opérations 64
3.10 Critères de qualité d'un type de schéma 64
3.11 Conclusion 65

Chapitre 4 Opérations de modification et de restructuration de schémas 67
4.1 La modification de schémas 67
4.1.1 Typologie des opérations de modification 67
4.1.2 Les opérations de modification de schémas : récapitulatif 70
4.1.3 Conclusion 70
4.2 Intégration de schémas et opérations de restructuration 71
4.2.1 Phase 1 : les conflits de représentation et leur détection 72
4.2.2 Phase 2 : la résolution de conflits et les opérations de restructuration 74
4.2.3 Phase 3 : la fusion de schémas 75
4.2.4 Intégration de schémas et invariants de schémas 77
4.2.5 Conclusion 77
4.3 Conclusion : Notion de transaction de modification de schéma 77

Chapitre 5 La Restructuration des données 79
5.1 Versions de schémas et d'objets 79
5.1.1 Pas de version et migration immédiate 80
5.1.2 Versions de schémas et d'objets 81
5.1.3 Conclusion 83
5.2 Versions de schémas et homomorphismes 84
5.2.1 Morphismes d'entités 84
5.2.2 Morphismes d'associations 87
5.2.3 Conclusion 89
5.3 Fonctions de correspondance et modification de schémas 89
5.3.1 Les opérations de renommage 91
5.3.2 Les opérations de changement de domaines 91
5.3.3 Les opérations de changement de structure 104
5.3.4 Fonctions de correspondance et modification de schémas : conclusion . . . 110
5.4 Fonctions de correspondance et intégration de schémas 110
5.4.1 Fonctions de correspondance et restructuration de schémas 110

5.4.2 Fonctions de correspondance et fusion de schémas 111
5.5 Conclusion 111

Chapitre 6 Interopérabilité orientée objets 113
6.1 Introduction 113
6.1.1 Dimensions de l'intégration : rappel 113
6.1.2 Caractéristiques de l'approche 115
6.2 L'intégration de données par l'exemple 116
6.3 Fondements de l'intégration de données 117
6.4 Modèle de données et règles de restructuration 119
6.4.1 Modèle de représentation de données 119
6.4.2 Règles de réécriture des structures 120
6.5 Un aperçu sur l'implémentation 122
6.6 Discussion 123
6.6.1 Granularité des objets et intéropérabilité 123
6.6.2 Données + Contrôle 125

Partie IV Conclusion et perspectives 127

Chapitre 7 Conclusion et perspectives 129
7.1 Intégration de schémas et intégration de données 129
7.2 Evolution de schémas et transactions 130
7.3 Les mécanismes d'intégration dans la migration de "systèmes du patrimoine" . . . 130
7.4 Contraintes d'intégrité dans le processus d'intégration 132

Partie V Annexes 135

Chapitre 1 Exemples de sessions dans un environnement fondé sur la modélisation de procédés 137

Chapitre 2 Les opérations de modification de schémas 143
2.1 La modification des définitions des types de base 143
2.2 La modification des définitions des types d'attributs 144
2.3 La modification de la relation de sous-type 145
2.4 La modification des compositions 146
2.4.1 Modification d'un composant d'une composition 146
2.4.2 Modification de la liste de composants d'une composition 147

2.5 La modification des types d'entité 147
2.5.1 Opérations sur un type d'entité 147
2.5.2 Opérations sur l'ensemble des types d'entité d'un type de schéma 148
2.6 La modification des types d'association 149
2.6.1 Opérations sur un type d'association 149
2.6.2 Opérations sur l'ensemble des types d'association 150
2.7 La modification d'un type de schéma 151

Chapitre 3 Les règles de ré-écriture de structures **153**

Bibliographie **157**

Table des figures

1.1 Scénarii d'intégration par les données 4
1.2 Approche naïve de l'intégration des données 5
1.3 Encapsulation des convertisseurs de données 6
1.4 Intégration forte "autour" d'une structure commune 6
1.5 Référentiel unique et convertisseurs 6
1.6 Composants d'un environnement fondé sur les modèles de procédés 8
1.7 Les niveaux de modèles et d'instances 9
1.8 Interopérabilité orientée processus vs interopérabilité orientée objets 14
1.9 Scénario d'implantation de nouveaux types dans un SGBD 18
1.10 Communication dans les systèmes multi-agents 23

2.1 Integration Layers 30
2.2 Purpose of the Model 31
2.3 Life Cycle Coverage 31
2.4 Role Coverage 31
2.5 Structuring Mechanisms 33
2.6 Part of the ISK Object Schema 38
2.7 ALF System Toolset 39
2.8 ALF System Structure 40

9 Mise à jour de données et modification de schémas 51

3.1 Représentation graphique et exemple de définition d'un type de base 55
3.2 Représentation graphique et exemple de définition d'un type d'attribut 56
3.3 Lien de composition vs lien d'attribut 57
3.4 Représentation graphique et exemple de définition d'une composition 57
3.5 Schématisation d'un arbre d'héritage 59
3.6 Représentation graphique des définitions de types d'entité 60
3.7 Représentation graphique de la définition du type d'entité EQUIPE 60
3.8 Représentation graphique d'une instance d'un type d'entité 61
3.9 Représentation graphique des définitions des types d'association 62
3.10 Représentation graphique des définitions des types de schéma 63
3.11 Schématisation de spécifications d'opérations 65

4.1 Liste des opérations de modification d'une spécification de schéma (1) 71
4.2 Liste des opérations de modification d'une spécification de schéma (2) 72
4.3 Une typologie des différences de représentation dans des schémas 73
4.4 Restructuration d'un type de base en un type d'entité (1) 75

4.5 Restructuration d'un type de base en un type d'entité (2) . . . 75
4.6 Fusion de deux types d'entité . . . 76
4.7 Fusion par généralisation . . . 76
4.8 Fusion de deux types d'association . . . 76
4.9 Violation de l'héritage simple après intégration . . . 77

5.1 Trois versions de schéma . . . 80
5.2 Schématisation d'une version "primaire" d'objet . . . 83
5.3 Trois versions de schéma (rappel) . . . 86
5.4 Typologie des opérations de modification de schémas . . . 90
5.5 Exemple de schéma . . . 90
5.6 Calcul de correspondances d'identifiants et relation de sous-type . . . 94
5.7 Changement de domaine et relation de sous-type (1) . . . 99
5.8 Changement de domaine et relation de sous-type (2) . . . 99
5.9 Changement de domaine et relation de sous-type (3) . . . 99
5.10 Modification de la relation de sous-type . . . 100
5.11 Ajout de type d'entité et relation de sous-type . . . 108
5.12 Suppression de type d'entité et relation de sous-type . . . 109
5.13 Suppression de type d'entité et relation de sous-type . . . 110
5.14 Restructuration d'un type d'attribut en un type d'entité . . . 111
5.15 Fusion de deux types d'entités (rappel) . . . 111

6.1 Exemple d'environnement . . . 114
6.2 Fondements du processus d'intégration par les données . . . 117
6.3 Exemple de structure de données . . . 120
6.4 Suite de réécriture de structures . . . 122
6.5 Exemple d'environnement (bis) . . . 124
6.6 Couplage des dimensions données et contrôle . . . 125

7.1 Migration de logiciels du patrimoine . . . 131

1.1 **Screen images of correct executions** . . . 138
1.2 **Screen images of "Can it be done" guidance facility** . . . 139
1.3 **Screen images of "How To Do IT"** . . . 140
1.4 **Screen images of "Plan Generator" execution** . . . 141

2.1 Suppression de type d'entité et relation de sous-type . . . 149

3.1 Extension/Restriction de listes d'attributs ((R1), (R2)) . . . 153
3.2 Eliminer/Imbriquer un produit cartésien dans un autre ((R3), (R4)) . . . 154
3.3 Imbrication/Désimbrication un produit cartésien ((R5), (R6)) . . . 154
3.4 Imbrication/Désimbrication un produit cartésien dans un ensemble ((R7), (R8)) . . . 155
3.5 Distributions de l'ensemble et de l'union ((R9), (R10)) . . . 155
3.6 Imbriquer/désimbriquer une union de types dans une union de types ((R11), (R12)) . . . 156
3.7 Structures et règles invariantes (R13) . . . 156

Première partie

Le décor : problématique et domaines d’application

1

Intégration et évolution de systèmes logiciels

1.1 Introduction : De l'univers du discours

L'objectif premier des environnements intégrés de production de logiciels est d'apporter divers types de supports aux acteurs impliqués dans les activités de développement (programmeurs, concepteurs, gestionnaires, etc) afin d'accroître leur productivité et d'augmenter la qualité des produits [199]. Ils devraient contribuer à l'organisation et à la gestion de différents types d'informations, c'est-à-dire, tant ceux relatifs aux activités de développement que ceux liées à sa gestion. Ces données sont de plus en plus souvent gérées par un système de gestion de bases de données dédiés au génie logiciel. Par ailleurs, les environnements logiciels donnaient la primauté aux produits à développer en fournissant les outils nécessaires à leur production. Plus récemment, les travaux sur les environnements basés sur la connaissance se préoccupent également de la façon dont les produits sont fabriqués. Le rôle dévolu à cette catégorie d'environnements, dits *environnements fondés sur les modèles de procédés de fabrication de logiciels*, peut être plus clairement illustré par analogie avec celui d'un système de contrôle dans une chaîne de fabrication industrielle où des machines agissent sur des objets transitant à travers divers postes de travail et où le système de contrôle supervise et synchronise les activités et le transport des objets, conformément à un procédé de fabrication donné. Le système de contrôle peut être capable de se conformer à un seul modèle de procédé, auquel cas peu de flexibilité est possible sur la chaîne. Une alternative est de disposer d'un système de contrôle plus général, c'est-à-dire capable de prendre en compte différents modèles, ceux-ci pouvant être des instanciations de modèles génériques. A titre d'exemple, on peut imaginer un *modèle générique* de production de véhicules et des *instances spécifiques de ce modèle* pour produire des modèles particuliers de véhicules. Les instances spécifiques peuvent utiliser des machines différentes, leurs tâches peuvent avoir des durées et des enchaînements différents, etc. Cela suppose l'existence de moyens d'expression, d'instanciation et d'exploitation des modèles pour *réaliser les activités* et pour *raisonner sur les actions* effectuées. Pour ce faire, l'environnement doit collecter, organiser et gérer les informations nécessaires au fonctionnement et à la communication entre les mécanismes qui le composent. Ces informations, regroupées dans la *base d'objets* de l'environnement, décrivent les modèles, leurs instances et l'état du logiciel en cours de développement.

En outre, à la différence de la production industrielle où le comportement des machines peut être figé et déterministe, le développement de logiciel peut nécessiter des remises en cause fréquentes de par la diversité, la complexité ou la durée de ses tâches, rendant ainsi primordiaux les problèmes de *modification* et *d'évolution* des modèles de procédés, de leurs instances, des objets logiciels en cours

de production et des outils utilisés dans l'environnement.

Pour introduire les différentes facettes des problèmes traités dans notre étude, nous mettrons l'accent sur l'organisation et la diversité des *mécanismes* et des *structures* des objets dans un environnement. Plus précisément, nous nous focaliserons sur la représentation des objets et l'interopérabilité des outils. Par *interopérabilité,* nous entendons la possibilité qu'a un outil O_1 d'agir sur des objets produits par un outil O_2 parce que O_1 suit O_2 dans le cycle de développement ou que O_1 vient remplacer O_2 dans l'environnement, ou encore parce que O_1 et O_2 sont des outils fonctionnellement équivalents (exemple : deux éditeurs de textes, deux outils de dessins, etc). Par ailleurs, si O_2 peut agir sur les objets produits par O_1, nous dirons que O_1 et O_2 sont *intégrés par les données* et *interopérables*. La figure 1.1 schématise des scenarii d'intégration d'outils par les données. La partie gauche montre le cas

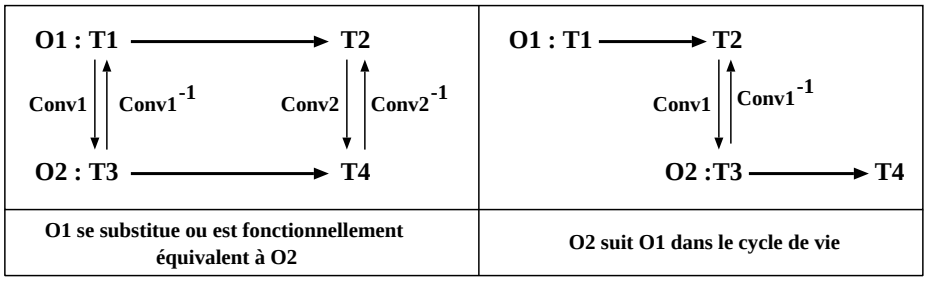

FIG. 1.1 – *Scénarii d'intégration par les données*

où les outils $O1$ et $O2$ veulent inter-opérer sur des objets de types $T3$ et $T1$, alors que la partie droite de la figure montre le cas où $O2$ veut agir sur les résultats de $O1$. Dans chacun des cas, des différences dans les représentations des types des objets requièrent des transformations d'objets (matérialisées par les convertisseurs de données $Conv^i$ sur la figure 1.1).

Il est important de souligner qu'outre l'intéropérabilité, l'intégration par les données est un facteur important *d'évolution* des environnements, c'est-à-dire favorisant leur capacité d'accueil de nouveaux outils ou de remplacement d'un outil par un autre avec *ré-utilisation* des objets existants (comme schématisé par la partie gauche de la figure 1.1 où O_2 viendrait à remplacer O_1 et qu'il existe des objets persistants produits par O_1).

Dans le paragraphe 1.2, nous montrons, de façon informelle, les problèmes posés par l'interopérabilité et l'évolution en fonction d'une typologie simplifiée des environnements (celle-ci étant délibérément orientée vers nos objectifs, on pourra se référer à [170] pour une classification différente, fondée sur une analogie avec les organisations sociales (individu, famille, ville, etc) et à [134] pour une caractérisation basée sur la technique d'intégration et le degré de généricité des environnements). Le paragraphe 1.3 montre et justifie la diversité des évolutions nécessaires, le paragraphe 1.4 contient une brève description des approches visant à résoudre le problème de l'interopérabilité.

La catégorie des environnements fondés sur la notion de modèle de procédés fait intensivement appel à des concepts et des techniques provenant du domaine des bases de données et de celui de l'intelligence artificielle. Pour illustrer les problèmes communs aux domaines du génie logiciel, des bases de données et de l'intelligence artificielle et pour montrer les contributions respectives de ces domaines, nous exposons successivement,

– dans le paragraphe 1.5, quelques aspects de l'évolution dans le domaine de la programmation,

– dans le paragraphe 1.6, des problèmes d'intégration et d'interopérabilité dans les bases de données fédérées ou réparties (paragraphe 1.6.1) et les bases de données extensibles (paragraphe 1.6.2),

– et dans le paragraphe 1.7 nous considérons l'évolution des bases de connaissances et le raisonnement sur les actions dans les systèmes évolutifs.

Le paragraphe 1.8 résume nos contributions et indique la structure de leur présentation.

1.2 Environnements de développement de logiciels et intégration

1.2.1 Notion d'environnement

De manière très générale, nous pouvons caractériser les environnements par trois composantes [169] ([158] y ajoute une composante *"personnes"* dont nous ne tiendrons pas compte ici) :

– une composante *"disciplines"* comprenant l'ensemble des règles, des stratégies, des contraintes relatives au projet logiciel,

– une composante *"mécanismes"* constituée par l'ensemble des outils de l'environnement,

– et une composante *"structures"* contenant les objets sur lesquels agissent les mécanismes dans le respect des disciplines.

Afin d'indiquer les approches possibles pour l'intégration par les données, nous allons examiner quatre catégories d'environnement, en ne nous préoccupant pas de la catégorie, plus récente, constituée par les environnements de développement d'environnements (*Meta-CASE Environments*).

1.2.1.1 Environnements "boites à outils"

Dans la *catégorie la "plus primitive"* où un environnement est constitué par *une collection* d'outils, les disciplines ne sont pas prises en compte par l'environnement. Par ailleurs, si des incompatibilités existent dans les représentations des objets de O_1 et de O_2, on est amené à développer des convertisseurs de structures, comme schématisé par les figures 1.2 et 1.3, qui visent à pallier les différences de représentations en vue de permettre l'interopérabilité. Ces convertisseurs peuvent être indépendants des outils (figure 1.2) ou incorporés à ceux-ci (figure 1.3).

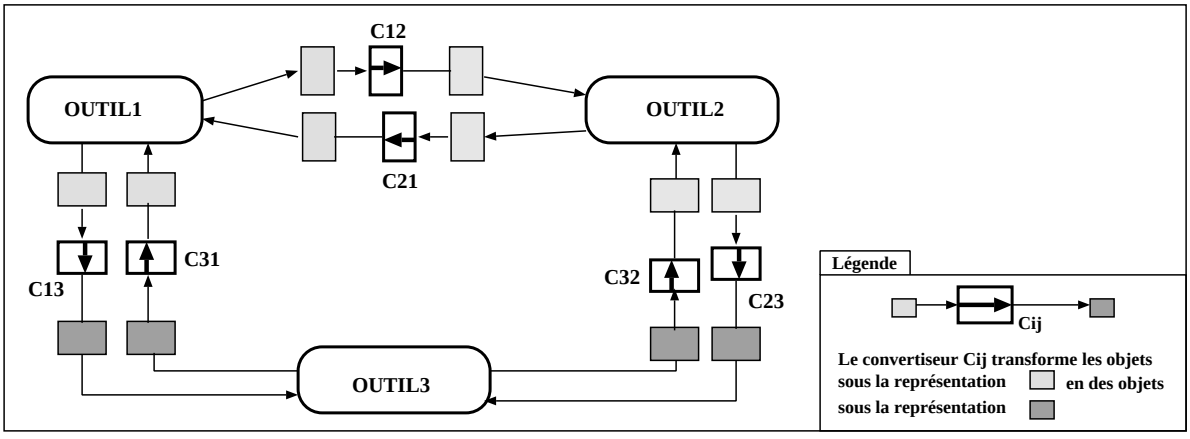

FIG. 1.2 – *Approche naïve de l'intégration des données*

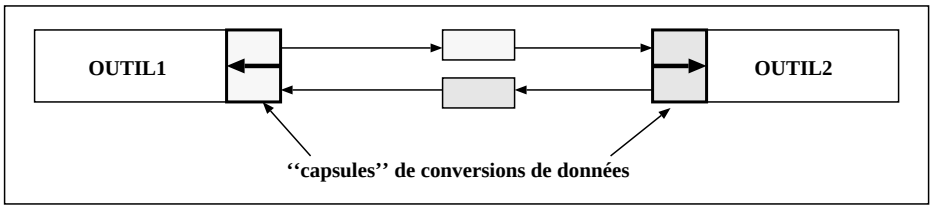

FIG. 1.3 – *Encapsulation des convertisseurs de données*

1.2.1.2 Environnements dédiés

Une deuxième catégorie d'environnements, constituée par les environnements dédiés (à un domaine d'application, à une méthode de conception ou de spécification, etc), se fonde généralement sur une structure de représentation commune aux outils où l'hétérogénéité des représentations ne se pose plus (figure 1.4). Les outils sont conçus et réalisés en fonction de la structure adoptée : on parlera, dans

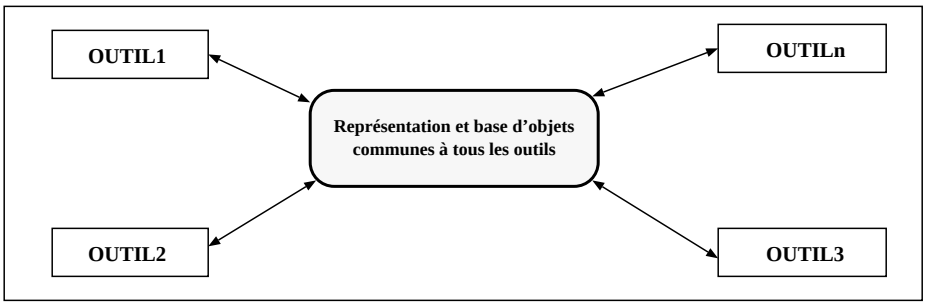

FIG. 1.4 – *Intégration forte "autour" d'une structure commune*

ce cas, *d'intégration a priori*. Ce type d'environnements est *fortement intégré* autour de la structure commune, mais il pêche par ses capacités limitées à supporter des modifications tant de la structure que des outils. En effet, l'incorporation d'un nouvel outil, ou *intégration a posteriori*, ne peut se faire que si l'outil à incorporer "connaît" la structure de données commune. Si ce n'est pas le cas, il doit être soit modifié soit étendu par des mécanismes d'accès à cette structure. La figure 1.5 montre deux approches possibles. La première, appliquée à $OUTIL_2$, consiste à encapsuler les mécanismes d'ac-

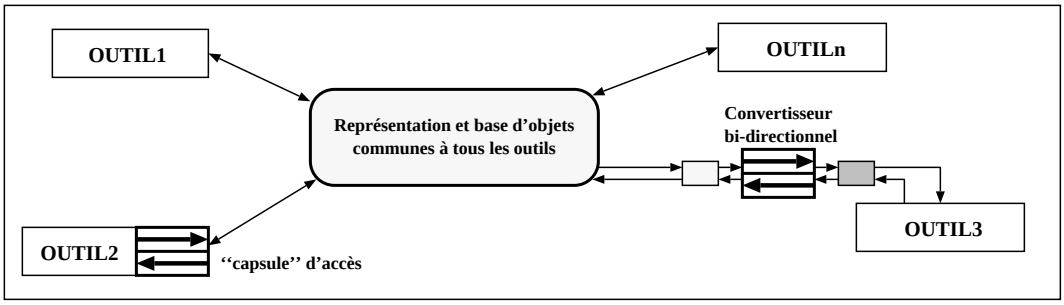

FIG. 1.5 – *Référentiel unique et convertisseurs*

cès dans l'outil. La seconde, schématisée par l'$OUTIL_3$, suggère le développement d'un convertisseur bi-directionnel (des structures de données de l'outil vers la structure de représentation commune et réciproquement). Dans cette approche, la structure commune joue le rôle de *représentation canonique*

et la communication d'un objet *Q* entre deux outils O_1 et O_2 est opérée en *deux étapes* : conversion de *Q* dans la représentation canonique puis conversion dans la structure cible. Cette approche réduit le nombre $n * (n - 1)$ de convertisseurs requis par l'approche naïve (figure 1.2) à $2 * n$, n étant le nombre d'outils s'échangeant des objets. Elle présente néanmoins, l'inconvénient de ne pas être évolutive dans la mesure où toute modification de la structure commune ou des structures de données d'un outil nécessite la modification des convertisseurs.

Par ailleurs, dans ce type d'environnements, l'enchaînement des outils est généralement prédéfini et immuable, ce qui limite leur capacité d'évolution, notamment pour l'intégration de modèles de procédés.

1.2.1.3 Environnements flexibles

Une troisième catégorie d'environnements est constituée par une composition flexible d'outils où la définition de la composition (qui peut par exemple, être réalisée à l'aide de *make* [87] ou d'un *shell script*) revient à instancier l'environnement. Les "disciplines" peuvent être fixées pour chaque composition. Le problème de l'interopérabilité des outils peut être résolu soit par le biais de convertisseurs, soit par le biais d'un référentiel commun aux outils. Dans ce dernier cas, du fait de la flexibilité des compositions, la structure de ce référentiel doit pouvoir être définie pour chaque nouvelle composition d'outils. Elle doit aussi pouvoir évoluer pour prendre en compte les représentations des données d'éventuels nouveaux outils. Les environnements fondés sur une base de données commune aux outils [103, 101, 59], tels ceux développés au-dessus de PCTE [95, 193], font partie de cette catégorie : ils se basent sur une modélisation abstraite des objets de l'environnement, appelée *schéma des données* ou *schéma des objets*. Les outils de l'environnement accèdent alors aux objets par le biais d'une partie du schéma, appelée *sous-schéma ou vue*. Néanmoins, si ce type d'environnement offre des mécanismes communs aux outils pour accéder et modifier les descripteurs d'objets[194], c'est-à-dire leurs attributs, à notre connaissance, ils n'offrent pas de moyens pour adapter le contenu d'un objet D_1 produit par un outil O_1 en cas d'incompatibilité de sa représentation avec celle des objets sur lequel opère un outil O_2. Nous nous retrouvons alors dans une situation similaire à celle de la catégorie précédente d'environnements où l'interopérabilité se résout par modification des outils ou par développement de convertisseurs ad hoc.

1.2.1.4 Environnements et modèles de procédés

Une quatrième catégorie d'environnements est celle dotée de capacités de définition de compositions flexibles des outils et des disciplines : ceux sont les environnements fondés sur les modèles de procédés de production de logiciels [90]. Nés vers le milieu des années 1980 [202, 129, 178, 130, 160, 161, 25], la notion de modèles de procédés logiciels (*Software Process Model*) concerne la description explicite des "disciplines" qui régissent les activités liées au développement de logiciels et la prise en compte de cette description par un environnement. En d'autres termes, ce type d'environnement ne se préoccupe pas uniquement des objets logiciels à produire, mais il tient compte de la façon dont ces objets sont produits. En outre, ses "mécanismes" et ses "structures" ne sont plus alors limités à ceux afférents aux objets du logiciel à produire, mais ils doivent être étendus par des moyens permettant la spécification et l'utilisation des modèles de procédés.

La spécification des modèles comprend la description d'une connaissance variée, incluant, notamment, une modélisation des types des objets, des activités et des règles régissant ou contraignant à la fois les objets et les activités, voire même, les acteurs impliqués dans un développement [79]. Une spécification de modèles de procédés peut alors être utilisée pour conduire un ou plusieurs développements. Cela passe par une phase *d'instanciation* où des objets du modèle peuvent être créés et des

outils attachés à des activités (on parlera alors *d'instance de modèles de procédés*).

L'exploitation des modèles par l'environnement (ou *activation*) poursuit divers buts, dont l'assurance de la *conformité* du processus effectif de développement aux modèles spécifiés, la *prédiction* d'états futurs du développement et *l'explication* d'états antérieurs [38]. L'environnement devra, de ce fait, être doté de capacités *d'exécution* et de *raisonnement sur les actions* qu'il effectue. Les mécanismes nécessaires à la réalisation de ces objectifs, que nous dénommerons *moteur de procédés* (figure 1.6), utilisent

– des objets représentant la spécification des modèles de procédés,

– des objets décrivant des modèles de procédés en cours d'utilisation,

– et des objets décrivant les états successifs du logiciel en cours de développement.

La figure 1.6 montre les composants essentiels des environnements visés. Ils comprennent *deux catégories d'outils* : celle des outils permettant la spécification, l'instanciation, la validation, . . . , de modèles de procédés, et celle comprenant les outils spécifiques au développement. Les spécifications de modèles de procédés sont persistants et gérés comme des objets à part entière. Le *moteur de procédés* est lui-même composé d'un ensemble d'outils dont le fonctionnement utilise à la fois la *base des "objets procédés" et celle des objets logiciels*, cette dernière décrivant l'état d'un logiciel en cours de production.

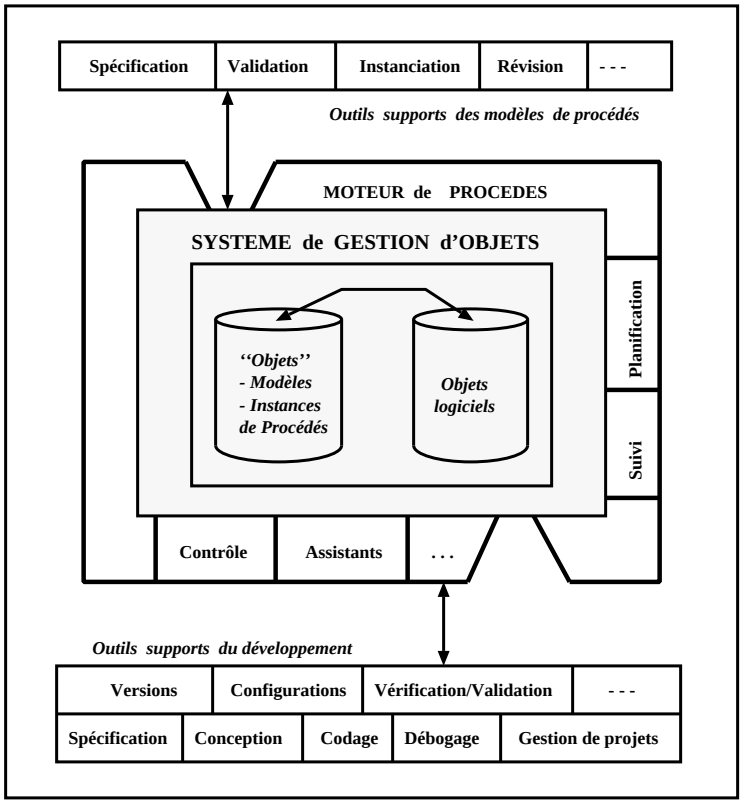

FIG. 1.6 – *Composants d'un environnement fondé sur les modèles de procédés*

La figure 1.7 schématise les différents niveaux de modèles que nous venons d'évoquer. Celui qualifié de *"méta-modèle"* décrit les démarches de conception, de spécification, de validation, d'instan-

ciation, . . . , de modèles. Il peut lui-même être spécifié en termes de (méta-)modèles de procédés.

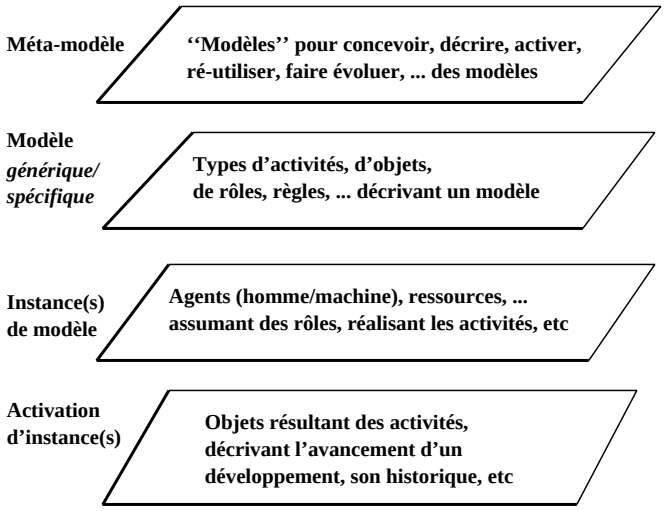

FIG. 1.7 – *Les niveaux de modèles et d'instances*

Le niveau *"modèle générique/spécifique"* comprend la spécification de modèles de procédés. Quand ceux-ci sont *génériques*, ils peuvent être adaptés pour produire des modèles spécifiques. Alors que les deux niveaux précédents sont des descriptions abstraites, le niveau *"instance de modèle"* comporte des objets concrets, comme l'indication des ressources allouées à des tâches, la répartition de celles-ci entre des personnes, le choix des outils logiciels nécessaires à la réalisation des activités [1], etc. Enfin, le dernier niveau, celui de *l'activation* d'une instance de modèle, est celui de l'exécution effective d'un développement sous la supervision du *moteur de procédés*.

Chaque niveau suppose la disponibilité de mécanismes assurant la gestion des objets qui lui sont associés. Comme les outils logiciels dédiés au développement, ceux spécifiques aux modèles de procédés nécessitent également de communiquer et de s'échanger des informations, leur exécution nécessite d'être contrôlée et coordonnée, etc : en un mot, ils ont besoin d'être intégrés comme doivent l'être les outils d'un environnement.

1.2.2 Les dimensions de l'intégration

En fait, le caractère *intégré* d'un environnement ne se mesure pas uniquement aux mécanismes et aux structures. Ainsi, le modèle de référence de l'ECMA (European Computer Manufacturers Association) [83] pour les environnements assistés (CASE, Computer Assisted Environments) qui se veut un cadre conceptuel et fonctionnel pour décrire et comparer les systèmes, identifie huit dimensions pour l'intégration : la base d'objets, l'intégration par les données, les outils, la gestion des tâches, les messages ou communication entre les outils, l'interface utilisateur, la sécurité, et l'administration et la configuration de la plateforme support de l'environnement. Chaque dimension est analysée selon une perspective conceptuelle, un point de vue externe (vision de l'utilisateur) et une perspective interne (point de vue du système). Pour chaque dimension, le modèle définit des règles, des opérations, la variété de types de données requises, ainsi que les relations entre les dimensions.

1. Nous verrons plus tard (chapitre 2), qu'en fait, le choix des outils peut aussi être concrétisé dynamiquement, c'est-à-dire pendant l'activation des modèles de procédés.

Une autre caractérisation [201], sur laquelle nous nous appuierons, distingue cinq dimensions :

(i) *La plateforme* : il s'agit d'un ensemble de services qui libère les applications des préoccupations de "bas niveaux", telles que la gestion des processus ou celle de la répartition. Il peut s'agir entre autres, de plateformes générales, type Unix, ou de plateformes spécialisées, type PCTE.

(ii) *La présentation* : du point de vue de l'utilisateur, tous les outils de l'environnement doivent avoir la même apparence et les mêmes modalités de mise en œuvre (les interfaces Mac Intosh en sont un exemple représentatif).

(iii) *L'intégration des données* : cette dimension concerne le partage et l'échange de données entre les outils, ainsi que la gestion des relations entre les objets produits par les outils. Comme nous avons essayé de le montrer dans la classification des environnements qui précède, l'intégration des données peut être atteinte par le biais d'un référentiel unique commun aux différents outils ou à l'aide de mécanismes ad hoc assurant, quand nécessaire, des transformations d'objets.

(iv) *L'intégration par le contrôle* : les outils doivent pouvoir se transmettre des messages, se communiquer des événements et, éventuellement, y réagir.

(v) *L'intégration des procédés* : c'est la possibilité de modéliser a priori la façon dont devrait se dérouler un développement et de contrôler l'exécution du modèle.

Dans la suite de ce document, nous ne considérerons que les trois dernières dimensions, préoccupations essentielles de notre activité de recherche. Par ailleurs, il est à noter que ces dimensions ne sont pas isolées l'une de l'autre. L'intégration par le contrôle peut nécessiter des services de la dimension données dans la mesure où, par exemple, la structure d'un message reçu par un outil ne se conforme pas à celle admise par l'outil. D'autre part, les mécanismes associés à l'intégration des procédés font recours à la fois à la dimension contrôle (par exemple, simplement pour activer un outil) et à la dimension données.

Les besoins en intégration selon les trois dimensions que sont les données, le contrôle et les procédés, sont nécessités par la diversité des connaissances à incorporer au sein d'un environnement, par celle de leur formulation et par celle des mécanismes utilisés pour leur exploitation. En effet, nous verrons, dans le chapitre 2, que la prise en compte de modèles de procédés dans les environnements logiciels peut faire appel à des mécanismes fondés sur des paradigmes et des formalismes différents mais utilisant des connaissances et des objets communs. Aussi,

(i) Quelle représentation adopter pour les connaissances et les objets ?

(ii) Quelles solutions apporter pour permettre l'interopérabilité des mécanismes sur cette connaissance et sur ces objets ?

(iii) Lorsque des modifications sont apportées aux spécifications des connaissances et des types des objets, comment adapter les mécanismes et les objets, à la fois pour préserver l'interopérabilité et pour permettre la réutilisation des instances d'objets produites en vertu de la connaissance initiale ?

sont les questions essentielles aux réponses desquelles nous essayons d'apporter une contribution.

Analysons, toujours de façon informelle, la diversité des besoins en évolution afin de caractériser précisément nos objectifs.

1.3 Evolution des environnements intégrés

La nature dynamique et "non linéaire" des activités de développement de logiciels, comme toute activité de conception, amène à des remises en cause de décisions ou de résultats antérieurs. Ces remises en cause peuvent concerner les objets du logiciel en cours de développement (paragraphe 1.3.3), les mécanismes supports du développement (paragraphe 1.3.1) ainsi que les objets décrivant les modèles de procédés (paragraphe 1.3.2). En fait, si nous nous référons de nouveau au schéma de la figure 1.7, l'évolution peut concerner chacun des plans de la figure. Les conséquences d'une modification dans un plan peuvent être "horizontales" et "verticales", c'est-à-dire concerner des objets du même niveau, et des objets des niveaux supérieurs et inférieurs.

Il faut, à ce propos, clairement faire la distinction entre la *modification de l'état* des objets (ou *mise à jour*), que nous ne considérons pas comme une évolution, et la *modification de leur spécification* qui, elle, fait partie de ce que nous entendons par évolution. En effet, de façon générale, la mise à jour de l'état des objets n'a pas d'incidence sur l'état d'autres objets (à l'exception parfois, d'objets dérivés ou dépendants de ceux modifiés). Par contre, la modification des spécifications peut avoir des répercussions sur la structure et les états d'objets existants : on parlera alors de *révision* (Dans la suite, nous utiliserons indifféremment les termes *révision* et *évolution*). Le but des mécanismes d'évolution est alors d'identifier ces répercussions et d'adapter les objets concernés aux nouvelles spécifications.

1.3.1 Evolution de la collection d'outils d'un environnement

L'évolution des outils d'un environnement, du fait de l'introduction d'un nouvel outil ou du remplacement d'un outil existant, est concernée par l'intégration par les données. En effet, quelle que soit la structure de l'environnement, qu'il soit fondé sur une base d'objets commune aux outils ou non, un des buts assignés à l'intégration par les données est de permettre *la réutilisation d'objets existants* par l'outil nouvellement incorporé et ce, dans la mesure du possible, même si la structure des données admises par ce dernier n'est pas exactement conforme à celle des objets à réutiliser. Nous verrons, dans le paragraphe1.4, les stratégies possibles pour atteindre cet objectif. Nous indiquerons également l'approche que nous privilégions, ses principes et ses fondements formels étant détaillés dans le chapitre 6 de la partie III de ce document.

Dans le cas des environnements fondés sur une base de données, les outils ont accès à cette base par l'intermédiaire de sous-schémas décrivant, notamment, les types des objets manipulés par l'outil et leurs relations. L'incorporation d'un nouvel outil peut alors nécessiter l'extension du schéma de la base par celui du nouvel outil : on parlera alors *d'intégration de schémas* [22]. Nous en exposerons la problématique dans le paragraphe 1.6 et nous détaillerons nos contributions dans le chapitre 4 de la partie III.

1.3.2 Evolution des modèles de procédés

Une modélisation de procédés est généralement constituée d'un ensemble de spécifications de modèles de procédés (appelés parfois *fragments de modèles*) où chaque spécification comporte trois types de descriptions : la description des activités que nous appellerons *opérations*, celle des *types des objets* produits ou manipulés par les activités, et celle des diverses "disciplines" à respecter, que nous appellerons *règles* régissant les procédés. Les spécifications des modèles peuvent être structurées et être liées par divers types de relations. Ainsi, un modèle de procédé peut être raffiné ou décomposé en *sous-modèles* ou *fragments*. Par exemple, un modèle de procédé *"Développement"* peut comporter un modèle *"Spécification"* et un modèle *"Codage"*. En outre, les spécifications des modèles peuvent

partager des spécifications de certains de leurs composants. Par exemple, le type d'objet *"Planning"* peut être utilisé dans le modèle de procédé *"Gestion de projet"* et celui de *"Développement"*.

L'évolution de modèles peut alors consister en la modification de l'état d'un modèle ou d'un sous-modèle en cours d'activation (cet état est décrit, rappelons le, par des valeurs d'objets), la modification de la spécification d'un modèle ou d'un sous-modèle existant et l'ajout ou la suppression de la spécification d'un modèle ou d'un sous-modèle. La prise en compte de ces opérations nécessite d'établir leur "correction" en tentant de détecter d'éventuelles incohérences qui seraient introduites et de déterminer leurs incidences sur les spécifications d'autres procédés, sur les objets décrivant les instances de procédés en cours d'activation et sur ceux décrivant le logiciel en cours de développement.

Les propositions actuelles varient des plus statiques au plus dynamiques. Les premières soit anticipent partiellement certains changements [106, 190], soit ne répercutent pas les changements d'un modèle sur les modèles instanciés : dans ce dernier cas, cela revient à produire un nouveau modèle par modification de modèles existants. Des approches plus dynamiques permettent la propagation immédiate, "opportuniste" ou différée des changements. Ainsi, dans SPADE [19], une distinction est faite entre des copies actives et non actives d'objets. Les propagations peuvent être immédiates sur les copies non actives, différées sur celles actives et "opportunistes", c'est-à-dire lors de la création d'un nouvel objet. Dans EPOS [117], le changement est vu comme une gestion de configurations de modèles nécessitant ainsi des mécanismes de gestion de versions. Des changements pouvant induire des migrations de classes sont interdits sur les modèles instanciés ou en cours d'activation. Des changements ayant un moindre impact sont autorisés, mais la résolution d'éventuelles inconsistances introduites est à la charge de l'utilisateur. Pour modifier le modèle lui-même, EPOS propose également des mécanismes et une démarche (sous forme de modèle de procédés) pour analyser la demande de changement et évaluer son impact avant de l'appliquer. Les changements ne préservant pas l'unicité des noms et l'héritage simple sont interdits.

A l'heure actuelle, nos travaux n'ont concerné que l'évolution des spécifications des types des objets d'un modèle de procédé, spécifications considérées comme un schéma de données. Nous exposerons, dans le chapitre 4 de la partie III, nos propositions en distinguant des opérations atomiques permettant des *modifications "locales"* d'un schéma de types d'objets et des opérations complexes exprimées comme une composition d'opérations atomiques et permettant *l'intégration de schémas*. La spécification d'un schéma de types d'objets est la donnée de la spécification d'un ensemble de types et de leurs relations. Elle est exprimée dans un modèle de représentation orienté objets [123, 41] comprenant des constructeurs de types et des relations inter-types communément admis (attribut, agrégation, sous-type, composition). Les diverses opérations proposées sont *contraintes par des critères de qualité, ou propriétés invariantes*, que les schéma des types d'objets doivent vérifier. L'impact des modifications sur les utilisations des opérations des types et sur les instances des objets existants est aussi considéré : nous montrerons quelques voies possibles pour adapter les instances des objets et leurs opérations aux modifications subies par leur type.

1.3.3 Evolution des objets logiciels

Ce qui distingue un environnement fondé sur les modèles de procédés du domaine de la programmation "classique" c'est, notamment, la diversité des activités qui y sont menées, celle des outils servant à la réalisation de ces activités et celle des types des objets que l'environnement a à gérer. Ceux-ci sont généralement complexes et caractérisés par un type de *contenu* (exemple : "objet" spécification, code source, etc), des descripteurs ou *attributs* (comme le nom et la taille de l'objet) et d'éventuelles *relations* avec d'autres objets (comme, par exemple, celles liant une spécification à son auteur ou un module logiciel à la liste de ses composants). Dans les environnements fondés sur une base d'objets commune, la spécification des types des objets est représentée par un schéma, au sens classique du terme dans

le domaine des bases de données traditionnelles. La mise à jour, définie comme la modification des valeurs des descripteurs ou de celle du contenu des objets, est essentiellement réalisée par les outils de l'environnement ou par les mécanismes de gestion de la base d'objets.

Sur un autre plan, certaines opérations de modification des spécifications de modèles de procédés ou d'instances de modèles peuvent aussi induire des modifications d'objets (par exemple, une nouvelle répartition des tâches dans un projet en cours affecte les relations entre des objets *"Personnes"* et des objets logiciels dont celles-ci ont la charge).

Pour notre part, nous n'aborderons ni la mise à jour des objets ni leur révision qui pourrait faire suite à l'évolution d'instances de modèles de procédés en cours d'activation. Nous ne considérerons (chapitre 4 de la partie III) que la modification de la spécification des types des attributs et des types de liens associés à un type d'objet car, à la différence des systèmes de gestion de fichiers classiques, dans les environnements fondés sur une base de données, la liste des attributs des objets peut être quelconque et n'est pas limitée à celle prédéfinie dans le système support. Le comportement des attributs caractérisant les objets est alors

– soit "centralisé" au niveau du système de gestion d'objets (SGO) : les outils rendent compte des actions effectuées, le SGO met à jour les attributs en conséquence,

– ou "répartie" dans les outils qui manipulent les objets : chaque outil met à jour les attributs qui le concernent à travers le sous-schéma qui lui est associé.

1.3.4 Conclusion

L'évolution des environnements intégrés présente, comme nous venons de le voir, diverses facettes intimement liées et où les objets de l'environnement et leur représentation jouent un rôle central. Cela suppose la disponibilité de mécanismes de spécification et d'implantation de nouveaux types de données, de mécanismes de modifications atomiques ou complexes de spécifications. La plus grande difficulté nous semble être la détermination des conséquences d'une modification et leur propagation en vue de la réutilisation, à moindre coût, des objets modifiés. Voyons comment des mécanismes permettant l'interopérabilité dans les environnements, peuvent contribuer à la levée de ces difficultés.

1.4 Interopérabilité et polymorphisme

Rappelons que nous considérons l'interopérabilité comme la faculté qu'a un outil logiciel d'agir sur des objets dont les types peuvent "ne pas être exactement" ceux des arguments de l'outil. Deux approches, non exclusives, peuvent être adoptées dans le cas où la structure des objets ne se conforme pas aux types des objets admis par l'outil en question : l'interopérabilité orientée processus (ou traitement) et l'interopérabilité orientée données (ou objets). Passons les successivement en revue.

1.4.1 Interopérabilité orientée processus

L'intéropérabilité orientée processus (*Process-Oriented Interoperability*) consiste à modifier le code d'un outil en fonction des différences entre les types. En d'autres termes, si on considère un outil *O* comme une fonction d'un type *D*, argument de l'outil, dans un type *R*, "résultat" de l'outil, pour appliquer *O* à des objets de type D' (D' différent de *D*), cette approche préconise la révision du corps de l'implémentation de *O* pour l'adapter au type D'. Cette démarche peut être envisagée en l'absence d'instances persistantes [15] d'objets du type *D*. Néanmoins, d'un point de vue pratique, elle présente l'inconvénient d'être difficilement réalisable du fait du coût probable de la modification du

code source de l'outil, à supposer que celui-ci soit disponible. D'autre part, si deux outils O_1 et O_2 de l'environnement produisent des objets de types respectifs *D* et D', et si *O* suit O_1 et O_2 dans le cycle de vie du logiciel, l'adaptation du code de *O* au type D' a pour conséquence que les instances résultant de O_1 risquent de plus pouvoir être traitées (à l'exception du cas particulier où D' est *sous-type* de *D*).

De ce fait, pour pouvoir opérer sur les instances existantes de type *D*, on peut envisager la co-existence de deux versions de *O*, l'une agissant sur les instances du type *D*, et l'autre, appelons la O', opérant sur celles du type D', rejoignant ainsi la notion de *polymorphisme ad hoc sans coercion* [58] , c'est-à-dire un polymorphisme sans conversion de types [2]. D'un point de vue pratique, deux voies sont possibles pour implémenter cette solution. La première (figure 1.8, (a)) consiste à considérer que *O* et O' sont des outils distincts et à les invoquer par des noms différents (avec des arguments du "bon type"). La seconde voie, montrée par la partie (b) de la figure 1.8, suppose que l'on dispose de mécanismes de *surcharge* (overloading) et de *liaison dynamique*: *O* et O' seraient invoqués par le même nom (O_{Poly} sur la figure), le système décidant quel code exécuter en fonction des types des arguments de l'invocation. Néanmoins, dans un cas comme dans l'autre, nous ne pouvons pas considérer que *O* et O' sont interopérables puisque chacun opère sur ses "propres types".

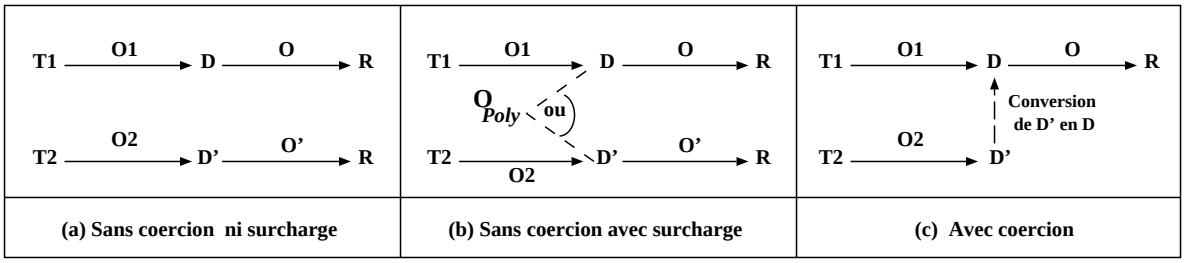

FIG. 1.8 – *Interopérabilité orientée processus vs interopérabilité orientée objets*

1.4.2 Interopérabilité orientée objets

La seconde approche pour parvenir à l'interopérabilité des outils, que nous appellerons *interopérabilité orientée objets* [126, 167, 42], ne nécessite pas de modification du code de *O*. Elle vise à adapter les instances du type D' pour qu'elles se conforment au type *D*. *O* pourra ainsi indifféremment opérer sur des instances de chacun des deux types (figure 1.8, (c)). Cette approche s'apparente à la définition et à l'implantation de fonctions avec *polymorphisme ad hoc et coercion*.

Dans l'état actuel de nos travaux, nous avons privilégié la seconde approche. Nous essayons de définir des fonctions de coercion de D' en *D* après avoir préalablement établi une relation de sous-type entre D' et *D*. Comparée à l'approche naïve, comme celle schématisée par la figure 1.2, ou à celle de conversion en deux étapes (schématisée par l'*OUTIL*$_3$ de la figure 1.5), les fonctions de coercion définissant les convertisseurs sont engendrées dynamiquement facilitant ainsi l'évolution des structures des types des objets. Nous en parlerons en détail dans le chapitre 6 de la partie III de ce document.

1.5 Evolution des programmes

Dans le domaine de la programmation, l'évolution d'un programme P, défini comme une fonction d'un type de données D dans un type de résultat R, peut concerner la sémantique de P, que nous

2. Rappelons qu'en termes d'implémentation, le *polymorphisme universel* signifie que le même code d'une fonction est exécuté pour des arguments de n'importe quel type admissible par la fonction et que le *polymorphisme ad hoc* signifie qu'un code différent *peut être exécuté* pour chaque type d'argument.

assimilerons à son contenu (ou corps de P), ou les types D et R, que nous assimilerons à son interface. Dans les méthodes informelles de développement, la modification de P est généralement faite "à la main", même si des outils d'aide peuvent parfois contribuer à l'identification des conséquences d'une modification et à la localisation des changements à effectuer [6]. Des approches plus formelles, comme celles basées sur des techniques transformationnelles [17, 9, 62] ou sur des techniques de démonstration de théorèmes [140, 141], offrent davantage d'automatisation du processus. D'autre part, les techniques d'encapsulation [144], couplées aux mécanismes de liaison dynamique, de surcharge et d'identification dynamique de types (*dynamic type checking*), permettent de rendre les *clients* (c'est-à-dire les programmes qui utilisent une ou plusieurs des fonctions (ou *services*) offertes par P) insensibles aux modifications du contenu mais pas à celles des interfaces. En effet, la modification de ces dernières a des conséquences sur tous les clients de P. L'approche *interopérabilité orientée processus* préconiserait d'identifier les clients et d'adapter leurs codes aux modifications des interfaces des services. L'approche *interopérabilité orientée données*, quant à elle, ne se préoccuperait pas des clients, en se contentant d'adapter dynamiquement les paramètres des invocations des services aux changements de leurs interfaces.

Nos travaux n'ont, jusqu'à maintenant, considéré que des objets persistants. Une de leurs extensions que nous exposerons en conclusion de ce travail consistera à appliquer l'interopérabilité orientée objets aux données transiantes, c'est-à-dire aux paramètres non persistants des appels de services ou d'outils d'un environnement.

1.6 Interopérabilité et évolution dans les bases de données

A l'instar de l'interopérabilité dans les environnements logiciels, le domaine des bases de données connaît une intense activité tendant à offrir la possibilité d'exploiter des données stockées dans différentes bases, situées sur un ou plusieurs sites et gérées par le même type de système de gestion de bases de données (SGBD) ou par des SGBD différents [162, 181]. Cette utilisation

(i) se veut *transparente*, c'est-à-dire que l'utilisateur ne doit pas être concerné par les problèmes de *localisation* des données ni par ceux de l'hétérogénéité éventuelle des SGBD.

(ii) et préservant *l'autonomie* des SGBD, c'est-à-dire l'aptitude d'un SGBD à fonctionner indépendamment d'autres SGBD.

Nous montrerons dans le paragraphe 1.6.1 les types d'approches que l'on peut adopter pour permettre l'échange de données entre SGBD puis, dans le cadre plus général de la *fédération* de bases de données, pour permettre la coopération de SGBD.

Sur un autre plan, les problèmes *d'évolution* [76] des bases de données présentent également des similitudes avec ceux évoqués dans le cadre des environnements. Ils concernent à la fois les bases existantes et leurs systèmes de gestion. *L'évolution des bases* est le plus souvent abordée sous l'angle structurel, c'est-à-dire par le biais de mécanismes de modification et d'intégration de schémas [22, 98] tels ceux évoqués plus haut et que nous détaillerons dans le chapitre 4 de la partie III. D'autres approches se fondent sur des modèles temporels [184, 185] ou historiques [143, 70] : elles ne seront pas considérées ici. Du point de vue de la logique, un schéma peut être vu comme une *théorie* admettant une infinité de *modèles* dont un seul est privilégié, celui représenté par l'instance de la base de données. Une mise à jour est alors une transition d'état opérée sur la base et elle n'est correcte que si le nouvel état de la base "est toujours un modèle de la théorie". Dans ce contexte, l'évolution de schéma peut être assimilée à un changement de la théorie. L'approche, dans un cadre opérationnel, qui consisterait à produire un modèle de la nouvelle théorie n'est pas viable. Le point de vue pratique requiert des

mécanismes de transformation du modèle de l'ancienne théorie en un modèle de la nouvelle. Il requiert également des mécanismes d'adaptation au nouveau modèle des transitions qui étaient opérationnelles sur l'ancien modèle. Cette vision est attrayante mais elle bute sur la difficulté que présente la définition logique de systèmes dynamiques (les propositions contenues, entre autres, dans [28, 124] et [176] constituent indéniablement des contributions dignes d'intérêt).

Quant à *l'évolution des systèmes*, elle peut se matérialiser par la disponibilité de mécanismes d'extension des fonctionnalités d'un SGBD ou par un changement de SGBD, auquel cas une migration des données et des applications existantes est nécessaire [109, 128, 147, 142, 36]. Nous considérerons, dans le paragraphe 1.6.2, deux types d'extension : celle concernant l'introduction de nouveaux types de données dans les SGBD et celle visant à étendre un SGBD par un système de règles, l'étude de ces deux types d'extension se justifiant par leur utilité dans les environnements fondés sur les modèles de procédés. En effet, la spécification de modèles de procédés comporte la spécification de types de données qui peuvent ne pas être des types prédéfinis dans l'environnement. Par ailleurs, l'exploitation des *règles* spécifiant les disciplines auxquelles un développement doit se soumettre peut nécessiter des mécanismes de déduction et des mécanismes de déclenchement d'actions similaires à ceux proposés dans le domaine des bases de données déductives [92, 93, 94] et des bases de données actives [24, 75], respectivement.

1.6.1 Fédération et répartition

Un des mécanismes le plus souvent utilisé pour *l'échange de données* entre SGBD consiste à *exporter* des données d'une base B_1 sous une représentation "pivot" puis de les *importer* dans une base B_2. Cette représentation, la plus élémentaire étant une représentation ASCII, peut être fondée sur un modèle standard [100, 85]. Cette approche va à l'encontre de la transparence. D'autre part, elle ne permet pas aux SGBD gérant les différentes bases de coopérer dans la gestion des données, c'est-à-dire qu'un programme ne peut pas accéder ou manipuler simultanément des données rangées dans différentes bases.

La fédération de SGBD est une collection de systèmes qui coopèrent dans la gestion de bases de données multiples. La fédération peut être centralisée sur un site ou distribuée sur différents sites d'un réseau. Elle peut être *homogène* ou *hétérogène*, c'est-à-dire utilisant un ou plusieurs types de SGBD. La notion de *transparence* consiste à donner aux utilisateurs et par conséquent aux programmes, une vision unique des bases faisant partie de la fédération. En d'autres termes, le schéma des données de la fédération inclut la description de toutes les bases qui y participent et un programme peut accéder ou mettre à jour des données rangées dans différentes bases, sans se soucier de leur localisation [162, 34].

Cela suppose :

(i) que les schémas des différentes bases sont intégrés en un seul,

(ii) qu'il existe un mécanisme capable

- de *décomposer* une requête adressée à la fédération, en sous-requêtes, chaque sous-requête ne concernant qu'une seule base,
- d'adresser les sous-requêtes aux sites appropriés,
- de constituer le résultat final à partir des résultats des sous-requêtes,

et de contrôler le déroulement de ce processus.

Dans ce domaine, nous sommes également confrontés à des problèmes d'intégration de données et de logiciels. *L'intégration des schémas* des bases de la fédération peut se fonder sur un modèle de

représentation de données "pivot" à partir duquel sont opérées des applications (mapping) sur les modèles de représentation de chaque SGBD de la fédération. De même, lors du traitement d'une requête exprimée dans un langage L, si elle est décomposée en n sous-requêtes, il est nécessaire de traduire chacune de celles-ci dans les langages $L_1, \cdots, L_n$ des SGBD auxquels elles doivent être adressées. On imagine aisément les difficultés techniques et la multiplicité des convertisseurs requis. Les solutions actuelles du marché, comme celles proposées par Sybase 10 [72] ou Oracle 7 [159], offrent des correspondances ou "passerelles" entre les SGBD les plus répandus en utilisant le modèle relationnel de données comme modèle pivot. Sybase 10 offre, en outre, des outils de développement de nouvelles correspondances.

Ainsi, l'hétérogénéité des modèles de représentation et des langages de manipulation de données, que ce soit dans le traitement coopératif de données par fédération ou dans le cadre d'échange de données entre bases, conduit au développement de convertisseurs ou "passerelles" entre les différents modèles et systèmes. Les approches les plus usuelles s'apparentent aux mécanismes de *conversion en deux étapes* évoqués dans le paragraphe 1.2.1 et dont nous avions souligné les limites.

1.6.2 SGBD extensibles

L'extension des fonctionnalités d'un SGBD peut se fonder :

(i) sur le *couplage* de systèmes hétérogènes, comme dans le cas des SGBD déductifs [96],

(ii) sur l'extension d'un langage [179, 16, 63, 7] ou d'un système existant [188, 97],

(iii) sur le développement d'un système "ouvert" dont les composants sont modifiables (comme c'est le cas dans *STARBURST* [133]),

(iv) ou des systèmes "ouverts" conçus comme des "briques" à assembler selon les besoins (approche adoptée notamment, dans *GENESIS* [23]).

Dans ce paragraphe, nous considérons deux types d'extension : l'extension des types de données et l'extension de SGBD par un système de règles.

1.6.2.1 Systèmes de types de données extensibles

Les SGBD actuels offrent un ensemble de types prédéfinis pour décrire les objets et leurs attributs. *Un courant de travaux* visant à étendre le système de types d'un SGBD est celui des *"langages de programmation de bases de données"* [53] (Database Programming Language ou DBPL) ou celui de la *programmation persistante* [14]. L'objectif est de donner aux objets d'une base le même statut que les variables d'un programme pour pouvoir manipuler les uns et les autres de façon homogène (c'est-à-dire ne pas avoir à écrire des programmes multi-langages, comme c'est généralement le cas actuellement où un programme peut comporter des séquences exprimées dans un *langage dit hôte*, tel que C ou Ada, et des séquences exprimées dans le langage du SGBD, tel SQL). L'approche DBPL permet de réduire *"l'impedance mismatch"*, c'est-à-dire de pallier les différences entre le système de types du langage hôte et celui du SGBD. Elle offre également la possibilité de définir, d'implanter et d'utiliser de nouveaux types de données, tirant ainsi bénéfice des résultats obtenus dans le domaine de la programmation [132, 105, 108, 107].

Pascal-R [179], *PS-Algol* [16, 14] et *Galileo* [7] sont parmi les langages représentatifs de ce courant. *Pascal-R*, un des pionniers du domaine, intègre un système relationnel et le langage Pascal. Le système de types du langage comprend un type *"relation"* et un type *"base de données"*. Le langage est étendu par des possibilités de manipulation des instances de ces types. Néanmoins, la définition de nouveaux

types est limitée aux constructeurs offerts par Pascal. *PS-Algol* est un langage où la persistance n'est pas limitée aux fichiers mais étendue à n'importe quel type d'objet d'un programme (variable, procédure, etc). Son successeur, *Napier* [53] incorpore un système de types plus riche et un contrôle de types plus puissant. Enfin, *Galileo* [7] est basé sur ML et est un langage de programmation de bases de données avec un système de types polymorphes.

Un autre courant vise à permettre la définition et l'implantation de nouveaux types de données par extension du système de types d'un SGBD. C'est le cas notamment de POSTGRES [187] et de la majorité des systèmes de gestion de bases de données orientées objets, dont O_2 [81], Orion [125], Gemstone [136, 54]. La principale difficulté, qui nous rappellera les problèmes d'intégration par les données, est la représentation dans la base, des instances d'un nouveau type de données et leur manipulation par les opérations du type. Examinons la figure 1.9, inspirée de [163], qui schématise une opération Op de profil $T1 \rightarrow T2$, $T1$ et $T2$ supposés ne pas être des types de données prédéfinis du SGBD.

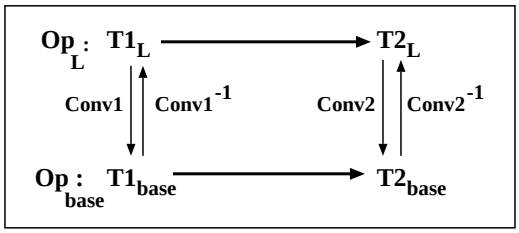

FIG. 1.9 – *Scénario d'implantation de nouveaux types dans un SGBD*

Soient $T1_L$ et $T2_L$ les représentations de $T1$ et $T2$ dans le langage L d'implémentation du corps de Op. Soient également $T1_{base}$ et $T2_{base}$ les représentations de $T1$ et de $T2$ dans la base. La matérialisation des effets de l'opération sur des objets de la base nécessite la conversion $Conv1^{-1}$ de $T1_{base}$ en $T1_L$, l'application de l'opération sur l'objet ainsi obtenu et la conversion $Conv2$ du résultat, de $T2_L$ en $T2_{base}$. Les fonctions de conversion $Conv1$ et $Conv2^{-1}$, inverses respectives de $Conv1^{-1}$ et de $Conv2$, jouent un rôle similaire lorsqu'il s'agit de ranger des objets de type $T1$ dans la base ($Conv1$) ou d'en extraire des objets de type $T2$ ($Conv2^{-1}$). Dans les SGBD actuels où le système de types est figé, ces fonctions de conversion sont prédéfinies. Dans les systèmes extensibles, elles doivent être définies pour chaque nouveau type de données. Une approche alternative, ne nécessitant pas de convertisseur mais plus difficilement réalisable et moins évolutive, consisterait à implanter les opérations des nouveaux types en utilisant le langage et les structures de représentation interne du SGBD (on notera l'analogie avec l'interopérabilité orientée objets dans les environnements logiciels comme schématisé par la figure 1.1).

Ainsi, comme dans le cas de l'interopérabilité des outils des environnements de développement de logiciels, l'introduction d'un système de types extensible dans un SGBD peut faire appel à des mécanismes de *coercion* de types. De façon similaire, nous allons essayer de montrer que l'extension des SGBD par des capacités déductives ou actives amène à considérer le même type de problème.

1.6.2.2 Les règles dans les SGBD

L'introduction d'un système de règles dans les SGBD vise plusieurs objectifs, parmi lesquels figurent la protection des données, leur intégrité [49] et la définition de données *"dérivées"* [94, 37], c'est-à-dire de données dont les valeurs ne sont pas stockées explicitement mais "calculées" à partir de celles stockées et de règles de déduction. Nous distinguerons deux catégories de règles : les règles

déductives et les règles actives.

a) Règles déductives
L'introduction de capacités de déduction dans les SGBD s'est essentiellement appuyée sur le modèle relationnel de données [71] et la logique du premier ordre. Dans un cadre logique, une base de données relationnelle peut être vue [175] :

(i) *dans une approche "interprétative" (théorie du modèle),* comme un *modèle* des contraintes d'intégrité de la base. Une requête est alors une formule $R(\vec{X})$, et son évaluation consiste à trouver toutes les substitutions à $\vec{X}$ qui rendent $R(\vec{X})$ vraie, $\vec{X}$ étant les variables libres de $R(\vec{X})$.

(ii) *dans une approche théorie de la preuve,* comme une théorie T du premier ordre où, évaluer une requête $R(\vec{X})$ consiste à prouver $R(\vec{X})$ étant donnée la base comme prémisse. Dans ce contexte, une base de données déductive est obtenue en adjoignant à la théorie T des formules (ou règles de déduction) définissant des relations dérivées.

D'un point de vue pratique, une base de données déductive est constituée de deux ensembles de données : une base de données "extensionnelle" (ou base de faits) vue comme des tables relationnelles et une base de données "intentionnelle" (ou base de règles) décrite par des formules exprimées dans une des variantes du langage logique de données DATALOG [61]. La réalisation de SGBD dotés de capacités déductives se fait par couplage ou par intégration [96, 37] de mécanismes de gestion des données et de mécanismes de déduction. Les expériences menées se fondent généralement sur le couplage ou l'intégration d'un SGBD relationnel et de PROLOG. Le principal problème à résoudre pour réaliser de tels systèmes, est de concilier le paradigme "un ensemble à la fois" des SGBD relationnels (la réponse à une requête adressée au SGBD est généralement un ensemble de n-uplets) et le paradigme "un n-uplet à la fois" (chaque étape d'unification ne considère que deux littéraux) des mécanismes de résolution "à la Prolog".

Un problème secondaire qui s'apparente à celui de l'intégration par les données, est la disponibilité de mécanismes de transformation des représentations des données, notamment lorsque le mécanisme de déduction doit opérer sur des données en provenance de la base relationnelle. Ce problème est facilement résolu dans la cadre du couplage d'un SGBD relationnel et d'un système de résolution comme PROLOG : chaque n-uplet $< a_1, a_2, \cdots, a_n >$ d'une relation R est réécrit en une instance d'un prédicat $R(a_1, a_2, \cdots, a_n)$ [175].

La disponibilité d'une formalisation logique des bases de données relationnelles communément admise a facilité l'émergence de résultats fondamentaux concernant l'interrogation dans les SGBD déductifs (la caractérisation syntaxique de programmes DATALOG "sûrs" [18, 61], c'est-à-dire corrects, complets et se terminant, étant, à notre avis, un des résultats les plus importants). Par contre, la mise à jour en présence de règles qui, à de rares exceptions, ne peut être réalisée que sur la base extension, ne bénéficie pas encore d'un cadre formel adéquat, témoins la diversité des propositions en la matière [137, 138, 139, 150, 27, 52, 127, 177]. Le problème rejoint celui, complexe, de la modification de théories logiques et s'apparente fortement à la révision des bases de connaissances dont nous parlerons dans le paragraphe 1.7.

L'introduction de règles actives dans les SGBD est une approche qui mélange les paradigmes déclaratif et procédural et qui permet, entre autres, de contribuer à la mise à jour en présence de règles (en attendant une formalisation correcte du problème ?).

b) Règles actives dans les bases de données
Les règles actives peuvent servir, entre autres, à la vérification de contraintes d'intégrité sur les données, au contrôle des accès et à la propagation d'actions comme effet de bord d'une autre action. Leur forme syntaxique la plus usuelle comporte la spécification d'un événement, d'une condition et d'une action (*règle ECA*) ;

sur Evénement *si* Condition *alors* Action

Une sémantique statique étant difficilement définissable, on s'accorde généralement sur une sémantique procédurale : "Si l'événement survient et que la condition est vérifiée, alors déclencher l'action". L'implantation d'un systèmes de règles nécessite d'apporter des réponses notamment, aux problèmes :

(i) de *couplage* : quand déclencher l'action spécifiée par une règle qui devient activable ? Dès l'arrivée de l'événement qui la "garde", de façon différée, etc ?

(ii) de *conflit* de déclenchement, lorsque plusieurs règles sont candidates à l'activation,

(iii) de *terminaison* dans le cas de déclenchement d'actions "en cascade" (enchaînement des règles),

(iv) *d'interaction des actions d'une règle* avec celles du programme ou de la transaction qui ont rendu la règle déclenchable. C'est-à-dire, d'une part, est-ce-que l'action de la règle a un statut de transaction ou de sous-transaction de la transaction qui l'a rendue active ? D'autre part, si un incident survient pendant ou après le déclenchement d'une règle, les actions effectuées par la règles doivent elles être défaites, c'est-à-dire, leurs effets sur la base doivent-ils être annulés (notion de *Rollback*) au même titre que celles de la transaction ou pas ? Et enfin, dans un contexte multi-utilisateurs, que faire si les objets sur lesquels doit agir la règle sont déjà détenus par une transaction et que la règle doit être déclenchée immédiatement ?

A notre connaissance, il n'y a pas de solution générale à ce type de problèmes, mais une variété de solutions ad hoc, comme l'extension de la spécification des règles par des modalités de déclenchement (immédiat, différé, dès que possible), des priorités entre les règles pour résoudre les conflits de déclenchement ... Ainsi, *POSTGRES* [188], dans sa première version, exécute immédiatement une règle et considère sa partie action comme partie intégrante de la transaction qui l'a rendue activable. *STARBURST* [133] diffère l'exécution des règles à la fin de la transaction alors que *HiPAC* [74] permet de choisir le mode de couplage (immédiat, différé ou "découplée" i.e. considérer la règle comme une transaction à part entière). Dans GOODSTEP [4], un événement pouvant déclencher plusieurs règles, un ordre entre les règles, modifiable par l'utilisateur, est défini par le système. Le couplage est immédiat ou différé et non découplé.

c) Règles et environnements
Nous sommes confrontés à la même diversité de questions dans les environnements fondés sur les modèles de procédés dans la mesure où les règles de ces modèle peuvent prendre la forme de règles actives ou de règles déductives. En outre, nous pouvons également être amenés à résoudre des problèmes d'interopérabilité entre le mécanisme d'évaluation de la partie condition d'une règle et celui de gestion de données.

Enfin, un dernier aspect lié à la nature dynamique des environnements et des bases de données actives est celui qui concerne l'arrivée d'événements remettant en cause des actions en cours d'exécution. Nous avons évoqué, dans le paragraphe 1.2.1, le fait que les modèles de procédés pouvaient servir

à des fins de prédiction, notamment par le biais de génération de plans d'actions pour atteindre un état du logiciel en cours de développement. Des événements modifiant l'état des objets de l'environnement, voire l'évolution des modèles de procédés, peuvent conduire à la révision de plans en cours d'exécution.

Le même type de problème est considéré dans le domaine de l'évolution des bases de connaissances et celui des systèmes réactifs : c'est l'objet du paragraphe qui suit.

1.7 Interopérabilité et évolution dans les systèmes à bases de connaissances

Le domaine des bases de données et celui de l'intelligence artificielle ont parfois été opposés, arguant du fait que le premier se préoccupe davantage de la modélisation d'un univers en termes de structures de ses données et de la gestion de ces structures, alors que le second se focalise sur la "conception de systèmes capables de reproduire le comportement de l'être humain dans ses activités de raisonnement" [113]. Les convergences entre les deux domaines, comme nous l'avons déjà évoqué dans le paragraphe 1.6.2, se sont concrétisées par le développement de systèmes de déduction coopérant avec des systèmes de gestion de données. Elles se sont également manifestées dans d'autres cadres, comme celui de la conception de bases de données [47, 189, 32], celui du traitement "intelligent" des requêtes [115], de l'optimisation sémantique de celles-ci [65] ou celui de la satisfaction des contraintes d'intégrité [154, 149, 148]. L'idée d'environnement logiciel "intelligent" n'est pas récente, elle aussi. Dès 1973, Winograd [203] évoquait quelques exigences fondamentales pour une aide intelligente au développement de logiciel, une de ces exigences étant qu'un environnement logiciel "comprenne" non seulement *ce qu*'il fait, mais aussi *pourquoi* il le fait. Le rôle des modèles de procédés est non seulement d'y contribuer mais aussi d'adjoindre aux environnements des capacités de raisonnement sur les activités qu'ils supportent.

Dans de ce paragraphe nous n'avons pas la prétention de passer en revue tous les points de convergence des trois domaines. Nous nous concentrerons sur deux domaines de l'intelligence artificielle, les systèmes évolutifs et les systèmes de raisonnement distribué, pour leurs contributions potentielles au développement des environnements fondés sur les procédés et la communion de certaines de leurs préoccupations avec des préoccupations présentes dans le domaine des environnements fondés sur les modèles de procédés.

1.7.1 Evolution des bases de connaissances

Le rôle actif d'un environnement intégrant les modèles de procédés suppose la disponibilité de mécanismes d'exécution et de supervisison des activités, s'apparentant ainsi aux systèmes physiques évolutifs [82] où la dimension temporelle est importante. En effet, pour assumer ses fonctions, l'environnement doit maintenir les informations décrivant les activités menées au fur et à mesure de leur déroulement. Il devrait également être à même de modifier son comportement en fonction des observations effectuées, c'est-à-dire de l'évolution du développement en cours. Cela suppose des capacités de révision des connaissances sur l'état des objets et des instances de procédés. *On notera la nuance entre le terme* évolution *tel qu'il a été utilisé jusqu'à maintenant (modification de spécifications de modèles de procédés ou de schémas d'objets) et sa signification dans les systèmes dits évolutifs où l'état d'une base évolue pour refléter les changements constatés dans le monde qu'elle représente*. Rappelons aussi la distinction entre la *mise à jour* d'une base, qui consiste à modifier son état quand le monde représenté change, et sa *révision* [8, 120, 151] qui est l'arrivée d'une information remettant en cause ce qui

était connu sur le monde représenté. Keller et Winslett [121] avaient introduit la même nuance dans les bases de données relationnelles incomplètes en faisant la distinction entre les mises à jour de n-uplets, ou *change-recording updates*, qui simulent les effets d'événements survenus dans le monde représenté, et celles d'ajout de connaissances, ou *knowledge-adding updates*, qui décrivent des changements de croyances. Katsuno et Mendelzon [120] ont étendu ces travaux à des bases de connaissances propositionnelles, en formalisant les notions de mise à jour de n-uplets et de mise à jour de connaissances. Ils s'appuient sur les *postulats rationnels* de la révision (*postulats AGM*) d'Alchourrón, Gärdenfors et Makinson [8], pour proposer une axiomatisation des mises à jour.

La différence intuitive entre les deux ensembles de postulats peut se résumer comme suit. Etant donnée une base de connaissance représentée par une théorie ψ, la réviser par une formule μ en satisfaisant les postulats AGM revient à choisir parmi les modèles de μ ceux qui sont "les plus proches" des modèles de ψ, la distance entre les modèles étant définie par une relation d'ordre sur les modèles. La nouvelle théorie "ψ révisée par μ" est déterminée par les modèles retenus. Les méthodes de mise à jour choisissent, pour chaque modèle M de la base de connaissance ψ, l'ensemble des modèles de μ qui sont les plus proches de M : la nouvelle théorie "ψ mise à jour par μ" décrit l'union de ces modèles. Ces approches présentent l'intérêt évident de donner une sémantique formelle à la révision et à la mise à jour. Néanmoins, elles ne proposent pas de "solutions procédurales" pour concrétiser les opérations.

La difficulté réside dans le fait qu'une définition naturelle et immédiate de ces opérations ne peut pas être facilement formulée. Cela amène à considérer ce que l'on appelle communément le *problème du décor* (*Frame problem*) et ses deux facettes :

(i) *le principe d'inertie* : comment spécifier ce qui ne change pas?

(ii) *le problème de ramification* : comment calculer efficacement ce qui a dû changer?

Indépendamment du cadre formel sous-jacent, les approches proposées sont confrontées à la maîtrise de l'explosion combinatoire engendrée par le *problème du décor*. Celle-ci est amplifiée par la diversité des cas de dysfonctionnement et leur prise en compte. La dimension du problème est accentuée dans les systèmes physiques qui évoluent dans le temps en fonction de leur dynamique propre ou d'actions et d'événements externes. Ce type de système se doit de maintenir une connaissance évolutive sur son propre fonctionnement et sur celui de ses composants. Il doit incorporer différents niveaux de contrôle pour assurer la supervision des activités et le diagnostic de dysfonctionnements éventuels. Il doit aussi être à même de prédire ses états futurs et suggérer des actions à entreprendre pour remédier aux dysfonctionnements constatés, ou des voies à suivre pour atteindre correctement un des états futurs. En ce sens, nous assimilons totalement un environnement logiciel fondé sur les modèles de procédés à ce type de système, où le *moteur de procédés* joue le rôle de système de supervision. Nous détaillerons dans le chapitre 2, les fonctionnalités attendues de ce type d'environnement, les stratégies de réalisation, les divers formalismes et mécanismes que nous avons expérimentés et ceux que nous avons adoptés dans le cadre du projet ESPRIT ALF [25, 33, 56].

1.7.2 Interopérabilité dans les systèmes de raisonnement distribué

Les systèmes de raisonnement distribué, ou systèmes multi-agents, sont constitués d'agents coopérants de façon centralisée ou répartie à la résolution d'un problème [112]. Ils se partagent les connaissances et les tâches pour conduire un raisonnement. Dans les systèmes à tableau noir (*Blackboard*) ou dans les systèmes d'acteurs, les agents n'ont généralement pas une connaissance complète du problème ou de la solution et chaque agent peut être spécialisé dans un sous-domaine. La coopération est réalisée par une structure de données commune (le tableau noir) ou par communication de messages, respectivement (figure 1.10).

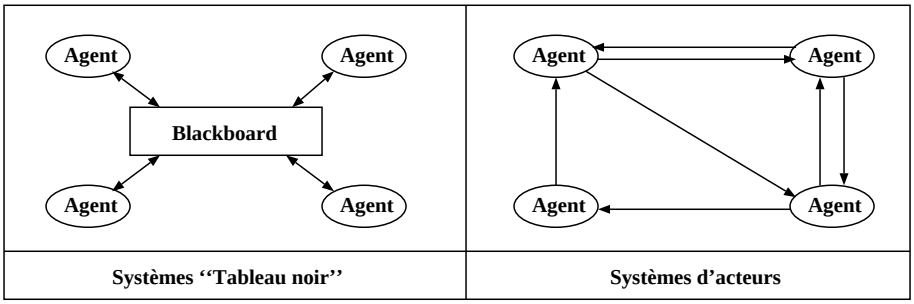

FIG. 1.10 – *Communication dans les systèmes multi-agents*

Dans le cas de *systèmes physiquement distribués*, les agents sont généralement autonomes, donc capables de fonctionner indépendamment des autres agents et de résoudre seuls les problèmes. Néanmoins, leur collaboration nécessite des mécanismes similaires à ceux évoqués dans le cadre des bases de données fédérées : décomposition de problèmes, transport de sous-problèmes entre les sites, coordination des activités entre les différents sites, etc.

Lorsque la connaissance est exprimée dans le même formalisme, il n’y a pas de problème d’interopérabilité entre les agents. Ce n’est évidemment plus le cas si les agents sont disparates : des besoins identiques à ceux évoqués dans le domaine des environnements logiciels et des bases de données se font alors ressentir pour assurer la coopération et l’interopérabilité entre les agents. Il n’est alors pas surprenant de constater que des travaux s’apparentant à ceux visant à définir un format [100, 85] ou des mécanismes [156, 157, 168] standards d’échange de données entre des outils et des environnements, sont entrepris actuellement dans le domaine des connaissances. Ces travaux visent la définition :

(i) de *langages de communication* entre les agents : ainsi, *KIF* (*Knowledge Interchange Format*) [99] peut être considéré comme un “langage pivot” de description de connaissances, et *KQML* (*Knowledge Query and Manipulation Language*) [88, 89] est un langage de communication entre agents asynchrones autonomes. *KQML* complémente *KIF* et des propositions telles que CORBA [156, 157] (ces dernières spécifient des services distribués pour l’échange d’objets entre des outils ou entre des plateformes technologiques éventuellement hétérogènes).

(ii) de *services d’interopérabilité* entre agents assurant, notamment, la décomposition de problèmes, la traduction de messages et leur routage basé sur le contenu,

(iii) et de *protocoles de collaboration ou de compétition* entre les agents.

Même si, comme nous venons de le voir, des rapprochements entre les systèmes à bases de connaissances distribués et les environnements logiciels sont possibles des points de vue de l’intégration et de l’interopérabilité, dans la suite de ce document, nous n’en ferons pas mention. Nous nous référerons uniquement à l’architecture de ces systèmes dans l’étude des stratégies de réalisation des environnements fondés sur les modèles de procédés.

1.8 Structure et contenu du document

Ce chapitre a voulu introduire la problématique de l’évolution, de l’interopérabilité et de l’intégration dans les environnements fondés sur les modèles de procédés de production de logiciels.

(i) Sans prétendre à l’exhaustivité (par exemple, nous avons passé sous silence l’interopérabilité entre les réseaux), nous avons *illustré la diversité des domaines* où des problèmes d’intégration et

d'évolution sont rencontrés. *L'évolution* est considérée selon deux aspects : le premier, celui de la *mise à jour*, est relatif à la prise en compte des événements survenus dans un monde dans sa représentation artificielle qu'est une base de données. Le second aspect est celui de la *révision* ou prise en compte de changements dans ce qui était cru sur un monde.

L'intégration présente différentes facettes et est une qualité nécessaire dans les systèmes logiciels complexes, qu'il s'agisse d'un programme d'application, d'un outil de développement, d'un système de gestion de bases de données, d'un système à base de connaissance ou d'un environnement de développement de logiciels. Cette qualité favorise la *communication et la coopération* entre les composants de ces systèmes logiciels.

(ii) Nous avons essayé de *faire ressortir la complémentarité* des domaines de la programmation, du génie logiciel, des bases de données et de l'Intelligence Artificielle. En effet, de plus en plus d'environnements logiciels se fondent sur un système de gestion d'objets offrant les mêmes services qu'un SGBD (spécialisé), en substitution au traditionnel système de gestion de fichiers. Par ailleurs, les travaux sur les types abstraits de données ont largement influencé ceux sur l'extension du système de types des SGBD. Ces capacités d'extension sont nécessitées par la diversité et la complexité des objets requis par les applications, comme le développement de logiciels, dont les objets sont difficilement modélisables à l'aide des types simples usuels. Elles sont également requises du fait des remises en cause probables de la structure des objets durant le développement de ces applications. Enfin, l'intelligence artificielle, domaine par excellence de la mécanisation du raisonnement est "un passage obligé" dès que l'on veut doter un environnement de développement de logiciels de capacités actives "intelligentes", comme le suivi des activités, la détection de dysfonctionnements ou la prédiction.

(iii) Enfin et surtout, nous avons essayé de montrer que les mécanismes assurant l'intégration, l'interopérabilité, l'évolution et la communication dans un système logiciel ou un environnement sont indissociables.

La suite de ce document détaille nos travaux.

La partie II comporte un chapitre dédié aux environnements fondés sur les modèles de procédés. *Dans la version préliminaire que constitue ce document*, nous nous contenterons d'une analyse critique de l'expérience menée dans le cadre du projet ESPRIT ALF [56]. Cette analyse nous permettra de situer et de motiver nos travaux actuels. Elle se fonde sur un document que nous avons rédigé suite à l'évaluation du projet dont nous avons eu la charge et la responsabilité [35, 33, 31]. Ce document est annoté (sous formes d'encadrés) par nos contributions majeures au projet et par les besoins non satisfaits dans le cadre du projet ou pour lesquels les solutions adoptées sont jugées insatisfaisantes.

Des informations complémentaires sont disponibles dans le document *"Recueil d'articles"* joint à celui-ci. On y trouvera :

- une présentation du système ALF [56],
- une analyse des mécanismes d'intégration [33] adoptés dans le projet,
- une description des mécanismes de raisonnement nécessaires dans les environnements fondés sur les modèles de procédés [38, 40],
- et des articles montrant nos "jeunes" travaux d'exploration du domaine [39, 29].

Le détail de travaux plus récents liés à l'intégration, à l'interopérabilité et à l'évolution est exposé dans les chapitres 3 à 6 de la partie III.

Le chapitre 3 introduit le modèle de représentation de données que nous avons défini comme support de l'étude [41]. Les concepts de ce modèle appelé GORM, pour "Generic Object-Relationship Model", (type d'entité, type d'association, liens de composition et de sous-types, etc) sont décrits de façon intuitive et une formalisation est proposée pour chacun d'eux.

Le chapitre 4 décrit les services proposés pour permettre des modifications simples ou complexes de schémas d'objets [45, 44]. Nous y présenterons :

(i) une typologie des *opérations de modification*. Celle-ci détermine un ensemble *d'opérations de base* de modification de schémas, chaque opération s'interprétant comme la modification de la spécification d'un type abstrait complexe. La classification de ces opérations fait ressortir trois catégories. *La première* est constituée d'opérations de *renommage* de types d'objets dans un schéma qui n'ont pas d'effets importants sur le schéma ou les bases existantes. *La deuxième catégorie* comprend les *opérations qui changent le domaine* de valeurs d'une fonction dans un schéma. Cette catégorie permet de mettre en exergue des cas décidables et des cas indécidables de migration des données et des programmes. Brièvement dit, si une relation formelle est identifiée entre l'ancien et le nouveau domaine, on peut, de façon déterministe, établir une correspondance entre les valeurs des deux domaines. Dans le cas où une relation formelle entre les deux domaines ne peut pas être exhibée, nous sommes "condamnés" à produire des solutions ad hoc. Enfin, *la troisième catégorie* comprend les opérations dites de *changement de structure* des types des objets d'un schéma. Les opérations de cette catégorie tombent généralement dans les cas où la migration est indécidable et où seules des solutions ad hoc nous semblent possibles.

(ii) un processus d'intégration de schémas, vu comme la construction de la spécification d'un type complexe "synthèse" de la spécification de deux types. L'exposé rapide de ce processus (implanté dans [44]) nous permettra de faire ressortir quelques opérations "usuelles" mises en œuvre lors de l'intégration de schémas : ces opérations seront appelées *opérations de restructuration de schémas*.

Le chapitre 5 débute par une discussion informelle du problème de la prise en compte des modifications d'un schéma en présence de données et de programmes. Nous y analysons les politiques de *migration immédiate et différée* et les stratégies de gestion de versions d'objets et de schémas. Nous montrons que les mécanismes nécessaires sont indépendants des politiques et des stratégies et qu'ils peuvent se formaliser en termes *d'homomorphismes de schémas*. Des homomorphismes "génériques" sont définis pour chacun des types de concept de GORM ayant le statut d'objet, au sens du paradigme objet, à savoir les types d'entités et les types d'associations. Nous proposons, ensuite, la définition de ces homomorphismes pour chaque catégorie d'opérations de modification de schéma décrite dans le chapitre 4. Ces définitions prennent la forme d'une sémantique statique et de plusieurs interprétations procédurales, ces dernières montrant que les fonctions définies peuvent être opérationnellement utilisées dans l'une ou l'autre des politiques ou des stratégies. Nous montrons enfin, que des opérations complexes de modification de schémas, comme par exemple, les opérations de restructuration, s'expriment en termes de composition des opérations de base au sein de *transactions de modification de schéma*.

Dans le chapitre 6, nous présenterons notre approche et nos propositions en matière d'interopérabilité orientée objets [43, 166, 167, 165]. Elles se fondent sur l'identification d'une relation de sous-type, entre les types des objets sur lesquels l'interopérabilité est désirée et la génération (assistée) de fonctions de coercion. Nous décrirons :

(i) le modèle d'abstraction des structures des données sur lesquelles on désire assurer l'interopérabilité,

(ii) des règles de ré-écriture des structures des types des données et de leurs instances,

(iii) et nous ferons le lien avec le chapitre précédent, pour montrer comment ces propositions peuvent contribuer à la résolution des cas *indécidables* de modification de schémas.

En conclusion de cette partie, nous situerons nos propositions en matière d'intégration et d'interopérabilité dans une vision des architectures des environnements où les facilités d'intégration sont applicables tant à des objets de fine granularité qu'à ceux de grosse granularité et où elles sont utilisables par n'importe quel composant de l'environnement (outil, programme, SGBD, etc). Nous montrerons la stratégie que nous avons adoptée et qui est en cours de réalisation : elle consiste à coupler les fonctionnalités des dimensions données et contrôle de l'intégration.

Nous terminerons (*chapitre 7*) par un bilan critique de nos travaux.

(i) Nous mettrons en évidence les liens entre l'évolution de schémas et l'intégration des données. Nous situerons également nos contributions dans une démarche de migration de "systèmes logiciels du patrimoine" (*Legacy Systems*) [36].

(ii) Nous exposerons la démarche que nous comptons adopter pour étendre nos travaux à la prise en compte de contraintes sémantiques sur les types de données ou de schémas à intégrer. Ceci nous amènera à considérer des évolutions autres que structurelles où des mécanismes de *révision* des croyances devront être mis en œuvre.

Deuxième partie

Environnements intégrés fondés sur les modèles de procédés

2

Environnements intégrés et modèles de procédés

Avertissement

Ce chapitre ne se veut absolument pas être un chapitre général sur les environnements fondés sur les modèles de procédés. Il est "orienté" vers la présentation et l'analyse critique d'un projet de type *"Recherche et Développement"* mené dans le cadre du programme européen ESPRIT (1988-1992). Le lecteur intéressé pourra avantageusement consulter [90].

2.1 Introduction

The software process is a set of related activities involved in the production and the evolution of software systems. It addresses technical as well as managerial aspects throughout the software life cycle. Most of the current software processes are implicit, except those who are hard-coded in some CASE-tools. The integration of explicitly described software processes (Software Process Model) in an environment aims at providing computerized facilities to support software development activities. It is also intended at making the environment play an active role during the development. For instance, an environment fitted with software process support facilities (process-centered environment) may decide on its own to recompile a source file whenever it is modified. It can also forbid starting an activity when predefined conditions are not satisfied, and suggest a series of actions to be performed prior to the refused activity . . .

Thence, the integration of Process Models into an environment requires making the environment informed about the activities to be carried out, about people who will carry out these activities, about conditions for an activity to start or to be considered as terminated, and so on. And the development of process-centered environments requires :

– *Process modeling concepts :* existing models [122] rely on various paradigms. Some of them are centered around the activities which constitute the process, others focus on the products, while other ones focus on the policies to be obeyed during the effective process.

– *Process model description formalisms :* they range from natural language expression to programming language-like formalisms. In the later case, the formalisms rely on various paradigms (procedural, declarative, object-oriented, . . .).

– *Process enaction mechanisms :* a software process model can be used for different purposes : assessment, improvement, automation, learning, . . . And a process-centered environment must provide mechanisms to support, assist, guide, control and monitor processes that run according to a given model.

The ALF Project falls in this category of environments. It is concerned with Knowledge Based Systems and advanced Information Systems (IS) techniques in Software Engineering Environments. Its objective was to progress on the ways of building customizable Software Engineering Environments (SEE). Considering the definition of a SEE as "a name given to a set of tools, structures, rules and procedures that together provide a framework for software development and support" [86], or as "a collection of computer-based facilities to support the activities of programmers, software engineers, system designers, project managers, etc, to achieve higher productivity and higher product quality" [199], we notice that both the definitions focus on moving from a "mechanist" point of view to another one which is more oriented toward users activities and goals. The ALF project wanted to improve the 1987-1988 state-of-the-art SEEs building strategies by introducing a higher layer in a SEE which provides at least two functionalities. The first one concern "intelligent" user assistance: as far as possible, provide the largest kinds of assistance to any type of user or any type of role. The second functionality is to provide facilities that permit to a project leader to define its own assisted software development environment.

Initially, the ALF project was viewed as a process-centered environment. Revisions of the project has lead to the development of a platform based on PCTE [196] to support the development of process-centered Software Engineering Environments [200, 191, 119, 62]. The ALF system encompasses tools to describe and to enact process models. These tools can be viewed as integrated "horizontal tools" (figure 2.1, adapted from [201]), while PCTE is used as a repository manager and as the virtual operating environment.

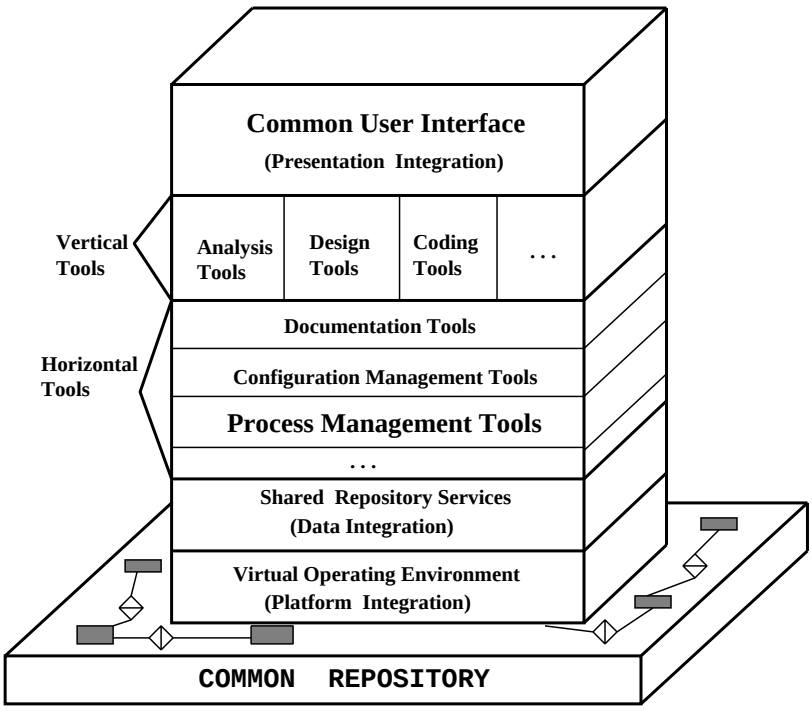

FIG. 2.1 – *Integration Layers*

This paper proposes a critic overview of the project following the project assessment [35, 31]. Sec-

tion 2.2 presents and discusses the ALF concepts for software process modeling. Section 2.3 analyzes the software process description language, while section 2.4 presents and discusses the ALF system structure and functionalities. Section 2.5 ends the paper by concluding remarks.

2.2 The ALF Concepts for Process Modeling

The design and the choice of the ALF software process description model, called Model for Assisted Software process (MASP), are based on a multi-paradigm approach (The rationales for that are developed in [78, 195]). In order to give the software process model and the ALF system a high degree of generality, the model is neither specific to a development method, nor life-cycle model dependent [155], nor programming-language dependent [191]. It is mainly designed for controlling and automating software processes. Assistance support is also a big concern of the model. Learning, in the sense of learning machines in Artificial Intelligence, is not dealt with. The figures 2.2 and 2.3 compares ALF's purposes to similar systems or projects.

Purpose	Projects or Systems
Communication	MVP
Control	**ALF**, MARVEL, Arcadia, ISTAR, SPECIMEN
Automation	All
Analysis	MELMAC
Evolution	Adele-2, EPM, MELMAC, Arcadia, EPOS
Improvement	Not demonstrated by any model or system
Learning	Not demonstrated by any model or system
Standard	No standard exists in the field

FIG. 2.2 – *Purpose of the Model*

Activity	**Systems or Projects**
Whole life-cycle	**ALF**, OIKOS, MELMAC, EPOS, SPECIMEN
Version and Configuration Management	Adele-2
Development and Maintenance	MARVEL

FIG. 2.3 – *Life Cycle Coverage*

Further, the concepts enable the specification of processes for individual working as well as for group working (figure 2.4), and they also enable a wide coverage of roles [80].

Scale	**Systems or Projects**
Individual	**ALF**, MARVEL, Adele-2
Group	**ALF**, Arcadia, EPOS

FIG. 2.4 – *Role Coverage*

2.2.1 Process Modeling Concepts

There is a common agreement on the fact that a Software Process Model should at least, describe object-structures and relationships, operations that act on these, and policies or constraints that govern the process. The expression of all these aspects in an assisted software process model seems to be a problem very close to the specification of external system behavior, in which a great number of approaches are well known (such as Finite state machines, decision tables and decision trees, State-charts, Petri nets) and they have a lot of common aspects. They use operations and/or abstractions of operations. Operations act on objects which may be described within an object schema. Actions are undertaken under the control of events or conditions (FSM, Petri Nets, SDL) and many approaches propose features to express orderings of actions. Furthermore, some of them use logical constraints to characterize particular states of the process.

In order to support the description of a reasonable number of existing methods, our model is a combination of these features. Therefore, an ALF software process model, called Model for Assisted Software Process (MASP), is composed with:

– `an object model:` it is a typed Entity-Relationship data schema extended with sub-type relationship and multivalued attributes,

– `an operator model` in which each operator-type is defined by a profile, a precondition and a postcondition that are used for acceptability and control of an operation and as basis for reasoning. Each operator type will be implemented by an operator, using an Instantiation tool (see section 2.4.2.1).

– `a rule model` which describes actions to be triggered. A rule has a condition part which expresses *when to do* an action (immediately, as soon as possible, at user defined instants, or when a given condition becomes true), and an action part which notifies *what to do*,

– `an ordering model` which specifies constraints on operators activation, like precedence or simultaneity.

– `a characteristic model` to express integrity constraints and the ultimate goal of a model.

2.2.2 Structuring Mechanisms and Component Relationships

The basic components of a model are linked to each other by a set of relationships and construction mechanisms. Object-types are described by a kind of *aggregation mechanism* where attribute-types are grouped together to describe object properties. Further, a variety of relationships among object-types are expressible (type importation, type extension, subtyping ...).

Relationships among operator types and object types are expressed by using typed formulae as preconditions and postconditions, where variables can range over the types in the object model. Similarly, relationships between the rule model, the object model and the operator model are *de facto* expressible since a rule can encompass logical expressions and operator-type names.

Further, relationships between operator types can be explicitly mentioned in the orderings. ALF proposes classical control structures (sequence, iteration, ...) to relate operators to each others, and, at the same time, to enable expressing constraints among them. In systems like MARVEL [119] or OIKOS [11], these kinds of control structures are implicit. MARVEL uses operator's preconditions and postconditions to automatically determine the applicability of a sequence of operators. No parallelism

is expressible in MARVEL. Arcadia [191] uses the Ada tasking mechanisms, and EPOS [73] uses static planning and dynamic triggering.

Finally, relationships can exist between MASPs thanks to the *MASP structuring* mechanism. Indeed, an operator-type may be refined into a MASP, with its proper object model, operator model, and so on. This mechanism encourages a disciplined approach to process modeling. Therefore, a software process model can be described as a hierarchy of MASPs.

The figure 2.5 summarizes the ALF structuring capabilities with regard to some existing systems.

Mechanism	**Systems or Projects and associated features**
Decomposition	**ALF**: Sub-Models, subtyping; EPOS: subtyping, task decomposition; Adele-2: task structuring
Composition	**ALF**: type extension; HFSP, MVP, Statemate
Specialization/Generalization	**ALF**: subtyping and type extension; MVP
Reuse	**ALF**: Importation, by referring to some component names
Model Versioning	EPOS: type versioning
Multiple View	**ALF**: PCTE dependent; MVP, Statemate
Inheritance	**ALF**: PCTE dependent; Adele-2, MARVEL and EPOS: "naturally" induced by their Object-Oriented Approach

FIG. 2.5 – *Structuring Mechanisms*

2.3 The Software Process Description Language

The foundations of the MASP description language (MASP/DL) are induced by the underlying influences of the model. This section shows the impact of these influences on the MASP/DL design, and discusses the deficiencies and the expressive power of the language.

2.3.1 Foundations of the MASP/DL

The object-model description language is PCTE's data definition language. It permits the specification of the structure of the object types (attribute types which describe an object type) and the relationships between the object types (link or pair of link-types). It also permits the re-use of object type and attribute type descriptions to support type importation and type extension.

Examples

- *Attribute type definitions*

```
Version#:   INTEGER := 0;
tested:     BOOLEAN := FALSE;
```

- *Object type definitions*

```
c_program_dir: SUBTYPE OF sys-dir;
c_program : SUBTYPE OF program;
```

- *Object type extension*

```
EXTEND c_program WITH
        ATTRIBUTE tested, version_number;
        LINK
```

```
        src : COMPOSITION LINK TO c_source;
        obj : COMPOSITION LINK TO c_object;
END c_program
```

Characteristics appear as logical expressions (**EVALUATE** clause) associated with an optional event-part (**ON** clause) which indicates *when* the expression must be evaluated.

Example

```
ON UPDATE OBJECT body_compiled OF body_unit
        EVALUATE body_unit(_b) AND body_compiled(_b,TRUE)
```

Logical expressions are built using conventional comparison predicates, logical connectors and quantifiers, and a set of predicate symbols that issue from a predicative view of the object model. The later set encompasses two classes of predicate symbols: The first class contains "meta-predicate" symbols that logically describe the object model. Meta-predicates are intended to be evaluated against META-DATA under the Closed World Assumption [175] (CWA), i.e. against information that concern the model itself (see section 2.4). For example, the following sentences describe the predicative view of the object type *c_program* and its associated link-types *src* and *obj*.

- *IS_AN_OBJECT_TYPE*(c_program) $\wedge$ *IS_A_SUBTYPE*(c_program, program)
  $\wedge$ *OBJECT_ATTRIBUTES*(c_program, [tested, version#])
- *IS_A_LINK_TYPE*(src) $\wedge$ *LINKS*(src, c_program, c_source)
  $\wedge$ *CATEGORY*(obj, COMPOSITION)
- *IS_A_LINK_TYPE*(obj) $\wedge$ *LINKS*(obj, c_program, c_object)
  $\wedge$ *CATEGORY*(obj, COMPOSITION)

Pour l'élaboration de la définition du langage des expressions logiques utilisées dans les *Characteristics*, dans la spécification des opérations et dans les expressions des *règles*, nous nous sommes inspirés des travaux sur la logique et les bases de données relationnelles. Plus précisément, la reformulation logique proposée par Reiter dans [175] nous a été d'un grand secours. Notre avons proposé une reformulation logique du modèle d'objets adopté. [175] fait correspondre à chaque relation n-aire $R(D_1, \cdots, D_n)$, sur les domaines $D_1, \cdots, D_n$, un prédicat $r(arg_1, \cdots, arg_n)$ de même arité que R dont l'extension est obtenue en faisant correspondre à chaque tuple $R(v_1, \cdots, v_n)$ de la relation R, une instance $r(v_1, \cdots, v_n)$ du prédicat r. En outre, à chaque domaine D_i est associé un prédicat unaire $D_i(_)$ dont l'extension est obtenue par projection des relations sur les "colonnes" qui prennent leurs valeurs dans le domaine. On aboutit ainsi à une théorie typée du premier ordre.
Dans notre cas, le besoin de traiter les descriptions de modèles de procédés comme des objets nous a amené à faire la distinction[a], entre ce que nous venons d'appeler les *méta-prédicats* et les *prédicats*. Le modèle de données sous-jacent s'appuie sur deux concepts : celui de type d'entité (ou type d'objet) et de type de liens (un type d'association est défini dans ce modèle par un couple de types de liens). Nous avons alors été amenés a distinguer *deux ensembles de "méta-prédicats"*, un par concept. *Concernant les types d'objets*, au lieu d'associer comme le fait Reiter pour les relations, à chaque type d'objet $O(a_1, \cdots, a_n)$ un prédicat de même arité que O, nous avons opté pour une solution consistant à associer à chaque type d'objet O d'un schéma une instance *Is_An_Object_Type(O)* du "méta-prédicat" unaire *Is_An_Object_Type(_)* qui joue le même rôle que le prédicat de domaines de Reiter. Pour spécifier les attributs associés à un type d'objet, nous avons opté pour une solution similaire à celle proposée dans *LDL* [69] où un argument d'un prédicat peut être un ensemble (voir $DATALOG^{ens}$, l'extension de DATALOG avec des ensembles notamment dans [61]) : à chaque type d'objets $O(a_1, \cdots, a_n)$, où les a_i sont les attributs de O, est associé une instance du "méta-prédicat" *Object_Attributes(O, [liste des noms des types des attributs de O])*.
Nous avons adopté une approche similaire pour définir logiquement *les types de liens* : l'extension du "méta-prédicat" *Is_A_Link_type(_)* énumère les types de liens d'un schéma, l'extension du "méta-prédicat" *Links(_, _)* décrit les types des objets source et cible de chaque type de lien et l'extension du "méta-prédicat" *Category(_, _)* décrit la nature de chaque type de lien.

[a] Cette distinction est introduite pour les besoins de l'exposé : formellement, un "méta-prédicat" est un prédicat!

The second class of predicates encompasses predicate symbols that apply to instances of the object model, and they are evaluated against the object base (called DATA in section 2.4), under the CWA too. In the example below, *body_unit(_b) AND body_compiled(_b, TRUE)* intuitively means that _b is an instance of the object-type *body_unit* and that *body_compiled* is a boolean attribute of *body_unit* that must evaluate to TRUE. This can be more formally stated using predicates and meta-predicates as follows:

IS_AN_OBJECT_TYPE(body_unit) $\wedge$ *OBJECT_ATTRIBUTE*(body_unit, _x)
$\wedge$ *IS_ELEM*(body_compiled, _x) $\wedge$ *IS_AN_OBJECT*(_b)
$\wedge$ *IS_OF_TYPE*(body_unit, _b)
$\wedge$ *OBJECT_ATTRIBUTE_VALUE*(body_compiled, _b, TRUE)

La définition des *prédicats* a été opérée selon la même démarche que celle suivie pour la définition des "méta-prédicats". L'extension des prédicats *Is_An_Object(_)* et *Is_A_Link(_)* identifie les instances des types d'objets et des types de liens du schéma, l'extension du prédicat *Is_Of_type(_, _)* identifie le type de chaque instance. En outre, les valeurs des attributs des objets sont logiquement spécifiées par l'extension du prédicat *Object_Attribute_Value(_, _)*.

La multiplication des prédicats, comparée à l'approche de Reiter pour les bases de données relationnelles, peut à première vue, sembler inutilement complexe. Néanmoins, elle a le mérite de nous permettre de rester dans le cadre de la logique du premier ordre et de définir une sémantique logique des formules construites à l'aide de ce langage ([30] est une contribution (partielle) à la définition de cette sémantique). Comme nous l'avons déjà mentionné, celle-ci est définie sous l'hypothèse du monde clos (*Closed World Assumption*).

Néanmoins, *d'un point de vue opérationnel*, les extensions des prédicats ne sont pas représentées sous leur forme logique mais elles sont gérées par le Système de Gestion d'Objets (Emeraude-Pcte) utilisé pour le développement du système ALF. Nous reviendrons sur ce point dans le paragraphe 2.4. Nous dirons, pour terminer que ce langage a servi dans d'autres travaux, comme ceux développés dans [12] qui s'appuient sur une logique des croyances, ceux développés dans [102] qui étendent le langage d'expression logique par des modalités temporelles et ceux décrits dans [198] pour la validation statique de descriptions de modèles de procédés.

The rule model description language is influenced by production rules. Indeed, the right-hand-side part of a rule (**THEN** clause) specifies an action to be performed when the rule becomes firable, and the left-hand-side part (**IF** clause) specifies *when* the rule is evaluated (**ON** clause) and *under what a condition* the action is to be performed (**EVALUATE** clause).

Example

IF *ON EXIT OPERATOR* compile(_x)
EVALUATE successful_compilation(_x) *EVALUATED* TRUE
THEN link(_x,/lib/maths.l)

The operator model definition language is influenced by abstract data types specifications [105]. Each operator-type is specified giving a signature, a precondition, and a postcondition. An additional feature indicates whether the operator may be processed by the system alone, or it needs any user's cooperation (the **KIND** clause). This information is useful for the "active role" of the environment i.e. the environment may decide on its own to activate a non-interactive operator. Moreover, the MASP/DL allows importation of operator-type definitions from a MASP into another one.

Examples

audit: (IN _p: project; OUT _r: report)
PRECOND: audit_authorized(_p) = TRUE
POSTCOND: audit(_p) = TRUE AND proj_to_rep(_p, _r, NO_KEY)
KIND: *INTERACTIVE*

IMPORT audit *AS* imported_audit

The ordering model is described as annotated path expressions. The specification of authorized operators' sequences (e.g. *edit(x) ; compile(x)*), simultaneity (e.g. *{print(x)}*), concurrency (e.g. *read(x)* || *print(x)*), grouping (e.g. *(lex(x); parse(x)*), alternative (e.g. *print(x)* | *pretty_print(x)*) and iteration (e.g. [*test(x)*] *(1 TO 5)*) are among the possibilities which can be formulated in the Ordering Definition Language.

2.3.2 Discussion

A MASP description encompasses structural aspects (the object model), behavioural aspects (the operator model), control aspects (characteristics, preconditions and postconditions, orderings) together with policy aspect (rules and orderings). But, the MASP/DL does not enable encapsulating object-types together with their proper operator-types and constraints, and it does not incorporate supports for user-defined data types. For instance, the MASP/DL enables using sets and lists of items, but the classical operations on these data types are not available (except the set membership operation). That means that the MASP description language is not a strongly typed language. These drawbacks are mainly due to PCTE limitations. More precisely, they are due to the choice of the object-model of PCTE as an object-model. (Alternatives are being explored to give the MASP an "object-oriented flavor" [41, 66, 57]).

Further, neither the MASP concepts, nor the MASP/DL can apprehend fine knowledge about the objects involved in a software process. Indeed, the model only deals with gross-grained objects, and it cannot help in specifying their fine structures. Consequently, consistency of objects' contents cannot be ensured. The values that are associated to the objects' attribute-types are the only means to decide whether an object is consistent or not. In addition, the MASP/DL does not incorporate powerful features to express temporal constraints, such as the time duration of an activity (it may be partly expressed using DATE attribute-types and event specification).

From a theoretical point of view, the hybrid nature of the model makes it hard to find out a unique formal basis (contributions to a formal semantic definition is done in [118]). As a first consequence, we tried to give the MASP/DL an operational semantics (see section 2.4), and as a second consequence, formal properties, like consistency and completeness of a given MASP description, cannot be proved. This obliged us to change a fundamental work on MASP consistency checking into a less ambitious one which consists in tracking MASP inconsistencies [64, 111, 198]. But, to our knowledge, no existing process model incorporates features to prove formal properties of a model.

Furthermore, according to MASP writers point of views, the MASP/DL requires a long learning curve, and it produces descriptions that are hard to read and hard to understand. Further, it requires particular skills : database and knowledge base modeling, logical formulation . . .

Despite these criticisms, the MASP/DL proved to be powerful, as stated by the evaluation [122] done during the sixth International Software Process Workshop [116]. Moreover, we consider the current version of the MASP/DL as a first layer which needs more abstract layers (especially, to make it less PCTE-dependent) fitted with appropriate tooling for the design and the specification of a model.

2.4 The ALF system

The ALF system is implemented on top of PCTE, and it benefits from PCTE's services, like networking facilities, execution control and management of the object store. Those services have been augmented with additional ones for dealing with the notion of software process model. Indeed, the enaction mechanisms (see the role of the enaction toolset in section 2.4.3.1) require the availability of the various types of knowledge encompassed by a MASP description, together with knowledge about the state of a project under-development (which activities have been performed, what objects have been

produced, and so on). Both kinds of knowledge are maintained in an “ad-hoc” PCTE object base whose data schema is partly shown in figure 2.6. It encompasses two kinds of data: *META-DATA* which is a set of information related to the knowledge that is described by a MASP, and *DATA* which is a set of information that concern on-going software processes. Both types of data are used by the *MASP enaction toolset*.

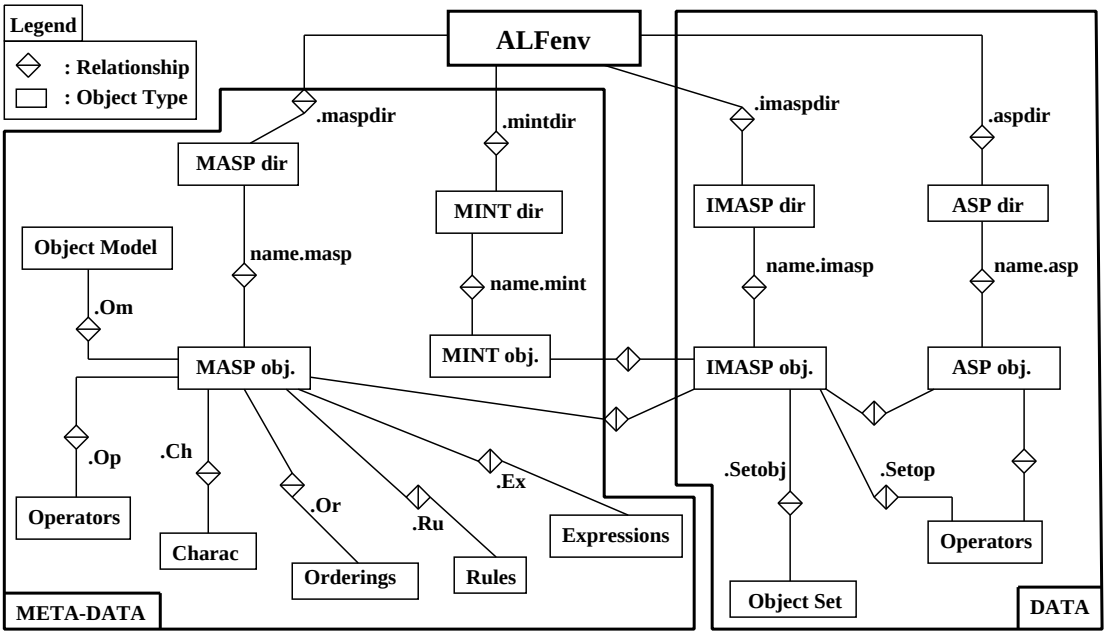

FIG. 2.6 – *Part of the ISK Object Schema*

META-DATA is similar to the data dictionary in relational database management systems. It is produced and maintained by some of the *MASP management toolset*, while DATA is created and maintained by the environment toolset and by the *INSTANTIATOR* (figure 2.7).

The ALF system architecture (figure 2.8) is defined by four interacting functional layers. These are examined and criticized below.

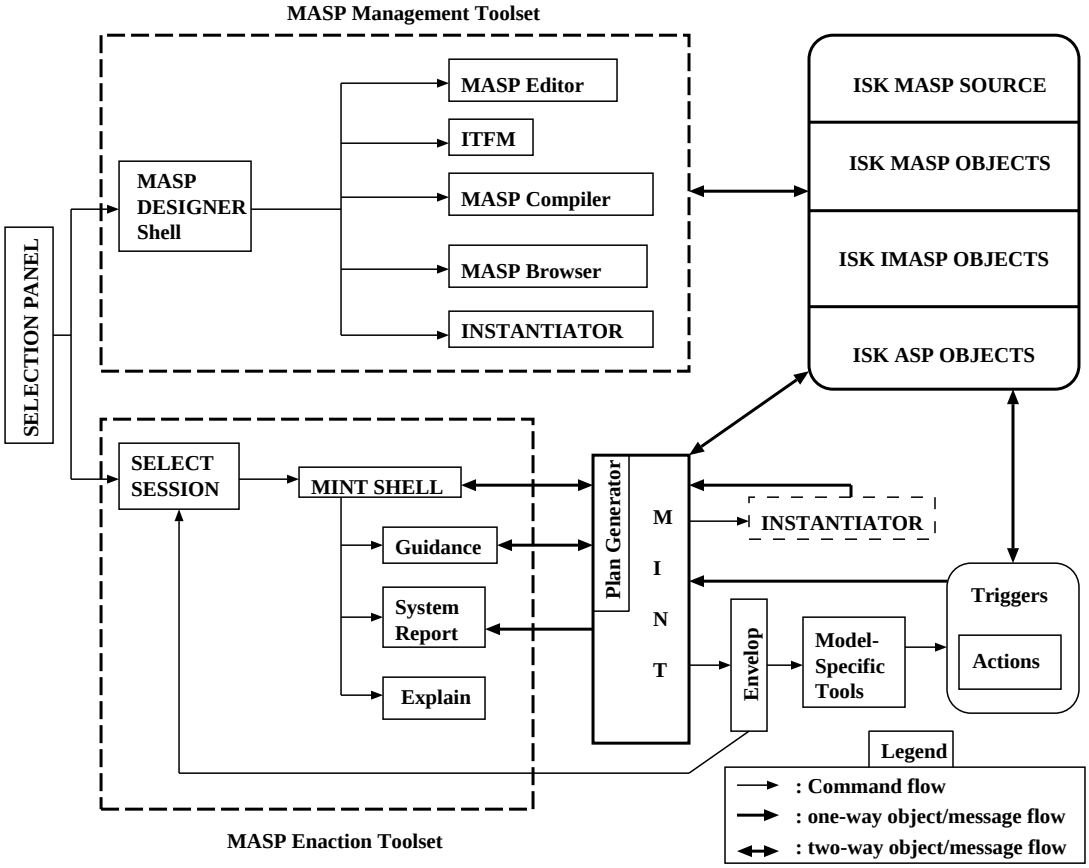

FIG. 2.7 – *ALF System Toolset*

Nous noterons que le choix de représentation des objets ne diffère pas de celui communément adopté dans les SGBD traditionnels : le schéma d'une base est lui-même géré comme une base. Dans notre cas, les descriptions des modèles de procédés (dont un schéma partiel est donné dans la partie gauche de la figure 2.6) sont gérés comme des objets persistants dont les valeurs servent à évaluer ce que nous avions dénommé *"méta-prédicats"* dans la formulation logique du langage de description de modèles de procédés. Les valeurs des types des objets contenus dans le modèle d'objets des procédés sont aussi des objets persistants : ces valeurs constituent la base de données d'un projet logiciel en cours de développement et des modèles de procédés en cours d'activation.

Nous noterons également que nous nous trouvons dans une architecture similaire à celle des SGBD déductifs que nous avions évoqués dans le paragraphe 1.6.2.2. Nous disposons d'une représentation non logique des extensions des objets dans la base d'objets et de formulations logiques mettant en jeu des prédicats dont l'évaluation peut requérir l'accès aux valeurs des objets des modèles de procédés (c'est la cas des "méta-prédicats") et l'accès aux valeurs des objets qui décrivent l'état des procédés en cours d'activation ou du logiciel en cours de développement (cas des "prédicats"). Comme nous avions déjà eu l'occasion de le mentionner dans le cadre des bases de données déductives, des mécanismes de correspondance entre les deux types de représentation sont nécesssaires. En fait, dans le cadre de la réalisation du système ALF, comme nous allons le voir, le problème a été éludé en donnant une implantation procédurale aux expressions logiques.

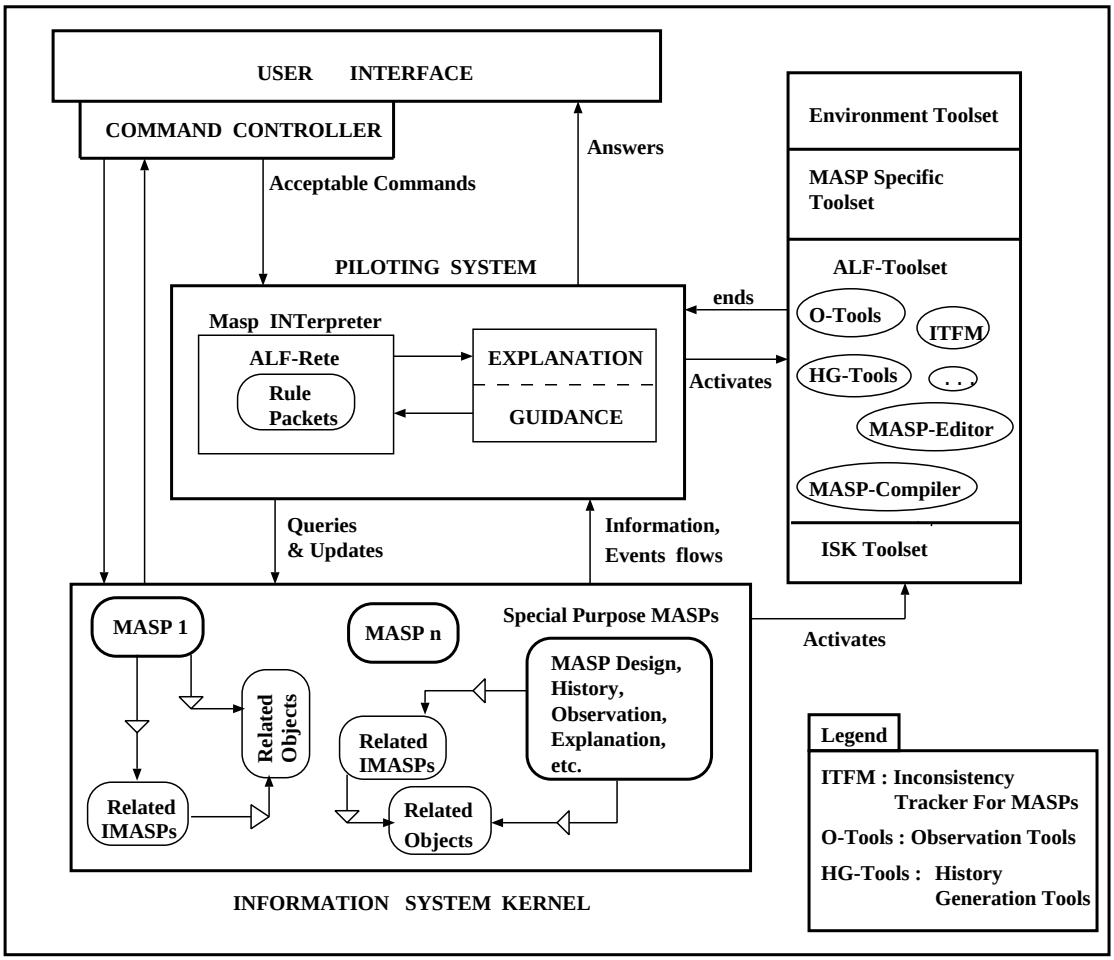

FIG. 2.8 – *ALF System Structure*

2.4.1 The User Interface Management System

The *User Interface* layer provides different interfaces to the various kinds of users. Namely, interfaces will be provided for designing MASPs and checking their consistency, other ones for software developers and project managers . . . , but in the current status of the system, when logged-in, a user is only offered the MASP management and the MASP enaction menus (respectively called MASP Designer Shell and SELECT SESSION in figure 2.7). The ALF User Interface Management System comprises Presentation, Dialogue Control and Application Interface Level layers. The presentation layer consists of a Presentation Manager, a Window Manager and the X-window system. The dialogue controller understands how user/application dialogues are built and handles parts of them on behalf of the application. Dialogues are defined by dialogue objects like buttons, windows and menus. Finally, the Application Layer is an event-driven system which interacts with the Dialogue Controller.

2.4.2 The Information System and the MASP Management Toolset

The *information system* layer (figure 2.8) corresponds to the data integration layer. It uses the services of the Object Management System of PCTE.

2.4.2.1 The MASP management toolset

The Information System Kernel (ISK) provides a data schema which models the Meta-MASP concepts (see figure 2.6 for a partial view of the ISK schema). The META-DATA part of the ISK schema is instantiated by some of the MASP management toolset. Indeed, the *MASP Editor* is a syntax-driven tool that permits a user to input a MASP description and to store it in the ISK. Following the MASP edition, the *Inconsistency Tracker For MASPs* (*ITFM*) can help in a partial semantic validation of the syntactic description. The ITFM mainly detects undefined references (like missing object-type or operator-type definitions) and some inconsistencies in expressions that appear in the rule model, in the characteristics and in the operator model. Afterward, the *MASP Compiler* proceeds in the instantiation of the META-DATA. It produces three outputs: the PCTE representation of the MASP description according to the data schema, a set of triggers associated with the events that are present in the MASP, and a set of rule packets that are required by the MASP INTerpreter (MINT).

The DATA part of the ISK schema is instantiated by the *INSTANTIATOR* tool and by the environment toolset. Indeed, follow-up to the MASP compilation, the INSTANTIATOR may be invoked to establish links between operator-types and actual tools in the environment toolset. Actual tools are constituted of "classical" development tools (for instance, the *edit* and the *compile* operator-types may be instantiated by *emacs* and *cc* tools) and possibly, by MASP specific tools, i.e. special purpose user-developed tools. The result of the INSTANTIATOR activity is called an Instantiated MASP (IMASP).

An IMASP contains a set of objects that are instances of the MASP objects types, a set of operators that conform to the MASP operator types, and the MASP prescriptions (rules, orderings, characteristics). An IMASP can share instances with another one so long as the types used in their respective MASPs are compatible.

A structured set of MASPs is instantiated into a structured set of IMASPs that share object and operator instances. It can be associated to a specific role in a team and can be considered as a working space for the role [80]. Therefore, the project manager, the software developers, the quality officer . . . can have their own process models. Again, instance sharing can be used to allow different roles to share objects and operators, and to permit to a role to introduce new prescriptions on the shared components. In this way, one role may influence another process.

It is worth noticing, first, that ALF enables different instantiations for a single MASP, that means that a given MASP may be used for different developments. Second, ALF does not require a complete instantiation of a MAP, i.e. all the operator-types need not have an operator statically associated to them. Indeed, the INSTANTIATOR tool may be dynamically activated during the MASP enaction whenever one attempts to call an operator which is not linked to an operator-type.

Moreover, in order to integrate a tool in an ALF-based environment, ALF provides a mechanism that permits one to encapsulate the tool in an envelop. The envelop contains a *preface* that specifies actions to be performed before activating the tool, a *body* that maps the formal specification of the operator-type to the call message structure of the actual tool and a *postface* that indicates actions to be performed at the end of the tool execution (like updating the DATA part of the object store).

2.4.2.2 Discussion

Therefore, in order to achieve strong data integration [33], the ISK representation of the ALF object store is common to all the system components. Initially, we envisaged the development of *Renderer Agents* to transform the unique form of representation into "ad hoc" ones, whenever this is needed (for instance, a renderer agent would present, in a PROLOG-like form, information extracted from the ISK to the ITFM, and another agent would present information in the accurate form to the ALF's inference engine). However, during the system implementation, it was found more efficient and more

convenient to start either from the syntactic description of a MASP (for the ITFM needs) or from the MASP Grammar (to produce the rule packets in the compilation phase), rather than starting from the ISK representation.

Further, a set of primitives are provided for making the ALF components and the ISK communicate to retrieve, store, modify or delete information. But, similar capabilities are not yet provided to the user to permit her/him to query a MASP or part of it, except the *MASP Browser* that enables the browsing of the META-DATA.

Comme nous l'avions fait remarquer plus haut, la nature procédurale de l'implantation n'a pas nécessité le développement de mécanismes d'interopérabilité des composants du système ALF. Cependant, si cette solution a été adoptée, en partie à cause de la nature "recherche-développement" du projet, elle nous a amené à considérer de façon plus précise les problèmes d'interopérabilité entre des systèmes logiciels hétérogènes. Particulièrement, dans le cadre des environnements fondés sur les modèles de procédés l'hétérogénéité se pose en termes de représentation des objets et en termes de paradigmes des composants de l'environnement. Certains sont des logiciels "procéduraux" classiques et d'autres se fondent sur des mécanismes de raisonnement. Le rapprochement [32] que nous envisageons avec les travaux déjà cités [99, 88, 89] sur l'interopérabilité dans le domaine de l'intelligence artificielle ne peut, à notre avis qu'être bénéfique.

2.4.3 The Piloting System Layer and the MASP Enaction Toolset

The *piloting system* layer is in charge of coordinating the activity of the system's components. It is the layer where the MASP is enacted by the MINT. The MINT is implemented by the ALFRete inference engine and PCTE's triggering, communication and concurrency control mechanisms. ALFRete extends the rule-based XRete [197] system with communication facilities with PCTE's OMS [194], with backward reasoning, composite structure facilities and dynamic activation of the rule packets that issue from the MASP compilation.

2.4.3.1 The enaction toolset functionalities

The initial objectives of the enaction were :

- to enable a user to do actions,
- to gather observation data about the users and the system itself,
- to control the state of the object base and the applicability of the operators,
- to take initiatives on behalf of the users,
- to provide various guidance facilities, including *what to do next, how does "something" work, how to do "something", what are the activities that can be performed*,
- to provide various kinds of explanations to the users.

The cooperation of the MASP concepts to these objectives and the limits of the implementation are presented hereafter (see [38] for more details).

The first reaction of the MINT to a user's initiative is to *control* whether the activity the user wants to perform is legal or not, considering the current status of the software process and the process model. The control is performed thanks to the ISK knowledge: is the operator linked to an operator-type? are its parameters of the accurate types? is the operator-type's precondition satisfied? is the initiative

conform to the orderings? . . . . The *System Report* tool permanently informs the user about the activity of the ALF system. Follow-up to the control phase, we may be faced to two situations: either `the control is successful` and the operator is activated (see figure 1.1 in appendix 1), or `it is unsuccessful` and the MINT rejects the user's demand. In the latter case, *guidance* facilities may help in understanding what happened (*Can it be done* guidance facility) and in identifying the actions that must be performed prior to the initial demand (*How to do it* facility). *Can it be done* verifies that an action is a legal one, and if it is not, it produces the reason why (figure 1.2 in appendix 1). *How to do it* provides the user with a list of actions (a plan) a user can perform to reach a specified goal (figure 1.3 in appendix 1).

Operator planning generation (*Plan Generator*) and execution facilities can also be activated by the MINT whenever it notices a prescription violation (precondition, characteristic, ordering) (figure 1.4 in appendix 1). Furthermore, a flow of triggers and events may be passed by the ISK to the MINT to inform the latter about what happened in the object base. The raised events may then make some rules candidate to firing, and the MINT ensures effective firing of the candidate rule(s).

2.4.3.2 Discussion

The current implementation only deals with characteristics which encompass an event-part. The limitation is due to technical reasons and to performance considerations : indeed, for instance, how to implement a process which permanently checks characteristics violation, and in such a case, are we prevented from the situation where the system permanently tries to enforce the characteristics ? Further, the plan generator does not consider the orderings while generating plans, and it works on formulae that are free of disjunctions (i.e. without the logical OR). This problem is related to truth maintenance in knowledge based systems, and it is not fully mastered to-day.

Moreover, if the plan generation or the execution of a plan fails, the consequences of the user's initiative should be (carefully) undone. Indeed, classical rollback techniques cannot be applied to software development activity: that is, the result of an activity, such as edit, for example, cannot be removed just because something wrong happened. Software processes are long-term processes (longer than a session) that share persistent objects, and a model for the management of long-term transactions is needed. This model should allow negotiated roll-backs. In this context, no specific recovery mechanism is provided by the ALF system. However, PCTE and some tools provide ALF with transactions, backups and archives. But, the current ALF system does not manage long term or ested long-term transactions (see [104] for prospective work on the topic).

From another standpoint, the scope of guidance facilities have been limited to *can it be done* and *how to do it*. *What can be done* and *what if* facilities have been designed but are not yet implemented. And *how does it work* is not designed nor implemented. Furthermore, for the moment, the *explanation* facility has been restricted to on-line predefined manuals on the various topics related to the ALF concepts and system. Finally, process history generation mechanisms are implemented by system reports that keep track of all the user's initiatives, and this mechanism is obviously insufficient to meet the requirements of an audit service. Nevertheless, the ability to model observations as a proper MASP may be considered as an intermediate solution.

2.4.4 The executive system layer

The *executive system* layer is in charge of executing operations. It manages and controls system processes called Assisted Software Processes (ASP). Those are created and launched by the MINT on an IMASP. In this framework, PCTE basic mechanisms are intensively solicited to safely ensure concurrency control and data sharing.

However, as mentioned before, recovery mechanisms and long term transaction mechanisms are not available in the current ALF system. Further, the various kinds of parameter passing (IN, INOUT, . . .), which are permitted in operator-type specifications, are not implemented due to the lack of accurate concurrency control mechanisms.

2.5 Concluding remarks

We conclude this paper by three considerations.

The first one concerns the process model notion. Indeed, the specification of a model appeals to knowledge acquisition techniques, and as such it gives rise to many questions. The most important one is the validity of the specified model. It seems clear that an incorrect model will deliver an incorrect support. In the framework, considering the difficulty to achieve formal validation of a process model, we feel that simulation techniques could be used as well. In the post-ALF context, we envisage using these techniques to validate a compiled MASP before instantiating the MASP-based environment. From another standpoint, process modeling should be a progressive and an incremental activity. Indeed, we feel that the completeness of a process model may not be a desired quality of a model. But, merging of models or dynamically changing parts of a model may be more useful mechanisms under the condition that the result of the change remains valid. In this context, we are investigating the potential benefits of an object-oriented approach to process model evolution.

Les travaux exposés dans la suite de ce document constituent une étape vers la prise en compte de l'évolution et de la définition progressive et incrémentale des modèles de procédés. Dans le paragraphe 1.3 du chapitre précédent, nous avions discuté les diverses facettes de l'évolution des environnements. Les contributions que nous détaillons dans la partie III vont dans cette voie.

Finally, the modeling of creative tasks revealed to be the most difficult part in the specification of a process model. A trivial solution would consist in modeling such an activity as an operator type that will be instantiated by a knowledge-based tool. That means that the knowledge that is related to the creative tasks is included into the tools that support the tasks and not in the process model. Similarly, the extension of our approach to capture fine knowledge on the objects, requires introducing additional features to our model, and the contributions, toward this aim, of strong data typing and accurate support mechanisms are ways being explored.

The second concluding consideration relates to the scope of the enaction. We tried to take into account the state-of-the-art in psychological experimentation with man-machine communication in the design and the implementation of the ALF system. We avoided over-automation of process-model enaction to keep the system "humanly" understandable. Indeed, the need and the benefits of a wide range of assistance functions have not to be proved, but the system's initiatives should be tunable and in a certain way, they should not escape the user control. Nevertheless, to-day, we cannot state any sound result on the user acceptance of the ALF system.

The final consideration concerns the benefits of the layered architecture for process-centered environment. It is clear that the choice of PCTE as the underlying development platform alleviated the development effort and helped in focusing on process-model related mechanisms. But, at the same time, it impacted the design of the process model. Nevertheless, we feel that the prominent industrial

objective of the ALF project (provide PCTE with an initiative engine) has been reached, even though the current MINT implementation does not meet all the initial requirements. Further, the ALF experiment brought partial evolutions to PCTE. Triggering mechanisms and multi-valued attributes are among the extensions that were needed in the ALF project and that are achieved. However, from the object model perspective, the evolution toward a state-of-the-art (object-oriented "flavor") object model seems costly. Further, PCTE extension with long-term transactions needs more deep studies before being implemented.

From another standpoint, the current prototype proves the validity of the approach toward process-centered environments. Nevertheless, the system needs severe performance improvements, as it needs appropriate tooling to support process-model design and verification.

Acknowledgements : This work was partially sponsored by the Commission of the European Communities under the ESPRIT programme (Project Ref. N◦ 1520).

Troisième partie

Evolution, Intégration et Interopérabilité Orientée Objets

Avant propos : Evolution et Intégration de schémas

Dans des systèmes logiciels dont la durée de vie peut s'étendre sur plusieurs années, des remises en cause de décisions de conception ou de réalisation interviennent naturellement. Ces remises en cause sont nécessitées par la découverte d'éventuelles erreurs dans la solution qu'est censé apporter le système logiciel, par la nécessité de lui faire assurer de nouvelles fonctionnalités ou encore par le désir d'en améliorer la qualité ou les performances, le cas extrême étant la migration de la totalité du système logiciel pour l'adapter à de nouvelles technologies ou pour le faire fonctionner sur une nouvelle plateforme. Dans chaque cas de figure, il semble évident que le système résultant des adaptations ou des améliorations apportées ne devraient pas avoir d'incidences "fâcheuses" sur les services rendus par l'ancien système. Comme nous l'avons évoqué dans le chapitre introductif à ce document, les modifications peuvent indifféremment concerner la sémantique du système ou de ses composants, la nature de ses entrées ou de ses sorties. Bien que des techniques, comme l'encapsulation ou la liaison dynamique, favorisent l'évolution des systèmes avec ré-utilisation de composants existants, elles ne répondent pas à tous les besoins, comme par exemple, la prise en compte de modifications apportées à des interfaces de fonctions ou de méthodes.

Dans le cas de systèmes logiciels fondés sur des bases de données, le schéma de la base et les sous-schémas associés jouent le rôle d'interfaces abstraites pour les programmes qui opèrent sur la base. Même si les SGBD actuels, et particulièrement les SGBD relationnels, assurent une indépendance entre les programmes et les données, ils ne supportent que partiellement des restructurations profondes des bases dont ils assurent la gestion et encore moins la répercussion des restructurations sur les programmes existants. Ces tâches sont généralement laissées à la charge de l'administrateur de la base et des programmeurs.

- Quels types de modifications peut-on opérer sur les schémas de données?

- quels types de support peut-on offrir pour gérer les conséquences des modifications à la fois sur les données et les programmes existants?

sont les deux questions dont traite cette partie. Nous distinguerons *deux catégories d'opérations* que l'on peut effectuer sur un schéma de données. *La première* permet d'opérer des modifications que nous qualifierons de locales, consistant essentiellement en la modification d'un des constituants d'un schéma, comme changer le type d'un attribut d'un objet, ajouter un type d'objet, etc. *La deuxième catégorie* d'opérations permet de produire un schéma incorporant les abstractions exprimées dans plusieurs autres : il s'agit de *l'intégration de schémas*. Dans chacun des cas, des critères de qualité du résultat obtenu doivent être satisfaits. En outre, les effets de bord, tant sur les instances de données que sur les programmes, doivent être maitrisés. En supposant que les conséquences d'une modification de schéma soient identifiées (souvenons-nous des propos tenus, dans le chapitre 1, sur l'état actuel des travaux sur

la révision des bases de données et des bases de connaissances et sur l'évolution des modèles de procédés), *diverses politiques*, non exclusives l'une de l'autre, peuvent être suivies. *La première* pourrait consister à propager immédiatement les effets induits par une modification : cela suppose l'arrêt du système et l'application des modifications les programmes et sur les objets. Mais, certains de ces derniers, comme les objets détenus par des transactions actives, peuvent être inaccessibles voire dans un état incohérent. De ce fait, la mise en œuvre de cette politique ne nécessite pas le maintien de représentations multiples des objets ou des programmes mais elle peut entraîner l'avortement de transactions. Cette politique semble approriée dans le cas de changements majeurs ou de correction d'erreurs graves.

Une autre approche consisterait à différer la matérialisation des effets de bord des modifications sur les objets et les programmes, jusqu'à ce que le système soit dans un état cohérent (fin des transactions actives ou suspendues, fin d'une étape du cycle de vie ou de l'activation de l'instance du fragment de modèle de procédé décrivant une suite d'activités, etc). Cette stratégie peut aussi nécessiter l'arrêt du système, comme elle peut requérir des mécanismes de gestion de *versions multiples* de schémas. En effet, dans des applications comme le développement de logiciels, les transactions peuvent être de longue durée : doit-on alors attendre leur terminaison avant de rendre effectives les modifications d'un schéma, ou peut-on utiliser l'ancienne version du schéma, pour les activités ou les transactions en cours, et la nouvelle version pour celles devant être initiées? Dans le premier cas, les modifications ne sont visibles qu'à la fin des transactions actives sans qu'une nouvelle transaction puisse être débutée : on parlera alors de *politique différée atomique*. On remarquera que cette politique revient à une politique immédiate sans avortement de transactions. Dans le second cas, les modifications sont visibles pour des objets venant à être créés ou pour des objets existants qui ne sont pas détenus par des transactions. Ceux détenus par des transactions pourront être adaptés au fur et à mesure de leur libération ou lors d'une prochaine utilisation. On parlera alors de *politique différée progressive ou "opportuniste"* (Nous reviendrons plus en détail sur la relation entre les évolutions de schémas et les transactions dans la discussion qui clôt cette partie).

La philosophie qui a guidé nos travaux dans ce domaine est de ne privilégier aucune des politiques qui viennent d'être citées, mais d'essayer d'identifier les mécanismes qui permettraient d'adopter indifféremment l'une ou l'autre parmi elles. En d'autres termes, nous avons voulu identifier et proposer un ensemble de moyens pouvant constituer une partie des services d'un sous-système d'intégration, laissant à l'utilisateur, dans l'état actuel de nos travaux, la charge de composer les services offerts pour concrétiser telle ou telle politique [3].

Deux traits caractérisent l'approche que nous avons adoptée pour conduire cette étude :

1. *typage* des schémas et de leurs composants,
2. et *modularité* de leur spécification en vue de favoriser à la fois leur définition incrémentale et la ré-utilisation de spécifications de composants à des degrés différents de granularité.

En effet, le modèle d'abstraction de données retenu (voir le chapitre 3) est un modèle fortement typé bâti sur

- *des types de base* prédéfinis atomiques (comme les types usuels entier, réel ou booléen) ou structurés (comme des ensembles, des produits cartésiens ou des listes).
- *des types structurés* construits
  - par agrégation de types de base ou de types construits,
  - par spécialisation ou généralisation de types construits.

3. On peut parfaitement imaginer que la politique à adopter soit exprimée comme un modèle de procédé d'intégration.

Chaque spécification de type est alors ré-utilisable dans la conception initiale ou lors de la modification d'un schéma. Cette vision amène en définitive à considérer l'élaboration d'un schéma comme une suite d'applications d'opérations de construction de types, résultant en un *nouveau type de schéma*. Celui-ci peut alors être utilisé pour décrire des bases, ou pour produire un nouveau schéma par modification ou par intégration de schémas existants. En d'autres termes, un type de schéma $\mathcal{T}$ détermine une classe de bases de données $\mathcal{B}_{i.T}$ de schéma $\mathcal{T}$ (figure 9).

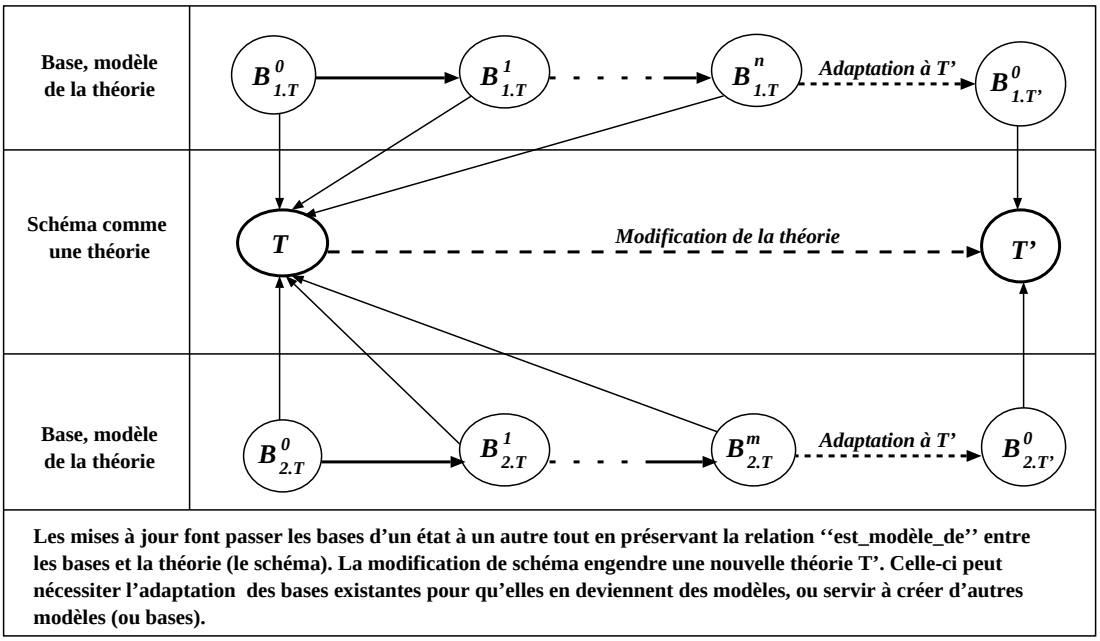

FIG. 9 – *Mise à jour de données et modification de schémas*

Si des modifications sont apportées à $\mathcal{T}$ pour produire un nouveau type de schéma $\mathcal{T}'$, elles n'ont pas d'incidences sur les bases $\mathcal{B}_{i.T}$ et le type de schéma résultant, $\mathcal{T}'$, a le même statut que $\mathcal{T}$, c'est-à-dire que nous ne considérons pas que $\mathcal{T}'$ est une version de $\mathcal{T}$. Par contre, lorsque des modifications sont opérées sur le schéma qui leur est associé (c'est-à-dire, sur une copie de $\mathcal{T}$ et non sur $\mathcal{T}$), elles peuvent affecter des bases existantes $\mathcal{B}_{i.T}$ d'une part, et d'autre part, le schéma résultant est considéré comme une version de celui dont il dérive.

Les effets des modifications du schéma sur les objets peuvent dans certains cas être des restructurations simples d'objets (c'est, par exemple, le cas où la modification consiste en l'ajout d'un type d'attribut à un type d'objet) ou des processus plus complexes et plus coûteux. C'est, notamment le cas quand une hiérarchie de spécialisation est modifiée ou quand un type d'attribut est "élevé au grade" de classe d'objets ou de type d'entité, entraînant ainsi de migrations d'instances entre différentes classes d'objets, processus coûteux et parfois non automatisable.

Par analogie avec la logique, un type de schéma est assimilé à une *théorie* $\mathcal{T}$ et une base de données $\mathcal{B}$, à un des *modèles* de cette théorie. Les mises à jour de $\mathcal{B}$ produisent un nouvelle interprétation qui doit toujours être un modèle de $\mathcal{T}$. La modification de schémas s'interprète alors comme la définition d'une nouvelle théorie $\mathcal{T}'$, c'est-à-dire comme un changement de la théorie. La prise en compte des incidences des modifications sur une base, consiste à produire un modèle de $\mathcal{T}'$ par modification d'un modèle de $\mathcal{T}$. Dans cette vision logique, l'intégration de deux schémas, spécifiés par les théories $\mathcal{T}_1$ et $\mathcal{T}_2$, peut être vue comme une opération résultant en

– une théorie $\mathcal{T}_3$ qui est "l'union" des théories $\mathcal{T}_1$ et $\mathcal{T}_2$,

– un modèle de $\mathcal{T}_3$ obtenu par "fusion" des bases $\mathcal{B}_1$ et $\mathcal{B}_2$, modèles de $\mathcal{T}_1$ et $\mathcal{T}_2$ respectivement.

L'approche purement logique de changement de théories nous semble plus prometteuse d'un point de vue fondamentale, mais également plus ardue. Aussi, avons-nous adopté une approche de type algébrique, c'est-à-dire fondée sur un modèle de représentation des structures des schémas et des bases, et des opérations agissant sur ces structures.

Dans l'état actuel de nos travaux, nos propositions comprennent, outre les opérations de modification et d'intégration de schémas,

1. un mécanisme d'instanciation des objets permettant de maintenir des versions d'instances d'objets se conformant à différentes versions de schémas,
2. des règles, de réécriture des invocations des opérations d'un type, utilisables pour tenter d'adapter les programmes existants aux modifications subies par le type,
3. et un ensemble d'opérateurs de restructuration d'objets.

Ce dernier faisant l'objet du chapitre 6, dans ce qui suit, nous allons successivement :

(i) présenter, de façon intuitive et formelle, le modèle d'abstraction de données retenu comme support de l'étude (chapitre 3),

(ii) passer en revue les deux catégories d'opérations permettant de faire évoluer des schémas proposées (chapitre 4)

(iii) exposer les concepts et les mécanismes proposés pour contribuer à la maitrise des conséquences des changements effectués sur les instances de données et les programmes existants. Dans ce cadre, nous reviendrons, dans le chapitre 5,

- sur les politiques et les stratégies de migration (immédiate, différée, avec ou sans versions de schémas et d'objets), où nous montrerons que les besoins sont indépendants de la politique et de la stratégie. Nous montrerons également que la réponse à ces besoins nécessite la définition de fonctions de correspondance entre le ou les schémas initiaux et le schéma final.
- Ces fonctions sont alors définies pour chaque opération de modification de schéma : une sémantique statique et des interprétations procédurales sont définies pour chacune d'elles.
- Enfin, nous montrons que les opérations usuelles dans l'intégration de schémas s'expriment simplement comme une composition des opérations de modification de schémas.

Nous terminerons par une analyse des limites et des améliorations possibles de nos propositions. Nous insisterons plus particulièrement sur leur mise en œuvre dans le cadre des systèmes dynamiques où nous nous situons.

3

GORM : un modèle d'abstraction de données

3.1 Caractéristiques du modèle

Le modèle de représentation que nous avons adopté mêle des concepts des modèles sémantiques [114, 164] et des concepts des modèles objets [13]. Il ne s'agit nullement de proposer un nouveau modèle mais essentiellement de fixer des concepts et des constructeurs, aujourd'hui communément admis dans la modélisation d'objets complexes, sur lesquels fonder notre étude.

GORM, ou Modèle Générique Objets-Relations, s'apparente au modèle entité-association étendu par des relations de sous-type et de composition, et par la possibilité d'attacher des opérations (que nous appellerons également *méthodes* ou définition du *comportement*) aux types des objets d'un schéma. GORM comprend

1) des types de base,

2) un constructeur de types d'entités,

3) un constructeur de types d'associations,

4) et les relations d'attributs, de composition et de sous-type.

Nous passons en revue ces concepts en nous appuyant sur une représentation diagrammatique et une spécification formelle. Les définitions abstraites des concepts seront représentées graphiquement sous forme d'arbres dont les nœuds désignent des types. Nous utiliserons alors les conventions suivantes :

– la valeur d'un nœud représentant un type de base est le nom du type de base,

– $*$, X et [] sont les valeurs des nœuds ensemble, produit cartésien et liste, respectivement.

Les labels associés aux arcs s'interprètent comme des fonctions du type du nœud source de l'arc dans celui du nœud cible.

Dans un second temps, nous donnerons une définition plus formelle des concepts de GORM qui nous permettra d'identifier "naturellement" les opération de modification que l'on peut effectuer sur un schéma. En effet, celui-ci est considéré comme un type complexe dont la définition est composée de définitions de types d'objets et de relations entre ces types. Chaque définition est elle-même considérée comme un type et pourra être modifiée par le biais des opérations de modification associées au type

support de la définition. Dans les définitions formelles, nous distinguerons la définition des types et celle de leurs instances. Nous fairons usage des notations suivantes :

- [ ] dénote une liste d'éléments,
- { } dénote un ensemble,
- et $< a_1 : T_1, \cdots, a_n : T_n >$ désigne un n-uplet où les a_i sont les noms des fonctions de projection et les T_i leur co-domaine. Par ailleurs, si t est un n-uplet, $t.a_i$ désignera la valeur de sa projection sur a_i.

En outre, les *espaces des noms* sont désignés par :

- *NomTypBase* pour les types de base,
- *NomTypAttribut* pour les noms des attributs,
- *NomDomaineAttribut* pour les noms des domaines de ces derniers,
- *NomTypEntité* pour les types d'entités,
- *NomRôleComposition* pour les rôles des types d'entité dans les compositions éventuelles de types d'entités,
- *NomTypAssociation* pour les types d'associations,
- *NomRôleAssociation* pour les rôles des types d'entité dans les types d'association,
- et *NomTypSchéma* pour les types de schémas.

Ces espaces sont disjoints deux à deux et chaque type est identifié par son nom dans son espace. Par ailleurs, les seuls "types d'objets" dont les instances ont une identité propre sont les types d'entités et les types d'associations. A chacun de ces types est implicitement associé un *espace d'identifiants* dénommé *OID(NomDuType)*[4].

3.2 Les types de base

3.2.1 Définition d'un type de base : *DefTypBase*

Une distinction est faite entre *des types de base atomiques prédéfinis* (entier, réel, booléen, chaîne de caractères) et *des types de base structurés* (ensemble, listes, produit cartésien). Les types de base sont définis

- par leur nom, lorsqu'il s'agit de types prédéfinis ou de types définis par renommage d'un type de base atomique prédéfini,
- ou par un nom et une expression caractérisant leur espace de valeurs (restriction d'un domaine, énumération de ses valeurs, etc). Par exemple, *Tranche_Age* $\stackrel{def}{=}$ *[18..25]* est une définition de type de nom *Tranche_Age* ayant comme type sous-jacent le type prédéfini *entier*.

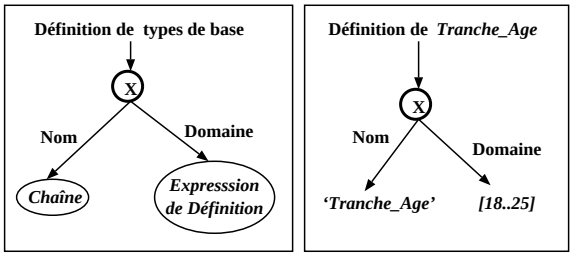

FIG. 3.1 – *Représentation graphique et exemple de définition d'un type de base*

Dans les illustrations graphiques, un type de base prédéfini est représenté par un nœud contenant le nom du type, l'arbre de la figure 3.1 montrant la schématisation des définitions des types de base non prédéfinis.

$$DefTypBase \stackrel{def}{=\!=} < \text{Nom: } NomTypBase, \\ \text{Domaine: } ExpressionDeDéfinition >$$

NomTypBase, l'espace des noms des types de base est constitué des noms des types prédéfinis et de ceux des types redéfinis. Les noms de cet espace sont utilisés pour typer des attributs.

3.2.2 Instance d'un type de base

Les valeurs des types de base sont des "chaînes" satisfaisant l'expression de définition. Elles n'ont pas le statut d'objet, au sens du paradigme objet, c'est-à-dire qu'il ne leur est pas associé d'identifiant. En outre, le comportement de ces types est

- soit celui usuel des types prédéfinis,
- soit hérité du type sous-jacent, lorsqu'il s'agit d'un type obtenu par renommage ou par restriction du domaine d'un type prédéfini.

3.3 La relation d'attribut

3.3.1 Définition d'un type d'attribut : *DefTypAttribut*

Intuitivement, un attribut est un couple <Nom, Domaine> servant à décrire une propriété d'un type d'entité ou d'un type d'association. Formellement, un attribut se définit comme une fonction du type de l'objet auquel il est attaché, dans un domaine de valeurs. Dans GORM, ce domaine peut être un type de base ou un type d'entité, et l'image de la fonction attribut peut être un élément du domaine de l'attribut (*attribut à valeur unique ou mono-valué*) ou un sous-ensemble du domaine (*attribut à valeurs multiples ou multi-valué*).

Sur l'exemple de la figure 3.3, *Nom* est un attribut mono-valué (de profil $EQUIPE \longrightarrow Chaîne$) et *Réunions* est multi-valué (profil : $EQUIPE \longrightarrow Date^n$).

4. Formellement, cela signifie que nous nous permettons de confondre un nom de type d'entité ou d'association et une "variable" du type

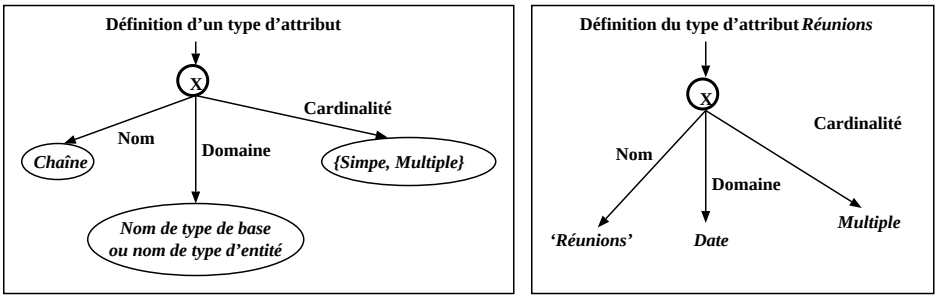

FIG. 3.2 – *Représentation graphique et exemple de définition d'un type d'attribut*

DefTypAttribut $\stackrel{def}{=}$ < Nom: *NomTypAttribut*,
Domaine: *NomDomaineAttribut*
Cardinalité: {*'Simple', 'Multiple'*}>

avec *NomDomaineAttribut* = *NomTypBase* ∪ *NomTypEntité*

3.3.2 Instance d'un type d'attribut : *InstanceAttribut*

Si le domaine d'un type d'attribut est un type de base, ses valeurs sont naturellement prises dans le domaine du type. Par contre, les valeurs d'un attribut dont le domaine est un type d'entité sont prises dans l'espace des identifiants des entités du type considéré. Ainsi, si *A* est une définition d'un type d'attribut, une instance de *A* est définie par :
Instance(A) $\stackrel{def}{=}$ $< A.Nom : A.Valeur >$, avec

A.valeur $\stackrel{def}{=}$ Si *A.Domaine* ∈ *NomTypBase*
Alors Si *A.Cardinalité* = 'Simple'
Alors "une chaîne" du domaine
Sinon {Chaînes du domaine}
Finsi
Sinon /* Le domaine est un type d'entité */
Si *A.Cardinalité* = 'Simple'
Alors un identifiant i, i ∈ *OID(A.Domaine)*
Sinon {i | i ∈ *OID(A.Domaine)*}
Finsi
Finsi

3.4 La relation de composition

La composition permet de décrire la structure de certains types d'objets complexes. Elle permet de définir des imbrications de types d'objets. Elle se distingue des attributs multivaluées, dont le domaine est un type d'entité, par le fait qu'elle peut comporter une liste de types d'objets. En un sens, elle peut s'interpréter comme une agrégation d'attributs à valeurs dans des types d'entités et à laquelle on attache une sémantique particulière (*Partie_De, Is_Part_Of*) .

3.4.1 Définition d'une composition : *DefComposition*

La partie gauche de la figure 3.3 montre la modélisation d'une *équipe* de développement comme une composition d'un *analyste* et de plusieurs *programmeurs* et la partie droite, comme deux attributs du type d'entité *équipe* : *Lanalyste* et *LesProgrammeurs*, mono-valué (représenté par un arc à une seule "tête") et multi-valué (représenté par un arc à "double tête"), respectivement.

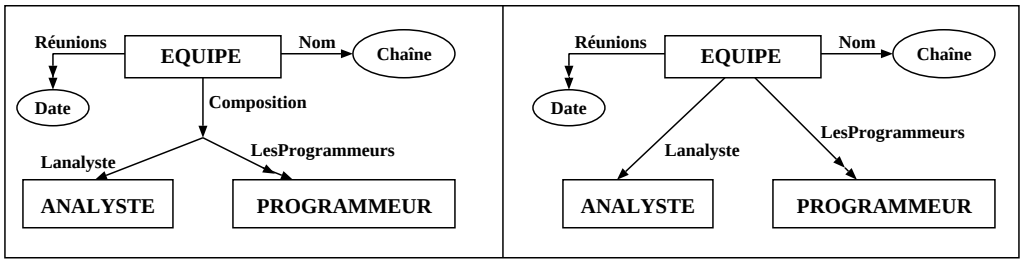

FIG. 3.3 – *Lien de composition vs lien d'attribut*

Une composition est définie par la donnée des types d'entités qui la constituent et que nous appellerons *types des composants*. Chacun d'eux est caractérisé par un nom (appelé *rôle* du composant) et une *cardinalité*, entier strictement positif, indiquant le nombre maximum d'instances de chaque composant dans une instance de la composition (Dans ce qui suit, la cardinalité sera dénotée par *"Simple"* (représentée par un arc à une seule "tête") ou *"Multiple"* (représenté par un arc à "double tête") pour indiquer respectivement une participation unique ou pas). En outre, chaque type de composant peut être caractérisé par :

- sa *dépendance* vis-à-vis du type d'entité *composé* : celle-ci est dite *forte* si une instance du composant ne peut pas persister après la suppression du composé, ou *faible*, dans le cas contraire (la composition de nature *faible* rejoint la notion d'entité faible de Chen [68]).

- sa nature, *partagée* ou *exclusive*, signifiant qu'une instance peut ou ne peut pas apparaître dans des instances différentes du type d'entité composé.

La définition d'une composition, comme schématisée par la figure 3.4, est spécifiée par un ensemble de produits cartésiens, représenté par une liste, où chaque produit cartésien décrit un des composants. D'où la définition formelle :

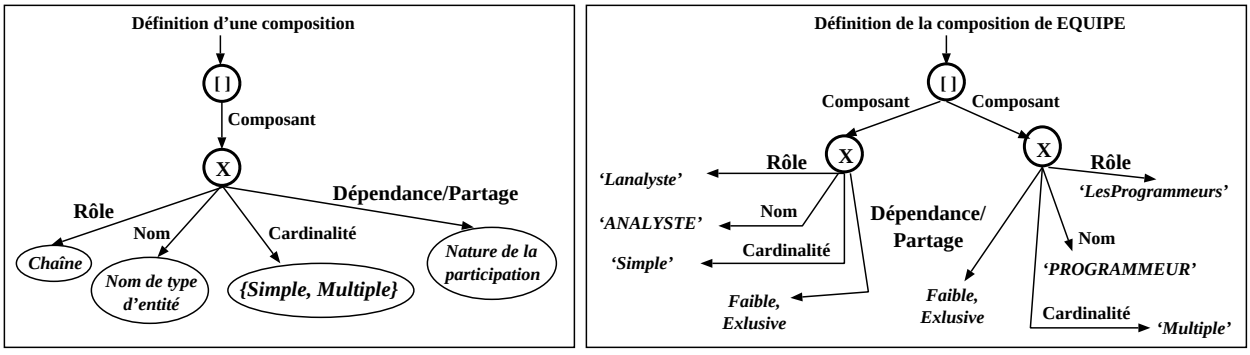

FIG. 3.4 – *Représentation graphique et exemple de définition d'une composition*

DefComposition $\stackrel{def}{=\!=}$ [<Rôle: Chaîne,
NomComposant: *NomTypEntité*
Cardinalité: {*'Simple', 'Multiple'*}
Dépendance-Partage: {*Forte, Faible*} x {*Partagé, Exclusive*}>]

3.4.2 Instance d'une composition : *InstanceComposition*

Une instance d'une composition est alors déterminée par les identifiants des instances de ses composants. Par exemple,

[*Lanalyste:* a_1, *LesProgrammeurs:* $\{p_{14},\ p_{21}\}$]

est une instance de la composition d'une *équipe* E, où a_1 est l'identifiant de l'analyste et $p_{14},\ p_{21}$ sont les identifiants des programmeurs de l'équipe.

De façon plus générale, si *C* est une définition de composition, une instance de *C* est définie par :

InstanceComposition $\stackrel{def}{=\!=}$ [<Rôle: Valeur>]

où la valeur V_i d'un élément i de la liste est un identifiant ou un ensemble d'identifiants de l'espace des identifiants du type d'entité domaine du composant i de la liste de définition de la composition, soit plus formellement,

$V_i \stackrel{def}{=\!=}$ Si *C*.Cardinalité$_i$ = 'Simple'
Alors un identifiant i, i $\in$ *OID(C*.NomComposant$_i$*)*
Sinon {i | i $\in$ *OID(C*.NomComposant$_i$*)*
Finsi

Une composition n'a pas d'identifiant propre : au sein d'une entité, elle se comporte comme un attribut à valeurs dans un produit cartésien d'identifiants d'entités.

On notera que les attributs multi-valués, ceux à valeurs dans des types d'entités et les compositions sont des constructions qui permettent de décrire des types d'objets complexes, c'est-à-dire dont la structure ne comporte pas que des types atomiques. On remarquera, en outre, que la notion de composition se distingue du *constructeur de groupement* [114] ou de "convoi" [110] par le fait que ses composants peuvent être de n'importe quel type d'entité et, particulièrement, ils peuvent ne pas être des sous-types d'un même type (hormis la racine de l'arbre représentant la relation de sous-type).

3.5 La relation de sous-type

3.5.1 Définition d'une relation de sous-type : *DefSousType*

La relation de sous-type permet de classifier des ensembles d'entités selon diverses perspectives. Elle relie deux types d'entités : un sous-type et un super-type. Il lui est associé une *sémantique d'inclusion* de l'ensemble des instances du sous-type dans celui des instances du super-type. Les "propriétés" (attributs, composition, associations) et le comportement du super-type sont héritées par chacun de ses sous-types. Dans notre modèle, nous considérons qu'un type d'entité a au plus un super-type, nous limitant ainsi à un *héritage simple*. La relation de sous-type définit un *ordre partiel* sur les types d'entités et elle sera matérialisée par un arbre enraciné en un nœud "constante" *Entité* (voir l'exemple de la figure 3.5). Une relation de sous-type est définie par un ensemble de couples de noms de types

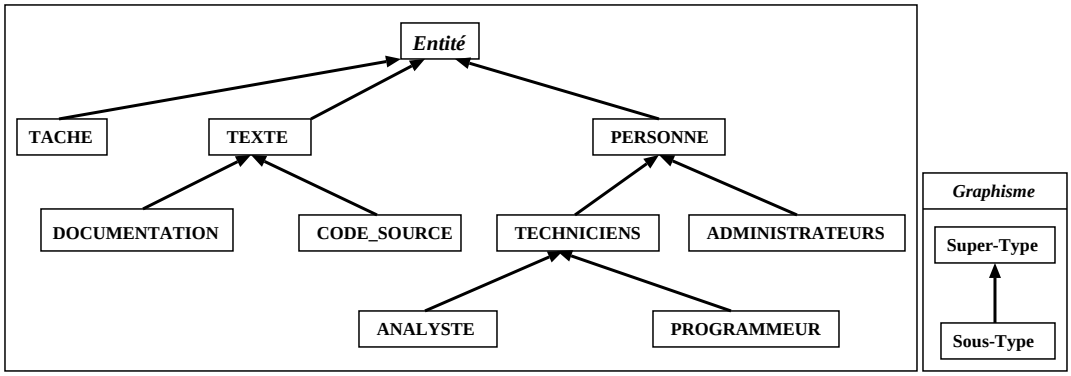

FIG. 3.5 – *Schématisation d'un arbre d'héritage*

d'entités. Soit, plus formellement :

$$\textit{DefSousType} \stackrel{def}{==} \{< \text{Sous-Type} : \textit{NomTypEntité}, \\ \text{Super-Type} : \textit{NomTypEntité} \cup \text{`Entité'} >\}$$

3.5.2 Instance d'une relation de sous-type : *InstanceSousType*

La fermeture transitive de la relation peut être matérialisée par un graphe. Nous considérons alors qu'une instance de la relation est un ensemble d'ensembles de couples d'identifiants d'instances d'entités, chaque ensemble étant l'instance de la relation entre deux types d'entités. C'est-à-dire, pour chaque couple <Sous-Type : E_1, Super-Type : E_2 > d'une définition *DefSousType*, on dispose d'une instance définie par :

$$InstanceSousType_{E_1,E_2} \stackrel{def}{==} \{< \text{Sous-Type} : \textit{OID}(E_1), \\ \text{Super-Type} : \textit{OID}(E_2) >\}$$

3.6 Les types d'entité

3.6.1 Définition d'un type d'entité : *DefTypEntité*

Un type d'entité est défini par la donnée :

- de son nom,
- de la liste des attributs associés. Ces attributs prennent leurs valeurs soit dans des types de base, soit dans des types d'entités.
- éventuellement, de la définition d'une relation de composition,
- et du nom du super-type d'entité immédiat (ou la "constante" *Entité* si le type de l'entité n'a pas de super-type).

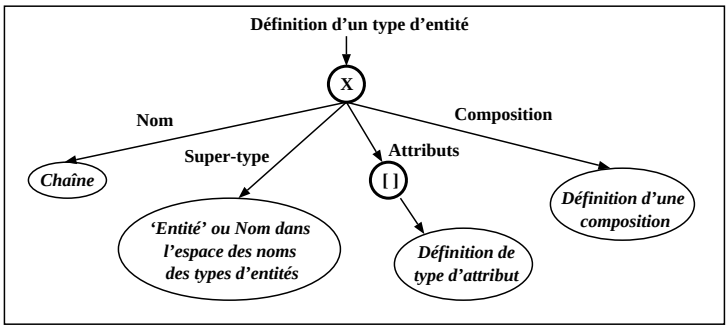

FIG. 3.6 – *Représentation graphique des définitions de types d'entité*

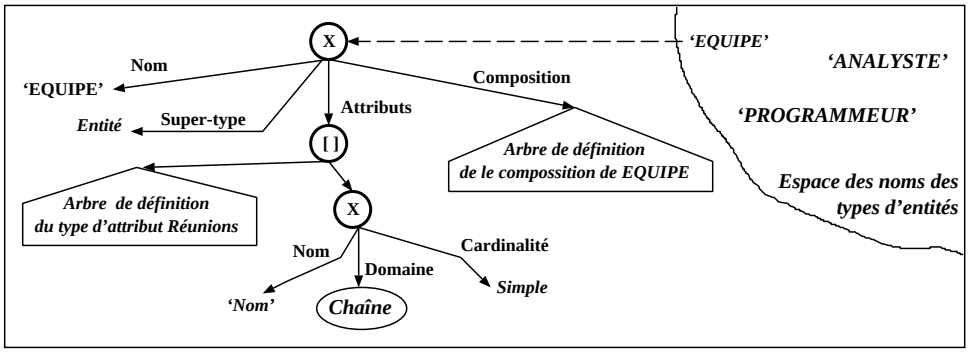

FIG. 3.7 – *Représentation graphique de la définition du type d'entité EQUIPE*

L'espace des noms des types d'entités comprend les noms des types des entités définis. La figure 3.6 montre la représentation graphique de la définition d'un type d'entité et la figure 3.7 montre celle de la définition du type d'entité *EQUIPE*. Formellement, nous distinguerons les types des entités munis d'une relation de composition (*DefTypEntitéComposé*) de ceux qui ne le sont pas (*DefTypEntitéNonComposé*).

$$\begin{aligned} \textit{DefTypEntitéComposé} \stackrel{def}{=\!=} < & \text{Nom} : \textit{NomTypEntité}, \\ & \text{NomSuperType} : \textit{NomTypEntité} \cup \textit{'Entité'}, \\ & \text{LesAttributs} : [\textit{DefTypAttribut}], \\ & \text{Composition} : \textit{DefComposition} > \end{aligned}$$

DefTypEntitéNonComposé, une définition de type d'entité non composé, n'est pas dotée de l'attribut *Composition* et *DefTypEntité*, une définition d'un type d'entité est alors une *DefTypEntitéComposé* ou une *DefTypEntitéNonComposé*.

3.6.2 Instance d'un type d'entité : *InstanceEntité*

Les instances d'un type d'entité, que nous appellerons *entités*, sont munies d'un identifiant et sont définies par un produit cartésien comprenant :

– l'identifiant de l'entité,

– celui de l'instance de son super-type,

– les valeurs des attributs associés, à valeurs dans les types de base,

– les identifiants des éventuelles entités, valeurs des attributs à valeurs dans des types d'entités,

– et, éventuellement, une instance d'une relation de composition (voir l'exemple de la figure 3.8).

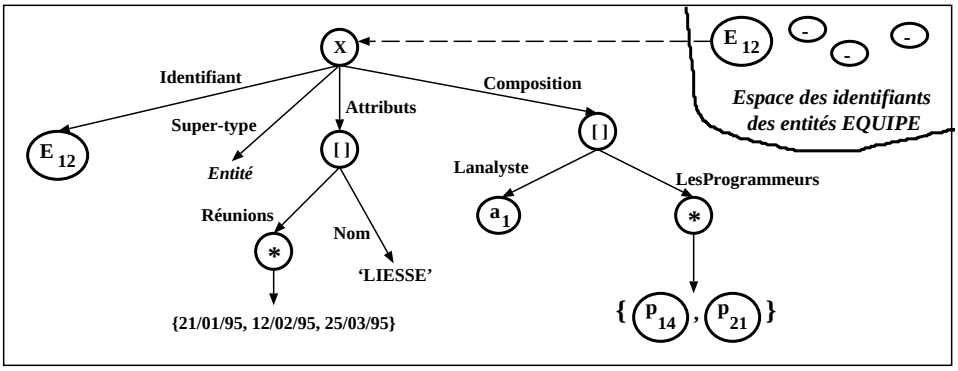

FIG. 3.8 – *Représentation graphique d'une instance d'un type d'entité*

Toutes les instances d'un type d'entité sont *implicitement* contenues dans un ensemble se comportant comme un ensemble de produits cartésiens. Les identifiants des entités d'un type donné sont regroupés dans l'espace des identifiants des instances du type d'entité.

Nous ne donnons ci-dessous que la définition d'une instance d'une entité composée, celle-ci se distinguant d'une instance d'une entité non composée par l'absence d'instance de l'attribut *Composition*. Soit *TE* une définition de type d'entité et TE' celle de son éventuel super-type dans un schéma S, une *entité* de type *TE* est alors définie par :

InstanceEntitéComposée $\stackrel{def}{=}$ <Identifiant : *OID(TE.Nom)*,
IdentifiantSuperType : *OID(*TE'*.Nom)* $\cup$ *'Entité'*,
[*InstanceAttribut*],
InstanceComposition>,

le couple $st \stackrel{def}{=} < Sous-Type : OID(TE.Nom), Super-Type : OID(TE'.Nom) >$ appartenant à l'instance de la relation de sous-type, i.e.

$$< Sous-Type : TE.Nom, Super-Type : TE'.Nom > \in DefSousType_S$$

et $st \in InstanceSousType_{TE.Nom,TE'.Nom}$.

3.7 Les types d'association

3.7.1 Définition d'un type d'association : *DefTypAssociation*

Un type d'association est une agrégation de types d'entités et d'une liste, éventuellement vide, de types d'attributs. Il est défini par (figure 3.9)

– le nom du type d'association,

– les types des entités associés, appelés *participants*; dans notre modèle, nous nous limitons à des types d'associations binaires.

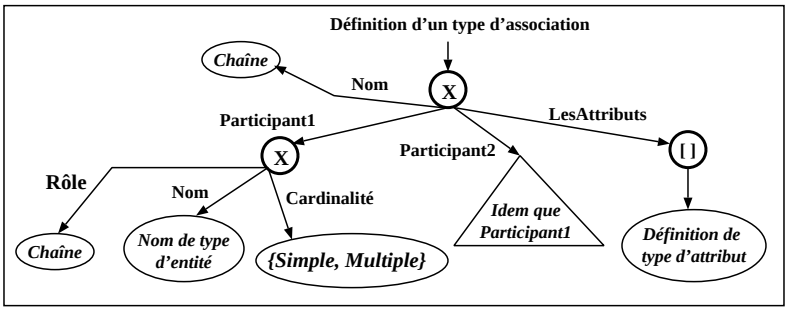

FIG. 3.9 – *Représentation graphique des définitions des types d'association*

– et, éventuellement, les types des attributs du type d'association.

Les noms des types d'associations sont également regroupés dans un espace des noms, appelé *NomTypAssociation*.

Classiquement, un *rôle* est attribué à chaque type d'entité participant à un type d'association. En outre, une cardinalité est associée à chaque type d'entité dans un type d'association. Nous interpréterons la définition d'un type d'association comme la définition d'une relation, au sens mathématique, entre les types des entités associées et les types d'attributs éventuels.

$$
\begin{aligned}
\textit{DefTypAssociation} \stackrel{def}{=\!=} <\ & \text{Nom} : \textit{NomTypAssociation}, \\
& \text{Participant}_1 : <\ \text{Rôle}_1 : \textit{NomRôleAssociation}, \\
& \qquad\qquad \text{NomParticipant}_1 : \textit{NomTypEntité}, \\
& \qquad\qquad \text{Cardinalité}_1 : \{\text{`}\textit{Simple}\text{'}, \text{`}\textit{Multiple}\text{'}\}>, \\
& \text{Participant}_2 : <\ \text{Rôle}_2 : \textit{NomRôleAssociation}, \\
& \qquad\qquad \text{NomParticipant}_2 : \textit{NomTypEntité}, \\
& \qquad\qquad \text{Cardinalité}_2 : \{\text{`}\textit{Simple}\text{'}, \text{`}\textit{Multiple}\text{'}\}>, \\
& \text{Attributs} : [\textit{DefTypAttribut}]>
\end{aligned}
$$

3.7.2 Instance d'un type d'association : *InstanceAssociation*

Une instance d'un type d'association a une identité propre et est définie par la donnée

– de l'identifiant de l'instance,

– des identifiants des entités associées,

– des valeurs des attributs éventuels, ces valeurs pouvant être des éléments des domaines des attributs ou des éléments de l'espace de identifiants des entités.

$$
\begin{aligned}
\textit{InstanceAssociation} \stackrel{def}{=\!=} <& \text{Identifiant} : \textit{OID(TA.Nom)}, \\
& \text{Rôle}_1 : r_1, \\
& \text{Rôle}_2 : r_2, \\
& [\textit{InstanceAttribut}]>
\end{aligned}
$$

$r_k \in OID(TA.Participant_k.NomParticipant_k)$ (k ∈ {1, 2}), est un identifiant de l'espace des identifiants des entités du type du participant à l'association. .

3.8 Le type schéma

3.8.1 Définition d'un type de schéma : *DefTypSchéma*

La spécification d'un *type de schéma* (figure 3.10) est constituée par

– un ensemble non vide de définitions de types d'entités,

– un ensemble de définitions de types d'associations,

– les définitions des types de base, domaines des attributs caractérisant les types d'entités ou d'associations,

– et la description du graphe de la relation de sous-type, exprimée sous la forme d'un ensemble de couples de noms de types d'entités, désignant le sous-type et le super-type, respectivement.

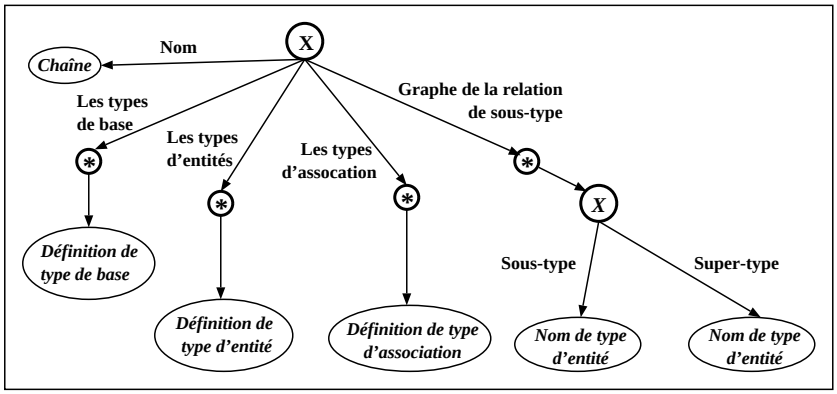

FIG. 3.10 – *Représentation graphique des définitions des types de schéma*

$$\begin{aligned} \textit{DefTypSchéma} \stackrel{def}{=\!=} < & \text{Nom} : \textit{NomTypSchéma}, \\ & \{\textit{DefTypBase}\}, \\ & \{\textit{DefTypEntité}\}, \\ & \{\textit{DefTypAssociation}\}, \\ & \textit{DefSousType} > \end{aligned}$$

Une telle spécification est *identifiée* par un nom unique dans un espace des noms des types de schémas et elle peut être utilisée pour décrire une ou plusieurs bases.

3.8.2 Instance d'un type de schéma : *InstanceSchéma*

Nous appellerons *instance d'un type de schéma*, ou simplement *schéma* une utilisation d'une spécification d'un type de schéma. Si *TS* est une définition d'un type de schéma, alors une de ses instances est décrite par :

$$\textit{InstanceSchéma} \stackrel{def}{=\!=} < \text{Nom} : \textit{TS.Nom} >$$

3.8.3 Instance d'une base de données

Dans le cadre de la formalisation qui vient d'être proposée, une base de données est considérée comme une *réalisation* d'un schéma et elle est définie par :

- le nom d'une instance d'un type de schéma,
- le nom de la base,
- une instance par type d'entité définie dans le type de schéma,
- une instance par type d'association définie dans le type de schéma,
- et une instance d'une relation de sous-type.

Soit plus formellement,

$$\begin{aligned} \mathit{Base} \stackrel{def}{=\!=} <&\text{Nom} : \mathit{NomBase},\\ &\mathit{InstanceSchéma},\\ &\text{Entités} : \{\mathit{InstanceEntité}\},\\ &\text{Associations} : \{\mathit{InstanceAssociation}\}\\ &\text{Sous-Types} : \{\mathit{InstanceSousType}\} > \end{aligned}$$

3.9 Types et opérations

Dans l'exposé des concepts de GORM, nous avons fait allusion au comportement de chacun des types de données adoptés pour les représenter. Nous avons également précisé que certaines opérations étaient héritées du type sous-jacent à la représentation ou du super-type (dans le cas de certains types d'entités). En outre, de nouvelles opérations peuvent être spécifiées pour chacun des types présentés dans le paragraphe précédent, chaque opération étant décrite par

- son nom,
- une composante *syntaxique* précisant le domaine et le co-domaine sous la forme de deux produits cartésiens de noms de types de base ou d'entités,
- et une composante *sémantique* spécifiée par deux expressions logiques décrivant, respectivement, la pré-condition et la post-condition de l'opération.

(Les représentations graphiques des concepts du modèle sont implicitement étendues par la définition de la structure de la spécification d'un ensemble d'opérations, telle que schématisée sur la figure 3.11).

3.10 Critères de qualité d'un type de schéma

Une spécification d'un type de schéma doit satisfaire certaines propriétés invariantes. En outre, les opérations de modification d'une spécification doivent produire un résultat satisfaisant ces invariants. De ce fait, les conditions d'application et les effets de toute opération de modification d'un type de schéma sont déterminées, d'une part, par des contraintes inhérentes au modèle GORM et d'autre part, par des propriétés que doit satisfaire une spécification de schéma.

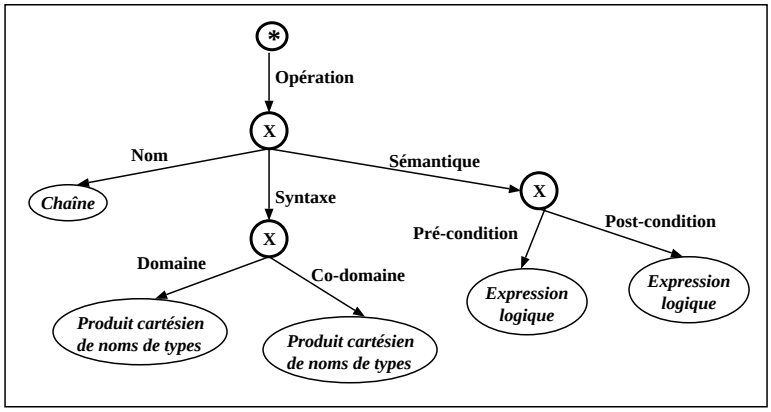

FIG. 3.11 – *Schématisation de spécifications d'opérations*

Nous distinguons des critères "locaux", ou critères auxquels doivent se soumettre les types des nœuds d'une spécification (c'est-à-dire, chaque spécification de type faisant partie de la spécification d'un type de schéma), et des critères "globaux" que doit satisfaire le type de schéma dans son entité.

Les critères locaux dérivent naturellement des définitions des concepts du modèle de représentation adopté. Par exemple, le domaine de *Cardinalité* est {Simple, Multiple}, celui d'un attribut est un nom de type de base ou un nom de type d'entité défini dans le type de schéma, etc. Nous n'en ferons pas l'énumération ici.

Quant aux *critères globaux*, nous en avons fixé trois :

(i) *Unicité des noms* : les espaces des noms des constituants d'une spécification (attributs, types d'entité et d'associations, rôles, etc) sont disjoints.

(ii) *Acyclicité de la relation de sous-type* : le graphe représentant la fermeture transitive de la relation de sous-type est un arbre : chaque nœud a un seul père et il n'existe pas d'arcs réflexifs,

(ii) *Connexité de la relation de sous-type* : l'arbre de la relation de sous-type est enraciné au nœud prédéfini *Entité*.

Les critères dits locaux et le critère d'unicité des noms sont utilisés comme pré-conditions des opérations de modification de schémas, les deux autres étant utilisés comme des post-conditions : c'est l'implémentation qui en a été faite dans [45, 44] où une opération de modification n'est exécutée que si sa pré-condition est vraie et les effets de l'opération ne sont validés que si le résultat satisfait la post-condition et les deux derniers critères de qualité.

3.11 Conclusion

Il convient de souligner que le modèle que nous venons de présenter ne se veut pas être un nouveau modèle de représentation de données. Il a pour humble objectif de nous servir de support à l'étude des opérations de modification et d'intégration de schémas, indépendamment de tout système existant. Nous ne manquerons pas de constater, dans la suite de ce document, que l'effort de formalisation qui en a été proposée (côté par lequel pêchent la plupart des modèles du même type, à l'exception de certains d'entre eux, dont [2]), facilitera l'expression des effets de bord des opérations de modification ou d'intégration de schémas qui est l'objet du prochain chapitre.

4

Opérations de modification et de restructuration de schémas

4.1 La modification de schémas

Au vu des concepts qui viennent d'être exposés, la modification d'un type de schéma ou d'une des ses instances consistera en la modification d'un des constituants de sa spécification. En d'autres termes, les spécifications des types contenues dans celle d'un schéma sont considérées comme des objets à part entière.

La diversité des opérations que l'on pourra effectuer, peut aisément être imaginée à partir de la représentation du type schéma, telle qu'elle est schématisée sur la figure 3.10, et telle qu'elle vient d'être formalisée : chaque "nœud" libellé par un constructeur de type (ensemble, liste, produit cartésien) est susceptible de se voir appliquer une des opérations du constructeur. Cette vision se généralise "récursivement" aux éléments composants une spécification d'un type de schéma, à savoir les types d'entités (figures 3.6 et 3.4), les types d'associations (figure 3.9) et les opérations associées à chacun de ces types (figure 3.11). Par ailleurs, chaque valeur attachée à une feuille est sujette à modification.

Cette approche, qui s'apparente à celles adoptées notamment dans [20, 183], permet d'identifier une liste suffisamment complète d'opérations dont certaines, comme celles consistant en un renommage d'un des "objets" de la spécification d'un type de schéma, ont des conséquences mineures et maîtrisables sur les données et les programmes existants, alors que d'autres, comme la modification de la hiérarchie de sous-types, peuvent conduire à de profondes restructurations.

Le paragraphe 4.1.1 donne une vision synthétique des opérations de modification de la spécification d'un type de schéma. Le paragraphe 4.1.2 contient l'énumération des opérations proposées. Une discussion informelle de l'impact de chacune d'elles sur les données et sur les programmes est donnée dans l'annexe 2.

4.1.1 Typologie des opérations de modification

Les opérations modifiant une spécification d'un type de schéma se rangent en trois catégories : changement de noms de types par renommage, changement de domaines et changement de structure.

4.1.1.1 Les opérations de renommage

Cette catégorie d'opérations concerne tout type nommé dans la spécification d'un schéma, à savoir :

1. le type du *schéma*,

2. les *types d'entités*, et pour chaque type d'entité,
   - les noms des *types d'attributs* du type d'entité,
   - les *rôles* des types des entités impliqués dans la définition de la *composition* du type d'entité,
3. les *types d'associations* et pour chaque type d'association,
   - les noms des types d'attributs du type d'association,
   - les *rôles* des types des entités impliqués dans la définition du type d'association.
4. et les noms des *types de base* non prédéfinis.

Les opérations de cette catégorie n'ont pas d'effet sur les instances des objets existants, mais elles en ont sur la désignation des objets dans les programmes et dans les opérations ou méthodes du type sur lequel la modification est effectuée. En effet, toute référence à un objet du type dont le nom a changé devient invalide : un simple renommage du type dans ces références peut alors suffire à résoudre ce problème.

4.1.1.2 Les opérations de changement de domaines

Cette catégorie d'opérations concerne :

1. *les types de base* dont l'expression de définition peut être modifiée,
2. le domained'un *type d'attribut* qui était :
   - un type de base, peut devenir un autre type de base ou un type d'entité,
   - un type d'entité, peut devenir un autre type d'entité ou un type de base.
     Le changement de domaine, qui consiste à affecter un nouveau type d'entité en remplacement d'un type d'entité, s'applique également
     (a) aux types d'entités participant à un type d'association,
     (b) à ceux entrant dans la définition d'une composition
     (c) et à ceux liés par une relation de sous-type.

L'impact des opérations de cette catégorie dépend de la relation qui peut exister entre l'ancien et le nouveau domaine, que nous dénoterons $\overline{Dom}$ et *Dom*, respectivement.

1) Le type de *Dom est sous-type* de celui de $\overline{Dom}$:
Ayant adopté l'inclusion des domaines comme sémantique de la relation de sous-type, *formellement*, la migration des valeurs appartenant à l'intersection des domaines peut être faite.

Du point de vue opérationnel, vu qu'il n'existe pas de valeurs dans *Dom*, l'intersection de *Dom* et de $\overline{Dom}$ est constituée par les valeurs de $\overline{Dom}$ "du type" de *Dom* : cela signifie qu'il ne faut faire migrer que les valeurs de $\overline{Dom}$ qui *satisfont* le type de *Dom* : pour les types de base, un mécanisme de *"type checking"* est nécessaire et, dans le cas de domaines de types entités, la migration pourra être effectuée en exploitant l'instance de la relation de sous-type.

2) Le type de $\overline{Dom}$ *est sous-type* de celui de *Dom* :
Dans ce cas, *formellement*, toutes les valeurs de $\overline{Dom}$ migrent dans *Dom*, toujours du fait de la sémantique de l'inclusion de la relation de sous-type.

Néanmoins, du *point de vue opérationnel*, lorsque le type de *Dom* est un type d'entité, l'espace des identifiants de ses instances étant différent de celui de $\overline{Dom}$, la migration est possible lorsque *Dom* et $\overline{Dom}$ appartiennent à la "même branche" de l'arbre d'héritage : elle nécessite la mise en œuvre d'un algorithme exploitant le graphe de la fermeture transitive de la relation de sous-type (cet algorithme est décrit dans le paragraphe 5.3) du chapitre 5.

3) Il n'existe *pas de relation formelle* entre *Dom* et $\overline{Dom}$:
C'est le cas, par exemple, où le type de $\overline{Dom}$ est un type de base et celui de *Dom* est un type d'entité. Ce cas est généralement indécidable et il nécessite la mise en œuvre de mécanismes ad hoc, tels ceux que nous préconisons dans le chapitre 6.

Entrent aussi dans cette catégorie d'opérations, les opérations qui changent la cardinalité d'un type d'attribut, d'un type d'entité dans un type d'association ou dans une composition de type d'entité (passage d'une cardinalité *'Simple'* à une cardinalité *'Multiple'* ou inversement). Rappelons qu'un attribut s'interprète comme une fonction à valeur dans le type du domaine de l'attribut. *Formellement*, si T est le type d'entité ou d'association auquel est rattaché un attribut $< a : t >$, a de cardinalité simple s'interprète comme une fonction de profil :

$$a : T \longrightarrow t$$

(La même interprétation s'applique aux rôles des participants à un type d'association et aux rôles des composants d'un type d'entité).

Le passage d'une cardinalité dite *simple* à une cardinalité multiple, signifie que le profil de la "fonction attribut" a est changé en :

$$a : T \longrightarrow t^n$$

Nous constatons que, formellement, les effets des modifications de cardinalités ne peuvent pas être traités correctement, le nouveau et l'ancien domaine de l'attribut étant des types totalement différents. Néanmoins, du point vue opérationnel, dans le cas du passage d'une cardinalité simple à une cardinalité multiple, on peut raisonnablement proposer de restructurer la représentation de l'attribut en une structure ensembliste et initialiser celle-ci par la valeur de l'attribut existante.

Enfin, la modification de la nature de la participation d'un type d'entité dans une composition est également classée dans la catégorie des opérations de changement de domaines. Une participation d'un type d'entité TC, composant d'un type d'entité TE, dont la nature *'Partagé'* devient *'Exclusive'* signifie, formellement, que toute *entité* de type TC ne peut, désormais, être l'image que d'une seule fonction rôle parmi toutes celles définies dans les compositions des types d'entités du schéma. Opérationnellement, comment identifier l'entité qui désormais détient l'exclusivité ? En effet, si

- tc est une instance du composant TC concerné par la modification,
- que $te_1, \cdots, t_n$ sont des entités *partageant* tc par composition
- et que, désormais, tc doit être le composant exclusif de te_j, ($j \in [1..n]$)

alors tc doit être retiré des compositions de toutes les entités autres que te_j.

Tous les autres cas de modification de la nature des participations (de *'Exclusive'* à *'Partagé'*, de *'Faible'* à *'Forte'* ou de *'Forte'* à *'Faible'*) n'ont aucune incidence sur les instances d'objets, les mécanismes supports du modèle de données étant supposés "connaître" le comportement de ces natures de compositions.

4.1.1.3 Les opérations structurelles

Sont regroupées dans cette catégorie :

1. les opérations qui modifient la structure du domaine d'un type de base (comme le passage d'un type atomique à un type produit cartésien, ou inversement),
2. celles d'ajout et de suppression de spécifications de types d'entités et de types d'associations,
3. les opérations d'extension ou de restriction de la liste des types d'attributs d'un type d'entité ou d'un type d'association,
4. et celles d'ajout ou de suppression de composants dans la définition de la composition d'un type d'entité

Nous étendrons cette catégorie d'opérations par celles qui sont nécessaires dans un processus d'intégration de schéma et dont nous parlerons dans le paragraphe 4.2.

De façon générale, le gestion des impacts des opérations de la catégorie structurelle requiert des mécanismes ad hoc implémentant des solutions "intuitivement correctes". Nous montrons, de façon informelle, dans l'annexe2 et, de façon plus formelle dans le paragraphe 5.3 du chapitre 5, les types de solutions que l'on peut y apporter.

4.1.2 Les opérations de modification de schémas : récapitulatif

L'ensemble des opérations disponibles pour opérer des modifications "locales" sur un type de schéma est énuméré dans les tableaux des figures 4.1 et 4.2 où elles sont classées par type de concept du modèle GORM sur lequel elles sont définies.

4.1.3 Conclusion

La présentation qui vient d'être faite de la diversité des opérations de modification auxquelles peut être soumis un schéma et l'analyse des effets de bord induits par certaines d'entre elles, donnent un aperçu sur la complexité de la maîtrise des propagations de ces effets de bord. En outre, cette analyse a été faite en supposant que l'on ne disposait pas de mécanismes de gestion de versions de schémas et de données. Or ces derniers sont nécessaires dès que l'on veut adopter une politique de migration différée des données et des programmes. Nous exposerons nos propositions dans ce domaine dans le chapitre 5.

Dans l'état actuel de la réalisation, les opérations ont été implantées sans propagation d'effets de bord, mais avec préservation des critères de qualité d'un schéma. Celui-ci est considéré comme un multi-graphe où chaque opération de modification concerne un de ses nœuds, ceux-ci pouvant eux-mêmes être des graphes (voir [44] pour les détails d'implémentation).

Il faut enfin, noter que ces opérations constituent une partie de celles requises dans un environnement de conception de bases de données. La différence essentielle est qu'en phase de conception initiale, les ensembles de descriptions des composants d'un schéma sont *vides* et il n'existe pas d'instances de données ni de programmes. Le processus de conception résulte en des ensembles de spécifications obtenus par ajouts de définitions de types. Les opérations de modification sont également utilisables durant ce processus, pour éventuellement remettre en cause des décisions de conception. Elles n'auront d'effets, dans ce cas, que sur les définitions des types.

Une alternative à la démarche de conception "à partir du vide" consiste à ré-utiliser des schémas existants et les modifier pour satisfaire les objectifs du système à réaliser. La ré-utilisation de schéma

Opérations de modification d'un type de schéma
(ChgTypSchéma.NomOpération)

NomOpération	*Rôle*
Nom	Changer le nom du type
	Toutes les opérations ci-dessous sont applicables à un type de schéma

Opérations de modification des types de base
(ChgTypBase.NomOpération)

NomOpération	*Rôle*
Nom	Changer le nom d'un type de base
ExprDef	Changer l'expression de définition d'un type de base

Opérations de modification des types d'attributs
(ChgTypAttr.NomOpération)

NomOpération	*Rôle*
Nom	Changer le nom
Domaine	Changer le domaine en un autre type de base ou d'entité
Cardinalité	De *'Simple'* à *'Multiple'* ou inversement

Opérations de modification de la relation de sous-type
(ChgSousType.NomOpération)

NomOpération	*Rôle*
AjtLien	Créer un lien de sous-type entre deux types d'entités
SuprLien	Supprimer un lien de sous-type

Opérations de modification des composants dans une composition de types d'entités
(ChgComposant.NomOpération)

NomOpération	*Rôle*
Rôle	Changer le nom de la fonction rôle associée à un composant
Nom	Changer le co-domaine d'une fonction rôle
Cardinalité	De *'Simple'* à *'Multiple'* ou inversement
Nature	Un couple (x, y) $\in$ *{'Partage', 'Exclusif'}* x *{'Faible', 'Forte'}*

FIG. 4.1 – *Liste des opérations de modification d'une spécification de schéma (1)*

intervient également dans le cas de l'extension d'un schéma existant par un autre schéma : on parle alors *d'intégration de schémas*.

4.2 Intégration de schémas et opérations de restructuration

Comme pour la modification de schémas, l'intégration peut survenir en présence ou non de bases de données et de programmes. *Dans le premier cas*, il s'agit d'incorporer à un schéma existant les types d'un autre schéma, ces derniers pouvant s'interpréter comme des spécifications des sous-schémas de nouveaux programmes devant agir sur la base ou comme des sous-schémas associés à des outils devant être intégrés dans un environnement fondé sur une base d'objets commune aux outils, etc.

Dans le second cas, le processus d'intégration de schémas fait partie d'une démarche de conception incrémentale : le schéma final résulte d'une suite d'intégrations de modélisations partielles. Dans chacun des cas, conceptuellement, l'intégration est vue comme un mécanisme qui "fusionne" les multigraphes représentant les schémas à intégrer. Formellement, il en résulte un nouveau type de schéma.

L'objectif de ce paragraphe n'est pas de proposer un mécanisme d'intégration de schémas (une

Opérations de modification de la composition de types d'entités
(ChgComposition.NomOpération)

NomOpération	*Rôle*
AjtComposant	Ajouter une définition de composant à une définition de composition
SuprComposant	Retirer une définition de composant

Opérations de modification d'un type d'entité
(ChgTypEntité.NomOpération)

NomOpération	*Rôle*
Nom	Renommer un type d'entité
SuperType	Changer le nom du super-type
AjtAttribut	Ajouter un type d'attribut à la liste des attributs caractérisant un type d'entité.
SuprAttribut	Retirer un type d'attribut de la liste

Opérations de modification de l'ensemble des types d'entités

NomOpération	*Rôle*
AjtTypEntité	Ajouter une définition de type d'entité.
SuprTypEntité	Supprimer une définition de type d'entité.

Opérations de modification d'un type d'association
(ChgTypAssociation.NomOpération)

NomOpération	*Rôle*
Nom	Renommage d'un type d'association
Rôle	Changer le nom de la fonction rôle d'un type d'entité participant
NomParticipant	Changer le co-domaine d'une fonction rôle
Cardinalité	idem que *ChgTypAttr.Nom*
AjtTypAttribut	Ajouter un type d'attribut à la définition du type d'association
SuprTypAttribut	Supprimer un type d'attribut de la définition

Opérations de modification de l'ensemble des types d'associations

NomOpération	*Rôle*
AjtTypAssociation	Ajouter une définition de type d'association.
SuprTypAssociation	Supprimer une définition de type d'association.

FIG. 4.2 – *Liste des opérations de modification d'une spécification de schéma (2)*

abondante littérature existe sur le sujet et [44] montre, notamment, l'intégration de schémas GORM). Mais, partant d'un processus général d'intégration en trois phases (détection de conflits entre les schémas à intégrer, résolution de conflits et fusion), nous montrons les problèmes à résoudre dans chacune des phases. Nous montrerons plus particulièrement, que les deux dernières phases de ce processus mettent en œuvre des opérations que nous appellerons *opérations de restructuration de schémas* qui peuvent être exprimées à l'aide de compositions d'opérations de modification présentées plus haut.

4.2.1 Phase 1 : les conflits de représentation et leur détection

Une spécification de schéma est le résultat, exprimé à l'aide d'un modèle de représentation, d'une abstraction d'un domaine d'application ou d'une de ses parties. Cette abstraction peut, de toute évidence, être différente selon ses auteurs. Les différences peuvent être de simples *homonymies ou synonymies* : des symboles différents ou le même symbole désignent la même abstraction ou des abstractions différentes, respectivement. Elles peuvent aussi être *structurelles* : par exemple, les "hauteurs"

des arbres de spécialisation d'un type d'entité sont différentes dans le couple de schémas, des concepts différents du modèle de représentation sont utilisés dans chacun des schémas pour représenter la même partie du domaine d'application, etc. Le problème est amplifié lorsque le couple de schémas n'est pas exprimé dans le même modèle de représentation : nous ignorerons ce cas de figure, que nous avions évoqué dans le chapitre 1, notamment dans le cadre de la fédération de bases de données.

Différentes classifications des conflits de représentation existent dans la littérature : la figure 4.3, inspirée de [180], illustre l'une d'elles qui nous semble suffisamment représentative.

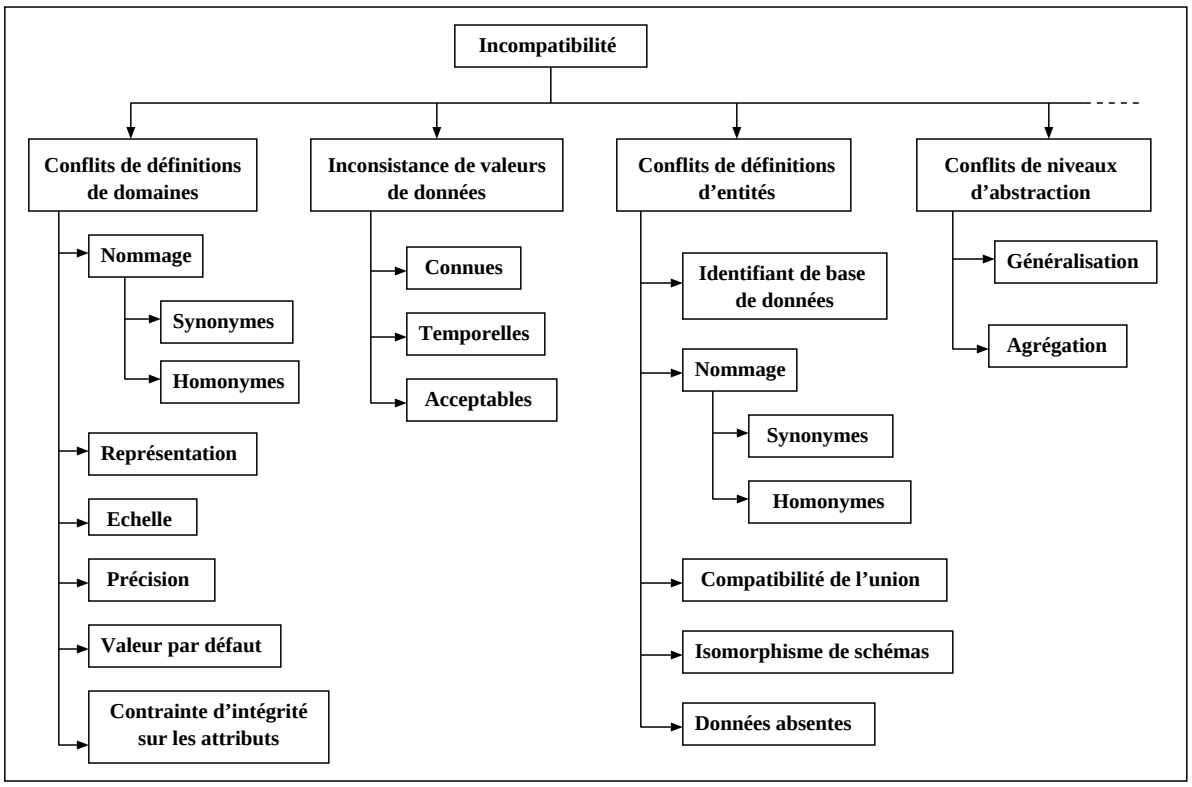

FIG. 4.3 – *Une typologie des différences de représentation dans des schémas*

La détection, à proprement parler, des conflits de représentation est un processus non déterministe nécessitant une connaissance du domaine d'application. Elle peut être réalisée manuellement ou de façon assistée. Certaines approches se basent sur un dictionnaire sémantique liant des types de données des schémas et les objets qu'ils représentent dans le domaine d'application [182], d'autres utilisent une notion de proximité sémantique et de calcul incertain [180]. [186] examine des instances de données pour tenter d'identifier des relations sémantiques, [46] se fonde sur l'unification de structures et la classification automatique et [192] prend en compte le comportement des types des objets pour identifier leurs éventuelles similitudes.

Pour notre part, nous nous cantonnons à identifier des similitudes entre des composants d'un couple de schémas à intégrer, en nous fondant sur les noms associés aux abstractions ou les constructeurs utilisés pour exprimer ces abstractions. Ainsi, par exemple (voir [45, 44] pour une liste exhaustive et une formalisation des relations de similitude engendrées par comparaison de couples de schémas),

– *un type* T_1 (d'attribut, d'entité ou d'association) *dans un schéma* S_1 est supposé similaire à *un type* T_2 *dans un schéma* S_2, si T_1 et T_2 ont le même nom. Concrètement, cela signifie que le nom donné à T_1 est associé, dans le second schéma, à un type de même structure ou de structure

différente de celle de T_1.

- deux types d'attributs sont supposés similaires s'ils ont le même domaine ou des domaines supposés similaires.
- deux types d'entités sont supposés similaires si l'intersection de leurs listes d'attributs ou celle de leurs listes de composants n'est pas vide.
- deux types d'association sont supposés similaires si l'intersection de leurs listes d'attributs n'est pas vide.

Ces similitudes sont de *simples hypothèses* devant être confirmées ou infirmées par un agent humain durant la phase de résolution de conflits.

4.2.2 Phase 2 : la résolution de conflits et les opérations de restructuration

La résolution de conflits de représentations consiste à "harmoniser" les abstractions formulées dans le couple de schémas à intégrer. Concrètement, cette phase se fonde sur les hypothèses de similitude engendrées dans la phase précédente. Celles qui sont confirmées peuvent conduire à des modifications ou à des restructurations dans l'un ou l'autre des deux schémas. Par exemple, un type d'attribut dans un des deux schémas peut être transformé en un type d'entité (ou réciproquement), un type d'entité dans un schéma peut prendre le même nom qu'un type d'entité dans l'autre schéma, etc.

En pratique, des opérations de modification de schémas, parmi celles présentées dans le paragraphe 4.1.2, sont utilisables pour concrétiser la résolution de conflits simples. Ainsi, le renommage de types permet d'éliminer les synonymies et les homonymies, le changement de domaines de types d'attributs permet de résoudre des différences de représentation, etc. Outre ces opérations, nous proposons six *opérations de restructuration* qui, elles, permettent de résoudre des différences d'abstraction :

1. Transformation d'un type de base en un type d'entité,
2. Transformation d'un type d'entité en un type d'association,
3. Transformation d'un type d'association en un type d'entité,
4. Transformation d'un lien de composition en un type d'association
5. Transformation d'un type d'association en un lien de composition,
6. Transformation d'un type d'attribut en une relation de sous-type.

Ces transformations sont exprimées comme une composition d'opérations de modification de schémas (ou transaction de modification de schéma). Nous allons l'illustrer sur la première transformation, en invitant le lecteur à se référer à [44] pour une présentation détaillée. Cette transformation peut être décidée lorsqu'un "phénomène" du domaine d'application est abstrait par un type d'attribut à valeurs dans un type de base, dans un schéma, alors qu'il l'est par un type d'entité dans l'autre (c'est le cas de A sur la figure 4.4), et que la solution du second schéma doit primer sur celle du premier.

La figure 4.5 montre les restructurations possibles :

(i) A est renommé en A' et son domaine est le type d'entité A. Cette configuration du schéma S_1 est obtenue par la séquence :

- Ajouter la définition du type d'entité A au schéma S_1 (*ChgTypEntité.AjtTypEntité*),

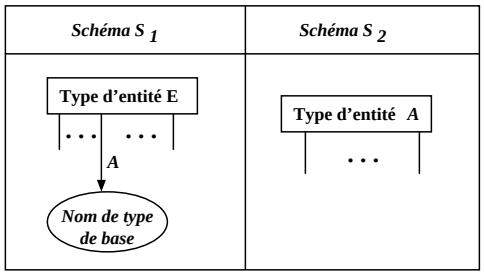

FIG. 4.4 – *Restructuration d'un type de base en un type d'entité (1)*

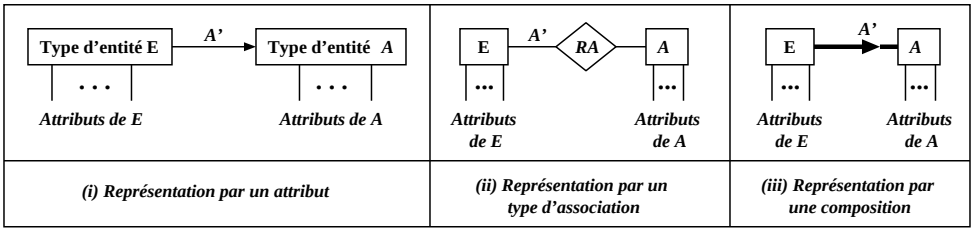

FIG. 4.5 – *Restructuration d'un type de base en un type d'entité (2)*

- Renommer le type d'attribut A en A' (*ChgTypAttr.Nom*),
- Affecter le type d'entité A comme domaine du type d'attribut A' (*ChgTypAttr.Domaine*).

(ii) A est renommé en A' et A' est un nom de rôle dans le type d'association qui lie les types d'entités E et A. Le nouveau schéma S_1 résulte de la séquence :

- Ajout de la définition du type d'entité A (*ChgTypEntité.AjtTypEntité*),
- Ajout de la définition du type d'association RA (*ChgTypAssociation.AjtTypAssociation*), avec E et A comme participants, A' étant le rôle de A dans RA. La cardinalité de A' est celle qu'avait A dans le schéma initial.

(iii) A est renommé en A' et A' est un nom de rôle dans la composition de E dont A est un composant. La séquence de modification de S_1 est similaire à celle qui précède, à savoir :

- Ajout de la définition du type d'entité A (*ChgTypEntité.AjtTypEntité*),
- Ajout de la définition d'une composition dans le type d'entité E (*ChgComposition.AjtComposant*), où A est un composant de rôle A' et de cardinalité identique à celle de A dans le schéma avant restructuration.

A l'issue de cette phase, qui consiste essentiellement en une suite d'opérations de modification ou de restructuration, le couple de schémas est mis en conformité : il ne reste plus qu'à produire le schéma résultant de l'intégration, par fusion du couple de schémas.

4.2.3 Phase 3 : la fusion de schémas

Cette phase procède par :

1) "Superposition" de types d'entités : un type d'entité E_1, dans un schéma S_1 et un type d'entité E_2, dans un schéma S_2, avérés similaires dans la phase précédente (i.e. ils ont le même nom, des

listes de propriétés éventuellement différentes, mais leur hypothèse de similitude a été confirmée), donnent naissance à un type d'entité de même nom dans le schéma résultat (figure 4.6). Ce type d'entité est caractérisé par l'union des propriétés (attributs, composition) des types d'entité (le cas où E_1 est strictement identique à E_2 n'est qu'un cas particulier de superposition). On

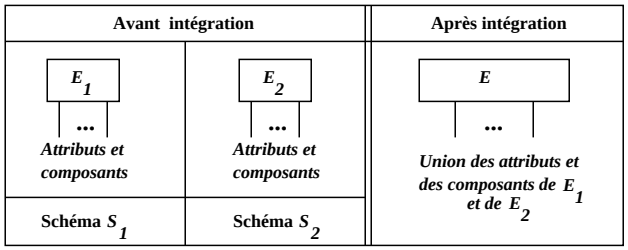

FIG. 4.6 – *Fusion de deux types d'entité*

notera que cette opération s'exprime à l'aide de l'opération d'ajout d'une définition de types d'entités dont la liste de types d'attributs et la composition éventuelle sont construites à partir des définitions des types des entités fusionnés et de leurs compositions.

2) Fusion par généralisation : si un lien de sous-type a été découvert entre un type d'entité E_1, dans un schéma S_1, et un type d'entité E_2, dans un schéma S_2, ce lien est créé dans le schéma résultant et les attributs communs aux deux types d'entité sont attachés au super-type (figure 4.7).

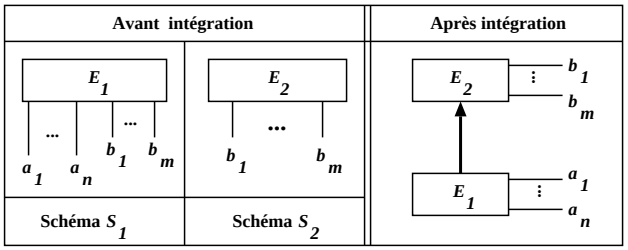

FIG. 4.7 – *Fusion par généralisation*

3) Fusion de types d'associations : les types des entités participant aux associations étant supposés intégrés, la fusion de deux types d'association (figure 4.8) consiste à fusionner leurs listes d'attributs. En d'autres termes, cela revient à ajouter une définition de type d'association au schéma résultat de l'intégration.

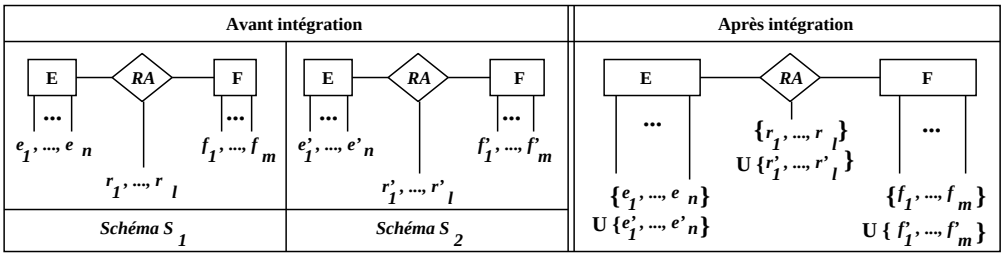

FIG. 4.8 – *Fusion de deux types d'association*

4.2.4 Intégration de schémas et invariants de schémas

Comme dans le cas des opérations de modification de schémas, les opérations de restructuration et de fusion, utilisées respectivement, dans les phases de mise en conformité et de fusion du processus d'intégration doivent satisfaire les critères de qualité d'un type de schéma : ceci est formellement démontré dans [45, 44], à l'exception d'un cas indécidable, schématisé sur la figure 4.9, qui viole la propriété d'arborescence du graphe de la relation de sous-type : supposons que les types d'entités E_1 et E_2 sont sous-types de E_{11} et de E_{22} dans leurs schémas respectifs. Supposons également que E_1 et

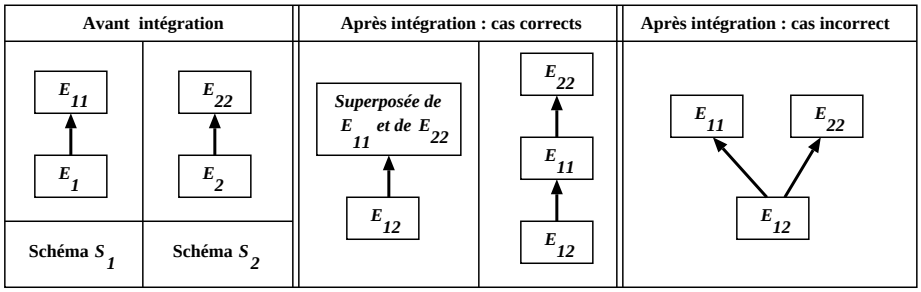

FIG. 4.9 – *Violation de l'héritage simple après intégration*

E_2 soient fusionnés par superposition en un type d'entité E_{12}. Si E_{11} et E_{22} sont aussi fusionnés par superposition ou qu'un lien de sous-type les relie alors le graphe de la fermeture transitive de la relation de sous-type est toujours un arbre. Dans le cas où aucun lien n'existe entre E_{11} et E_{22}, cette propriété n'est pas satisfaite, E_{12} "devenant" sous-type de E_{11} et de E_{22}. Cette situation peut dénoter une erreur de conception et met davantage en exergue le rôle de la connaissance sur le domaine d'application et celui de l'agent humain dans le processus d'intégration.

4.2.5 Conclusion

Dans ce paragraphe nous avons présenté, de façon très brève, un procédé d'intégration de schémas de données. En outre, nous avons essayé de montrer que ce procédé s'appuyait fortement sur les opérations de modifications "locales" de schémas exposées dans le paragraphe 4.1.2. L'intégration de schémas est alors considérée comme une opération "complexe" de modification qui trouve son application tant dans une approche de conception modulaire que dans une perspective d'évolution de bases existantes.

4.3 Conclusion : Notion de transaction de modification de schéma

L'orientation que nous avons prise, pour l'implantation de chacune des opérations de modification de schémas, est de ne pas propager leurs effets de bord. En effet, le but ultime de notre approche est de fournir ces opérations comme base d'un "langage de programmation" des modifications de schémas. Nous pensons qu'un système support est incapable de déterminer les finalités d'une modification et ainsi, "'coder en dur" les propagations peut ne pas satisfaire les intentions de la modification. Néanmoins, un support automatisé peut être apporté pour identifier les types et les objets concernés par les effets de bord. Un autre type de support est une évaluation du coût d'une modification "locale" d'un schéma qui permettrait à son auteur de mesurer l'ampleur des conséquences de l'opération.

L'idée est alors de permettre la composition d'opérations, au sein de *transactions de modifications de schémas*, pour permettre la prise en compte de leurs effets induits : c'est, par exemple, le cas de

l'implémentation du processus d'intégration de schémas décrite dans [44] qui :

- se fonde sur un mécanisme de raisonnement hypothétique dans les phases de comparaison et de mise en conformité,
- réalise la mise en conformité et la fusion de schémas par des compositions d'opérations de modification
- et produit un schéma satisfaisant les critères de qualité évoqués dans le paragraphe 3.10.

Cependant, quand les opérations de modification sont utilisées pour opérer une intégration ou une conception de schémas, les critères de qualité d'un schéma peuvent n'être satisfaits qu'à la fin du processus d'intégration ou de conception. Cela a pour conséquence que des états intermédiaires incohérents doivent être admis et que les transactions de modification de schémas prennent le statut de *longues transactions*. Des propositions comme celles décrites dans [57] devraient contribuer à la gestion de ces dernières. En outre, lorsque les modifications de schémas concernent le schéma d'une base de données existante, il est clair que des restructurations ou des modifications apportées au schéma ont des incidences, parfois non négligeables, sur les données et les programmes existants. L'activité engendrée par la prise en compte de ces incidences, doit faire partie de la transaction de modification de schéma.

Le chapitre qui suit est dédié à la présentation de diverses stratégies permettant de répercuter les effets de bord des opérations de modification et de restructuration de schémas ainsi qu'à la définition de ces effets de bord.

5

La Restructuration des données

Dans ce que nous avions appelé, en introduction de cette partie du document, *politique de migration immédiate* et *politique différée atomique*, il n'est pas nécessaire de maintenir des versions différentes d'objets ou de schémas : les modifications ou les restructurations de schémas ne sont visibles qu'après la répercussion de leurs effets de bord sur le objets et les programmes existants. Ces politiques ne nous semblent pas adaptées dans le cadre de certaines applications, comme la CAO ou le génie logiciel, où les transactions ne sont pas de courte durée. L'attente de la fin des transactions actives, pour répercuter les modifications, peut être longue, d'une part, et d'autre part, leur avortement peut ne pas être maîtrisable du point de vue de ses effets sur les objets modifiés par les transactions. Néanmoins, ces politiques nous semblent viables dans le cas de changements fondamentaux opérés sur la spécification de schémas et risquant de compromettre le projet en cours de développement.

Dans la politique *différée opportuniste*, les effets de bord des modifications de schémas sont appliqués aux objets au fur et à mesure de leur "libération" par les transactions ou lors de la première utilisation qui suit la modification de schéma. Dans ce cas de figure, cela signifie qu'il existe différentes versions de schémas et d'objets. Dans ce paragraphe, nous examinons plus en détail, les stratégies candidates à la répercussion des effets de bord des modifications de schémas, en insistant plus particulièrement, sur les mécanismes qui paraissent nécessaires pour pouvoir adopter l'une ou l'autre des politiques citées. Ces mécanismes consistent essentiellement en le maintien de relations entre différentes versions de schémas et d'expressions de *dérivation de valeurs d'objets* à partir de leur représentation selon une version de schéma. Les relations entre les versions de schémas et les expressions de dérivation de valeurs sont utilisables indépendamment de la politique de migration adoptée : dans une politique de migration immédiate, les relations et les expressions disparaissent à la fin de la migration, alors qu'elles persistent dans le cas des autres politiques.

Dans ce chapitre, nous allons montrer, de façon intuitive (paragraphe 5.1) les problèmes posés et les stratégies envisageables lors de la migration de données comme suite à un changement de la spécification d'un schéma. Nous formaliserons notre discours, dans le paragraphe 5.2, en termes d'homomorphismes entre versions de schémas et de données. Enfin, nous proposerons, dans le paragraphe 5.3 des homomorphismes pour des opérations de modification et des opérations de restructuration de schémas évoquées dans le chapitre précédent.

5.1 Versions de schémas et d'objets

Dans un premier temps, nous allons raisonner sur la base de l'exemple de la figure 5.1 qui montre trois versions d'un type d'entité spécifiant, par exemple, une tâche, avec, dans la première version, un *Nom*, celui de son *Responsable* et sa *Durée* exprimée par un couple de *Date*s.

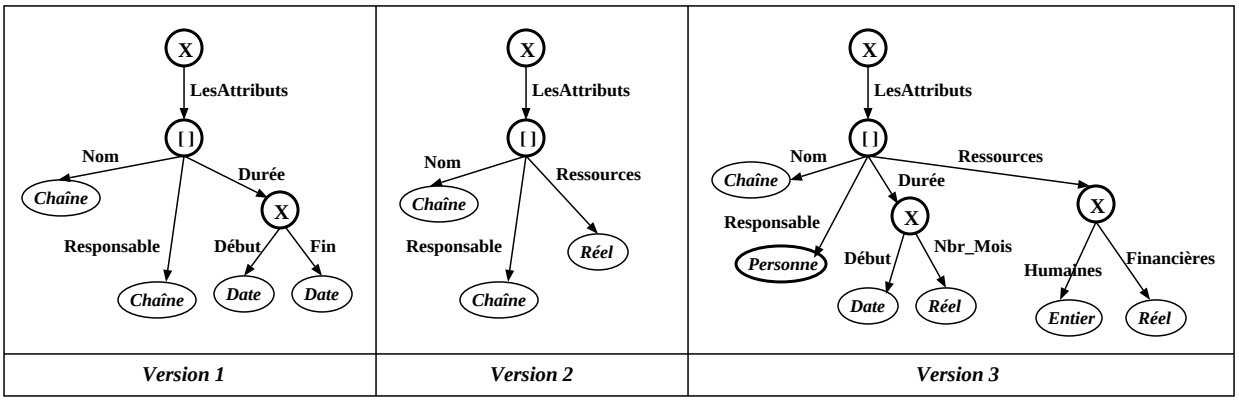

FIG. 5.1 – *Trois versions de schéma*

La liste des types d'attributs supposés caractériser le type d'entité a subi les modifications suivantes :

1. *de la version 1 à la version 2*, l'attribut *Durée* a disparu et un attribut *Ressources*, indiquant le montant des ressources financières allouées à chaque tâche, a été ajouté,

2. *de la version 2 à la version 3*, le domaine de *Responsable* devient le type d'entité *Personne*, l'attribut *Durée* apparaît de nouveau mais sous la forme d'un couple (*Date de début*, *Nombre de mois*). Enfin, *Ressources* est restructuré en un couple (ressources *Humaines*, ressources *Financières*).

Le premier cas, que nous examinerons, est celui où des versions successives du schéma ne sont pas maintenues (paragraphe 5.1.1). *Le second cas*, traité dans le paragraphe 5.1.2, est celui où des versions multiples de schémas et d'objets sont maintenues.

5.1.1 Pas de version et migration immédiate

Cela signifie que seule la dernière version de schéma est valide et qu'en outre, les objets doivent être restructurés pour s'y conformer. Sur notre exemple, le passage de la version 1 à la version 2 ne nécessite que la suppression des instances de l'attribut *Durée* de tous les objets du type : il y a *restructuration des instances sans coercion de valeurs*. Cela a, comme conséquence, que des programmes peuvent devenir inopérants sur les objets restructurés : sur le même exemple, toute référence à l'attribut *Durée*, ou à une de ses projections, devient indéfinie après migration de la version 1 à la version 2.

Si la version 3 venait à succéder à la version 2, la migration poserait davantage de problèmes :

1. *La restructuration des objets*, de sorte à ce qu'il se conforment à la nouvelle version, est réalisable par des extensions de la représentation du produit cartésien. Formellement, cela revient à considérer le type d'entité étendu par de nouveaux types d'attributs, comme un sous-type de celui de la précédente version.

2. *Les valeurs des objets* devraient être dérivées de celles présentes dans l'ancienne version. L'exemple montre différentes situations :

   – la valeur de *Ressources.Financières* est identique à celle de *Ressources* dans la version 2.

   – la nouvelle valeur de l'attribut *Responsable* est un identifiant du type d'entité *Personne* : comment la déterminer ?

   – la valeur des attributs *Durée* et *Ressources.Humaines* sont indéfinies.

3. *Des opérations ne sont plus applicables* sur des types d'attributs dont la spécification a changé : c'est le cas de *Responsable* et de *Ressources*. Dans ce cas précis, si nous nous situons dans une vision de la programmation par objets où les opérations ou méthodes sont invoquées par des messages de la forme *"<Désignation d'objet>.<Sélecteur de méthode>"*, on peut envisager la ré-écriture des messages *"Ressources.<Sélecteur de méthode>"* en *"Ressources.Financières.<Sélecteur de méthode>"*.

De façon générale, dans certaines situations *décidables*, une restructuration des types de données, une migration de leurs valeurs et une contribution à l'adaptation des programmes sont possibles. Nous avons déjà évoqué l'exemple de *Ressources.Financières*. Un autre exemple est illustré par l'attribut *Durée* dont la migration des valeurs, de la version 1 à la version 3, est réalisable par transformation de ses valeurs dans la version 1.

Les situations décidables et les règles associées pour concrétiser la migration sont détaillées dans le paragraphe 5.3 et nous reviendrons, dans le chapitre 6, sur les fondements de la restructuration des objets et de leurs valeurs, ainsi que sur un processus de restructuration basé sur l'identification de relations de sous-types et sur la génération de fonctions de coercion.

Examinons, maintenant, le cas où plusieurs versions de schémas et d'objets sont maintenues.

5.1.2 Versions de schémas et d'objets

Dans cette optique, chaque version d'objet se conforme à une version de schéma. La production d'une version initiale d'objet se conformant à une nouvelle version de schéma pose les mêmes problèmes que ceux esquissés dans le paragraphe qui précède.

Concernant les opérations sur les types des objets, on peut considérer deux cas de figures. *Dans le premier cas*, chaque version de schéma dispose d'une version des opérations sur ses types, c'est-à-dire que l'impact des modifications de schémas est répercuté immédiatement sur les opérations : solution de type *"interopérabilité orientée traitement"*, évidemment peu économique puisque les opérations des types non modifiés ne sont pas réutilisés. *Dans le second cas de figure*, seules les opérations sur les types modifiés sont dotées de "versions", pour certaines, ou rendues polymorphes, pour d'autres, le mécanisme de liaison dynamique et le polymorphisme permettant d'assurer le partage et la ré-utilisation.

Concernant la représentation des instances des objets, on peut opter pour deux stratégies.

(i) *Celle des copies multiples* d'objets : cela signifierait qu'il existe autant de bases de données que de versions de schémas et qu'un programme est attaché à une version de schéma et à la base correspondante.

Outre la redondance introduite par les copies, si les versions des bases de données sont considérées comme dépendantes, il est nécessaire de répercuter la mise à jour de chacune d'elles sur toutes les autres. Concrètement, cela signifie que toute modification d'un objet dans une version V_j d'un schéma S déclenche, par effet de bord, la mise à jour de l'objet dans toutes les versions existantes. Cela a comme conséquence que,

1. d'une part, des liens "sémantiques" entre les versions sont maintenus : ils expriment des correspondances entre des composants des spécifications des versions de schémas. Dans la figure 5.1, le fait que *Ressources*, dans la version 2, soit représenté par *Ressources.Financières*, dans la version 3, ou que *Durée*, dans la version 1, joue le même rôle que *Durée*, dans la version 3, et en est dérivable, sont des exemples de liens entre des versions de schémas.
2. et que, d'autre part, les opérations sur les types des objets sont disponibles sur les différentes versions.

Ainsi, outre ces liens "sémantiques" qui, en définitive, matérialisent l'historique des modifications subies par les spécifications des types d'un schéma, des "correspondances" doivent être établies entre les opérations associées aux types des objets dont les spécifications diffèrent dans les versions de schémas.

Le cas le plus favorable est celui où un type T_i^{j+1} dans une version V_{j+1} d'un schéma S est *sous-type* du type T_i^j dans la version V_j de S. Une des propriétés de la relation de sous-type étant qu'une instance du sous-type est utilisable partout où une instance de son super-type l'est, T_i^{j+1} "hériterait" alors, du comportement de T_i^j.

Dans les autres cas, nous rejoignons le discours tenu dans le chapitre introductif de ce document à propos de l'interopérabilité orientée traitements ou orientée objets : les opérations d'un type T_i^j peuvent être applicables au type T_i^{j+1} via le polymorphisme avec ou sans coercion. En d'autres termes,

1) si une fonction de coercion de T_i^j en T_i^{j+1} existe, les opérations de T_i^j sont ré-utilisables sur les objets de type T_i^{j+1} (*polymorphisme ad hoc avec coercion inter-versions de schémas et d'objets*).

2) Si tel n'est pas le cas, une opération O du type T_i aurait deux implémentations, l'une s'appliquant aux objets de type T_i^j et l'autre à ceux de type T_i^{j+1}. O sera *liée dynamiquement* à l'une ou à l'autre de ses implémentations, selon la version de l'objet sur laquelle elle est invoquée (*polymorphisme ad hoc sans coercion entre les versions de schémas et d'objets*).

(ii) *La seconde stratégie de représentation des objets* dans le cas où des versions sont nécessaires, est celle de la *copie "primaire"*. Dans ce cas, une version de schéma est privilégiée par rapport aux autres : la base se conforme à cette version privilégiée, les autres versions de schémas étant considérées comme des *clichés* ou des vues sur celle-ci, c'est-à-dire un sous-ensemble strict ou dérivable du schéma primaire. Cette stratégie soulève *deux types de problème*.

Le premier concerne le choix de la version à privilégier : toute autre version doit en être dérivable. Comme le montre l'exemple de la figure 5.1, la version 3 peut jouer le rôle de schéma primaire à condition que des liens sémantiques entre les versions soient explicités, comme dans le cas des copies multiples. En outre, la version privilégiée de schéma doit garder trace de l'historique des modifications subies par le schéma initial.

Le second problème concerne la manipulation des objets selon une des versions non primaires du schéma : les solutions qu'on peut y apporter sont similaires à celles discutées dans le cas des copies multiples, à la propagation des mises à jour près. En effet, l'accès à la valeur d'un objet selon une version de schéma peut être réalisée par dérivation à partir de la version primaire. Cela suppose que la spécification d'un schéma primaire S_p est une "synthèse" des spécifications des versions successives d'un schéma. S_p devra comporter, pour chacun des ses types T_j^p,

- l'indication de la version du type choisie pour représenter les instances de T_j^p dans le schéma primaire,
- et, pour chaque version V_i de schéma dans lequel un type T_k^i est défini, la version primaire comporte l'expression de la correspondance entre les valeurs du type T_j^p, telles que représentées dans S_p, et leurs représentations dans V_i.

La figure 5.2 illustre une représentation diagrammatique d'un schéma "primaire" synthétisant les trois versions de la figure 5.1 : chaque nœud est étiqueté par :

- une expression de la forme *Rep* = V_i où V_i désigne une version de schéma; cette expression indique la version de schéma selon laquelle le nœud est représenté,

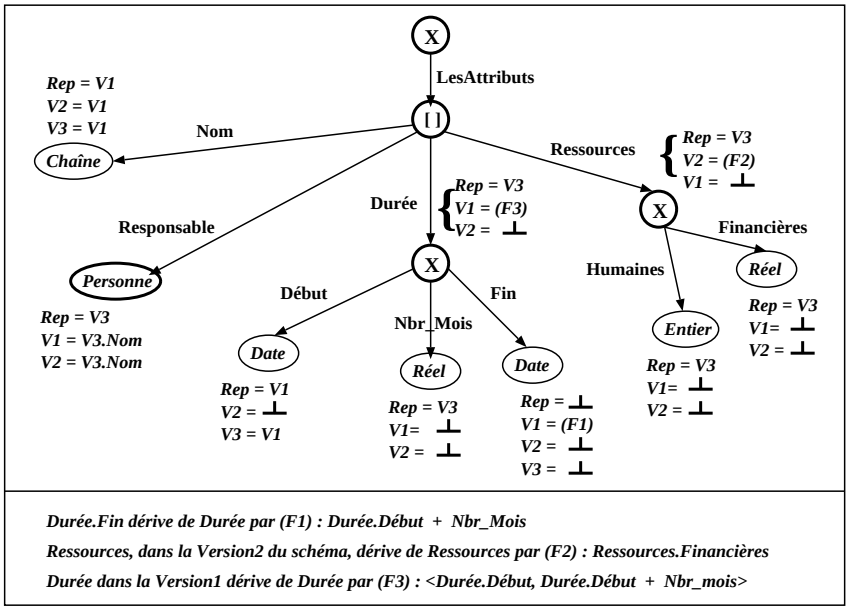

FIG. 5.2 – *Schématisation d'une version "primaire" d'objet*

– et par une liste d'expressions de la forme $V_i = Formule$ exprimant une fonction de dérivation de la valeur du nœud pour la version i du schéma, à partir de la version primaire.

Par exemple, les étiquettes attachées au nœud du domaine de *Nbr_Mois* signifient que la représentation est conforme à la spécification de ce nœud dans la version V_3 du schéma et que ce nœud n'est pas défini dans les autres versions ($V_1 = \perp$, $V_2 = \perp$). De même, l'étiquetage du nœud domaine de *Ressources* exprime le fait que sa représentation se conforme à la spécification du type dans la version 3 du schéma, que l'attribut n'est pas défini dans la version 1 du schéma ($V_1 = \perp$) et que sa valeur, dans la version 2 du schéma, est dérivable de celle de la version 3 par la formule (F_2). Enfin, l'étiquetage du nœud *Personne* stipule que la représentation est celle de la version V_3 et que les valeurs de ce nœud, dans les autres versions, sont dérivées par l'accès à l'attribut *Nom* de l'entité *Personne*.

En pratique, la stratégie de la copie primaire revient à considérer le schéma primaire comme une abstraction "pivot" selon laquelle sont opérées les instanciations des objets. La manipulation de ceux-ci selon une des versions de schéma se fait par conversion de cette représentation pivot vers la représentation dans la version de schéma et inversement.

5.1.3 Conclusion

Cette discussion informelle des effets de bord induits par la modification de la spécification d'un schéma, montre que, quelle que soit la politique et la stratégie de migration adoptée (immédiate, différée, copie primaire ou multiple), nous sommes amenés à définir des fonctions de dérivation ou de conversion de données et de programmes. La variété des modifications que l'on peut appliquer sur un schéma nous donne la mesure de l'ampleur de la tâche pour déterminer un ensemble complet de fonctions. Nous avons essayé de montrer que ces fonctions visaient, en fait, à définir un polymorphisme, inter-versions de schémas, des opérations des types d'un schéma. En outre, dans une politique de migration différée ou opportuniste, les fonctions sont persistantes alors qu'elles ne le sont pas dans le cas d'une migration immédiate.

Dans le paragraphe qui suit, nous proposons un exposé plus formel de l'approche que nous préco-

nisons, puis nous montrerons, dans le paragraphe 5.3 un ensemble de fonctions de dérivation en liaison avec les opérations de modification et de restructuration de schémas précédemment exposées dans le chapitre 4.

5.2 Versions de schémas et homomorphismes

Dans ce qui suit, nous dirons :

- qu'un type T_j^{i+1}, dans la version V_{i+1}^S d'un schéma S, est une *variante d'un type* T_j^i de la version V_i^S de S, si T_j^{i+1} est obtenu par l'application d'opérations de modification de schéma à la spécification du type T_j^i.
- qu'une spécification V_{i+1}^S d'un schéma S est une *version d'une spécification* V_i^S *d'un schéma S* si V_{i+1}^S résulte de l'application d'une suite d'opérations de modification ou de restructuration de V_i^S.

La correspondance entre les représentations des objets dans deux versions V_i et V_j de schémas est définie en termes d'homomorphismes entre les espaces des valeurs des types contenus dans les spécifications V_i et de V_j. Ces homomorphismes sont définis sur les objets munis d'une identité, aux sens du paradigme objet : dans notre cas, il s'agira des instances des types d'entités et des types d'associations. Concrètement, cela signifie que ces fonctions doivent être associées à chaque variante d'un type d'entité ou d'un type d'association.

Dans la formalisation ci-dessous, nous supposerons que l'on dispose d'une opération générique d'instanciation d'objets *Mk_NomDeType*, où *NomDeType* est un nom de type d'entité ou de type d'association. Cette opération crée une instance de *NomDeType*, l'ajoute à son espace de valeurs et rend comme résultat l'identifiant de l'objet créé.

5.2.1 Morphismes d'entités

5.2.1.1 Définition formelle

Rappelons qu'un type d'entité est défini par :

DefTypEntitéComposé $\stackrel{def}{=}$ $<$Nom : *NomTypEntité*,
NomSuperType : *NomTypEntité* $\cup$ *{'Entité'}*,
LesAttributs : [*DefTypAttribut*],
Composition : *DefComposition*$>$

DefTypEntitéNonComposé, une définition de type d'entité non composé, n'est pas dotée de l'attribut *Composition* et *DefTypEntité*, une définition d'un type d'entité est alors une *DefTypEntitéComposé* ou une *DefTypEntitéNonComposé*.

Rappelons, en outre, que si *TE* est une définition de type d'entité composé et TE' celle de son super-type, une *entité* de type *TE* est définie par :

InstanceEntitéComposée $\stackrel{def}{=}$ <Identifiant : *OID(TE.Nom)*,
IdentifiantSuperType : *OID(*TE'*.Nom)* $\cup$ *{'Racine'}*,
[*InstanceAttribut*],
InstanceComposition>

Enfin, une instance de *TE* est constituée par l'ensemble des *entités* de type *TE*.

Appliquée à un type d'entité de nom $TE.Nom$, *Mk_TE.Nom* a

– *comme arguments*

  – l'identifiant d'une instance du type d'entité $TE'.Nom$ super-type de *TE.Nom*,
  – la liste des valeurs des attributs du type d'entité,
  – et, éventuellement, une instance des entités entrant dans la composition.

– et *comme résultat* un identifiant de TE (*OID(TE.Nom)*)

Soit $< NomSuperType^i : NomTypEntite^i, a_1^i : t_1^i, \cdots, a_n^i : t_n^i >$ une dénotation simplifiée de la définition d'un type d'entité TE^i où les a_k^i ($1 \leq k \leq n$) désignent des noms d'attributs ou de rôles de composants de TE^i et les t_k^i leurs domaines respectifs. Soit également $< NomSupertype^j : NomTypEntite^j, a_1^j : t_1^j, \cdots, a_m^j : t_m^j >$ la dénotation simplifiée de la définition du type d'entité TE^j, *variante* de TE^i.

Faire correspondre à une instance e^i de TE^i une instance e^j de TE^j, revient à définir une transformation $\mathcal{T}_{i-j}$ du type spécifié par TE^i dans celui spécifié par TE^j. $\mathcal{T}_{i-j}$ est définie par la donnée d'un ensemble de fonctions $\mathcal{T}_{i-j}^{a_k}$ de correspondance entre les ensembles $\{< a_k^i : t_k^i >\}$ et $\{< a_l^j : t_l^j >\}$.

Plus formellement, si $Mk_TE^i.Nom(IdentifiantSupertype^i : os, a_1^i : v_1^i, \cdots, a_n^i : v_n^i)$ dénote l'instance e^i de TE^i (os étant l'identifiant du super-type de e^i et les v_k^i désignant les valeurs des attributs a_k^i dans leurs domaines respectifs), son adaptation à la définition de TE^j est définie par :

$$Mk_TE^j.Nom(\mathcal{T}_{i-j}^{NomSupertype^i}(os : OID(NomTypEntite^i)), \mathcal{T}_{i-j}^{a_1}(a_1^i : v_1^i), \cdots, \mathcal{T}_{i-j}^{a_n}(a_n^i : v_n^i)).$$

On notera que les transformations $\mathcal{T}_{i-j}^{a_k}$ s'appliquent au couple $< a_k^i : t_k^i >$, tout simplement parce que chaque membre du couple peut avoir subi une modification : les types d'attributs ou les rôles peuvent subir des renommages et leurs domaines respectifs peuvent changer. De ce fait, il est plus correct de définir $\mathcal{T}_{i-j}^{a_k}$ par un couple de fonctions :

– l'une, $\mathcal{T}structure_{i-j}^{a_k}$, de profil : $\{a_k^i\} \longrightarrow \{a_l^j\}$, qui décrit la correspondance entre les structures,

– et l'autre, $\mathcal{T}valeur_{i-j}^{t_k}$: $\{t_k^i\} \longrightarrow \{t_l^j\}$ qui décrit la correspondance entre les valeurs.

Par ailleurs, comme nous l'avons évoqué à travers l'exemple utilisé dans l'exposé informel,

1. des attributs peuvent ne pas être définis dans une des deux versions du schéma,
2. et, dans certains cas, des fonctions de correspondance ne peuvent pas être exhibées.

Les co-domaines des fonctions $\mathcal{T}structure$ et $\mathcal{T}valeur$ doivent alors être étendus comme suit :

– $\mathcal{T}structure_{i-j}^{a_k} : \{a_k^i\} \longrightarrow \{a_l^j\} \cup \{\perp\} \cup \{\text{`}Erreur'\}$, $\perp$ exprimant qu'il n'existe pas dans T^j d'attribut correspondant à a_k et *'Erreur'* signifiant que a_k n'est pas défini dans TE^i.

– $\mathcal{T}valeur^{t_k}_{i-j} : \{t^i_k\} \longrightarrow \{t^j_l\} \cup \{\omega\} \cup \{\surd\}$, ω signifiant qu'il n'existe pas de fonction de coercion du type de a_k, dans TE^i, dans son homologue dans TE^j et $\surd$ signifiant que la valeur de l'homologue de a_k dans TE^j est *inconnue*.

5.2.1.2 Exemples de morphismes

Illustrons ce discours sur notre exemple dont la représentation diagrammatique est rappelée dans la figure 5.3.

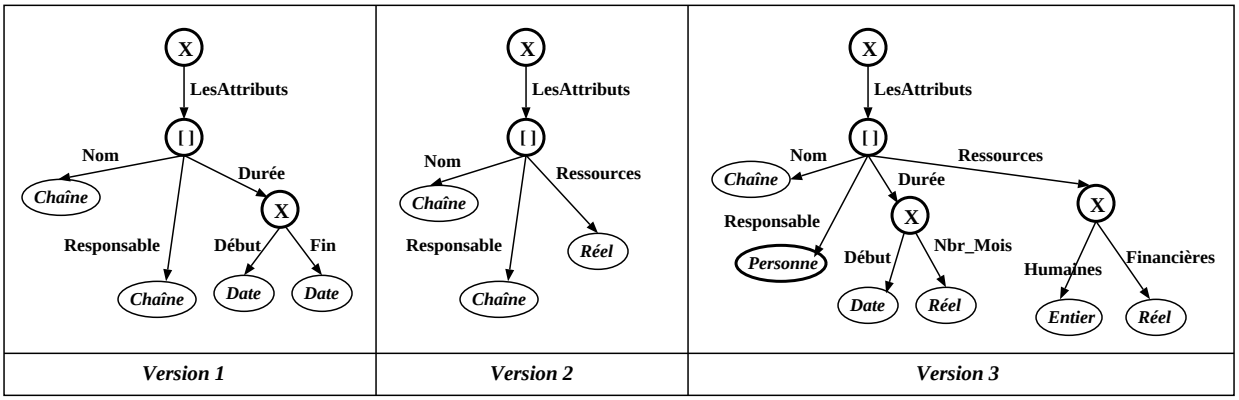

FIG. 5.3 – *Trois versions de schéma (rappel)*

- Correspondance de l'attribut *Nom* de la version 1 à la version 2 :

  - $\mathcal{T}structure_{1-2}(Nom^1) = Nom^2$
  - $\mathcal{T}valeur_{1-2}(Nom^1)$ = *Val*(Nom^1) (*Val* désignant une fonction générique d'accès à la valeur d'un objet).

  Ces expressions s'interprètent de la façon suivante : l'attribut *Nom* dans la version 2 du schéma correspond à l'attribut *Nom* dans la version 1 et sa valeur dans la version 2 est dérivée de la version 1 par *l'identité*.

- Correspondance de l'attribut *Durée* de la version 1 à la version 2 :

  - $\mathcal{T}structure_{1-2}(Duree^1) = \perp$
  - $\mathcal{T}valeur_{1-2}(Duree^1) = \omega$

- Correspondance de l'attribut *Fin* de la version 1 à la version 3 :

  - $\mathcal{T}structure_{1-3}(Fin^1)$ = Nbr_Mois^3
  - $\mathcal{T}valeur_{1-3}(Fin^1)$ = *Val*($Duree.Fin^1$) - *Val*($Duree.Debut^1$)

- Correspondance de l'attribut *Nbr_Mois* de la version 3 à la version 1 :

  - $\mathcal{T}structure_{3-1}(Nbr_Mois^3)$ = Fin^1
  - $\mathcal{T}valeur_{3-1}(Nbr_Mois^3)$ = *Val*($Duree.Debut^3$) + *Val*($Duree.Nbr_Mois^3$)

- Correspondance de l'attribut *Ressources* de la version 1 à la version 3 :

  - $\mathcal{T}structure_{1-3}(Ressources^1)$ = $Ressources.Financieres^3$

- $\mathcal{T}valeur_{1-3}(Ressources^1)$ = $< \surd,$ *Val*($Ressources^1$)>

- Correspondance de l'attribut *Ressources* de la version 3 à la version 1 :

  - $\mathcal{T}structure_{3-1}(Ressources^3) = Ressources^1$
  - $\mathcal{T}valeur_{3-1}(Ressources^3)$ = *Val*($Ressources.Financieres^3$)

Donnons, pour terminer, une *interprétation procédurale* de ces fonctions de correspondance.

5.2.1.3 Interprétation procédurale des morphismes

Dans le cas d'une migration des instances, $\mathcal{T}valeur_{i-j}(a_k)$ spécifie la fonction de coercion de la valeur de a_k de la version "source" V_i à la version "cible" V_i et $\mathcal{T}structure_{i-j}(a_k)$ spécifie la destination, dans la base de données, de la valeur de la fonction de coercion.

Dans le cas où des versions multiples des versions et des objets sont maintenues, les fonctions sont utilisées pour restructurer les objets et transformer leurs valeurs d'une représentation dans une version à une autre. Il en est de même *dans le cas où la stratégie "copie primaire"* est adoptée, la copie primaire jouant le rôle de la version source.

Enfin, on notera que ces fonctions de correspondance décrivent des coercions de types utilisables dans une *perspective d'adaptation des méthodes (ou opérations)* associées aux types. En d'autres termes et à titre d'exemple, supposons que l'on veuille changer la date de *Fin* (i.e. envoi du message approprié à "l'objet" *Fin*) et que les objets sont représentés conformément à la version 3 du schéma, les correspondances définies ci-dessus, pour l'attribut *Fin*, de la version 1 à la version 3, peuvent servir

(i) à *transformer*, par $\mathcal{T}valeur_{1-3}(Fin^1)$, les valeurs des paramètres de l'opération de modification,

(ii) et à "détourner" le message pour l'adresser à l'objet *Nbr_Mois*, la nouvelle destination étant spécifiée par $\mathcal{T}structure_{1-3}(Fin^1)$.

Le même raisonnement est tenu pour définir des correspondances entre des *variantes* de types d'associations dans des versions de schémas.

5.2.2 Morphismes d'associations

Souvenons-nous

1. qu'un type d'association est défini par :

$$\begin{aligned} \textit{DefTypAssociation} \stackrel{def}{=\!=} < & \text{Nom} : \textit{NomTypAssociation}, \\ & \text{Participant}_1 : < \text{Rôle}_1 : \textit{NomRôleAssociation}, \\ & \qquad \text{NomParticipant}_1 : \textit{NomTypEntité}, \\ & \qquad \text{Cardinalité}_1 : \{\textit{'Simple'}, \textit{'Multiple'}\} >, \\ & \text{Participant}_2 : < \text{Rôle}_2 : \textit{NomRôleAssociation}, \\ & \qquad \text{NomParticipant}_2 : \textit{NomTypEntité}, \\ & \qquad \text{Cardinalité}_2 : \{\textit{'Simple'}, \textit{'Multiple'}\} >, \\ & \text{Attributs} : [\textit{DefTypAttribut}] > \end{aligned}$$

2. qu'une instance d'un type d'association TA est définie par :

$$\begin{aligned} \textit{InstanceAssociation} \stackrel{def}{=\!=} < & \text{Identifiant} : \textit{OID(TA.Nom)}, \\ & \text{Rôle}_1 : r_1, \\ & \text{Rôle}_2 : r_2, \\ & [\textit{InstanceAttribut}] > \end{aligned}$$

$r_k \in OID(TA.Participant_k.NomParticipant_k)$, (k ∈ {1,2}), étant un identifiant de l'espace des identifiants des types d'entités participant au type d'association.

3. que toutes les instances d'un type d'association sont "groupées" dans un ensemble.

La définition de fonctions de correspondance entre deux versions d'un type d'association se fait de façon analogue au cas des types d'entités : *Mk_TA.Nom*, l'opération générique d'instanciation d'un type d'association de nom *TA.Nom, a comme arguments*

– les identifiants, r_1 et r_2, des entités associées

– et la liste $[a_k : v_k]$ des valeurs des attributs éventuels du type d'association.

Mk_TA.Nom rend comme résultat l'identifiant de l'objet créé (*OID(TA.Nom)*).

La transformation d'une instance ta^i d'un type d'association TA^i en une instance ta^j conforme à la variante TA^j de TA^i est obtenue grâce à un ensemble de fonctions de correspondance entre les identifiants des entités associées dans ta^i, les valeurs des attributs de ta^i et leurs homologues dans ta^j.

Formellement, si

Mk_TA.Nom(Role$_1^i$: r_1^i, Role$_2^i$: r_2^i, $a_1^i : v_1^i, \cdots, a_n^i : v_n^i$)

dénote l'instance ta^i de TA^i, et que ta^j, l'instance correspondante dans TA^j, est dénotée par,

Mk_TA.Nom(Role$_1^j$: r_1^j, Role$_2^j$: r_2^j, $a_1^j : v_1^j, \cdots, v_m^j : t_m^j$),

ta^j est obtenu par :

$$\textit{Mk_TA.Nom}(\mathcal{T}_{i-j}^{Role_1^i}(Role_1^i : r_1^i), \mathcal{T}_{i-j}^{Role_2^i}(Role_2^i : r_2^i), \mathcal{T}_{i-j}^{a_1}(a_1^i : v_1^i), \cdots, \mathcal{T}_{i-j}^{a_n}(a_n^i : v_n^i))$$

En ce qui concerne les attributs du type d'association, la correspondance est établie entre l'ensemble $\{a_k^i : t_k^i\}$ et l'ensemble $\{a_l^j : t_l^j\}$ $(1 \leq k \leq n$ et $1 \leq l \leq m)$ de façon analogue au cas des types d'entité, par

– une correspondance entre les structures :

$$\mathcal{T}structure_{i-j}^{a_k} : \{a_k^i\} \longrightarrow \{a_l^j\} \cup \{\perp\} \cup \{\text{`}Erreur\text{'}\},$$

– et une correspondance entre les valeurs :

$$\mathcal{T}valeur_{i-j}^{t_k} : \{t_k^i\} \longrightarrow \{t_l^j\} \cup \{\omega\} \cup \{\surd\}.$$

Concernant les participants du type d'association, les correspondances prennent en compte les noms des rôles des participants et leurs espaces de valeurs.

1) *Cas des rôles* : $\mathcal{T}structure_{i-j}^{Role_u^i} = Role_u^j$, $(1 \leq u \leq 2)$,
c'est-à-dire que $\mathcal{T}structure_{i-j}$ fait correspondre à un symbole de rôle dans TA^i un symbole de rôle dans TA^j, celui-ci pouvant être le symbole utilisé dans TA^i ($Role_u^i$) ou celui donné, par renommage, au participant du type d'association ($Role_u^j$).

2) *Cas des espaces de valeurs* : $\mathcal{T}valeur^{r_u^i}_{i-j} = r_u^j$, $(1 \leq u \leq 2)$,
c'est-à-dire que $\mathcal{T}valeur_{i-j}$ établit une correspondance entre les espaces des identifiants des entités associées. *D'un point de vue opérationnel*, cela signifie que si les domaines des participants du type d'association ont changé, il faudrait être capable d'exhiber une correspondance entre les anciens et les nouveaux domaines. Nous reviendrons plus en détail sur ce point dans le paragraphe 5.3.

5.2.3 Conclusion

Nous venons de montrer que la migration ou la gestion de versions de schémas et d'objets pouvait se formaliser en termes de couples de fonctions de correspondance entre des spécifications de types et des instances de types. Nous avons également montré, à travers les exemples de morphismes d'entités, que plusieurs interprétations procédurales pouvaient être données aux spécifications statiques des fonctions de correspondance : que ce soit pour la migration de données, la gestion de versions ou l'adaptation de méthodes associées aux types de données. Ceci conforte notre point de vue sur l'adéquation de nos propositions à l'une ou à l'autre des politiques de migration ou de stratégies de représentation des objets. Dans les deux prochains paragraphes, nous essayons de mettre en évidence des définitions de fonctions de correspondance entre des versions de schémas obtenues soit par l'application d'une opération de modification (paragraphe 5.3), soit par l'application d'une des opérations de restructuration ou de fusion utilisables durant le processus d'intégration de schémas (paragraphe 5.4).

5.3 Fonctions de correspondance et modification de schémas

Nous nous appuyons sur la classification des opérations de modification de schémas données dans le paragraphe 4.1.1 du chapitre 4 et rappelée dans le tableau de la figure 5.4.

Pour chaque classe d'opérations nous exhibons le couple de fonctions de correspondance $\mathcal{T}structure$ et $\mathcal{T}valeur$ entre deux *variantes* i et j de la spécification d'un type dans les spécifications des versions V_i^S et V_j^S d'un schéma S. Nous en donnons, dans les cas décidables, une sémantique statique et des interprétations procédurales. Dans certaines de ces interprétations, nous conviendrons que la désignation d'un objet est faite par une "expression de chemin" [5] qui prendra une des formes suivantes (les exemples de désignation d'objets se réfèrent à l'exemple de schéma dont la représentation diagrammatique est donnée par la figure 5.5) :

1) *pour les entités, instances d'un type d'entité* TE : Identifiant de l'entité (noté $OID_{TE.Nom}$) ou nom du type d'entité,

2) *pour les associations* : même type de désignation que les entités

3) *pour les attributs* : Un attribut étant rattaché à un type d'entité ou d'association, sa désignation est faite par : <Désignation de l'entité ou de l'association>.<Nom de l'attribut>.

Exemple : *Personne.NomP*, $OID_{Tache}.Ressources$

4) *pour les rôles dans une composition d'une entité* :

<Désignation de l'entité>.<Nom de la fonction rôle>

5. Une expression de chemin peut s'interpréter comme une forme syntaxique d'expression d'une composition de fonctions.

Classe d'opérations	Type d'opération
Renommage	Changement des noms des types de base non prédéfinis, Renommage de types d'attributs, d'entités, d'associations ou de schéma, Changement des noms des fonctions rôles dans les associations et dans les compositions
Changement de Domaine	Modification de l'expression de définition de types de base, Changement des domaines des types d'attributs, de ceux des participants aux associations et de ceux des composants dans les compositions, Modification de la relation de sous-type, des cardinalités, des natures des participations dans une composition
Changement de structure	Extension ou restriction de listes d'attributs, Ajout ou suppression de types d'entités ou d'associations, Ajout ou suppression de composants dans une composition, Changement du constructeur d'un type de base non prédéfini.

FIG. 5.4 – *Typologie des opérations de modification de schémas*

Exemple : *Equipe.LesProgrammeurs*

5) *pour les rôles dans une association* : <EouA>.<Nom de la fonction rôle>.<EouAttr>, où

- <EouA> est une désignation d'entité ou d'association,
- <EouAttr> est une désignation d'entité ou d'attribut de l'association.

Exemples : *Tache.À_la_Charge_De.NomEq*, OID_{Equipe}*.Chargée_De.Responsable.NomP*

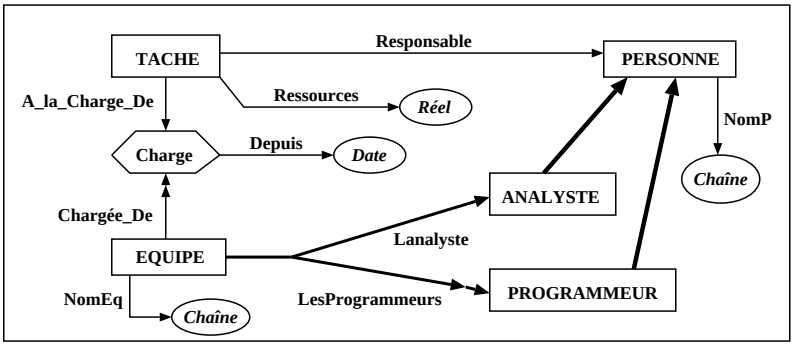

FIG. 5.5 – *Exemple de schéma*

5.3.1 Les opérations de renommage

De façon générale, le renommage de types n'est qu'une opération syntaxique sans effets sur les instances des objets existants.

- *Dans le cas d'un type de base non prédéfini*, le renommage n'a d'effet ni sur les instances ni sur la désignation des objets, car le nom d'un type de base n'est utilisé que pour définir le domaine d'un type d'attribut.

- *Dans les autres cas :* Soient a^i et a^j l'ancien et le nouveau nom. Soient également t_a^i et t_a^j leurs types respectifs dans V_i^S et V_j^S. Il est important de noter, que s'agissant d'un *renommage* de a^i, t_a^i est identique à t_a^j.

$$\mathcal{T}structure_{i-j}(a^i) = a^j$$
$$\mathcal{T}valeur_{i-j}(t_a^i) = Val(t_a^i)$$

En termes procéduraux, les opérations de renommage de types dans un schéma ont pour conséquences que le nouveau nom a^j remplace l'ancien (particulièrement dans les sous-schémas et les expressions de chemins où l'ancien nom apparaissait), les valeurs des "objets" du type t_a^i n'étant pas affectées par le changement.

5.3.2 Les opérations de changement de domaines

De façon générale, si le type $\overline{T}$ de l'ancien domaine est de "la même catégorie" que celle du type T de l'ancien domaine, c'est-à-dire que T est exprimé à l'aide du même concept de GORM que $\overline{T}$, un changement de domaine ressemblera *syntaxiquement* à un renommage, mais s'en distinguera *sémantiquement*. En effet, la représentation des valeurs de T et celle des valeurs de $\overline{T}$ n'est pas affectée par le changement de domaine. Par contre, les valeurs, elles, peuvent l'être. Dans ce cas, nous essayerons de mettre en évidence les cas *décidables* de correspondance entre les valeurs de T et celles de $\overline{T}$: il s'agit essentiellement du cas où T est sous-type de $\overline{T}$.

Néanmoins, certains changements de domaine, particulièrement ceux où $\overline{T}$ et T ne sont pas de la même catégorie (par exemple, l'ancien domaine d'un type d'attribut est un type de base et le nouveau est un type d'entité), ont des effets "profonds" souvent *indécidables* dont la prise en compte nécessite la restructuration de la représentation des valeurs du type et, quand possible, la correspondance entre les valeurs de $\overline{T}$ et celles de T.
Dans ce paragraphe, nous étudierons successivement :

1. le cas où le type du nouveau domaine et celui de l'ancien domaine sont des types de base,

2. celui où les deux domaines sont des types d'entités. Nous inclurons dans ce cas, l'étude de la modification de l'arbre d'héritage, car nous pouvons assimiler la définition d'un lien de sous-type entre deux types d'entités E_1 et E_2 à la définition d'une fonction des identifiants de E_1 dans ceux de E_2. Aussi, changer la définition d'un lien de sous-type revient à changer le domaine ou le co-domaine de la fonction, les deux étant des types d'entités. Nous verrons, ensuite,

3. le changement de la cardinalité d'un type d'attribut ou d'une participation à un type d'association ou à une composition de type d'entité,

4. le changement de la nature de la participation d'un composant dans une composition,

5. et enfin, nous terminerons par une discussion à propos de la situation où aucune relation formelle ne peut être établie entre *Dom* et $\overline{Dom}$.

D'autre part, dans chacun des deux premiers cas qui viennent d'être énumérés, nous distinguerons la situation où le type du nouveau domaine est sous-type de celui de l'ancien (que nous noterons, par abus d'écriture, $Dom \leq \overline{Dom}$) et la situation inverse où $\overline{Dom} \leq Dom$.

Mais au préalable, faisons remarquer deux points, l'un relatif à la décidabilité de la relation de sous-type ($\leq$) et l'autre concernant la correspondance des valeurs de $\overline{Dom}$ et de *Dom* dans le cadre de la relation $\leq$.

(i) *De la décidabilité de la relation* $\leq$

Lorsque les types de *Dom* et de $\overline{Dom}$ sont des types de base nous faisons la supposition que l'identification (que l'on sait *indécidable*) de la relation de sous-type qui les lie est faite. Par contre, si les types de *Dom* et de $\overline{Dom}$ sont des types d'entité, l'identification de la relation de sous-type est *décidable* car, dans un type de schéma, elle est définie en extension, i.e $Dom \leq \overline{Dom}$ ou $\overline{Dom} \leq Dom$ si le type de *Dom* et celui de $\overline{Dom}$ sont sur "la même branche" de l'arbre d'héritage (l'algorithme *Chemin*, défini dans le paragraphe 5.3.2.2, permet de vérifier qu'un chemin existe dans l'arbre d'héritage entre deux types d'entités).

(ii) *De la Correspondance entre les valeurs de* $\overline{Dom}$ *et de Dom*

Formellement, si $\overline{Dom} \leq Dom$ alors toute instance de $\overline{Dom}$ est aussi instance de *Dom*, par définition de la relation de sous-type. Cependant, si le type de *Dom* et celui de $\overline{Dom}$ sont des types d'entités, comme dans GORM les espaces des identifiants des types des entités sont distincts, les instances de $\overline{Dom}$ ne désignent pas "directement" des instances de *Dom*. Il est alors nécessaire d'établir une correspondance entre les espaces des identifiants des types de $\overline{Dom}$ et de *Dom*. De façon similaire, si $Dom \leq \overline{Dom}$, certaines instances de $\overline{Dom}$ risquent de ne plus être du type de *Dom*.

A cet effet, dans le cas des types d'entités, nous proposons dans le paragraphe 5.3.2.2, deux algorithmes, $Transitif^-$ et $Transitif^+$, qui permettent de calculer la correspondance entre les instances de $\overline{Dom}$ et celles de *Dom* dans le cas où $Dom \leq \overline{Dom}$ et celui où $\overline{Dom} \leq Dom$, respectivement.

D'un *point de vue opérationnel*, les deux remarques précédentes signifient que,

1. dans le cas où $\overline{Dom} \leq Dom$, toutes les valeurs de $\overline{Dom}$ peuvent migrer dans *Dom*
   (a) sans transformation aucune si les types des domaines sont des types de base,
   (b) par le biais de $Transitif^+$, dans le cas où les types de *Dom* et de $\overline{Dom}$ sont des types d'entités.
2. dans le cas où $Dom \leq \overline{Dom}$, certaines valeurs de $\overline{Dom}$ n'auront pas d'image dans *Dom* (elles ne sont pas du type de *Dom*) :
   (a) pour les types de base, un mécanisme de *"type-checking"* permettrait d'identifier les instances de $\overline{Dom}$ qui persistent comme instances de *Dom*,
   (b) et pour les types d'entités, $Transitif^-$ permettra d'identifier le sous-ensemble des identifiants de $\overline{Dom}$ qui ont un super-type dans *Dom* et établira la correspondance *partielle* entre les deux ensembles d'identifiants d'entités.

Examinons maintenant les différentes situations de façon plus détaillée.

5.3.2.1 Les types de *Dom* et de $\overline{Dom}$ sont des types de base (non prédéfinis)

Rappelons que la spécification d'un type de base non prédéfini est donnée par :

DefTypBase $\stackrel{def}{==}$ < Nom: *NomTypBase*,
Domaine: *ExpressionDeDéfinition*>

et qu'une *instance* est une *"chaîne"* vérifiant *ExpressionDeDéfinition*. En outre, un type de base n'a pas le statut de classes d'objets et il n'est utilisé, dans un type de schéma, que pour caractériser le domaine d'un type d'attribut. C'est-à-dire que, dans une notation simplifiée de la définition d'un type d'attribut, le nom $NomTb$ d'un type de base apparaît sous la forme $< NomTypAttribut : NomTb >$ et que toute valeur v de $NomTypAttribut$ vérifie l'expression de définition associée à $NomTb$ (ce que nous noterons par *ExprDef(v)*).

Soit $< a^i : t_a^i >$ un type d'attribut dans la version V_i^S d'un schéma S et $< a^j : t_a^j >$ sa *variante* dans V_j^S. Dans un changement de domaine d'un type de base, en fait a^i est *identique* à a^j mais l'expression de définition de t_a^i, notée $\overline{ExprDef}$ est modifiée (en *ExprDef*).

a) *Cas* $Dom \leq \overline{Dom}$

L'expression de définition $\overline{ExprDef}$ du type de $\overline{Dom}$ a été modifiée en $ExprDef$ et $ExprDef$ exprime une relation de sous-type entre t_a^j et t_a^i ($t_a^j \leq t_a^i$) : la fonction de correspondance des structures, $\mathcal{T}structure_{i-j}$, est *l'identité* et celle de correspondance des valeurs est définie par :

$\mathcal{T}valeur_{i-j}(t_a^i)$ = *Si* *ExprDef(Val*(t_a^i)*)*
Alors *Val*(t_a^i)
Sinon $\surd$
Finsi

où *Exprdef(x)* est évalué à *Vrai* si x satisfait *ExprDef*.

Du point de vue opérationnel, cette fonction signifie que dans une migration immédiate ou dans une "conversion" de valeurs entre les versions V_i^S et V_j^S du schéma, seules migrent, sans transformation, les valeurs des attributs de type t_a^i qui sont du type de *Dom*, c'est-à-dire qui vérifient ce que nous avons dénoté par *Exprdef(Val*(t_a^i)*)*. Par ailleurs, t_a^j étant sous-type de t_a^i, les opérations de ce dernier sont applicables aux instances de t_a^j par héritage.

b) *Cas* $\overline{Dom} \leq Dom$

Formellement, si la relation $\leq$ est démontrée, les valeurs de $\overline{Dom}$ sont incluses dans celles de *Dom*. Par ailleurs, la structure du domaine n'ayant pas changé, les fonctions $\mathcal{T}structure_{i-j}$ et $\mathcal{T}valeur_{i-j}$ sont des *identités*.

Du point de vue opérationnel, toutes les valeurs migrent, d'une version à l'autre du schéma, sans transformation, et les opérations du type de *Dom* leur sont applicables.

c) *Impact sur la désignation des objets*

Les domaines des types de base n'étant pas utilisés pour référencer des objets dans les méthodes ou les programmes, le changement de domaine d'un type de base en un autre type de base n'a pas d'incidence sur les expressions de chemins de désignation des objets.

5.3.2.2 Les types de Dom et de $\overline{Dom}$ sont des types d'entités

Le changement de domaine a consisté, dans ce cas, à modifier une définition $d^i \stackrel{Def}{==} < a^i : t^i >$, d'un type d'attribut ou d'une fonction rôle, en $d^j \stackrel{Def}{==} < a^j : t^j >$, où a^i est *identique* à a^j et où t_i et t_j sont des noms de types d'entités différents.

a) *Cas* $Dom \leq \overline{Dom}$

La correspondance des structures est toujours *l'identité* (au renommage du type du domaine près), vu que les domaines de valeurs de a^i et de a^j sont des identifiants d'entités. Néanmoins, la correspondance entre les espaces des identifiants de *Dom* et $\overline{Dom}$ doit être définie. Ainsi, pour chaque instance v_{a^i} de a^i dans $\overline{Dom}$, nous ferons correspondre une instance dans *Dom* dont le calcul se fonde sur la transitivité de la relation de sous-type. Intuitivement, à v_{a^i}, un identifiant du type d'entité de nom t^i_a, on fait correspondre v_{a^j}, un identifiant du type d'entité de nom t^j_a, tel que v_{a^j} "est transitivement" v_{a^i}.

La figure 5.6 montre le scénario de calcul de v_{a^j}. L'interprétation intuitive d'un lien de sous-type entre un type d'entité de nom X et un type d'entité de nom Y est qu'une entité x de type X *est une* (IS_A) *entité* y de Y. Dans notre modèle, le couple $< x, y >$ appartient à l'instance de la relation de sous-type qui lie X à Y ($< x, y > \in InstanceSousType_{X,Y}$). Ainsi, v_{a^j} est calculée par "transitivité en arrière" de la relation de sous-type, c'est-à-dire, sur le chemin $<t^i_a$, X, Y, . . . , Z, $t^j_a>$, en considérant les instances de relations de sous-types associées à chaque couple de type d'entités du chemin.

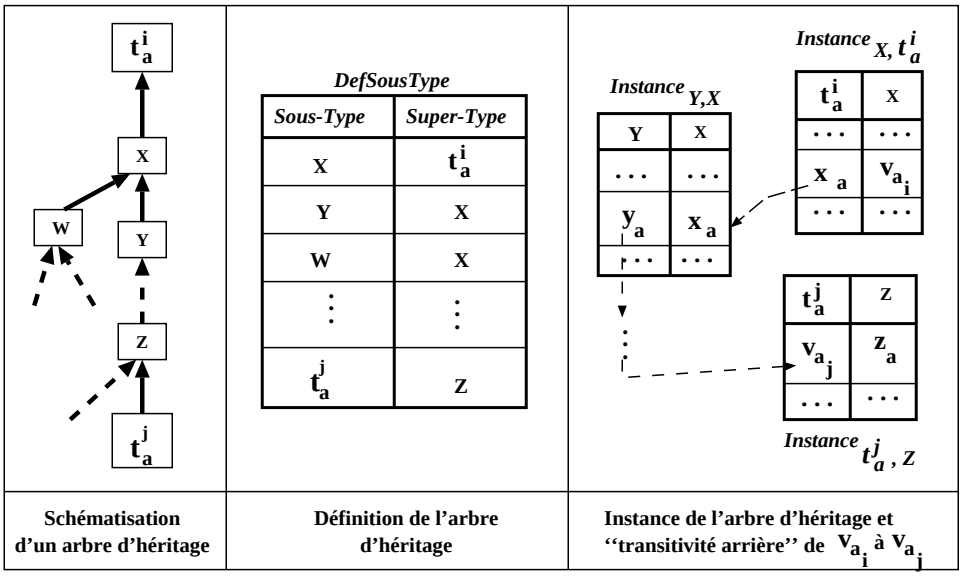

FIG. 5.6 – *Calcul de correspondances d'identifiants et relation de sous-type*

De façon plus formelle, la correspondance des valeurs entre *Dom* et $\overline{Dom}$ est définie par :

$Tvaleur_{i-j}(t^i_a) = Transitif^-(t^i_a, Val(a^i), t^j_a)$, où $Transitif^-$ est définie par l'algorithme décrit ci-dessous.

Rappelons que la relation de sous-type dans un type de schéma est définie par

DefSousType $\stackrel{def}{=\!=}$ {< Sous-Type : *NomTypEntité*,
Super-Type : *NomTypEntité* $\cup$ "Entité">}

Rappelons aussi que, pour chaque couple <Sous-Type : E_1, Super-Type : E_2> d'une définition *DefSousType*, où E_1 et E_2 sont des noms de types d'entités, on dispose d'une instance définie par un ensemble de couples d'identifiants d'entités :

$InstanceSousType_{E_1,E_2} \stackrel{def}{=\!=}$ {< Sous-Type : *OID(E_1)*,
Super-Type : *OID(E_2)*>}

Dans les algorithmes ci-dessous,

- $DefSousType_i^S$ désigne la définition de la relation de sous-type dans la version V_i^S du schéma et est considérée comme une table relationnelle,
- Π et σ dénotent la projection et la sélection relationnelles, respectivement,
- $InstanceSousType_{y.Sous-Type,y.Super-Type}$ désigne l'instance de la relation de sous-type qui lie les types d'entités de y, où y est un couple <Sous-type: E_1, Super-Type: E_2> appartenant à $DefSousType_i^S$,
- *R est une liste de couples de noms de types d'entités* décrivant un chemin de l'arbre d'héritage,
- *M contient la chaîne 'Entité'* s'il n'y a pas de chemin dans l'arbre d'héritage entre deux types d'entité ou si "l'extrémité supérieure" du chemin est la racine *'Entité'*.
- *Est_Vide*$_{Liste}(L)$ est une fonction qui rend vrai si la liste L est vide et faux sinon,
- $Vide_{Liste}()$ est une fonction qui crée une liste vide,
- $Vide_{Ensemble}()$ est une fonction qui crée un ensemble vide,
- $AjtTete_{Liste}(L,\ x)$ est une fonction qui insère l'élément x en tête de la liste L,
- $Ajt_{Ensemble}(E,\ x)$ est une fonction qui insère l'élément x dans l'ensemble E,
- $SuprQueue_{Liste}(L)$ est une fonction qui retire le dernier élément de la liste L,
- $SuprTete_{Liste}(L)$ est une fonction qui supprime le premier élément d'une liste L,
- *Premier(L)* et *Dernier(L)* sont des fonctions d'accès au premier et au dernier élément de L, respectivement.

Les arguments *Nouveau*, *Oid*, *Ancien* de la fonction *Transitif*$^-$ dénotent, respectivement, le nom du type de *Dom*, une valeur v_a^i de $\overline{Dom}$ et le nom du type de $\overline{Dom}$.

Transitif⁻(Nouveau: NomTypEntité, Oid: OID, Ancien: NomTypEntité): OID
/ Transitif⁻(Nouveau, Oid, Ancien) = X */*
DEBUT
R = Vide$_{Liste}$()
<R, M> = *Chemin(Ancien, Nouveau)*
Si M = *'Entité'*
Alors X = $\perp$
Sinon X = Correspondance^{-}(R, Oid)
Finsi
FIN

$Transitif^{-}$ utilise les fonctions :

– *Chemin(X: NomTypeEntité, Y: NomTypeEntité)* qui calcule le chemin du sous-type X au super-type Y et qui rend comme résultat le couple $< R,\ M >$:

Chemin(X, Y: NomTypEntité):<Liste(V,W: NomTypEntité), Chaîne>
/ Chemin(X, Y) =* $< R,\ M >$ **/*
DEBUT
W = $\Pi_{Super-Type}(\sigma_{(Sous-Type\ =\ X)}(DefSousType_i^S))$
Si W = *'Entité'*
Alors R = Vide$_{Liste}()$
M = *'Entité'*
Sinon *Si* W = Y
Alors R = $< X,\ Y >$
Sinon R = AjtTete$_{Liste}(Chemin(W,\ Y), < X,\ W >)$
Finsi
Finsi
FIN

– et $Correspondance^{-}$*(R: chemin, Oid: OID)* qui calcule la correspondance "en arrière" entre l'identifiant d'une *entité* de $\overline{Dom}$, identifiée par Oid, est celui d'une *entité* du nouveau type du domaine, si elle existe.

En d'autres termes, Oid est l'identifiant d'un type d'entité TE, "extrémité supérieure" du chemin, et le résultat est un identifiant X d'un type d'entité TX, "extrémité inférieure" du chemin, transitivement sous-type de TE, i.e. *l'entité'* identifiée par X "est transitivement" *l'entité'* identifiée par Oid. Dans la formulation de l'algorithme, $Y2SuperY$ dénote l'extension d'un lien de sous-type y (i.e. $InstanceSousType_{y.Sous-Type,\ y.Super-Type}$).

Correspondance⁻(R: chemin, Oid: OID): OID
/* *Correspondance⁻(R, oid) = X* */
DEBUT
Si $Est_Vide_{Liste}(R)$
Alors X = Oid
Sinon y = $Dernier_{Liste}$(R)
Y2SuperY = InstanceSousType$_{y.Sous-Type,\ y.Super-Type}$
w = $\Pi_{Sous-Type}(\sigma_{(Super-Type\ =\ Oid)}(Y2SuperY))$
Si w = $\perp$
Alors X = $\perp$ /* Pas de correspondance */
Sinon X = Correspondance$^{-}(SuprQueue_{Liste}(R),\ w)$
Finsi
Finsi
FIN

Ainsi, *du point de vue opérationnel*, $Transitif^{-}$ permet de déterminer si la relation $\leq$ est vraie entre *Dom* et $\overline{Dom}$ et si oui, c'est-à-dire si

(<R, M> = *Chemin*$(\overline{Dom},\ Dom)) \wedge \neg\ Est_Vide_{Liste}(R)$),

de calculer la correspondance entre des valeurs des deux domaines (grâce à *Correspondance⁻*). Ces valeurs peuvent alors être utilisées, tant dans une politique de migration immédiate que dans une stratégie avec versions de schéma et d'objets, pour modifier les références aux objets de $\overline{Dom}$ dans les attributs ou dans les rôles d'associations ou de compositions qui avaient le type de $\overline{Dom}$ comme domaine dans la version V_i^S d'un schéma S.

Voyons maintenant le cas inverse où les types de $\overline{Dom}$ et de *Dom* sont des types d'entités et où $\overline{Dom}\ \leq\ Dom$.

b) *Cas* $\overline{Dom}\ \leq\ Dom$

Ce cas est similaire au cas précédent, à la différence que le type de *Dom* est un "ancêtre" de celui de $\overline{Dom}$ dans l'arbre d'héritage. Aussi, la structure des valeurs n'est pas modifiée (encore une fois, *Dom* et $\overline{Dom}$ sont des ensembles d'identifiants d'entités) : $\mathcal{T}structure_{i-j}$ est, par conséquent, *l'identité*. Par contre, une correspondance entre les valeurs doit être établie, de façon analogue au cas précédent, par "transitivité en avant" sur l'arbre d'héritage, du type de $\overline{Dom}$ jusqu'à celui de *Dom*. Sur l'exemple de la figure 5.6, les "rôles" de t_a^i et de t_a^j sont inversés dans l'arbre d'héritage et il s'agit, cette fois-ci, de parcourir le chemin $<t_a^j$, Z, ... , X, $t_a^i>$ en considérant les instances de relations de sous-types associées à chaque couple de type d'entités du chemin. Soit, plus formellement,

$\mathcal{T}valeur_{i-j}^{t_a^i} = Transitif^{+}(t_a^i,\ Val(a^i),\ t_a^j)$, où $Transitif^{+}$ est définie par :

```
Transitif+(Nouveau: NomTypEntité, Oid: OID, Ancien: NomTypEntité): OID
    /* Transitif+(Nouveau, Oid, Ancien) = X */
DEBUT
    R = Vide_Liste()
    <R, M> = Chemin(Ancien, Nouveau)
    Si      M = 'Entité'
    Alors   X = 'Racine'
    Sinon   X = Correspondance+(R, Oid)
    Finsi
FIN
```

Correspondance$^+$ est "symétrique" à *Correspondance*$^-$: il calcule une correspondance entre des identifiants d'entités "de bas en haut" sur un chemin de l'arbre d'héritage. Comme dans *Correspondance*$^-$, $Y2SuperY$ dénotera $InstanceSousType_{y.Sous-Type,\ y.Super-Type}$, l'extension d'un lien de sous-type y.

```
Correspondance+(R: chemin, Oid: OID): OID
    /* Correspondance+(R, oid) = X */
DEBUT
    Si Est_Vide_Liste(R)
    Alors X = Oid
    Sinon   y = Premier(R)
            Y2SuperY = InstanceSousType_{y.Sous-Type, y.Super-Type}
            W = Π_{Super-Type}(σ_{(Sous-Type = Oid)}(Y2SuperY))
            X = Correspondance+(SuprTete_Liste(R), W)
    Finsi
FIN
```

L'interprétation opérationnelle de $Transitif^+$ est similaire à celle de $Transitif^-$. Néanmoins, on soulignera que la correspondance entre $\overline{Dom}$ et *Dom* est *totale* dans $Transitif^+$ alors qu'elle est *partielle* dans $Transitif^-$, par définition de la relation de sous-type.

On notera, pour clore ce paragraphe, que le changement d'un domaine "type d'entité" en un domaine "type d'entité" a des incidences sur la désignation des objets dans les méthodes ou les programmes, comme nous allons le voir immédiatement.

c) *Changement de domaines et désignation d'objets*

Considérons l'exemple de la figure 5.7 qui montre, dans sa partie gauche, le cas où *Personne*, le domaine du type d'attribut *Dirige* est changé en *Thésard* qui est sous-type de *Personne* (cas $Dom \leq \overline{Dom}$). Une expression de désignation d'objet de la forme "Directeur_Thèse.Dirige" est toujours valide : elle désignera un *Thésard* au lieu d'une *Personne* (du point de vue des instances, $Transitif^+$ est censé avoir été mis en œuvre pour actualiser les valeurs de l'attribut *Dirige*). De même, une expression de la forme *"Directeur_Thèse.Dirige.NomP"* reste valide, *NomP* étant hérité de *Personne* "via" *Etudiant*, dans la nouvelle version de schéma (partie droite de la figure 5.7).

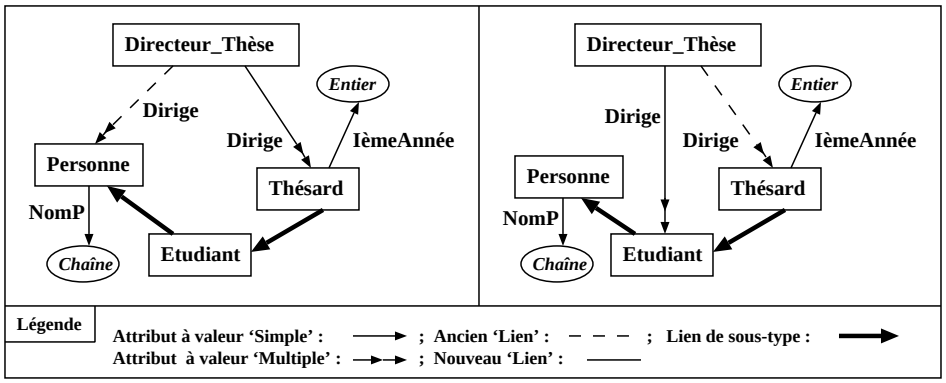

FIG. 5.7 – *Changement de domaine et relation de sous-type (1)*

Dans le cas contraire ($\overline{Dom} \leq Dom$), illustré par la partie droite de la figure 5.7, les références à des attributs de *Thésard*, comme dans l'expression *"Directeur_Thèse.Dirige.IèmeAnnée"*, deviennent invalides. *Opérationnellement*, cela signifie que lorsque $Dom \leq \overline{Dom}$, la ré-écriture des invocations de méthodes n'est pas nécessaire, alors qu'elle peut l'être dans le cas où $\overline{Dom} \leq Dom$.

On notera, pour terminer, que le même discours s'applique dans le cas où le domaine concerné est celui d'une fonction rôle dans une association ou dans une composition d'un type d'entité : nous laissons le soin au lecteur de s'en convaincre au vu des exemples des figures 5.8 et 5.9.

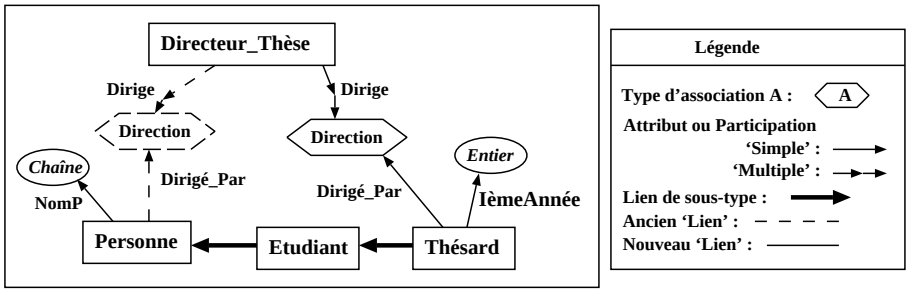

FIG. 5.8 – *Changement de domaine et relation de sous-type (2)*

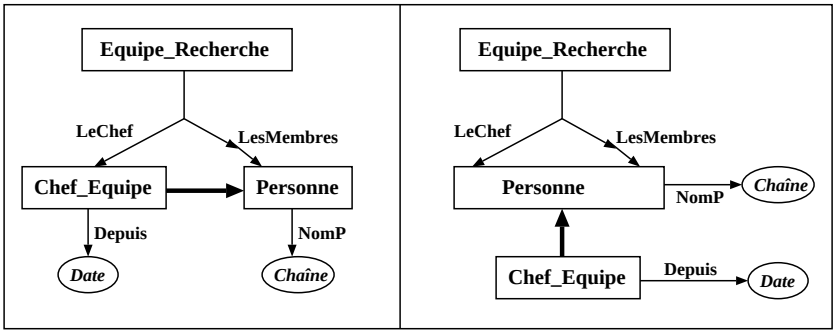

FIG. 5.9 – *Changement de domaine et relation de sous-type (3)*

5.3.2.3 Cas d'une modification de la relation de sous-type

La modification de la définition de la relation de sous-type peut être opérée par ajout ou par suppression d'un lien de sous-type entre deux types d'entités, *en préservant les invariants de schéma* (connexité, acyclicité et héritage simple). De ce fait, la suppression d'un lien de sous-type doit toujours être suivie de l'ajout d'un lien.

En effet, soit l^i = <Sous-Type: E_1^i, Super-Type: E_2^i>, le lien de sous-type supprimé dans la version V_i^S d'un schéma S, et soit $Super^{E_2^i}$, un nom de type d'entité, ou la constante *'Entité'*, le super-type de E_2^i.

Après la suppression de l^i, E_2^i est toujours lié à son ancien super-type $Super^{E_2^i}$. Par contre, E_1^i n'a plus de super-type : dans la version V_j^S du schéma, E_1^i, que nous dénommerons désormais E_1^j pour rester conforme aux notations adoptées, doit être lié à un type d'entité ou à la racine *'Entité'* pour préserver les invariants de schéma.

a) Cas où E_1^j est lié à *'Entité'* (Cas (i) de la Figure 5.10)

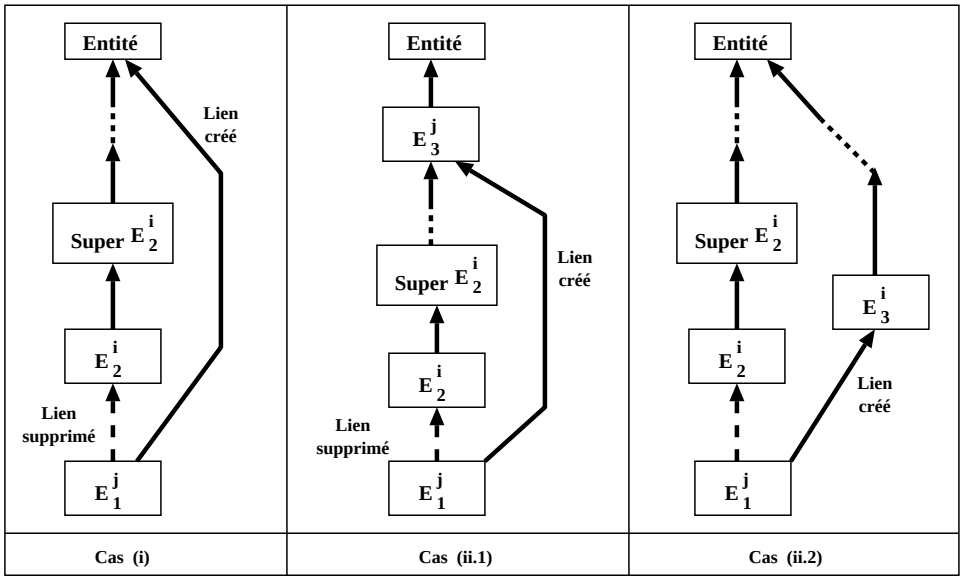

FIG. 5.10 – *Modification de la relation de sous-type*

Le super-type de toute *entité* de type E_1^j est la constante *'Racine'* de type *'Entité'* et l'instance du lien de sous-type est défini, dans ce cas par :

$$InstanceSousType_{E_1^j,\text{'}Entite\text{'}} = \{<\text{Sous-Type: OID}(e_1^i), \text{Super-type: 'Racine'}>\},$$

pour toute *entité* e_1^i de E_1^i.

Formellement, ceci nous amène à définir la correspondance structurelle entre les versions V_i^S et V_j^S de S comme une modification de le définition de la relation de sous-type et la correspondance des valeurs comme une correspondance entre $InstanceSousType_{E_1^i,E_2^i}$ et $InstanceSousType_{E_1^i,\text{'}Entite\text{'}}$. Soit

$$\mathcal{T}structure_{i-j}(l^i) = <\textit{Sous-Type: } l_i\textit{.Sous-type, Super-Type: 'Entite'}>$$

$$\mathcal{T}valeur_{i-j}(l^i) = \{<\textit{Sous-Type: Val}(l_i\textit{.Sous-type), Super-Type: 'Racine'}>\}$$

b) Cas où E_1^j est lié à un type d'entité autre que *'Entité'* dans la version V_j^S de schéma

Ce cas se ramène parfois à celui du paragraphe précédent où $Dom \leq \overline{Dom}$. En effet,

1. si E_3^j est le nouveau super-type de E_1^j et que E_1^j et E_3^j sont sur la même branche de l'arbre d'héritage dans la version V_i du schéma (Cas (ii.1) de la Figure 5.10), alors l'instance $InstanceSousType_{E_1^j,E_3^j}$ de la relation de sous-type, dans la version V_j^S de schéma, peut être calculée par l'algorithme $Transitif^+$.

2. Dans le cas contraire (Cas (ii.2) de la Figure 5.10), c'est-à-dire si E_1 et E_3 ne sont pas sur la même branche de l'arbre d'héritage, les fonctions de correspondance sont indéfinies car il n'existe pas de relation formelle entre E_1^j et E_3^j.

5.3.2.4 Cas d'une modification de cardinalité

Rappelons qu'une cardinalité peut être associée à un type d'attribut, au rôle d'un participant à un type d'association ou au rôle d'un composant dans une composition d'un type d'entité. Nous avons déjà eu l'occasion de dire que formellement, la modification d'un cardinalité revenait à changer le profil de la fonction attribut ou de la fonction rôle. En d'autres termes,

- si $d_i \stackrel{Def}{==} < a^i : t^i >$ est la définition d'une telle fonction dans une version V_i^S d'un schéma S,
- que $v_i \stackrel{Def}{==} < a^i : v^i >$ dénote une instance de d_i,
- et que $d_j \stackrel{Def}{==} < a^i : t^j >$ est la variante de d_i dans la version V_j^S de S,

alors si l'expression de définition de t^i est de la forme t_k (Cardinalité *'Simple'*), celle de t^j est de la forme *Ensemble(t_k)* (Cardinalité *'Multiple'*) (et réciproquement). *Structurellement*, cela signifie que la représentation des instances des objets "de type" d_i doit être modifiée pour pouvoir "accueillir" des ensembles de valeurs de t_k, au lieu d'une seule valeur (et réciproquement) : ce que nous dénotons formellement, dans le cas où une cardinalité *'Simple'* est modifiée en une cardinalité *'Multiple'*, par :

$$\mathcal{T}structure_{i-j}(d_i) = \{t^i\}.$$

Concernant les instances, pour chaque instance $v_i \stackrel{Def}{==} < a^i : v^i >$ de d_i, nous avons :

$$\mathcal{T}valeur_{i-j}(v_i) = \{Val(v^i)\}.$$

Dans le cas du passage d'une cardinalité multiple à une cardinalité simple, nous avons :

$$\mathcal{T}structure_{i-j}(d_i) = t^i \text{ et}$$
$$\mathcal{T}valeur_{i-j}(v_i) = \omega.$$

Du point de vue *opérationnel*, dans le premier cas, on peut considérer qu'une solution *"raisonnable"*, lors de la migration ou lors de la correspondance entre les instances de la version V_i^S et V_j^S de S, est celle exprimée par $\mathcal{T}valeur_{i-j}$ qui permet de constituer un ensemble contenant la seule valeur existante. Dans le cas inverse, nous considérons que la nouvelle valeur est indéfinie. Enfin, il est clair que les opérations ensemblistes qui s'appliquaient à la valeur d'un type d'attribut de cardinalité *'Multiple'* ne sont plus autorisées dans le cas du passage à une cardinalité *'Simple'*.

5.3.2.5 Cas d'une modification de la nature dans une composition

Rappelons d'abord la définition d'un type d'entité composé (*DefTypEntitéComposé*) et celle d'un composition (*DefComposition*).

DefTypEntitéComposé $\stackrel{def}{=}$ <Nom : *NomTypEntité*,
NomSuperType : *NomTypEntité* $\cup$ *'Entité'*,
LesAttributs : [*DefTypAttribut*],
Composition : *DefComposition*>

DefComposition $\stackrel{def}{=}$ [<Rôle: Chaîne,
NomComposant: *NomTypEntité*
Cardinalité: {*'Simple', 'Multiple'*}
Dépendance-Partage: {*Forte, Faible*} x {*Partagé, Exclusive*}>]

Soit TE une définition de type d'entité composé et E un de ses composants, c'est-à-dire que E apparaît dans la définition de $TE.Composition$. Comme nous l'avions déjà fait remarquer, le seul changement de la nature d'une composition qui peut avoir des incidences sur les instances des objets est celui où une participation 'Partagé' devient 'Exclusive'. En effet, dans ce cas, toutes les références à une *entité* e_c, dans les *entités* dans les types desquels E est un composant, deviennent invalides. Désormais, seule une référence à e_c devient autorisée. Par certains aspects, ce changement de la nature de la composition est analogue au changement de cardinalité, comme nous allons le voir plus formellement. Supposons que *Est_Partagé_Par* est une fonction qui à un nom de type d'entité E, associe les noms des types des entités dont E est un composant,

Est_Partagé_Par: NomTypEntite $\longrightarrow NomTypEntite^n$

Est_Partagé_Par(E) = $\{E_1,\ E_2,\ \cdots,\ E_n\}$, tel que

$\forall E_k,\ 1 \leq k \leq n,\ \exists C\ Est_Dans_{liste}(E_k.Composition,\ C) \wedge C.NomComposant = E$,
$Est_Dans_{liste}(L,\ x)$ étant un prédicat s'évaluant à vrai si x est dans une liste L.

Dans une participation de nature exclusive, $n = 1$ alors que $n \geq 1$ dans une participation de nature partagée. De ce fait,

- le changement d'une nature *'Exclusive '* en une nature *'Partagé'* signifie simplement que désormais sont autorisées des références multiples à une instance du type d'entité concerné. *En termes procéduraux*, cela signifie que les mécanismes support de GORM ne contrôleront plus l'unicité des références.

- *Dans le cas contraire*, comment identifier toutes les *entités* e_k^l, instances des E_k "partageant" E, qui référencent une *entité* e de E et comment n'en retenir qu'une seule, celle qui désormais "détient l'exclusivité" de la référence à *l'entité* e? *Du point de vue opérationnel*, l'identification des *entités* e_k^l est possible en associant à chaque couple de types d'entités $< E,\ E_k >,\ E_k \in$ *Est_Partagé_Par(E)*, une relation *Est_Partagé_Par*$(E,\ E_k)$. *Du point de vue opérationnel*, cela veut dire que des références inverses des liens de composition doivent être gérées. L'identification de *l'entité* détentrice de la référence exclusive reste néanmoins *indécidable*.

Les opérations de changement de domaines que nous avons étudiées jusqu'à maintenant supposaient *peu ou prou* qu'une relation formelle liait l'ancien et le nouveau domaine. Nous avons essayé de montrer que, dans ce cas, les situations étaient décidables et qu'une sémantique statique et une sémantique procédurale pouvaient être associées aux modifications. Malheureusement, ce n'est plus le cas lorsqu'aucune relation formelle ne peut être établie entre l'ancien et le nouveau domaine.

5.3.2.6 Cas où il n'y a pas de relation formelle entre *Dom* et $\overline{Dom}$

Illustrons ce cas par quelques exemples :

- <*Date: Chaîne*> est une *variante* de *Date:* <jj: 1..31, mm: 1..12, aa: 0..99>
- <*Note_Examen: 0..20*> est une *variante* de <*Note_Examen:* {A, B, C, D}>
- <*NomPersonne: Chaîne*> devient <*NomPersonne: Personne*>, *Personne* étant un type d'entité,
- <*Ressources: Réel*> devient *Ressources*: <Humaines: Entier, Financières: Réel> et, dans la nouvelle version de schéma, *Ressources.Financières* correspond à *Ressources* dans l'ancienne . . .

De façon générale, ce type de changement requiert la définition explicite des correspondances entre les variantes du type qui a subi la modification de domaine. Comme pour les cas précédents, ces correspondances concernent les structures aussi bien que les valeurs. Dans le chapitre 6, pour contribuer à la prise en compte de ce type de modification, nous proposons un ensemble de règles de ré-écriture de structures, un mécanisme de spécification de fonctions de correspondance et des opérateurs de transformation des valeurs.

A titre d'exemple, la prise en compte du changement de <*Ressources: Réel*> en *Ressources:* <*Humaines: Entier, Financières: Réel*>

1. utilisera les règles de ré-écriture qui permettent :
   - de transformer un type atomique T en un produit cartésien,
     (<*Ressources: Réel*> $\longrightarrow y :$ <*Ressources: Réel*>)
   - d'étendre un produit cartésien par une projection,
     (*y:* <*Ressources: Réel*> $\longrightarrow$ *y:* <*Humaines: Entier, Ressources: Réel*>)
2. utilisera des opérateurs de renommage,
   (*Rename(y, 'Ressources')*; *Rename('Ressources', 'Financières')*)
3. établira la correspondance entre la nouvelle et l'ancienne structure,
   (*Ressources.Humaines* = $\surd$ et *Ressources.Financières* = $\overline{Ressources}$)
4. et réalisera la transformation des valeurs.

La correspondance entre les variantes du type *Date* nécessite des transformations ad hoc (extraction, conversion de sous-chaînes).

Le processus interactif également proposé dans le chapitre 6, permet d'aider la découverte de ce type de séquence de transformations. Nous en parlerons de façon plus détaillée en temps utile. Passons maintenant à la troisième catégorie d'opérations de modification de schémas.

5.3.3 Les opérations de changement de structure

Nous classons dans cette catégorie d'opérations les opérations

- de changement du constructeur d'un type dans un type de base non prédéfini, comme l'exemple de la *Date* dans le paragraphe précédent,
- d'extension ou de restriction de la liste des attributs associée à un type d'entité ou à un type d'association,
- d'ajout ou de suppression d'un type d'entité dans la définition de la composition d'un autre type d'entité,
- d'ajout et de suppression d'un type d'association,
- et, enfin, d'ajout et de suppression d'un type d'entité du schéma.

Le premier type d'opération ne sera pas examiné ici car il rejoint le cas discuté précédemment, à savoir celui où il n'existe pas de relation formelle entre le nouveau et l'ancien type. Les autres types d'opérations sont brièvement passés en revue. Nous terminerons le paragraphe par la discussion de leurs effets sur la désignation des objets dans les méthodes ou les programmes.

5.3.3.1 Extension/restriction de listes d'attributs

Soit $l_i : \ < a_1^i : t_1^i, \cdots, a_n^i : t_n^i >$ la liste d'attribut dans la version V_i^S d'un schéma S, et soit $l_j :< a_1^j : t_1^j, \cdots, a_n^j : t_n^j, a_{n+1}^j : t_{n+1}^j >$ sa variante dans la version V_j^S de S, obtenue par *adjonction* de $a_{n+1}^j : t_{n+1}^j$ à l_i.

Formellement, l_j est *sous-type* de l_i [6] et les instances de l_i sont aussi instances de l_j. De ce fait, les fonctions de correspondance entre les versions de schéma s'énoncent comme suit :

• $\mathcal{T}structure_{i-j}(l_i) \ = \ < \mathcal{T}structure_{i-j}(a_1^i : t_1^i), \cdots, \mathcal{T}structure_{i-j}(a_n^i : t_n^i), a_{n+1}^j : t_{n+1}^j >$, avec

$\mathcal{T}structure_{i-j}(a_k^i : t_k^i) \ = \ < a_k^i : t_k^i >, \forall k \ 1 \leq k \leq n$ et

• $\mathcal{T}valeur_{i-j}(l_i) \ = \ < \mathcal{T}valeur_{i-j}(t_1^i), \ \cdots, \ \mathcal{T}valeur_{i-j}(t_n^i), \ \surd >$, avec

$\mathcal{T}valeur_{i-j}(t_k^i) \ = \ Val(t_k^i), \forall k \ 1 \ \leq \ k \ \leq \ n.$

L'interprétation procédurale que l'on peut donner à ces fonctions est que, structurellement, la liste des valeurs est étendue par un élément pour représenter l'attribut a_{n+1}^j ajouté à la liste, et que lors de la conversion initiale de la valeur de l_i en une valeur de l_j, la valeur a_{n+1}^j est indéfinie, celles des autres attributs étant identiques aux valeurs des attributs dans la version i du schéma.

On notera que la correspondance entre la version V_j^S et la version V_i^S du schéma exprime une *suppression d'attribut* dans la liste définie formellement par :

• $\mathcal{T}structure_{j-i}(l_j) \ = \ < \mathcal{T}structure_{j-i}(a_1^j : t_1^j), \cdots, \mathcal{T}structure_{j-i}(a_{n+1}^j : t_{n+1}^j) >$, avec

6. $< b_1 : u_1, \ \cdots, \ b_n : u_n, \ b_{n+1} : u_{n+1}, \ \cdots, \ b_m : u_m > \ \leq \ < a_1 : t_1, \ \cdots \ a_n : t_n >$ si et seulement si $u_i \ \leq \ t_i, \forall \ i, 1 \leq i \leq n$.

$\mathcal{T}structure_{j-i}(a_k^j : t_k^j) = < a_k^j : t_k^j >, \forall k\ 1 \leq k \leq n$, et
$\mathcal{T}structure_{j-i}(a_{n+1}^j : t_{n+1}^j) = \perp$

• $\mathcal{T}valeur_{j-i}(l_j) = < \mathcal{T}valeur_{j-i}(t_1^j), \cdots, \mathcal{T}valeur_{j-i}(t_n^j) >$, avec

$\mathcal{T}valeur_{j-i}(t_k^j) = Val(t_k^j),\ \forall k\ 1 \leq k \leq n$, et $\mathcal{T}valeur_{j-i}(t_{n+1}^j) = \omega$.

Opérationnellement, cela signifie que a_{n+1} n'est pas "visible" dans V_i^S et qu'il ne lui est pas associé de fonction de correspondance de valeurs.

5.3.3.2 Ajout/Suppression de types d'entités dans une composition

L'ajout ou la suppression d'un type d'entité dans la définition d'une composition présente une analogie très forte avec l'ajout ou la suppression d'un type d'attribut, aussi nous n'y reviendrons pas.

Néanmoins, il convient de remarquer que

- suite à la suppression d'un type d'entité $T_{Composant}$ de la composition d'un type d'entité composé $T_{Compose}$,
- si la nature de la participation de $T_{Composant}$ exprimait une dépendance *'Forte'* vis-à-vis de $T_{Compose}$,

le type d'entité $T_{Composant}$ est également supprimé comme sont récursivement supprimés les types des entités qui en dépendraient *fortement* (voir le paragraphe 4, plus bas, consacré à la suppression de types d'entités).

Enfin, on notera que les autres natures de la participation du type $T_{Composant}$ n'occasionnent pas de suppressions de types d'entités par effets de bord, de par la sémantique qui leur est associée.

5.3.3.3 Ajout/Suppression de définitions de types d'associations

a) *Dans le cas de l'ajout*, l'instance du type d'association est vide lors de la migration initiale de la version V_i^S du schéma à la version V_j^S. Par ailleurs, dans la stratégie de versions multiples, le type d'entité ajouté n'est pas "visible" dans la version V_i^S.

Soit de façon plus formelle :

$\mathcal{T}structure_{i-j}(R^i) = R^j$

$\mathcal{T}valeur_{i-j}(R^i) = \surd$

b) *Dans le cas de la suppression*, l'instance du type supprimé disparaît :

- *physiquement*, dans le cas d'une stratégie sans version,
- *logiquement*, dans une stratégie avec versions.

Soit,

$\mathcal{T}structure_{i-j}(R^i) = \perp$

$\mathcal{T}valeur_{i-j}(R^i) = \omega$

Ce type d'opérations a évidemment des effets de bord

1. sur les chemins de désignation d'objets,
2. et sur les types d'attributs du type d'association supprimé, c'est-à-dire l'ensemble :

$$\{\text{ta} \mid \exists x \quad x \in DefTypAssociation_i^S \wedge Est_Dans_{Liste}(x.Attributs, a) \\ \wedge\, a.Nom = ta\}$$

où $DefTypAssociation_i^S$ dénote l'ensemble des définitions des types d'associations de la version i d'un schéma S.

Les types d'attributs du type d'association supprimé sont pris en compte par l'opération *ChgTypAssociation.SuprTypAttribut*. Quant aux désignations d'objets, deviennent invalides, dans la nouvelle version de schéma, toutes celles qui contiennent un nom de type d'attribut ou un nom de fonction *rôle* du type d'association supprimé.

5.3.3.4 Ajout/Suppression de définitions de types d'entités

Nous ferons, tout d'abord, remarquer que ces opérations doivent préserver les invariants de schéma alors que les opérations d'ajout et de suppression de types d'associations ne risquaient pas de les compromettre (à l'unicité des noms près). Ainsi, *l'ajout* de la définition d'un type d'entité TE, pour être effective, devra être accompagnée de la création d'un lien de sous-type entre TE et la racine *'Entité'*, ou entre TE et un type d'entité existant : la composition de l'ajout de TE et de la modification de la relation de sous-type sera considérée comme une *transaction de modification de schéma*.

La suppression d'un type d'entité TE, quant à elle, a des conséquences sur tous les types du schéma où le type d'entité supprimé est référencé et, plus précisément sur :

– l'ensemble $ATTR_{TE}$ des types d'attributs ayant TE comme domaine, c'est-à-dire l'ensemble des définitions des types d'attributs défini par :

$$ATTR_{TE} \stackrel{Def}{==} \{A : DefTypAttribut \mid A.Domaine = TE.Nom\}$$

– l'ensemble $SousType_{TE}$ des liens de sous-type dans lesquels TE est super-type ou sous-type,

$$SousType_{TE} \stackrel{Def}{==} \{\, y \mid y = \text{<Sous-Type: X, Super-Type: Y>} \\ \wedge\, y \in DefSousType_i^S \wedge X = TE.Nom \vee Y = TE.Nom\}$$

– l'ensemble $Compose_{TE}$ des types d'entités dans les définitions desquelles TE est le domaine d'une fonction rôle dans une composition,

$$Compose_{TE} \stackrel{Def}{==} \{\text{T: DefTypEntité} \mid \exists\, C\ Est_Dans_{Liste}(T.composition, C) \\ \wedge\quad C.NomComposant = TE.Nom\}$$

– et l'ensemble $Assoc_{TE}$ des types d'associations où TE est le domaine d'une fonction rôle, i.e.

$$Assoc_{TE} \stackrel{Def}{==} \{R : DefTypAssociation \mid \\ R.Participant_1.NomParticipant_1 = TE.Nom \\ \vee\ R.Participant_2.NomParticipant_2 = TE.Nom\}$$

En outre, de façon similaire à l'ajout de types d'entités, la suppression et ses effets de bord sont considérés comme une *transaction de modification de schéma*.

a) Ajout d'un type d'entité à un type de schéma

Du point de vue opérationnel, cette opération ressemble à celle de l'ajout d'un type d'attribut. L'instance du type d'entité TE ajouté au schéma est vide dans la migration initiale de la version V_i^S à la version V_j^S. Dans le cas où des versions d'objets et de schémas sont maintenues, TE n'est pas "visible" dans la version V_i^S.

Cependant, le "point" d'insertion de TE dans l'arbre d'héritage mérite que nous y consacrions quelques lignes. L'insertion de TE, comme une feuille ou comme un sous-type de la racine de l'arbre, ne compromet pas la cohérence de l'instance de la relation de sous-type. Ce n'est plus le cas si TE est inséré comme un nœud "interne" de l'arbre car la valeur du chemin allant des sous-types de TE au super-type de celui-ci n'est *généralement* pas dérivable de l'instance de la relation de sous-type : l'instance de TE étant vide, il n'y a pas moyen d'établir formellement la correspondance entre les *entités* sous-types de TE et des *entités* de TE, ni entre celles-ci et des *entités* du super-type de TE. Le seul cas où, à notre avis, cette valeur de chemin est dérivable, est celui où une *définition intentionnelle* de la participation au nouveau lien de sous-type, entre TE et son super-type TX, peut être formulée. En effet, si $P_{TE-TX}(\vec{X})$ est une formule construite à l'aide des symboles $\vec{X}$ de la définition du type TX, TX étant le super-type de TE, alors :

– l'évaluation de $P_{TE-TX}(\vec{X})$ sur les *entités* de TX permettraient d'identifier celles qui sont des instances de TE : la valeur de la relation $InstanceSousType_{TE,TX}$ qui lie TE et TX peut alors être calculée en créant un identifiant de TE par *entité* satisfaisant $P_{TE-TX}(\vec{X})$, comme l'exprime l'algorithme suivant où $InstanceSousType_{TE.Nom,TX.Nom}$ est appelé *E2X*.

```
DEBUT
    E2X = Vide_Ensemble()
    Pour tout X ∈ OID(TX.Nom)
    Faire
        Si      P_TE-TX(Val(X))
        Alors   E = Mk_TE.Nom(< IdentifiantSuperType : X >)
                E2X = Ajt_Ensemble(E2X, < E, X >)
        Sinon   rien
        Finsi
    FinPour
FIN
```

– les valeurs des relations $InstanceSousType_{TY_k,TX}$, où TY_k sont les sous-types de TX, désormais sous-types de TE (voir l'illustration graphique de la figure 5.11), serviront alors à calculer les instances des relations $InstanceSousType_{TY_k,TE}$ selon l'algorithme qui suit où, pour en alléger l'expression,

  – Y_k2X dénotera $InstanceSousType_{TY_k,TX.Nom}$,
  – $E2X$ dénotera $InstanceSousType_{TE.Nom,TX.Nom}$,
  – Y_k2E dénotera $InstanceSousType_{TY_k,TE.Nom}$,

– et $\bowtie_{(C)}\ (R_1,\ R_2)$ dénotera la jointure relationnelle, (C) étant la condition de jointure.

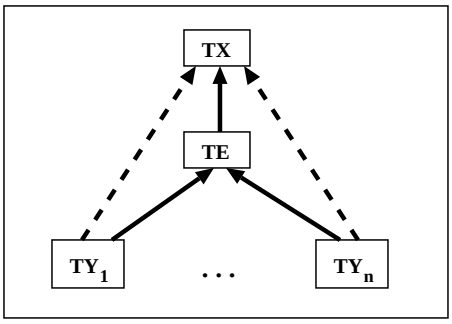

FIG. 5.11 – *Ajout de type d'entité et relation de sous-type*

DEBUT
Pour tout $TY_k\ \in\ \Pi_{Sous-Type}(\sigma_{(Super-Type\ =\ TX.Nom)})(DefSousType)$
Faire
$TY_k2TE\ =\ \Pi_{(Y_k2X.Sous-Type,\ E2X.Sous-Type)}(\bowtie_{(C)}\ (Y_k2X,\ E2X))$
où (C) est la condition : *Y_k2X.Super-Type = E2X.Super-Type*
FinPour
FIN

Le coût prohibitif de cette restructuration (parcours des instances de TX, créations d'instances "relais" de type TE, création d'une nouvelle extension de relation de sous-type pour chacun des sous-types de TX) explique probablement, pourquoi ce type d'opération est prohibé dans certains "systèmes du commerce".

b) Suppression d'un type d'entité dans un type de schéma

Voyons successivement l'impact de ce type d'opération sur les définitions des types d'un schéma puis sur les instances des objets. Soit E_s le nom du type d'entité supprimé :

- *Incidences sur le schéma* : supprimer la définition du type d'entité et propager sur les définitions :

1) *de la relation de sous-type* : cela revient à supprimer le nœud E_s du graphe représentant la relation, c'est-à-dire à supprimer les liens de sous-type de l'ensemble $SousType_{TE}$ défini plus haut :

– entre E_s et ses sous-types
– et celui entre E_s et son super-type.

Les sous-types éventuels de E_s peuvent "raisonnablement" devenir des sous-types du super-type de E_s, comme schématisé sur la figure 5.12.

2) *des types d'associations* : celles ayant comme participant le type d'entité supprimé devraient disparaître de la définition du schéma (c'est-à-dire les instances de tout $R\ \in\ Assoc_{TE}$).

3) *des compositions* : la suppression de E_s peut invalider toutes les définitions des compositions dans lesquels E_s est utilisé comme nom de composant (i.e. celles contenues dans $Compose_{TE}$).

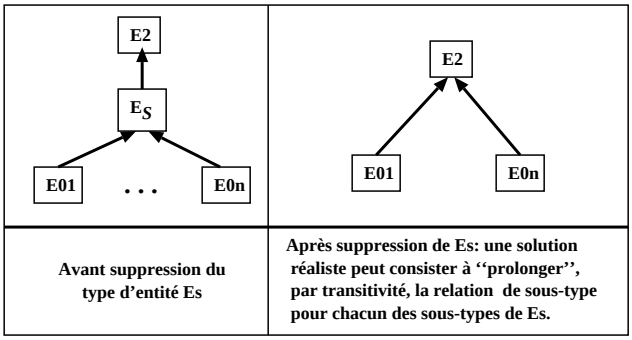

FIG. 5.12 – *Suppression de type d'entité et relation de sous-type*

4) *des attributs* : tous les attributs de $ATTR_{TE}$ dont E_s est le domaine doivent être soit redéfinis soit supprimés.

- *Incidences sur les données* :

1) les *entités* du type supprimé disparaissent : l'espace des identifiants et celui des instances sont détruits.

2) les *instances des types d'association* auxquelles participait le type d'entité supprimé disparaissent également (voir la suppression d'un type d'association dans un type de schéma),

3) la restructuration de la *relation de sous-type* dépend de la position du type d'entité dans le graphe de la relation. La nouvelle instance de celui-ci peut parfois être calculée, pour chaque sous-type E_s^k de E_S, le type d'entité supprimé, par 'transitivité en avant" sur le chemin allant de E_s^k à son nouveau super-type, si ce chemin existe (voir l'exemple de la figure 5.12),

4) les instances des types d'entité utilisant le type d'entité supprimé dans la définition d'une *composition* ou comme *domaine d'un attribut* doivent être restructurées conformément à l'analyse qui a été faite à propos des opérations de suppression de composants et d'attributs, respectivement.

La propagation des effets de bord de la suppression d'un type d'entité se matérialise, comme nous venons de l'évoquer, par une composition des opérations de modification de structures qui ont fait l'objet de ce paragraphe. Disons, pour terminer, quelques mots sur l'incidence de ce type d'opérations sur la désignation des objets.

5.3.3.5 Changement de structures et désignation des objets

De façon générale, la suppression d'un type dont le nom peut être utilisé dans une désignation d'objet invalide les chemins d'objets où ce nom figurait : c'est le cas de tous les types potentiellement concernés par un des changements qui ont été examinés dans ce paragraphe, à l'exception des noms des types de base. D'autre part, par effet de bord, dans le cas de la suppression d'un type d'entité, des chemins vers des attributs hérités peuvent devenir non valides. Par exemple, une expression de la forme $E_1.X$ où E_1 est un nom de type d'entité et X un attribut du super-type de E_1, n'est plus valide après la suppression du super-type de E_1 (figure 5.13).

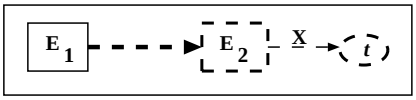

FIG. 5.13 – *Suppression de type d'entité et relation de sous-type*

5.3.4 Fonctions de correspondance et modification de schémas : conclusion

Nous avons voulu montrer, dans ce paragraphe, que certaines opérations de modification de schémas pouvaient avoir une sémantique formelle claire et des interprétations procédurales pour chaque politique de migration et chaque stratégie de représentation des objets introduites dans le paragraphe 5.1 du chapitre 4.

En outre, nous avons déjà eu l'occasion d'affirmer que notre objectif n'était pas de proposer un ensemble "fermé" de modification de schémas implantant une politique et une stratégie données, mais d'identifier un ensemble de services qui permettent de concrétiser des modifications simples de schéma et où des modifications complexes peuvent s'exprimer par une composition de ces services. C'est ce que nous allons examiner dans le paragraphe qui suit où nous montrons que les opérations de transformation de schémas utilisées dans un processus d'intégration de schémas, comme celui évoqué dans le paragraphe 4.2 du chapitre 4, s'expriment simplement à l'aide des opérations de modification comme des transactions de modification de schémas.

5.4 Fonctions de correspondance et intégration de schémas

La phase de mise en conformité du processus d'intégration de schémas et celle de fusion faisaient appel à quelques opérations "jugées raisonnables" de transformation de schémas appelées opérations de restructuration et opérations de fusion. Le dénombrement de ces transformations ne nous semble pas possible (et ne présente pas beaucoup d'intérêt dans cette étude) car elles visent généralement à apporter des solutions ad hoc à des incompatibilités détectées dans les schémas à intégrer. Ces incompatibilités naissent des choix de conception faits par l'un ou l'autre des concepteurs des schémas. Plus particulièrement, elles peuvent naître du fait que plusieurs concepts d'un modèle de représentation de données sont candidats pour abstraire un phénomène d'un domaine d'application. Par exemple, un concepteur sera "plus porté" à utiliser des types d'associations que des compositions, un autre aura tendance à ne pas faire usage des attributs de cardinalité multiple, leur préférant des types d'associations, etc.

Aussi, pour illustrer la notion de transaction de modification de schéma, nous ne considérons qu'une opération de restructuration et une opération de fusion et nous montrons comment elles peuvent s'exprimer comme des compositions des opérations de modification de schéma largement discutées dans le chapitre 4.

5.4.1 Fonctions de correspondance et restructuration de schémas

Parmi les opérations de restructuration, considérons celle qui consiste à transformer un type d'attribut A en un type d'entité A et dont les alternatives sont graphiquement rappelées sur la figure 5.14.

La correspondance structurelle entre l'ancienne version du schéma et la nouvelle s'exprime :

– *Dans le cas (i)*, dans le type d'entité E, par un renommage de l'attribut A en A' (*ChgTypAttribut.Nom*) et un changement du domaine du type de l'attribut (*ChgTypAttribut.Domaine*). Ainsi, la séquence {*ChgTypAttribut.Nom*; *ChgTypAttribut.Domaine*} constituerait le squelette de la transaction qui permettrait de matérialiser la restructuration.

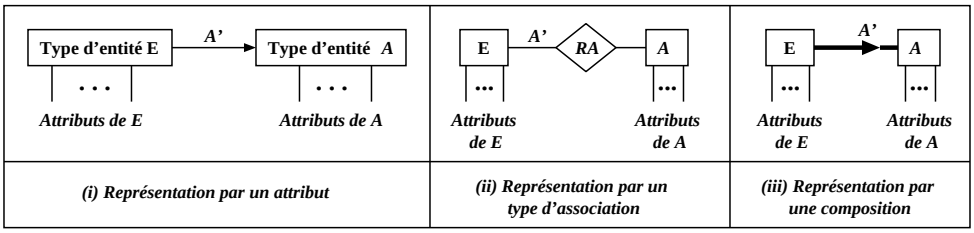

FIG. 5.14 – *Restructuration d'un type d'attribut en un type d'entité*

– *Le cas (ii)*, se concrétiserait par :
{*ChfTypEntité.SuprAttribut; ChgTypAssociation.AjtTypAssociation*},

– et *le cas (iii)* par :
{*ChfTypEntité.SuprAttribut; ChgTypEntite.AjtComposant*}.

En ce qui concerne la correspondance des valeurs des objets, nous nous retrouvons dans le cas où une relation formelle ne peut pas être exhibée de la sémantique de l'opération de restructuration, nous obligeant ainsi à recourir à des solutions ad hoc.

5.4.2 Fonctions de correspondance et fusion de schémas

Considérons l'opération de fusion de deux types d'entités, telle que schématisée sur la figure 5.15.

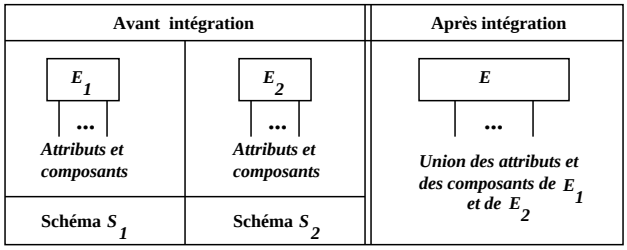

FIG. 5.15 – *Fusion de deux types d'entités (rappel)*

Formellement, cette opération s'exprime par {*ChgTypEntite.AjtTypEntité*} si E est un nouveau type d'entité ou par {*ChgTypEntité.AjtTypAttribut**} si E est un renommage de E_1 ou de E_2.

Néanmoins, cette opération à comme effets de bord, des modifications de liens de sous-type. En effet, tous les sous-types de E_1 et de E_2 deviennent sous-types de E. Par ailleurs, si E_1 et E_2 n'ont pas un super-type commun, autre que la racine *'Entité'*, nous avons déjà fait remarquer que le schéma résultant était incorrect car il ne satisfait pas les critères de qualité. En somme, nous sommes incapable dans ce cas de figure d'associer statiquement une sémantique non ambiguë à ce type d'opération.

Ceci montre davantage, s'il en est encore besoin, que toutes les opérations de modification de schémas ne pouvaient pas avoir une sémantique claire. Les effets de certaines d'entre elles peuvent être "programmés" à l'aide des opérations de modification de schémas dites locales, le système support se chargeant de valider ou d'annuler les actions du "programme".

5.5 Conclusion

Ce chapitre a montré nos propositions en matière de modification et d'intégration de schémas. Nous y avons défini un ensemble d'opérations qui agissent sur les spécifications des types qui constituent une

spécification de schéma. *La première catégorie d'opérations*, celle des opérations dites de modification de schémas est utilisable tant dans la conception initiale d'un schéma que dans sa modification durant la durée de vie des bases qui s'y conforment. Dans ce dernier cas, les effets d'une modification des types d'un schéma doivent être répercutés sur les données et sur les programmes existants. Dans cet objectif, nous avons présenté diverses politiques de migration d'une base et diverses stratégies de représentation des schémas et des objets. Ceci nous a permis de montrer que la détermination des mécanismes était indépendante des politiques ou des stratégies. En d'autres termes, les concepts et les mécanismes que nous proposons permettent de concrétiser indifféremment l'une ou l'autre des politiques ou des stratégies. Ces mécanismes se fondent sur des fonctions de correspondance définies entre deux variantes de types ou deux versions de schémas. L'étude de la diversité des fonctions a été menée en considérant une classification des opérations de modification de schémas en opérations de renommage, opérations de changement de domaines et en opérations de changement de structures de types. De façon générale, nous avons montré que lorsqu'un relation de sous-type est identifiable entre un type et sa variante, la migration des données est définissable de façon non ambiguë. Dans le cas où aucune relation formelle ne peut être établie entre un type et sa variante, on est conduit à définir, quand cela est possible, des solutions ad hoc. Ce cas de figure s'applique généralement aux opérations dites de changement de structures des types.

La seconde catégorie d'opérations, appelées opérations de restructuration de schémas, est utilisable dans le processus d'intégration de schémas. Les opérations de cette catégorie s'apparentent aux opérations de changement de structures. Compte tenu de la difficulté de cerner les types de restructurations que peut subir un schéma, nous avons défini quelques restructurations "intuitivement correctes" et nous avons montré qu'elles s'exprimaient à l'aide des opérations de la première catégorie. Cette particularité rejoint notre objectif général qui est d'identifier l'ensemble des mécanismes de base nécessités par l'intégration et l'inter-opérabilité des systèmes logiciels, indépendamment d'un domaine d'application particulier.

Une attention particulière a été consacrée à la formalisation des propositions exposées. Cet effort de formulation d'une sémantique statique nous distingue de beaucoup de travaux similaires dans le domaine, comme nous en distingue aussi, le fait que les propositions que nous avons formulées concernent les définitions des types, les instances de ceux-ci et les désignations des objets dans les méthodes ou opérations associées aux types, ces dernières considérations indiquant nos premiers pas vers la restructuration effective des programmes.

Nous avons également essayé de faire ressortir le fait que les effets de bord de certaines opérations pouvaient avoir un coût élevé pouvant ainsi faire surgir la question de leur opportunité ou de leur efficacité dans le cadre d'une base de données opérationnelles. Nous pensons, à ce propos, que des bases comportant peu d'instances, comme c'est souvent le cas des bases d'objets "non traditionnelles", peuvent être soumises à ce type d'opération.

Le chapitre qui va suivre complète nos propositions en matière de restructuration de schémas et d'objets en liaison avec les opérations dites de changement de structures des types de données.

6

Interopérabilité orientée objets

L'intégration par les données, que nous avons dénommée *interopérabilité orientée objets* vise à adapter des structures et des instances de données pour permettre l'interopérabilité de systèmes logiciels, ces derniers comme nous l'avons montré dans le chapitre 1, étant entendus au sens large du terme (programme d'application, outil de développement, SGBD, etc). Dans ce qui suit, nous considérons l'intégration par les données comme un processus qui vise à construire une relation de sous-type entre deux structures de données, une *source* et une *cible*. La structure source est la représentation concrète des données produites par un système logiciel S_1 et la structure cible est celle admise en entrée par un système S_2. L'objectif de l'intégration par les données est de rendre les sorties de S_1 conformes aux entrées de S_2, c'est-à-dire que l'interopérabilité orientée objets, à l'opposé de l'interopérabilité orientée processus, adapte les données plutôt que le code du logiciel.

Le processus qui est décrit ici compare d'abord les structures de données, puis essaie d'opérer la restructuration de la source en la structure cible. En cas de réussite, les instances de la structure source sont alors transformées en des instances se conformant à la structure cible. L'approche est fondée sur les spécifications abstraites de types de données et sur la notion de sous-type. Par ailleurs, nous optons délibérément, dans ce chapitre pour une présentation en liaison avec le *domaine d'expérimentation* que nous avons choisi, c'est-à-dire celui des environnements de développement de logiciels. Nous évoquerons à la fin du chapitre les relations entre les propositions qui vont être exposées et celles déjà présentées dans le cadre de la modification et de l'intégration de schémas.

Après un bref rappel des dimensions de l'intégration dans les environnements de développement de logiciels, ce chapitre montre le scénario de l'intégration, son fondement formel et l'implantation qui en est faite. Les limites de l'approche décrite et ses extensions sont également discutées.

6.1 Introduction

6.1.1 Dimensions de l'intégration : rappel

Nous avons déjà introduit le fait que l'intégration des outils d'un environnement pouvait être considérée selon cinq dimensions : plateforme, présentation, données, contrôle et procédé [201]. Illustrons les par l'environnement présenté dans la figure 6.1 et doté d'un outil de dessin de diagrammes Entité-Association (*ERDraw*), d'un transformateur de schémas Entité-Association en des schémas relationnels (*ER2Rel*) et d'un générateur de rapports (*Report*). Ces outils sont considérés comme "bien intégrés" du point de vue des cinq dimensions si, respectivement,

(i) ils s'exécutent sur une ou plusieurs plateformes technologiques qui assurent la transparence de la localisation des outils, celle des données, l'éventuelle répartition de celles-ci, etc,

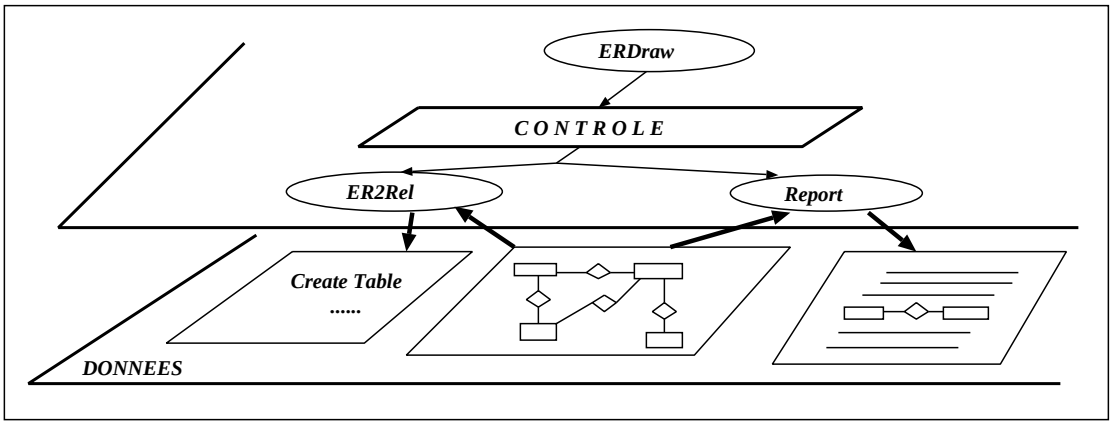

FIG. 6.1 – *Exemple d'environnement*

(ii) leurs fonctionnalités peuvent être invoquées par le biais d'interfaces homogènes ayant la même apparence pour l'utilisateur de l'environnement,

(iii) ils peuvent "partager" des données (par exemple, *ER2Rel* connaît la structure de objets produits par *ERDraw*),

(iv) ils peuvent communiquer : par exemple, *ERDraw* peut notifier des modifications de schémas existants à *ER2Rel* et à *Report* qui pourront réagir en conséquence,

(v) il existe un ensemble de règles régissant le fonctionnement de l'environnement. Ces règles peuvent être, par exemple, des contraintes d'enchaînement des outils (e.g. *ER2Rel* et *Report doivent suivre ERDraw* mais *peuvent* s'exécuter en concurrence), des règles de gestion des modifications e.g. toute modification d'un schéma doit être suivie de la production d'un schéma relationnel et d'un rapport de conception à jour), etc.

Rappelons aussi que la dimension données à laquelle ce chapitre est dédié, vise l'échange et le partage de données entre des systèmes logiciels autonomes ou coopérants. Dans une *"intégration a priori"*, les composants de ces systèmes sont supposés avoir été conçus et développés de sorte à pouvoir inter-opérer. A l'inverse, dans une *"intégration a posteriori"*, il s'agit de changer ou d'incorporer un composant sans perturber le système existant et plus particulièrement, permettre au nouveau composant de communiquer avec d'autres composants et d'opérer sur leurs résultats en dépit d'éventuelles différences qui pourraient exister entre les représentations de ces résultats et celles des données admises en entrée du nouveau composant. A titre d'exemple, en référence à l'environnement de la figure 6.1, l'incorporation d'un outil *ER2O2*, qui transforme un schéma Entité-Association en un schéma fondé sur un modèle orienté objets [48, 67], devrait être aussi transparente que possible, c'est-à-dire que, conformément à l'approche pour l'interopérabilité que nous privilégions ici, *ER2O2* devrait pouvoir utiliser les résultats de *ERDraw* directement ou via des mécanismes spécifiques. Ceux-ci peuvent être incorporés au nouvel outil lui-même ou réunis dans un service commun aux outils de l'environnement. Dans le premier cas de figure, cela signifie que d'une certaine façon, les mécanismes d'intégration sont répartis dans les divers composants de l'environnement, alors que dans le second cas, ils sont concentrés dans un *serveur d'intégration*. Nous sommes adeptes de la seconde vision car,

1. d'une part, elle va dans le sens de la "factorisation" des fonctionnalités (ou services) communes aux composants d'une plateforme technologique : c'est-à-dire rendre disponibles un ensemble de

fonctionnalités utilisables aussi bien par la plateforme elle-même que par les systèmes logiciels qu'elle accueille (SGBD, outils de développement, logiciels applicatifs, etc).

2. D'autre part, elle nous semble plus à même de favoriser *l'interopérabilité entre différentes plateformes ou différents environnements*.

Néanmoins, ces stratégies ne sont pas exclusives l'une de l'autre et l'adoption de l'une d'elles peut dépendre de la technologie de la plateforme sous-jacente à l'environnement et de la granularité des objets à intégrer : nous aurons l'occasion d'en discuter plus longuement dans la conclusion de ce chapitre (paragraphe 6.6).

6.1.2 Caractéristiques de l'approche

Si le problème, tel qu'il vient d'être exposé sous son angle applicatif, semble intuitivement clair, il n'en est pas de même du point de vue des fondements des approches pour sa résolution. Nous avions mis en évidence, dans le chapitre introductif (chapitre 1, paragraphe 1.2), diverses stratégies pour concrétiser l'intégration par les données (encapsulation des outils, conversion de données "point à point" ou en deux étapes via un modèle de données pivot). Nous avions également, dans le paragraphe 1.4.1 du même chapitre, montré que le problème revenait à la définition de fonctions avec *polymorphisme ad hoc et coercion*, c'est-à-dire qu'un même code implantant ces fonctions est exécuté pour différents arguments, avec éventuellement conversion des types de ces arguments. Dans le domaine d'application où nous nous situons, le problème est amplifié par la diversité, la complexité et l'évolutivité des types des objets manipulés par les composants d'un système logiciel, rendant difficile l'énumération des fonctions de coercion. Ainsi, une approche statique, telle celle proposée dans [126] ou suggérée dans une architecture à la CORBA [156, 157], ne nous semble pas appropriée car elle pêche par sa rigidité. Pour notre part, nous nous sommes orientés vers une approche dynamique et évolutive, c'est-à-dire une approche

(i) qui *détecte "au vol"* (c'est-à-dire lors de l'invocation d'un outil, d'une méthode sur un objet, voire d'une requête adressée à un serveur d'objets) la nécessité de mettre en œuvre le processus d'intégration, par comparaison des types source et cible objets du processus,

(ii) qui décide si l'intégration est réalisable en tentant *d'approcher une relation de sous-type* entre la source et la cible et en restructurant, éventuellement, la source à l'aide d'un ensemble de règles de ré-écriture,

(iii) et enfin, qui *engendre les fonctions de coercion* qui serviront à transformer une instance de la source en une instance conforme à la cible, les résultats de chaque étape de ce processus étant conservés à des fins de ré-utilisation.

Les principes, les fondements et la concrétisation de cette approche sont développés selon la progression suivante :

- dans le paragraphe 6.2, un exemple générique illustre le processus d'intégration,
- le paragraphe 6.3 contient la description du cadre formel de l'approche,
- dans le paragraphe 6.4, nous présentons le modèle de représentation des données support de notre étude et les règles de ré-écriture des structures exprimables dans ce modèle,
- l'implémentation du processus est rapidement évoquée dans le paragraphe 6.5.
- enfin, nous discuterons (paragraphe 6.6) l'extension de l'approche proposée à la dimension contrôle en vue de permettre l'interopérabilité sur des objets de fine granularité.

6.2 L'intégration de données par l'exemple

Considérons deux outils logiciels fonctionnellement équivalents, comme deux outils de dessin ou deux éditeurs de textes, devant inter-opérer sur des objets. *Un préalable* au processus d'intégration est de disposer d'une spécification des types des objets que les outils manipulent. Celle-ci décrit la structure des objets et "autant de sémantique que possible", c'est-à-dire le plus possible d'information sur les rôles joués par les éléments ou attributs composants les structures. Considérons que

$$\textbf{(i)} < a_1 : t_1, a_2 : t_2, \cdots, a_n : t_n > \text{ et}$$
$$\textbf{(ii)} < b_1 : u_1, b_2 : u_2, \cdots, b_m : u_m >$$

sont de telles spécifications structurelles où $<\ >$ symbolise le constructeur de n-uplets, les a_i et les b_j ($1 \leq i \leq n$ et $1 \leq j \leq m$) sont des symboles d'attributs et les t_i et les u_j sont leurs types respectifs, ceux-ci pouvant être des types simples ou complexes.

Une première phase va consister à comparer les deux définitions. Cette phase inclut la résolution de problèmes d'homonymie et de synonymie et rend comme résultats des listes

- *d'attributs communs*, c'est-à-dire des attributs jouant le même *rôle* dans les deux structures. Ces attributs peuvent ne pas être dénotés par le même symbole ni avoir le même type. Ainsi, un groupe d'attributs dans une structure peut avoir le même rôle qu'un seul attribut dans une autre. Par exemple, un triplet <jour: 1..31, mois: 1..12, an: 0..99> dans **(i)** peut jouer le même rôle qu'un attribut *Date: Chaîne* dans **(ii)**.

- *d'attributs de la structure source **(i)** qui n'ont pas d'équivalents dans la structure cible **(ii)***,

- et, à l'inverse, *une liste des attributs dans la structure cible qui ne figurent pas dans la source*.

Une deuxième phase tente de rendre la source *structurellement compatible* avec la cible, en réordonnant, en restructurant, en ignorant ou en en ajoutant des attributs.

La dernière phase se charge de la *compatibilité des domaines* des attributs. Il s'agit essentiellement de produire puis d'appliquer des fonctions de correspondance entre des ensembles de valeurs. Certaines de ces fonctions sont identifiables "automatiquement", comme l'extension d'un domaine entier à un domaine réel, et d'autres pas. Par exemple, un outil peut désigner des formes de figures géométriques par des chaînes de caractères ('oval', 'carré', 'polygone', etc) alors qu'un autre les spécifie à l'aide de valeurs entières : la correspondance entre les deux domaines est alors définie de façon externe en extension.

De façon plus générale, dans cette phase sont spécifiées en intention ou en extension les relations de sous-types éventuelles entre des composants de la structure source et des composants de la structure cible. C'est également dans cette phase que se situe l'expression de correspondances ad hoc entre des domaines entre lesquels aucune relation formelle ne peut être identifiée (voir le paragraphe 5.3 du chapitre 5). Ceci nous amènera à considérer les conditions d'application du processus et à tenir compte de sa nature indécidable.

De ce scénario, il ressort que le processus d'intégration suppose l'existence d'un modèle de représentation pour exprimer les structures source et cible et de règles de restructuration similaires à celles évoquées dans [3]. Le paragraphe qui suit présente le cadre général du processus d'intégration en se fondant sur l'équivalence de spécifications abstraites de données. Le modèle de représentation que nous avons adopté et les règles de restructuration que nous avons définies font l'objet du paragraphe 6.4.

6.3 Fondements de l'intégration de données

L'intégration de données peut être abstraite en termes de spécification et d'implémentation de types de données : un type abstrait de données peut être spécifié algébriquement [108, 105, 84] ou en utilisant un type support de la représentation du type abstrait, le type support pouvant lui-même être un type abstrait. Par exemple, un type chaîne peut être spécifié en donnant un ensemble d'axiomes qui lient les opérations du type sans fixer de structure d'implémentation. Alternativement, on peut spécifier le type chaîne en adoptant une structure de représentation, comme la valeur de la chaîne et une marque de fin ou un tuple de la forme <Longueur, Valeur> : dans ce cas, la sémantique des opérations du type est exprimée en utilisant les opérations du type support de la représentation (Par exemple, l'implémentation de la fonction de concaténation sera faite conformément à la représentation adoptée).

Maintenant, supposons qu'une telle représentation est fixée pour le type chaîne et que ses opérations ont été implantées en conséquence. Supposons également que des chaînes persistantes existent sous une représentation différente de celle adoptée pour le type chaîne. *Nous considérons que le processus d'intégration est un processus qui transforme les chaînes persistantes existantes en des chaînes dans la représentation adoptée afin de permettre à l'implémentation des opérations de leur être appliquée* (par opposition, encore une fois, à l'interopérabilité orientée processus qui adapterait les implémentations à la représentation des chaînes persistantes).

Après cette analogie de fine granularité, considérons, maintenant, comme illustré par la figure 6.2, un outil comme une opération et les types de ses paramètres et ceux de ses résultats comme des types (probablement) complexes. Soient T_1 un outil ($T_1 : I \longrightarrow O$), $REP_{1.I}$ et $REP_{1.O}$ les représentations respectives de I et de O. Soit également T_2 un outil dont les paramètres sont du même type que ceux des résultats de T_1 ($T_2 : O \longrightarrow O'$), mais dont la représentation $REP_{2.O}$ diffère de $REP_{1.O}$. Pour faire en sorte que T_2 puisse utiliser des objets résultats de T_1, il est nécessaire d'établir une correspondance entre les deux représentations et de transformer les objets de $REP_{1.O}$ en des objets de $REP_{2.O}$ (Le même scénario est applicable quand, dans l'environnement, T_1 est remplacé par un outil fonctionnellement équivalent dont la représentation des paramètres diffère de $REP_{1.I}$).

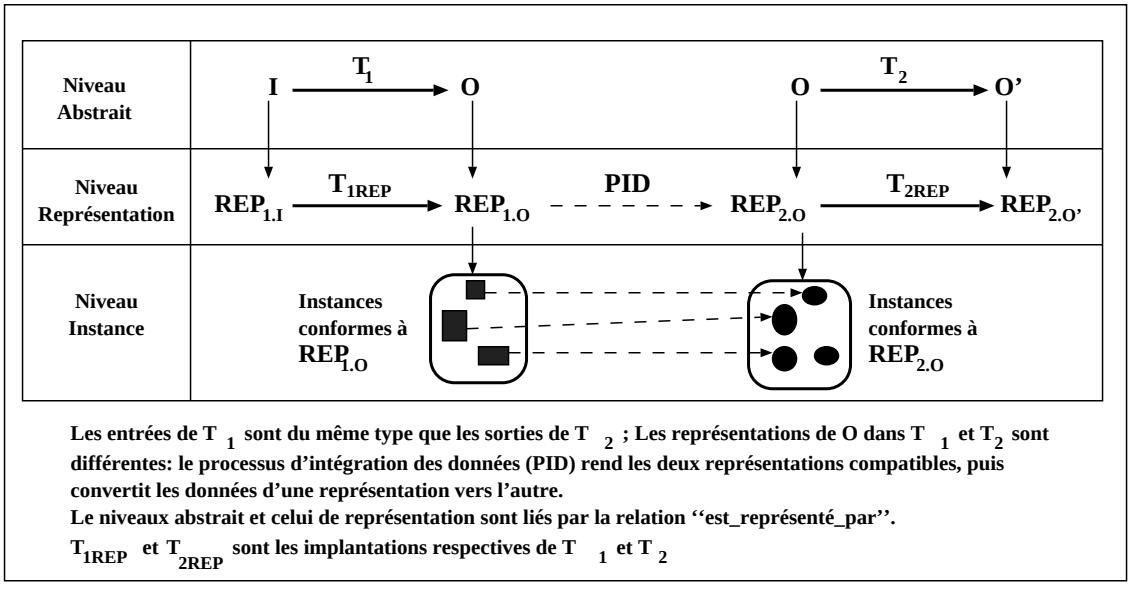

FIG. 6.2 – *Fondements du processus d'intégration par les données*

Mais sous quelle condition le processus peut-il réussir et être considéré comme correct? *Nous*

prétendons que le processus d'intégration de données est un processus qui tente de construire une relation de sous-type entre les deux types de données. En effet, si un type de données D_1 est sous-type de D_2 alors toute valeur de D_1 est utilisable partout où une instance de D_2 l'est, vu que D_1 est sous-type de D_2 si et seulement si chaque instance de D_1 et aussi instance de D_2 [58]. On peut alors raisonnablement penser qu'il suffit de "démontrer" que le type du résultat de T_1 est sous-type de T_2. Cependant, dans un cadre opérationnel, la preuve formelle ne suffit pas : la relation de sous-type est réflexive [7] mais néanmoins, comme le montrait l'exemple des chaînes, les implémentations des opérations sur le type ne peuvent pas indifféremment s'appliquer aux deux représentations du type suggérées ci-dessus.

Remarque : *On notera qu'en fait l'équivalence des spécifications de deux types de données est plus complexe car elle doit aussi tenir compte des comportements des types [84, 173, 192], c'est-à-dire de l'équivalence de leurs ensembles respectifs d'opérations. Pour notre part, nous faisons l'hypothèse (forte) que l'équivalence entre les opérations des types est montrée : concrètement, nous considérons que des outils logiciels équivalents (assimilés à des opérations, rappelons-le) sont regroupés dans des classes dont les extensions sont énumérées.*

En outre, la détermination de la relation de sous-type est indécidable. C'est pourquoi, en fait, nous essayons de l'approcher par ce que nous appelons la *relation d'intégrabilité* : dans ce qui suit, nous dirons *qu'un type D_1 est intégrable à un type D_2* s'il existe :

1. une transformation S qui rend D_1 structurellement compatible avec D_2 (intuitivement, S transforme la structure de D_1 en celle de D_2),

2. et une transformation C qui rend les instances de D_1 conformes à celles de D_2.

Plus formellement, en considérant la spécification d'un type [131] comme un triplet

<Nom du type, Type de l'espace des valeurs, Opérations>

et l'extension du type (ses instances) comme un triplet

<Identifiant d'instance, Valeur, Opérations>,

et, conformément à la remarque qui vient d'être faite, en ignorant les opérations sur les types,

- la transformation S est une correspondance entre les types des espaces de valeurs de D_1 et de D_2,

- et la transformation C est une correspondance entre les valeurs des instances de D_1 et celles de D_2.

Dans ce cadre, le processus d'intégration consiste à identifier et à implémenter S et C [8]. Concrètement, S et C sont des séquences d'opérations atomiques de transformation d'un modèle de représentation de données : ce modèle et les transformations associées font l'objet du paragraphe qui suit.

7. Elle est également transitive et anti-symétrique, définissant ainsi, un ordre partiel sur les types.
8. S et C rappellent, respectivement, les fonctions $\mathcal{T}structure$ et $\mathcal{T}valeur$ du chapitre 5.

6.4 Modèle de données et règles de restructuration

La première étape du processus d'intégration consiste à acquérir la description des structures de données à intégrer, c'est-à-dire la description de la structure source et de la structure cible. Il est clair que leur expression dans le même formalisme facilite leur comparaison. N'ayant pas (encore) de propositions à faire sur cette étape (nous en discuterons brièvement dans la conclusion de ce document), nous supposerons que les descriptions sont disponibles et qu'elles sont exprimées dans un *Langage de Définition d'Interfaces* (Interface Definition Language)

Dans ce paragraphe, nous introduisons de façon informelle le modèle de représentation des structures (il est formellement décrit dans [167, 42] et son langage de définition est donné dans [165]) puis nous présentons les règles de correspondances structurelles qui servent de base à la définition des fonctions de correspondance S du processus d'intégration.

6.4.1 Modèle de représentation de données

Le modèle comprend des types atomiques et des types construits à l'aide :

(i) *d'un constructeur de tuples labellés (ou produits cartésiens),* dénoté par :

X(Rôle$_1$: Type$_1$, . . . , Rôle$_n$: Type$_n$).

Le *constructeur de tuples* est similaire au constructeur d'enregistrements dans des langages de programmation comme PASCAL et par définition

> $X\,(a_1 : r_1, \cdots, a_n : r_n, \cdots, a_m : r_m)$ *est sous-type de* $X\,(a_1 : t_1, \cdots, a_n : t_n)$ si chaque r_i est sous-type du t_i correspondant, $\forall i\ 1 \leq i \leq n$ et $n \leq m$.

On notera qu'en pratique les attributs $[a_1,\ \cdots,\ a_n]$ sont ceux hérités par le sous-type. La conséquence de cette définition est que nous permettrons qu'un tuple T_1 soit étendu par des attributs pour le rendre conforme à la structure de l'un de ses sous-types T_2 et, inversement, la liste d'attributs de T_2 pourra être restreinte aux seuls attributs de T_1).

(ii) *d'un constructeur d'ensemble,* dénoté par **(Rôle: Type).*

La relation de sous-type entre des types ensembles est définie par :

> Si S_1 est un ensemble d'éléments de type t_1 et que S_2 est une ensemble d'éléments de type t_2, S_1 est sous-type de S_2 si et seulement si t_1 est sous-type de t_2.

Concrètement, cela signifie que dans le processus d'intégration de deux structures de type ensemble, on ne s'intéressera qu'aux types des éléments des deux structures.

(iii) *ou d'un constructeur d'union disjointe de types,* dénoté par

∪(Rôle$_1$: Type$_1$, . . . , Rôle$_n$: Type$_n$)

et dont la sémantique est qu'un seul des attributs de l'union est applicable. Ce constructeur est similaire à celui de construction d'enregistrement à structure variable (*variant record*) dans un langage comme PASCAL.

La définition de la relation de sous-type entre deux types union est similaire à celle des types tuples.

En pratique, la composition du constructeur de tuples et de celui d'union de types permet de décrire des spécialisations ou des généralisations disjointes (i.e. les extensions des sous-types sont disjointes

deux à deux) de types d'entités ou de classes d'objets ainsi que les types des attributs hérités. Par exemple, l'expression :

$$< a_1, \cdots, a_n, \cup (< a^1_{n+1}, \cdots, a^1_m >, < a^2_{n+1}, \cdots, a^2_m >) >$$

décrit une structure où $< a_1, \cdots, a_n >$ sont les attributs d'un type T, $< a^1_{n+1}, \cdots, a^1_m >$ sont ceux d'un type T_1 et $< a^2_{n+1}, \cdots, a^2_m >$ sont ceux d'un type T_2, T_1 et T_2 étant les sous-types de T.

Exemple : La figure 6.3, où une représentation graphique identique à celle introduite dans le chapitre 3 est utilisée, montre un tuple "composite" comprenant :

- les *tuples Date_Naiss* et *Eléments_de_Calcul*,
- un *ensemble Loisirs*
- et *l'union de types Montant_Salaire* qui exprime des attributs "alternatifs", c'est-à-dire un salaire fixe *Salaire* (pour les employés permanents, par exemple) ou des éléments *Taux_Horaire* et *Nbr_Heures* de calcul de salaire (pour des vacataires, par exemple).

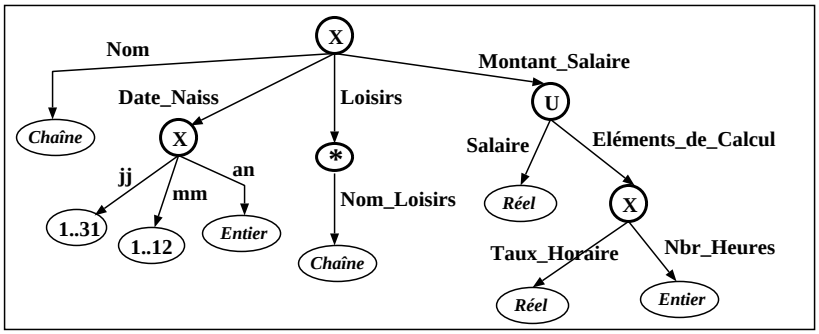

FIG. 6.3 – *Exemple de structure de données*

Remarque : On peut aisément montrer que les types d'un schéma GORM sont exprimables à l'aide de ce formalisme.

Voyons maintenant les types de restructuration que nous autorisons sur les structures exprimables à l'aide de ce modèle de représentation de données.

6.4.2 Règles de réécriture des structures

Ces règles sont élaborées en considérant toutes les compositions possibles des constructeurs du modèle de représentation. Avant de les exposer, faisons remarquer

1. que pour alléger leur formulation, nous omettrons les noms des attributs,
2. et que l'ordre des attributs dans une définition n'est pas important, c'est-à-dire que

$$X(T_1, \cdots, T_n) \equiv X(\Pi_{1-n}(T_1), \cdots, \Pi_{1-n}(T_n)) \text{ et}$$
$$\cup(T_1, \cdots, T_n) \equiv \cup(\Pi_{1-n}(T_1), \cdots, \Pi_{1-n}(T_n)),$$

où Π_{1-n} dénote une fonction de permutation.

Dans la formulation des règles ci-dessous le symbole '$\longrightarrow$' signifie, intuitivement, que la structure en partie gauche de la règle peut être ré-écrite en la partie droite. On trouvera dans l'annexe 3 des exemples d'application de chacune des règles exposées ci-dessous.

1. *Etendre la liste d'attributs d'un produit cartésien :*
   $X(\mathrm{T}_1, \ldots, \mathrm{T}_n) \longrightarrow X(\mathrm{T}_1, \ldots, \mathrm{T}_n, \mathrm{T}_{n+1}, \ldots, \mathrm{T}_m)$
2. *Restreindre la liste d'attributs d'un produit cartésien :*
   $X(\mathrm{T}_1, \ldots, \mathrm{T}_n, \mathrm{T}_{n+1}, \ldots, \mathrm{T}_m) \longrightarrow X(\mathrm{T}_1, \ldots, \mathrm{T}_n)$
3. *Eliminer un produit cartésien imbriqué dans un autre :*
   $X(\mathrm{T}_1, \ldots, X(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), \ldots, \mathrm{T}_n) \longrightarrow X(\mathrm{T}_1, \ldots, \mathrm{T}_i^1, \ldots, \mathrm{T}_i^k, \ldots, \mathrm{T}_n)$
4. *Imbriquer un produit cartésien dans un produit cartésien :*
   $X(\mathrm{T}_1, \ldots, \mathrm{T}_i^1, \ldots, \mathrm{T}_i^k, \ldots, \mathrm{T}_n) \longrightarrow X(\mathrm{T}_1, \ldots, X(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), \ldots, \mathrm{T}_n)$
5. *Distribuer un produit cartésien sur une union de types :*
   $X(\mathrm{T}_1, \ldots, T_{i-1}, \cup(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), T_{i+1}, \ldots, \mathrm{T}_n)$
   $\longrightarrow \cup(X(\mathrm{T}_1, \ldots, \mathrm{T}_{i-1}, \mathrm{T}_i^1, \mathrm{T}_{i+1}, \ldots, \mathrm{T}_n), \ldots, X(\mathrm{T}_1, \ldots, \mathrm{T}_{i-1}, \mathrm{T}_i^k, \mathrm{T}_{i+1}, \ldots, \mathrm{T}_n))$
6. *Distribuer une union de types sur un produit cartésien :*
   $\cup(X(\mathrm{T}_1, \ldots, \mathrm{T}_{i-1}, \mathrm{T}_i^1, \mathrm{T}_{i+1}, \ldots, \mathrm{T}_n), \ldots, X(\mathrm{T}_1, \ldots, \mathrm{T}_{i-1}, \mathrm{T}_i^k, \mathrm{T}_{i+1}, \ldots, \mathrm{T}_n))$
   $\longrightarrow X(\mathrm{T}_1, \ldots, T_{i-1}, \cup(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), T_{i+1}, \ldots, \mathrm{T}_n)$
7. *Désimbriquer un type ensemble dans un produit cartésien* :
   $X(\mathrm{T}_1, \ldots, *(\mathrm{T}_i), \ldots, \mathrm{T}_n) \longrightarrow *(X(\mathrm{T}_1, \ldots, \mathrm{T}_i, \ldots, \mathrm{T}_n))$
8. *Imbriquer un type ensemble dans un produit cartésien* [9] :
   $*(X(\mathrm{T}_1, \ldots, \mathrm{T}_i, \ldots, \mathrm{T}_n)) \longrightarrow X(\mathrm{T}_1, \ldots, *(\mathrm{T}_i), \ldots, \mathrm{T}_n)$
9. *Distribuer un constructeur d'ensemble sur une union de types :*
   $*(\cup(\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_n)) \longrightarrow \cup(*(\mathrm{T}_1), *(\mathrm{T}_2), \ldots, *(\mathrm{T}_n))$
10. *Distribuer une union de types sur un constructeur d'ensemble :*
    $\cup(*(\mathrm{T}_1), *(\mathrm{T}_2), \ldots, *(\mathrm{T}_n)) \longrightarrow *(\cup(\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_n))$
11. *Eliminer une union de types imbriquée dans une union de types :*
    $\cup(\mathrm{T}_1, \ldots, \cup(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), \ldots, \mathrm{T}_n) \longrightarrow \cup(\mathrm{T}_1, \ldots, \mathrm{T}_i^1, \ldots, \mathrm{T}_i^k, \ldots, \mathrm{T}_n)$
12. *Imbriquer une union de types dans une union de types :*
    $\cup(\mathrm{T}_1, \ldots, \mathrm{T}_i^1, \ldots, \mathrm{T}_i^k, \ldots, \mathrm{T}_n) \longrightarrow \cup(\mathrm{T}_1, \ldots, \cup(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), \ldots, \mathrm{T}_n)$
13. *Structures invariantes*, i.e. structures ne pouvant pas être réécrites en d'autres :
    - $*(*(\mathrm{T}))$
    - $\cup(\mathrm{T}_1, \ldots, X(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), \ldots, \mathrm{T}_n)$
    - $\cup(\mathrm{T}_1, \ldots, *(\mathrm{T}_i), \ldots, \mathrm{T}_n)$

9. Ces deux dernières opérations sont strictement équivalentes à la désimbrication et à l'imbrication de relations dans le modèle relationnel de données NF^2 [1].

Exemple d'application des règles : L'exemple de la figure 6.3 peut être restructuré en celui de la figure 6.4 par la séquence suivante :

1. Eliminer le produit cartésien imbriqué *Date_Naiss* (règle 3),
2. supprimer l'attribut *Montant_Salaire* (règle 2),
3. Désimbriquer l'ensemble *Loisirs* (règle 7)
4. et étendre la liste d'attributs avec *Prénom* (règle 1).

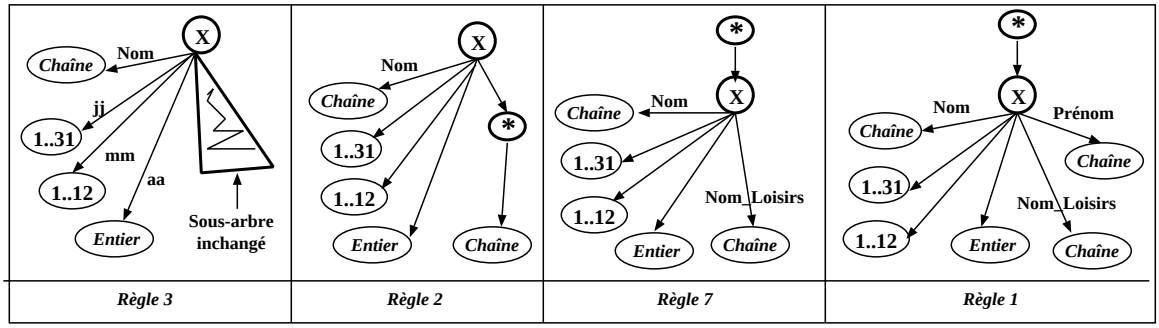

FIG. 6.4 – *Suite de réécriture de structures*

L'implantation faite du processus d'intégration [165] propose un mécanisme d'assistance à la découverte de ce type de séquence avant d'opérer la restructuration effective. Comme nous l'avons déjà mentionné, le processus étant indécidable, le prototype réalisé a été implémenté comme un système interactif basé sur la connaissance. Il est rapidement décrit dans le paragraphe qui suit.

6.5 Un aperçu sur l'implémentation

Les règles de ré-écriture de structures constituent le "cœur" de l'intégrateur de données. En fait, à chaque règle, à l'exception des invariants, est associé un opérateur qui transforme une structure conforme à la partie gauche de la règle en une structure conforme à la partie droite. Ces opérateurs constituent les opérations de base dont la composition permet d'exprimer le corps de la transformation S évoquée dans le paragraphe 6.3 : en d'autres termes, le résultat de la troisième phase du processus d'intégration, lorsque celui-ci réussit, est une séquence d'opérateurs de restructuration de la structure source en la structure cible.

Outre les transformations structurelles, des opérateurs sont également disponibles pour :

– *renommer* des types,

– *désigner* des relations de sous-types,

– et *spécifier* des fonctions de coercion.

Cette dernière catégorie d'opérateurs permet d'exprimer la fonction de transformation C qui, rappelons-le, a pour rôle de transformer le *contenu* de l'objet source.

D'autre part, dans l'implémentation actuelle, les outils "similaires" sont groupés dans des classes (par exemple, une classe *Edition* contient la désignation des *éditeurs* disponibles dans l'environnement, une classe *Dessin* contient les outils *graphiques*, etc) : intuitivement, cette notion de classe d'outils

résoud, par énumération des membres des classes, le problème de l'équivalence des opérations sur des types abstraits différents.

Le scénario du processus d'intégration est alors le suivant :

1. un outil est choisi dans une classe ainsi que l'objet sur lequel on désire appliquer l'outil,
2. le processus d'intégration est automatiquement activé si l'objet "paramètre" de l'outil n'a pas été produit par celui-ci (par exemple, on désire modifier à l'aide de l'outil de dessin *Xfig* une figure produite par l'outil *Tgif*),
3. un *sous-système d'intégration* tente alors de déterminer si les types de données des deux outils sont *intégrables*. Le sous-système est implanté comme un système à base de connaissances qui tente d'unifier les structures des deux types en se fondant sur les règles de ré-écriture de structures. L'unification des structures étant indécidable, le sous-système peut inter-agir avec un agent humain pour lever des ambiguïtés. Les informations recueillies sont utilisées pour résoudre ces ambiguïtés. Elles sont aussi conservées dans la base de connaissances du système, à des fins de ré-utilisation.
4. Si toutes les ambiguïtés sont résolues, le système crée une copie de l'objet qui se conforme à la structure de données admise par l'outil invoqué. Cette copie d'objets est obtenue grâce à la séquence d'opérations engendrée lors de la première phase du processus.

6.6 Discussion

L'approche exposée ici de *l'inter-opérabilité orientée objets (ou intégration des données)* allie deux préoccupations : l'une fondamentale et l'autre pragmatique. Nous avons essayé de montrer que formellement le problème était décidable lorsqu'une relation de sous-type existait entre les types source et cible de l'intégration et indécidable autrement. D'un point de vue pragmatique, dans chacun des cas, il peut être nécessaire de mettre en œuvre un procédé de restructuration des objets du fait des différences des implémentations des types à intégrer. Dans ce cadre, nous avons proposé un ensemble de règles de ré-écriture de structures que nous considérons "intuitivement correctes", c'est-à-dire qui expriment des restructurations "raisonnables". Nous sommes néanmoins conscient que la preuve formelle de leur correction et celle de la complétude de l'ensemble des règles restent à établir.

Sur un autre plan, le discours tenu dans ce chapitre s'est délibérément fait dans les termes du domaine d'expérimentation que nous avions retenu, c'est-à-dire celui des environnements de développement de logiciel. Néanmoins, nous estimons que les concepts et les mécanismes que nous proposons sont applicables à d'autres domaines ou à d'autres applications comme nous essaierons d'en convaincre le lecteur dans la conclusion de cette partie de ce document. Dans l'immédiat, nous discutons

– l'adéquation de nos propositions à la prise en compte de granularités diverses des objets dans un environnement,
– et une perspective à court terme de ce travail qui vise à coupler les fonctionnalités de la dimension données avec celles de la dimension contrôle dans les environnements.

6.6.1 Granularité des objets et intéropérabilité

Comme nous l'avions longuement discuté dans le chapitre d'introduction, la forme la plus primitive, et qui était la plus répandue, de l'intégration par les données consiste à faire partager des

fichiers par les outils. Néanmoins, les systèmes de gestion de fichiers ne sont pas adaptés pour la prise en compte des contraintes d'intégrité sur les données à partager, pour la gestion de versions, celle des accès concurrents, etc. Ces fonctions sont mieux maîtrisées par le système de gestion d'objets (SGO) dans les environnements fondés sur une base de données commune aux outils. Néanmoins, les SGO comme [135, 95] ne permettent généralement, qu'une intégration de *grosse granularité* : les descriptions des objets (par exemple, documents de spécification et de conception, fichiers de code source, etc) sont exprimées en termes d'attributs et de liens dans une spécification de schéma mais pas leurs contenus. En d'autres termes, le contenu d'un objet est généralement un fichier, traité comme un granule par un outil, celui-ci étant monolithique. Dans ce cas de figure, les communications inter-outils sont peu fréquentes. En considérant l'exemple d'environnement donné en introduction de ce chapitre (figure 6.5), *Report* ne peut pas "s'adresser" à *ERDraw* pendant que celui-ci est en train de contribuer à l'élaboration d'un schéma. En outre, l'incorporation d'un nouvel outil dans cet

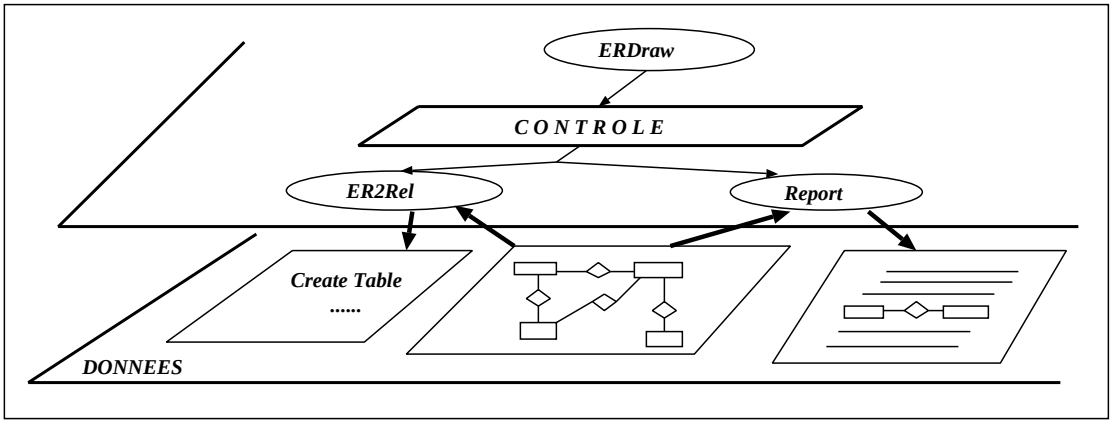

FIG. 6.5 – *Exemple d'environnement (bis)*

environnement nécessite souvent son "encapsulation" pour qu'il puisse communiquer avec le SGO et accéder à la base commune des objets. Nous avons avons déjà fait noter les inconvénients liés à ce type de solutions : multiplication des "capsules", manque de flexibilité réduisant les capacités d'évolution de l'environnement, etc. Cette approche se classe dans la stratégie que nous avions dénommée, dans l'introduction de ce chapitre, "répartition des services d'intégration" dans les outils.

A l'opposé, la stratégie "serveur d'intégration" nous semble mieux adaptée à une intégration de *fine granularité* où le contenu des fichiers serait décomposé en objets ou classes d'objets, avec les relations usuelles dans les approches orientées objets (*Est_un, Partie_de, utilise*, etc). Les outils, eux, ne seraient plus monolithiques mais ils sont vus comme un ensemble d'interfaces d'opérations ou de méthodes associées aux objets ou à leurs classes. Les communications inter-outils sont plus fréquentes dans cette stratégie. En fait, elles sont décrites par le réseau de messages inter-classes d'objet et leur prise en charge devient ainsi similaire à la prise en charge de l'envoi et de l'acheminement de messages dans les langages à objets. Les opérations d'évolution et d'intégration de schémas et celles assurant l'interopérabilité s'insèrent naturellement dans le serveur d'intégration.

Cette stratégie favorise, par essence, la ré-utilisation et l'évolution de l'environnement. Elle est plus à même de faciliter la communication entre des plateformes d'accueil des environnements ou la distribution et l'hétérogénéité d'un environnement. Les mécanismes d'intégration des données (i.e. ceux liés au polymorphisme des outils) et ceux de la communication inter-outils deviennent indissociables, comme nous allons le voir.

6.6.2 Données + Contrôle

Une perspective de ce travail (en cours de réalisation) est relative aux mécanismes de base qui semblent nécessaires dans les environnements. En effet, les dimensions de l'intégration dans ces derniers ne sont pas indépendantes les unes des autres et plus particulièrement, nous estimons que celle des données et celle dite du contrôle sont intimement liées. La majorité des solutions proposées (comme celles décrites dans [55, 91, 51]) pour le contrôle offrent des mécanismes de communication de messages entre les outils pouvant s'assimiler à des invocations de services inter-outils. Le couplage des dimensions données et contrôle permettrait d'adapter les messages d'invocation des services à des modifications des interfaces de ces services. La figure 6.6 montre le scénario visé.

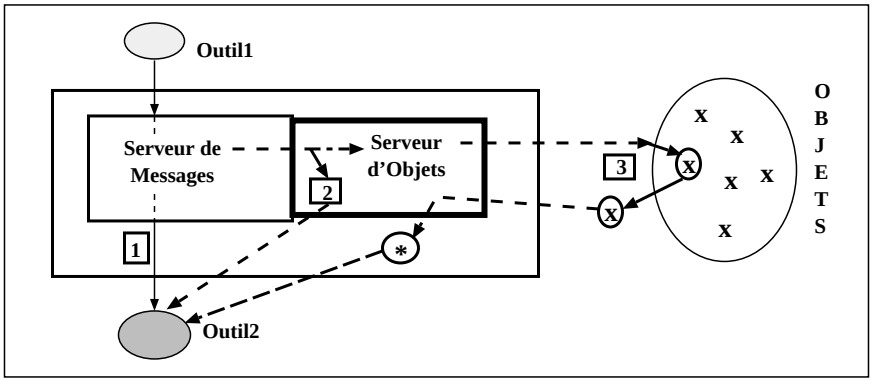

FIG. 6.6 – *Couplage des dimensions données et contrôle*

Le chemin [1] est celui généralement suivi [55, 174, 91, 77] : un outil communique avec un autre grâce à un service de messagerie fourni par l'environnement. Si nous considérons à nouveau l'exemple de la figure 6.5, *ERDraw* peut être vu comme un ensemble de services que peuvent invoquer *Report* et *ER2Rel*. Par exemple, *Report* peut demander à *ERDraw* d'extraire une partie d'un diagramme entité-association pour l'inclure dans un extrait de la documentation de la conception, et réciproquement, *ERDraw* peut demander à *Report* de mettre à jour un rapport sur une partie d'un schéma qui a été modifié. Si les interfaces de ces services sont modifiées, les invocations des services échouent. Le couplage des deux dimensions vise à éviter ces échecs en permettant aux messages de suivre le chemin [2] ou le chemin [3]. *Le chemin* [2] concerne les *données transiantes* (comme les paramètres des invocations) alors que *le chemin* [3] serait suivi par *les données persistantes.* Dans chacun des cas, le processus d'intégration, appelé *Serveur d'objets* sur la figure, peut être activé si la structure des valeurs des objets dans les messages (ou paramètres des invocations de méthodes) n'est pas conforme à celle des types admis "en entrée" du service invoqué. Le serveur d'objets orientera alors le message vers un des chemins en se fondant sur la comparaison de la description des structures des objets passées et celle des structures du service invoqué.

Cette vision des choses rejoint nos propos relatifs à la "factorisation" des services dans un environnement : le *serveur d'objets* fait usage des services (ou opérations) définis pour l'intégration des données. Ceux-ci comprennent notamment, les mécanismes d'implantation de l'inter-opérabilité inter-outils que nous considérons comme une forme de polymorphisme d'opérations de "grosse granularité" que sont les outils.

Quatrième partie

Conclusion et perspectives

7

Conclusion et perspectives

La discussion portera successivement sur :

(i) la relation entre le processus d'intégration de schémas et de celui d'intégration des données,

(ii) une approche pour l'implémentation des *transactions de modification de schémas*,

(iii) le rôle des mécanismes d'intégration dans un processus de migration de systèmes logiciels.

(iv) et sur l'extension du processus d'intégration des données à la prise en compte des contraintes d'intégrité sur les données (contraintes que nous avons délibérément occultées dans ce document),

7.1 Intégration de schémas et intégration de données

La question que l'on est en droit de se poser est "quelles sont les similitudes ou les différences entre l'intégration et la modification de schémas et l'intégration des données?" Nous avions fait remarquer, en introduction de ce chapitre, que la présentation de nos propositions en matière d'intégration de données serait faite dans les termes du *domaine d'expérimentation* que nous nous étions fixé. Nous allons dans ce qui suit, essayer de montrer la généralité de ces propositions et plus particulièrement la complémentarité des travaux sur l'intégration de schémas avec ceux sur l'intégration des données.

Les règles et les opérations mises en œuvre dans le processus d'intégration de données sont applicables

- *aux opérations de modification de schémas* de la catégorie que nous avions dénommée *opérations de changement de structure* dans les chapitres 4 et 5,
- et à celles qui, dans les mêmes chapitres, menaient à une situation où aucune relation formelle ne pouvait pas être exhibée entre un type dans une version V_i^S d'un schéma S et sa *variante* dans une version V_j^S de S.

En effet, la structure source joue le rôle d'un type T dans V_i^S et la structure cible joue celui de la *variante* de T dans V_j^S. Les différences entre les deux structures sont déterminées par l'opération de modification appliquée au type *source*, rendant inutile la phase initiale du processus général d'intégration de données (celle de comparaison des structures). De ce fait, la différence entre la phase de comparaison du processus d'intégration de données et celui d'intégration de schémas réside essentiellement dans la nature de leurs résultats : dans le premier cas, il s'agit de *similitudes certaines* entre deux structures alors que dans le second cas, ceux ne sont que des *assertions hypothétiques de*

similitude nécessitant d'être confirmées ou infirmées. Par ailleurs, ce qui distingue les deux processus, est que l'intégration de schémas produit un schéma à partir de deux autres alors l'intégration de données transforme la structure source en la structure cible. En ce sens, l'intégration de données ressemble à la *phase de mise en conformité* du processus d'intégration de schémas mais en imposant que les restructurations nécessaires ne soient appliquées qu'à la structure cible (alors que dans le cadre général de l'intégration de schémas, l'un ou l'autre des schémas à intégrer peut être l'objet de restructurations).

L'intégration des logiciels réalisés pour valider les propositions faites dans les deux cadres est en cours de réalisation en vue de parvenir à un "Laboratoire d'Intégration et d'Evolution des Systèmes Software et des Environnements" où nous expérimenterions les suites de ces travaux indépendamment d'une plateforme technologique particulière avant leur éventuel transport sur des plateformes industrielles.

7.2 Evolution de schémas et transactions

Le processus de modification d'intégration de schémas se déroule selon le scénario générique suivant :

1) vérifier que la modification désirée ne va pas à l'encontre des critères de qualité du schéma, c'est-à-dire que le schéma résultant les satisfait,

2) identifier les impacts de la modification sur les données et les programmes existants,

3) appliquer la modification, produisant ainsi une nouvelle version de schéma qui remplace la précédente ou co-existe avec elle,

4) propager les effets de bord de la modification sur les données et les programmes, selon une des politiques de migration.

Symboliquement, si $\mathcal{T}_\mathcal{S}$ est un type dont les instances sont des schémas de données, une opération générique O_M de modification d'un schéma S a comme domaine $\mathcal{T}_\mathcal{S}$ et la liste des types des paramètres de O_M, le co-domaine étant le type $\mathcal{T}_\mathcal{S}$. Dans une politique de propagation immédiate ou différée des effets de bord d'une modification, le scénario ci-dessus doit être exécuté comme une transaction "classique", alors que dans la politique opportuniste, il serait exécuté comme une suite de transactions.

Une perspective technique de notre travail est d'étendre les fonctionnalités des prototypes déjà réalisés [165, 44] par des mécanismes de génération de la suite de transactions. Celle-ci serait déclenchée "automatiquement" par le biais de mécanismes similaires à ceux mis en œuvre dans les bases de données dites actives. La gestion de la suite de transactions pourrait être dévolue à un système de gestion de transactions coopérantes comme celui en cours de développement dans [57] qui pallierait les inconvénients des mécanismes classiques d'invalidation des transactions en relâchant notamment la contrainte d'atomicité du fameux *"ACID test"*.

7.3 Les mécanismes d'intégration dans la migration de "systèmes du patrimoine"

La migration de systèmes logiciels du patrimoine [5], (*legacy systems*) peut se justifier par un changement de technologie de ces logiciels (par exemple, passer d'une "technologie Cobol" à un technologie

relationnelle ou de cette dernière à une technologie objets), par l'extension de leurs fonctionnalités ou leur intégration à des logiciels nouvellement développés. Les stratégies de migration peuvent consister à :

(i) *re-développer* des logiciels équivalents à ceux du patrimoine (à notre avis, peu réaliste et coûteux),

(ii) *encapsuler* le logiciel existant pour qu'il puisse s'intégrer dans un nouvel environnement ou une nouvelle plateforme technologique : cette approche est déjà plus attrayante que la précédente du point de vue correction du résultat (les logiciels existants ne sont pas modifiés) et de la "longueur" du chemin de migration. Néanmoins, elle ne favorise pas l'amélioration des logiciels du patrimoine ni la ré-utilisation de certains de ses composants.

(iii) *adapter* les logiciels existants à la technologie ou à la plateforme cible et saisir l'opportunité pour identifier des composants ré-utilisables : cette stratégie est de plus en plus adoptée et rend la ré-utilisation de logiciels similaire à sa maintenance [21].

La dernière de ces stratégies requiert des mécanismes de compréhension et de restructuration de l'existant. Dans le processus de migration schématisé par la figure 7.1 :

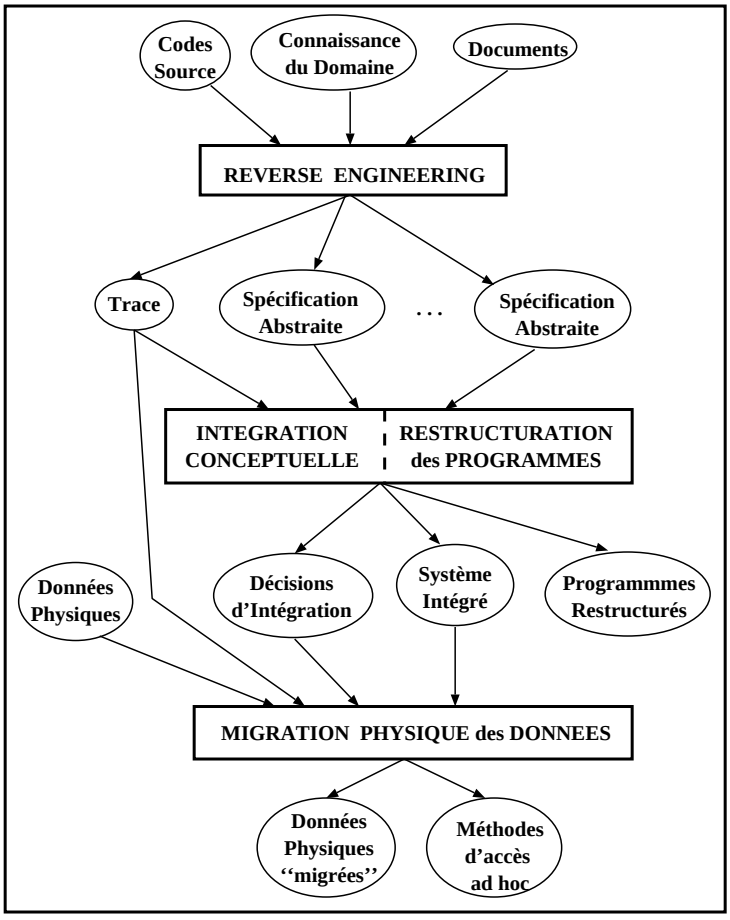

FIG. 7.1 – *Migration de logiciels du patrimoine*

1. dans l'étape de rétro-ingénierie qui vise essentiellement à comprendre l'existant [26, 147], des abstractions sont produites pour les données et les fonctions du logiciel du patrimoine,

2. les abstractions de données sont intégrées pour avoir une vision globale de l'ensemble du système et identifier les redondances ou les composants potentiellement ré-utilisables. Cette activité est complémentée par une restructuration des fonctions identifiées dans la première étape, conformément à l'intégration opérée sur les abstractions des données.

3. Enfin, faire physiquement migrer les données persistantes de l'ancien système vers le nouveau ou développer des méthodes d'accès spécifiques pour permettre l'interopérabilité entre le nouveau système et d'éventuelles parties de l'ancien qui n'auraient pas été sujettes à migration.

Il nous semble que, comme nous le discutons plus longuement dans [36], :

– *les opérations de modification de schémas* sont utilisables pour construire les abstractions de données (intuitivement, il s'agit de modifier un type de schéma initialement vide par des opérations d'ajout),

– *le processus d'intégration de schémas* est parfaitement adapté pour l'intégration des abstractions de données produite dans la première étape du processus de migration,

– et que les opérations décrites dans le cadre de l'intégration des données pour l'interopérabilité des outils trouvent leur utilité dans la troisième étape, soit pour réaliser la migration physique des données soit pour implanter les mécanismes d'accès spécifiques. En effet, dans le cas où la migration est partielle, c'est-à-dire qu'elle ne concerne pas la totalité des systèmes du patrimoine, le nouveau système et la partie inchangée de l'ancien système peuvent avoir besoin de communiquer et d'interopérer sur leurs données respectives.

Néanmoins, "un maillon manque à la chaîne", c'est celui qui permettrait d'extraire les abstractions de données à partir des logiciels existants : cette activité peut se fonder sur l'examen du code source du logiciel, celui des noms des données et de leurs structures, l'étude des fonctions opérant sur les données, etc. Elle peut également se fonder sur l'examen des valeurs des données (comme dans [171] où les auteurs tentent de déterminer le schéma d'un base de données relationnelle par des interrogations sur le contenu de la base). Une autre voie qui nous semble prometteuse, et dans laquelle nous sommes actuellement en train de diriger un travail exploratoire, est celle qui s'inscrirait dans le domaine de l'exploitation des "gisements" ou "mines" de données (*Data mining*) ou de la découverte des connaissances (*Knowledge Discovery* [145]).

7.4 Contraintes d'intégrité dans le processus d'intégration

Généralement, la spécification d'un type de données ou d'un type de schéma comprend une variété de contraintes que doivent satisfaire les objets du type. Ces contraintes peuvent être liées aux opérations (pré-conditions et post-conditions, par exemple), elles peuvent exprimer des invariants sur les espaces des valeurs du type ou encore être des contraintes dynamiques ou de transitions d'états. Dans [131], un essai de formalisation de la préservation des contraintes dans l'identification de la relation de sous-type est donné. Pour notre part, nous continuerons, pour les suites à court terme de ces travaux, à délibérément ignorer les contraintes éventuelles sur l'invocation des outils ou des opérations d'un type, nous concentrant sur les contraintes sur les espaces des valeurs.

La structure de données source, comme la structure de données cible peuvent être contraintes par des formules. Il semble "naturel" d'attendre du processus d'intégration qu'il produise des objets satisfaisant les contraintes de la structure cible, et inversement si on veut que le processus soit réversible (c'est-à-dire si on veut remettre l'objet transformé sous sa représentation source).

Conceptuellement, le problème est abordé de la façon suivante : soient P_s et P_c les contraintes sur les structures de données source *S* et cible *C*, respectivement. Soient également O_s un "objet source" et O_c l'objet produit par le processus d'intégration par transformation de O_s.

Dénotons $P_x(O_x)$ le fait qu'un objet O_x satisfait les contraintes P_x.

Nous considérons que le processus d'intégration préserve les contraintes si :

$$P_s(O_s) \rightarrow P_s^c(O_c) \wedge P_c(O_c)$$

i.e. si O_s satisfait P_s alors son transformé O_c satisfait P_c et P_s^c, où P_s^c est la reformulation de P_s dans le "vocabulaire" de la structure cible.

Cette vision de l'intégration des contraintes dans le processus d'intégration de données est en cours d'implémentation et se poursuivra "naturellement" dans un cadre plus général, similaire à celui tracé dans [131], à la prise en compte des contraintes sur les opérations des types. Cette perspective s'inscrit dans les travaux que nous envisageons sur l'étude de l'évolution des contraintes en liaison avec la révision dans les bases de données.

Cet objectif à long terme, que nous espérons atteindre par une démarche similaire à celle suivie jusqu'ici, c'est-à-dire mêlant des aspects fondamentaux pour justifier nos propositions et des aspects expérimentaux pour valider nos propositions, nous permettra alors d'offrir un cadre d'évolution des systèmes logiciels dans lequel l'évolution des environnements fondés sur les modèles de procédés ou celui des bases de données ne seraient que des domaines particuliers d'application.

Cinquième partie

Annexes

1

Exemples de sessions dans un environnement fondé sur la modélisation de procédés

The first screen shows the correct execution of the typed-in operator `Display_project_plans` applied to the object `sort` and the "clicked-on" invocation of the operator `Display_design` applied to the object `sort`. The second screen shows the result of `Display_design`.

FIG. 1.1 – **Screen images of correct executions**

The first screen shows the invocation of the `Can it be done` guidance facility to know if `Test sort` is an authorized operation. The second screen shows that it is not (the precondition is FALSE).

FIG. 1.2 – **Screen images of “Can it be done” guidance facility**

These screens show the invocation of `How to do it`. The result on the second screen is the generated plan.

FIG. 1.3 – **Screen images of “How To Do IT”**

The user wants to exit the MASP while the object `sort` is not yet tested (first screen). A plan is automatically generated and proposed to the user who decides what to do (second screen).

FIG. 1.4 – **Screen images of “Plan Generator” execution**

2

Les opérations de modification de schémas

La modification d'un type de schéma GORM ou d'une des ses instances consistera en la modification d'un des constituants de sa spécification. En d'autres termes, les spécifications des types contenues dans celle d'un schéma sont considérées comme des objets à part entière.

Ainsi, l'étendue des opérations que l'on pourra effectuer, peut aisément être imaginée à partir de la représentation du type schéma, telle qu'elle est schématisée sur la figure 3.10 du chapitre 4, et telle qu'elle a été formalisée dans le même chapitre : chaque "nœud" labellé par un constructeur de type (ensemble, liste, produit cartésien) est susceptible de se voir appliquer une des opérations du constructeur. Cette vision se généralise "récursivement" aux éléments composants une spécification d'un type de schéma, à savoir les types d'entités, les types d'associations et les opérations associées à chacun de ces types. Par ailleurs, chaque valeur attachée à une feuille est sujette à modification.

Cette approche, qui s'apparente à celles adoptées notamment dans [20, 183], permet d'identifier une liste suffisamment complète d'opérations dont certaines, comme celles consistant en un renommage d'un des "objets" de la spécification d'un type de schéma, ont des conséquences mineures et maîtrisables, sur les données et les programmes existants, alors que d'autres, comme la modification de la hiérarchie de sous-types, peuvent conduire à de profondes restructurations. Dans la présentation ci-dessous, les opérations sont rattachées aux définitions formelles des concepts du modèle de représentation et chaque opération est décrite par son rôle et ses effets de bord possibles sur la spécification, sur les instances de données et sur les programmes existants. La discussion des effets de bord met souvent en relief des situations dans lesquelles leur propagation est de nature indécidable. En outre, cette discussion est faite en *supposant que l'on ne dispose pas de mécanismes de gestion de versions* de schémas ou d'instances. Nous avons considéré cette éventualité dans le paragraphe 5 du chapitre 4. Enfin, nous supposerons qu'à chaque programme existant, il est associé un *sous-schéma*, sous-ensemble du schéma de la base sur lequel opère le programme.

2.1 La modification des définitions des types de base

Rappelons que par type de base, nous entendons un type prédéfini ou un type spécifié par renommage ou restriction du domaine d'un type prédéfini. Les opérations de modification concernant les

types non prédéfinis sont alors :

1. *ChgTypBase.Nom* : Changer le nom d'un type de base

   - *Incidences sur le schéma* : le type de base servant à caractériser le domaine d'un ou plusieurs attributs, cette opération peut induire le renommage d'un ou de plusieurs domaines d'attributs (Voir *ChgTypAttr.Domaine*).
   - *Incidences sur les données* : Aucune.
   - *Incidences sur les programmes* : Aucune.

2. *ChgTypBase.ExprDef* : Changer l'expression de définition d'un type de base

   - *Incidences sur le schéma* : Y incorporer la nouvelle expression de définition.
   - *Incidences sur les données* : Convertir les instances de sorte à ce qu'elles se conforment à la nouvelle expression de définition. Il est clair que la complexité de la conversion dépend de l'écart entre l'ancienne et la nouvelle expression de définition. Le cas le plus favorable est celui où la nouvelle définition est la formulation d'une relation de sous-type entre l'ancien et le nouveau type de base : dans ce cas, les instances peuvent ne pas être converties car, en vertu des propriétés de la relation de sous-type, une instance d'un sous-type est aussi instance de son super-type. Néanmoins, comment reconnaître qu'il s'agit d'une relation de sous-type ?

     Dans les autres cas, il faudra exhiber (manuellement ou de façon assistée) la fonction de conversion des instances du type. Des mécanismes comme ceux proposés dans le chapitre 6 peuvent y contribuer.
   - *Incidences sur les programmes* : Dans le cas favorable, il n'y a pas d'incidences car les instances d'un sous-type sont utilisables partout où une instance du super-type l'est. Dans les autres cas, les restructurations des programmes peuvent être coûteuses. Ainsi, si l'ancienne expression de définition du type caractérisait un type atomique et que la nouvelle définit un type à occurrence multiple (ensemble ou liste), la prise en compte de la modification nécessite l'introduction des méthodes associées au nouveau type (recherche, ajout, suppression d'un élément, parcours des éléments de l'occurrence, etc). Le problème inverse se pose dans le cas où la modification redéfinit un type à occurrence multiple en un type atomique : les méthodes ou opérations "ensemblistes" ne sont plus valides sur les instances du nouveau type.

2.2 La modification des définitions des types d'attributs

1. *ChgTypAttr.Nom* : Changer le nom d'un attribut

   - *Incidences sur le schéma* : Changer le nom de l'attribut dans le type d'entité ou le type d'association auquel l'attribut est associé.
   - *Incidences sur les données* : Aucune
   - *Incidences sur les programmes* : Un nom d'attribut étant considéré comme un nom de fonction de projection, le nouveau nom se substitue à l'ancien dans tous les programmes faisant usage de l'attribut en question.

2. *ChgTypAttr.Domaine* : Changer le domaine d'un attribut peut consister en :
   - désigner un type de base ou un type d'entité comme nouveau domaine d'un attribut dont l'ancien domaine est un type de base,
   - ou affecter un type d'entité ou un type de base comme nouveau domaine d'un attribut dont l'ancien domaine est un type d'entité.

   - *Incidences sur le schéma* : incorporer la modification dans le type d'entité ou le type d'association auquel est rattaché l'attribut.
   - *Incidences sur les données* : Le problème ne se pose pas dans le cas particulier où le nouveau nom du domaine est un simple renommage de l'ancien). Mais, dans le cas où la modification est un changement d'un type de base en un autre type de base, les incidences sur les données sont similaires à celle discutée dans le cadre de l'opération de changement de l'expression de définition d'un type de base (*ChgTypBase.ExprDef*). Dans les autres cas, il nous semble illusoire de proposer des mécanismes généraux d'adaptation des instances. En effet, si la modification fait passer le domaine d'un type de base à un type d'entité (ou inversement), comment faire correspondre les valeurs de l'attribut à des identifiants d'entités (et inversement) ? Et dans le cas où la modification a consisté à changer le domaine d'un type d'entité à un autre type d'entité, comment déterminer la correspondance entre les espaces des identifiants des deux types d'entité ?
   - *Incidences sur les programmes* : dans le cas où la modification a changé un type de base en un autre type de base, le problème rejoint celui traité dans l'opération *ChgTypBase.ExprDef*. Dans les autres cas, la correspondance entre les opérations associées à l'ancienne et au nouveau domaine de l'attribut peut être difficile à établir, nécessitant ainsi une restructuration manuelle des programmes qui utilisent l'attribut qui a été l'objet de la modification.

3. *ChgTypAttr.Cardinalité* : changer une cardinalité "Simple" en une cardinalité "Multiple" (ou inversement).

   - *Incidences sur le schéma* : incorporer la modification dans le type type d'entité ou le type d'association auquel est rattaché l'attribut.
   - *Incidences sur les données* : Restructurer pour permettre la représentation d'ensembles de valeurs ou, inversement, pour ne représenter qu'une seule valeur au lieu d'un ensemble. Au niveau des instances, nous pouvons imaginer que, dans le premier cas de figure, la valeur existant devienne un élément de l'occurrence multiple. Mais, dans le cas où une cardinalité multiple est changée en une cardinalité simple, comment choisir la valeur de l'attribut dans l'ensemble de valeurs ?
   - *Incidences sur les programmes* : similaires à celles discutées dans le cas du changement du domaine d'un type de base en un domaine à occurrence multiple, ou inversement.

2.3 La modification de la relation de sous-type

1. ChgSousType.AjtLien : Créer un lien de sous-type entre deux types d'entités

   - *Incidences sur le schéma* : Mise à jour de la définition de la relation de sous-type associée au schéma.

- *Incidences sur les données* : S'il n'existe pas d'instance des types d'entité concernés, la seule incidence est la création d'un ensemble vide destiné à contenir les instances du lien de sous-type créé. Dans le cas contraire, l'instanciation de la nouvelle relation de sous-type peut être automatisée si une définition intentionnelle calculable de l'appartenance à la relation peut être formulée.
- *Incidences sur les programmes* : les programmes qui utilisent des attributs ou des opérations de l'ancien super-type deviennent inopérants.

2. ChgSousType.SupLien : Supprimer un lien de sous-type

- *Incidences sur le schéma* : Mise à jour de la définition de la relation de sous-type en veillant à préserver la connexité du graphe de la relation.
- *Incidences sur les données* : l'instance de la relation supprimée devient vide.
- *Incidences sur les programmes* : idem que ChgSousType.AjtLien.

2.4 La modification des compositions

La définition de la composition d'un type d'entité peut être modifiée par changement des caractéristiques d'un des composants ou par modification de la liste des composants.

2.4.1 Modification d'un composant d'une composition

1. *ChgComposant.Rôle* :

- *Incidences sur le schéma* : mise à jour de la définition du composant dans la définition de la composition concernée
- *Incidences sur les données* : Aucune
- *Incidences sur les programmes* : Le rôle d'un composant est utilisable comme un nom de fonction d'accès aux instances du composant : un renommage de cette fonction est nécessaire dans les programmes qui en font usage.

2. *ChgComposant.Nom* : Changer le nom du composant

- *Incidences sur le schéma* : Répercuter le changement de nom.
- *Incidences sur les données* : si la modification consiste en un renommage du type d'entité du composant, les instances la composition ne sont pas affectées. Dans le cas contraire, comment les adapter ?
- *Incidences sur les programmes* :

3. *ChgComposant.Cardinalité* : voir la discussion concernant le changement de la cardinalité d'un type d'attribut

4. *ChgComposant.Nature* : Changer une participation *"Exclusive"* en *"Partagé"* (et réciproquement) ou une dépendance *"Faible"* en une dépendance *"Forte"* (et réciproquement).

- *Incidences sur le schéma* : Répercuter la modification

- *Incidences sur les données* : Dans le cas du passage d'une participation *"Partagé"* à une participation *"Exclusive"*, l'identification de l'entité à laquelle il faut "donner l'exclusivité" détermine les incidences. En effet, si
  - C est une instance du composant concerné par la modification,
  - que $E_1, \cdots, E_n$ sont des entités *partageant* C par composition
  - et que, désormais, C doit être le composant exclusif de $E_j, \ (1 \leq j \geq n)$

  alors C doit être retiré des compositions de toutes les entités autres que E_j.

  Dans les autres cas, les instances sont insensibles à la modification.
- *Incidences sur les programmes* : pas d'incidence, dans la mesure où le comportement des compositions est supposé être assuré par le SGBD support.

2.4.2 Modification de la liste de composants d'une composition

1. *ChgComposition.AjtComposant* : Ajouter une définition de composant à une définition de composition.
   - *Incidences sur le schéma* : répercuter sur la définition du type d'entité concerné
   - *Incidences sur les données* : restructurer la représentation de la composition pour pouvoir "accueillir" les identifiants des instances du nouveau composant. La répercussion sur les instances existantes de la composition ne nous semble réalisable qu'en extension.
   - *Incidences sur les programmes* : Aucune
2. *ChgComposition.SuprComposant* : retirer une définition de composant d'une définition de composition
   - *Incidences sur le schéma* : répercuter sur la définition du type d'entité concerné
   - *Incidences sur les données* : restructurer la représentation de la composition pour ne plus permettre l'accès au composant supprimé.
   - *Incidences sur les programmes* : tous les programmes qui utilisaient la fonction *Rôle*, associée au composant supprimé, deviennent inopérants.

2.5 La modification des types d'entité

Les modifications peuvent porter sur la spécification d'un type d'entité ou sur l'ensemble des spécifications des types d'entités d'un schéma. En ce qui concerne un type d'entité, les opérations que l'on peut effectuer sur un type d'attribut et des compositions ayant déjà été traitées, elles ne seront pas considérées à nouveau. De ce fait, concernant les attributs, nous n'examinerons que l'extension ou la restriction de la liste des attributs attachée à un type d'entité.

2.5.1 Opérations sur un type d'entité

1. *ChgTypEntité.Nom* : Renommer un type d'entité
   - *Incidences sur le schéma* : Incorporer le renommage en propageant sur les définitions des types d'associations, de la relation de sous-type et, éventuellement, sur celles de certains attributs ou certaines compositions.

- *Incidences sur les données* : seul le nom de l'espace des identifiants du type d'entité doit être changé.
- *Incidences sur les programmes* : renommage dans tout programme utilisant le type d'entité concerné par l'opération.

2. *ChgTypEntité.SuperType* : changer le nom du super-type d'un type d'entité. Cette opération consiste à supprimer puis à ajouter un lien de sous-type (voir plus haut).

3. *ChgTypEntité.AjtAttribut* : Ajouter un type d'attribut à la liste des attributs caractérisant un type d'entité.

   - *Incidences sur le schéma* : Incorporer la modification à la définition du type d'entité concerné.
   - *Incidences sur les données* : Restructurer les instances du type d'entité pour pouvoir représenter les instances du type d'attribut. Mais, quelle valeur initiale lui affecter ?
   - *Incidences sur les programmes* : a priori aucune.

4. *ChgTypEntité.SuprAttribut* : Supprimer un type d'attribut de la liste des attributs caractérisant un type d'entité.

   - *Incidences sur le schéma* : Incorporer la modification à la définition du type d'entité concerné.
   - *Incidences sur les données* : Restructurer les instances du type d'entité pour rendre inaccessible l'attribut en question.
   - *Incidences sur les programmes* : ceux utilisant l'attribut en question deviennent inopérants.

2.5.2 Opérations sur l'ensemble des types d'entité d'un type de schéma

Elles consistent à ajouter ou à supprimer un type d'entité dans une spécification de schéma.

1. *ChgTypEntité.AjtTypEntité* : Ajouter une définition de type d'entité.

   - *Incidences sur le schéma* : Ajouter la définition du type d'entité et propager sur la définition de la relation de sous-type.
   - *Incidences sur les données* : Le nouveau type d'entité doit être "inséré" dans le graphe de la relation de sous-type. L'instance de la relation entre le type d'entité et son super-type est vide. Dans le cas où une définition intentionnelle de l'appartenance à la relation de sous-type peut être formulée, on pourrait automatiser le calcul de son instance initiale.
   - *Incidences sur les programmes* : Aucune

2. *ChgTypEntité.SuprTypEntité* : supprimer une définition de type d'entité. Appelons E_s la définition du type d'entité supprimé.

   - *Incidences sur le schéma* : supprimer la définition du type d'entité et propager sur les définitions
     - *de la relation de sous-type* : cela revient à supprimer le nœud E_s du graphe représentant la relation, c'est-à-dire à supprimer des liens de sous-type : ceux entre E_s et ses sous-types et celui entre E_s et son super-type (Les sous-types éventuels de E_s peuvent "raisonnablement" devenir des sous-types du super-type de E_s, comme schématisé sur la figure 2.1).

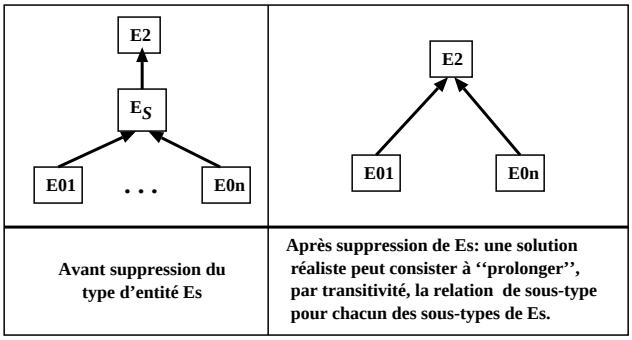

FIG. 2.1 – *Suppression de type d'entité et relation de sous-type*

- *des types d'associations*: celles ayant comme participant le type d'entité supprimé devraient disparaître de la définition du schéma.
- *des compositions*: la suppression de E_s peut invalider toutes les définitions des compositions dans lesquels E_s est utilisé comme nom de composant.
- *des attributs*: tous les attributs dont E_s est le domaine doivent être soit redéfinis soit supprimés.

- *Incidences sur les données*:
  1) les *entités* du type supprimé disparaissent: l'espace des identifiants et celui des instances du type d'entité sont détruits.
  2) les *instances des types d'association* auxquels participait le type d'entité disparaissent également,
  3) la restructuration de la *relation de sous-type* dépend de la position du type d'entité dans le graphe de la relation. La nouvelle instance de celle-ci peut, éventuellement, être calculée par transitivité (voir la figure 2.1),
  4) les instances des types d'entité utilisant le type d'entité supprimé dans la définition d'une *composition* ou comme *domaine d'un attribut* doivent être restructurées (voir les opérations *ChgComposition.SuprComposant* et *ChgTypEntité.SuprAttribut*).
- *Incidences sur les programmes*:

2.6 La modification des types d'association

Comme pour les types d'entité, la modification des types d'associations peut consister en la modification d'un type existant ou en l'extension ou la restriction de l'ensemble des types d'association d'un schéma. En outre, ne sont pas examinées ci-dessous, les modifications auxquelles peut être soumise la liste des attributs associée au type d'association, car elles sont similaires à celles étudiées dans le cadre de la modification de la liste des attributs d'un type d'entité.

2.6.1 Opérations sur un type d'association

1. *ChgTypAssociation.Nom*: Renommage d'un type d'association
   - *Incidence sur le schéma*: trivial
   - *Incidences sur les données*: l'espace des identifiants des instances du type d'association subit le même renommage que la définition du type d'association.

- *Incidences sur les programmes* : Aucune

2. *ChgTypAssociation.NomParticipant* : changer le nom d'un des types d'entité dans une définition d'un type d'association.

   - *Incidences sur le schéma* : incorporer le changement de nom
   - *Incidences sur les données* : nous ignorerons le cas trivial du simple renommage du nom du participant. Dans les autres cas, les instances des types d'association concernées ne peuvent, à notre avis, redéfinies qu'en extension.
   - *Incidences sur les programmes* : renommage du participant dans les programmes utilisant le type d'association concerné par la modification.

3. *ChgTypAssociation.Rôle* : renommage du rôle d'un type d'entité participant à la définition d'un type d'association.

   - *Incidences sur le schéma* : Répercuter le renommage dans la définition du type d'association.
   - *Incidences sur les données* : Aucune
   - *Incidences sur les programmes* : Le rôle étant utilisable comme un symbole de fonction d'accès, un renommage est nécessaire dans certains programmes.

4. *ChgTypAssociation.Cardinalité* : voir la discussion concernant le changement de la cardinalité d'un type d'attribut.

5. *AjtTypAttribut* : Ajouter un type d'attribut à la définition du type d'association.

6. *SuprTypAttribut* : Supprimer un type d'attribut de la définition.

2.6.2 Opérations sur l'ensemble des types d'association

1. *ChgTypAssociation.AjtTypAssociation* : Ajouter la définition d'un type d'association à la définition d'un schéma.

   - *Incidences sur le schéma* : Etendre l'ensemble des définitions des types d'associations par celle ajoutée.
   - *Incidences sur les données* : Aucune, l'ensemble des identifiants des instances du type d'association ajouté étant vide.
   - *Incidences sur les programmes* : Aucune.

2. *ChgTypAssociation.SuprTypAssociation* : Supprimer une définition de type d'association.

   - *Incidences sur le schéma* : supprimer le nom du type d'association de l'espace des noms correspondant.
   - *Incidences sur les données* : l'ensemble des instances du type d'association disparaît.
   - *Incidences sur les programmes* : naturellement, tous les programmes qui utilisaient le type d'association deviennent inopérants.

2.7 La modification d'un type de schéma

Toutes les opérations qui viennent d'être exposées sont applicables sur une type de schéma ou sur une de ses instances. Pour être complet, nous nous devons de proposer une opération de renommage d'un type de schéma, dont les répercussions peuvent se limiter à changer le nom de l'espace des instances du type de schéma objet du renommage.

3

Les règles de ré-écriture de structures

Les règles de ré-écriture de structure définies dans le chapitre 6 sont rappelées ci-desssous. Elles sont également illustrées graphiquement et par un exemple.

(R1) *Etendre la liste d'attributs d'un produit cartésien :*

$X(\mathrm{T}_1, \ldots, \mathrm{T}_n) \longrightarrow X(\mathrm{T}_1, \ldots, \mathrm{T}_n, \mathrm{T}_{n+1}, \ldots, \mathrm{T}_m)$

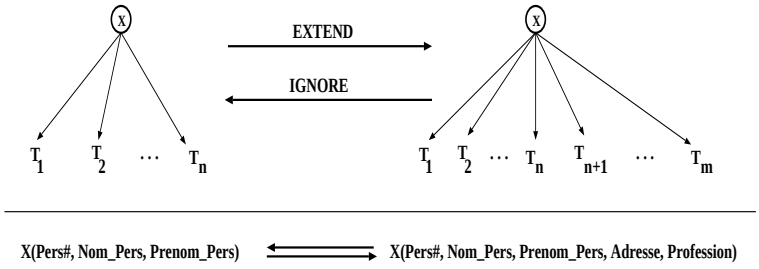

FIG. 3.1 – *Extension/Restriction de listes d'attributs ((R1), (R2))*

(R2) *Restreindre la liste d'attributs d'un produit cartésien :*

$X(\mathrm{T}_1, \ldots, \mathrm{T}_n, \mathrm{T}_{n+1}, \ldots, \mathrm{T}_m) \longrightarrow X(\mathrm{T}_1, \ldots, \mathrm{T}_n)$

(R3) *Eliminer un produit cartésien imbriqué dans un autre :*

$X(\mathrm{T}_1, \ldots, X(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), \ldots, \mathrm{T}_n) \longrightarrow X(\mathrm{T}_1, \ldots, \mathrm{T}_i^1, \ldots, \mathrm{T}_i^k, \ldots, \mathrm{T}_n)$

(R4) *Imbriquer un produit cartésien dans un produit cartésien :*

$X(\mathrm{T}_1, \ldots, \mathrm{T}_i^1, \ldots, \mathrm{T}_i^k, \ldots, \mathrm{T}_n) \longrightarrow X(\mathrm{T}_1, \ldots, X(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), \ldots, \mathrm{T}_n)$

(R5) *Distribuer un produit cartésien sur une union de types :*

$X(\mathrm{T}_1, \ldots, T_{i-1}, \cup(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), T_{i+1}, \ldots, \mathrm{T}_n)$
$\longrightarrow \cup(X(\mathrm{T}_1, \ldots, \mathrm{T}_{i-1}, \mathrm{T}_i^1, \mathrm{T}_{i+1}, \ldots, \mathrm{T}_n), \ldots, X(\mathrm{T}_1, \ldots, \mathrm{T}_{i-1}, \mathrm{T}_i^k, \mathrm{T}_{i+1}, \ldots, \mathrm{T}_n))$

(R6) *Introduire une union de types dans un produit cartésien :*

$\cup(X(\mathrm{T}_1, \ldots, \mathrm{T}_{i-1}, \mathrm{T}_i^1, \mathrm{T}_{i+1}, \ldots, \mathrm{T}_n), \ldots, X(\mathrm{T}_1, \ldots, \mathrm{T}_{i-1}, \mathrm{T}_i^k, \mathrm{T}_{i+1}, \ldots, \mathrm{T}_n))$
$\longrightarrow X(\mathrm{T}_1, \ldots, T_{i-1}, \cup(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), T_{i+1}, \ldots, \mathrm{T}_n)$

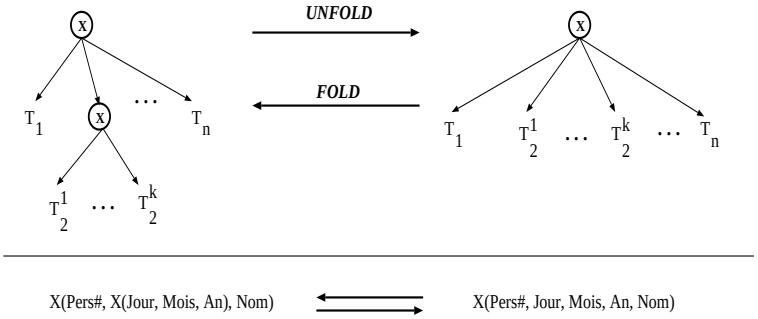

FIG. 3.2 – *Eliminer/Imbriquer un produit cartésien dans un autre ((R3), (R4))*

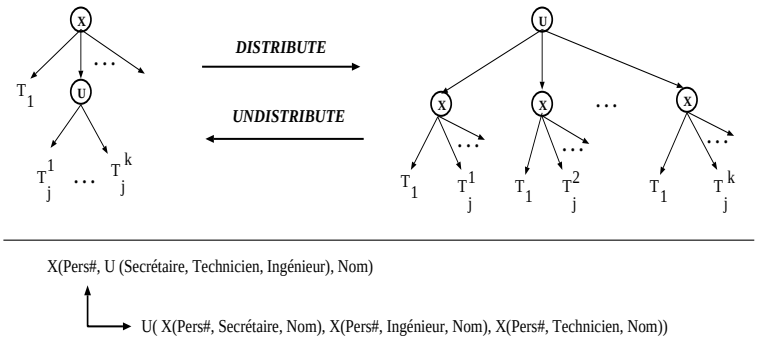

FIG. 3.3 – *Imbrication/Désimbrication un produit cartésien ((R5), (R6))*

(R7) *Désimbriquer un type ensemble dans un produit cartésien* :

$X(\mathrm{T}_1, \ldots, *(\mathrm{T}_i), \ldots, \mathrm{T}_n) \longrightarrow *(X(\mathrm{T}_1, \ldots, \mathrm{T}_i, \ldots, \mathrm{T}_n))$

(R8) *Imbriquer un type ensemble dans un produit cartésien* [10] :

$*(X(\mathrm{T}_1, \ldots, \mathrm{T}_i, \ldots, \mathrm{T}_n)) \longrightarrow X(\mathrm{T}_1, \ldots, *(\mathrm{T}_i), \ldots, \mathrm{T}_n)$

(R9) *Distribuer un constructeur d'ensemble sur une union de types :*

$*(\cup(\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_n)) \longrightarrow \cup(*(\mathrm{T}_1), *(\mathrm{T}_2), \ldots, *(\mathrm{T}_n))$

(R10) *Distribuer une union de type sur un ensemble :*

$\cup(*(\mathrm{T}_1), *(\mathrm{T}_2), \ldots, *(\mathrm{T}_n)) \longrightarrow *(\cup(\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_n))$

(R11) *Eliminer une union de types imbriquée dans une union de types :*

$\cup(\mathrm{T}_1, \ldots, \cup(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), \ldots, \mathrm{T}_n) \longrightarrow \cup(\mathrm{T}_1, \ldots, \mathrm{T}_i^1, \ldots, \mathrm{T}_i^k, \ldots, \mathrm{T}_n)$

(R12) *Imbriquer une union de types dans une union de types :*

$\cup(\mathrm{T}_1, \ldots, \mathrm{T}_i^1, \ldots, \mathrm{T}_i^k, \ldots, \mathrm{T}_n) \longrightarrow \cup(\mathrm{T}_1, \ldots, \cup(\mathrm{T}_i^1, \ldots, \mathrm{T}_i^k), \ldots, \mathrm{T}_n)$

10. Ces deux dernières opérations sont srtictement équivalentes à la désimbrication et à l'imbrication de relations dans le modèle relationnel de données NF^2 [1].

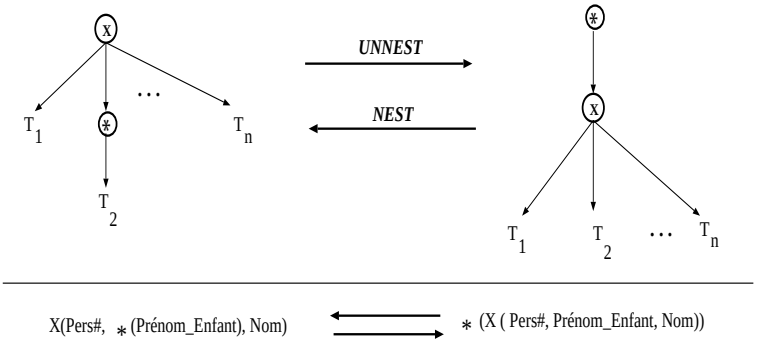

FIG. 3.4 – *Imbrication/Désimbrication un produit cartésien dans un ensemble ((R7), (R8))*

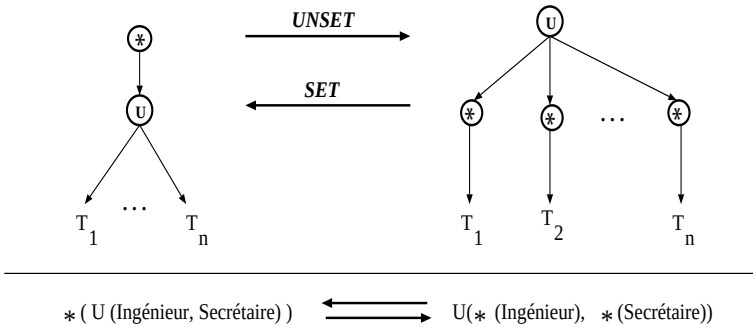

FIG. 3.5 – *Distributions de l'ensemble et de l'union ((R9), (R10))*

(R13) *Structures invariantes*, i.e. structures ne pouvant pas être réécrites en d'autres :

- $*(*(T))$
- $\cup(T_1, \ldots, X(T_i^1, \ldots, T_i^k), \ldots, T_n)$
- $\cup(T_1, \ldots, *(T_i), \ldots, T_n)$

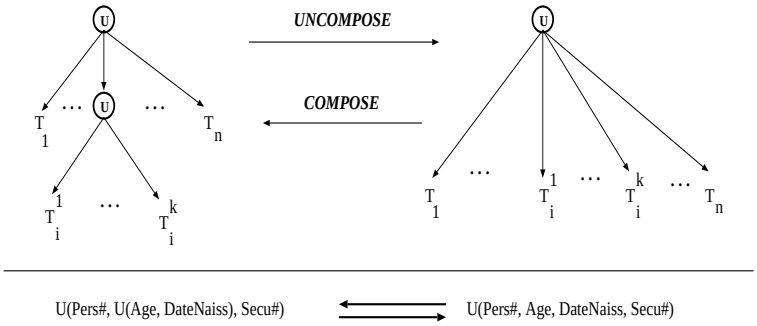

FIG. 3.6 – *Imbriquer/désimbriquer une union de types dans une union de types ((R11), (R12))*

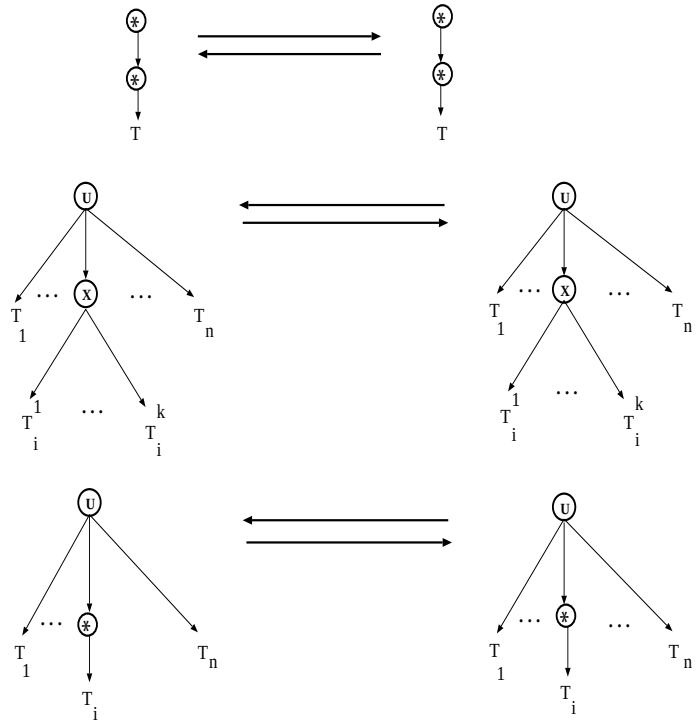

FIG. 3.7 – *Structures et règles invariantes (R13)*

Bibliographie

[1] Abiteboul (S.), Fischer (P.C.) et Eds (H.J. Schek). – *Nested Relations and Complex Objects in Databases*. – Springer-verlag, 1989, *Lecture Notes in Computer Science*, volume 361.

[2] Abiteboul (S.) et Hull (R.). – IFO: a Formal Semantic Database Model. *ACM Transactions On Database Systems*, vol. 12, n4, December 1987, pp. 525–565.

[3] Abiteboul (S.) et Hull (R.). – Restructuring Hierarchical Database Objects. *Theoretical Computer Science*, vol. 62, 1988, pp. 3–38.

[4] Adiba (M.) et Collet (C.). – Règles Actives dans un SGBD Orienté Objets. *In : [75]*. – Paris, France, Décembre 1993.

[5] Aiken (P.), Muntz (A.) et Richards (R.). – DOD Legacy Systems: Reverse Engineering Data Requirements. *CACM*, vol. 37, n13, May 1994, pp. 26–40.

[6] Ajila (S.), Basson (H.) et Boudjlida (N.). – Software Process Assistance: a Case Study of the Impact of Object Modification during Software Development. *In : Proceedings of the 1rst African Conference on Research in Computer Science*. – Yaoundé, Cameroun, October 1992.

[7] Albano (A.), Cardelli (L.) et Orsini (R.). – Galileo: a Strongly typed, Interactive Conceptual Language. *ACM Trans. On Database Systems*, vol. 10, n2, June 1985, pp. 230–260.

[8] Alchourrón (C.E.), Gärdenfors (P.) et Makinson (D.). – On the Logic of Theory Change: Partial Meet contraction and Revision Functions. *Journal of Symbolic Logic*, vol. 50, n2, June 1985, pp. 510–530.

[9] Alexandre (F.), Finance (J.P.) et Quéré (A.). – SPES : Un système de transformation de programmes logiques. *In : Colloque de programmation logique*. – Trégastel, France, 1988.

[10] Allen (J.), Fikes (R.) et Sandewall (E.) (édité par). – *Proceedings of the 2nd International Conference on the Principles of Knowledge Representation and Reasoning, KR'91*. – Cambridge, MA, April 1991.

[11] Ambriola (V.), Ciancarini (P.) et Montangero (C.). – Software Process Enactment in Oikos. *In : Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*. – Irvine, CA, December 1990.

[12] Arbaoui (S.). – *PEACE: un formalisme fondé sur une logique modale pour la modélisation et la mise en œuvre des processus logiciels e'volutifs et non-monotones*. – Thèse de PhD, Université Pierre Mendès France, Grenoble II, Juillet 1993. (Doctorat d'université).

[13] Atkinson (M.), Bancilhon (F.), de Witt (D.), Dittrich (K.), Maier (D.) et Zdonick (S.). – The Object-Oriented Database System Manifesto. *In : Proceedings of the First Conference on Deductive and Object-Oriented Databases*. – Kyoto, Japan, December 1989.

[14] Atkinson (M.P.), Bailey (P.J.), Chisholm (K.), Cockshott (P.) et Morrison (R.). – An approach to persistent programming. *Computer Journal*, vol. 26, n4, November 1983, pp. 360–365.

[15] Atkinson (M.P.), Buneman (P.) et Morrison (R.) (édité par). – *Data Types and Persistence*. – Springer Verlag, 1988, *Topics in Information Systems*.

[16] Atkinson (M.P.), Chisholm (K.) et Cockshott (P.). – *PS-Algol, an Algol with a Persistent Heap*. – Internal Report nCSR-94-81, Edinburgh, University of Edimburgh, Dept. of Computer Science, 1981.

[17] Balzer (R.). – A 15 Years Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, vol. 11, n11, November 1985, pp. 1257–1268.

[18] Bancilhon (F.) et Ramakrishnan (R.). – An Amateur's Introduction to Recursive Query Processing Strategies. *In : ACM SIGMOD*. – Washington DC, May 1986.

[19] Bandinelli (S.), Fuggetta (A.) et Ghezzi (C.). – Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering*, vol. 19, n12, December 1993, pp. 1128–1144. – Special Issue on Process Evolution.

[20] Banerjee (J.), Chou (H.T.), Kim (H.J.) et Korth (H.F.). – Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD Record*, vol. 16, n3, 1987, pp. 311–322.

[21] Basili (V.R.). – Viewing Maintenance as Reuse-Oriented Software Development. *IEEE Software*, vol. 7, n1, January 1990, pp. 19–25.

[22] Batini (C.), Lenzerini (M.) et Navathe (S. B.). – A Comparative Analysis of Methodologies for Databases Schema Integration. *ACM Computing Surveys*, vol. 18, n4, December 1986, pp. 323–364.

[23] Batory (D. S.) et al. – GENESIS: An Extensible Database Management System. *IEEE Trans. on Soft. Eng.*, vol. 14, n11, November 1988, pp. 1711–1730.

[24] Bayer (P.) et Fox (J.). – *State-of-the-Art Report on Reactive Processing in Database Technology and in Artificial Intelligence*. – Technical Report n ECRC-93-8, München, Germany, ECRC (European Computer-Industry Research Centre), 1993.

[25] Benali (K.), Boudjlida (N.), Charoy (F.), Derniame (J-C.), C.Godart, Ph.Giffiths, V.Gruhn, Ph.Jamart, Legait (A.), Oldfield (D.E.) et Oquendo (F.). – Presentation of the **ALF** project. *In : Proceedings of the International Conference on System Environment Development and Factories*. – Berlin, Germany, may 1989.

[26] Biggerstaff (T.), Mitbander (B.) et Webster (D.E.). – Program Understanding and the Concept Assigment Problem. *CACM*, vol. 37, n13, May 1994, pp. 72–82.

[27] Bonner (A.J.). – Hypothetical Datalog: negation and linear recursion. *In : ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'89)*, pp. 286–300. – Philadelphia, PA, March 1989.

[28] Bonner (A.J.) et Kifer (M.). – *Transaction Logic Programming (or A Logic of Procedural and Declarative Knowledge)*. – Technical Report nCSRI-270, University of Toronto, Computer Systems Research Institute, June 1994.

[29] Boudjlida (N.). – L'assistance dans les ateliers de logiciel. *In : 1er séminaire international de génie logiciel*. Université ES-SENIA - Oran - Algérie. – Novembre 1988.

[30] Boudjlida (N.). – *Formal Foundations for Value type quantification and Some Others Semantic Features*. – Technical Report nALF/NCY-NB/CT-4, ALF-Consortium, June 1990. Issue 0.1.

[31] Boudjlida (N.). – *Assessment of the ALF Project*. – Technical Report nALF/NCY-NB/WP-4/3-1-D1, CRIN-Nancy 1 University, ALF-Consortium, January 1992. Issue 1.1.

[32] Boudjlida (N.). – Knowledge in Interoperable and Evolutionary Systems. *In : KRDB'95, Workshop on "Reasoning about Structured Objets: Knowledge Representation Meets Databases"*. – Bielefeld, Germany, September 1995. (Position Paper).

[33] Boudjlida (N.) et Basson (H.). – Integration Mechanisms in ALF, a Process Model-Based Project Support Environment. *In : [152]*, pp. 315–324. – 1992.

[34] Boudjlida (N.) et Belhamissi (Y.). – Traitement de Requêtes Réparties : des Modèles à leur implémentation. *In : Actes des Journées Internationales des Sciences Informatiques, JISI'94*. – Tunis, Tunisie, Mai 1994.

[35] Boudjlida (N.), Benali (K.) et Charoy (F.). – *Assessment Criteria for the ALF Project*. – Technical Report nALF/NCY-NB/CT-21, CRIN-Nancy 1 University, ALF-Consortium, December 1991. Isssue 0.2.

[36] Boudjlida (N.), Bouneffa (M.A.) et Perrin (O.). – *Data Integration in a Legacy-Systems Migration Process*. – Research report, CRIN-UHP Nancy 1, January 1995. (submitted for publication to the 6th workshop "Empirical Studies of Programmers, Washington DC, January 1996).

[37] Boudjlida (N.) et Chabrier (J.J.). – Bases de données déductives et types abstraits de données : le système expérimental SINDBAD. *In : Actes du Congrés matériels et logiciels de 5ème génération*. – Paris - France, 1985.

[38] Boudjlida (N.), Charoy (F.), Derniame (J.C.) et Godart (C.). – Modeling of Software Process Assistance. *In : CASE'89 - 3rd Workshop on Computer-Aided Software Engineering*. – London, July 1989.

[39] Boudjlida (N.), Gervaise (O.), Godart (C.) et Derniame (J.C.). – Fundamental Concepts and Techniques for Environment Kernels - Invited Paper. *In : 2nd Congress on databases - Korea Information Science Society*. – Seoul - Korea, February 1988.

[40] Boudjlida (N.) et Khammaci (T.). – Knowledge-Based Software Assistant : A Knowledge Representation Model and its Implementation. *Intelligent Systems Review*, vol. 2, n 3/4, Spring/Summer 1990, pp. 54–68.

[41] Boudjlida (N.) et Khammaci (T.). – Towards an Object-Constructor Database Support for Software Environments. *In : Proceedings of 1991 SCS Western Multiconference*. – Anaheim - CA, January 1991.

[42] Boudjlida (N.) et Perrin (O.). – A Formal Framework and a Procedural Approach for Data Integration. *In : Proceedings of the International Conference on Systems Integration, ICSI'94*. pp. 476–485. – São Paulo, Brazil, August 1994.

[43] Boudjlida (N.) et Perrin (O.). – A Formal Framework and a Procedural Approach for Data Integration. *In : in [153]*, pp. 476–485. – 1994.

[44] Bouneffa (M.A.). – *Gestion d'Objets Complexes : Modélisation, Evolution, Intégration*. – Thèse de PhD, CRIN-Université Henri Poincaré Nancy 1, Mai 1995.

[45] Bouneffa (M.A.) et Boudjlida (N.). – Schema Evolution in CASE Databases. *In : Proceedings of the 2nd Intern'l Conf. on Software Quality Management*, pp. 663–678. – Edimburgh, July 1994.

[46] Bouzeghoub (M.) et Comyn-Wattiau (I). – View Integration by Semantic Unificaation and Transformation of Data Structures. *In : Proceedings of the 9th International Conference on Entity-Relationship Approach*. – Lausanne, Switzerland, October 1990.

[47] Bouzeghoub (M.), Gardarin (G.) et Métais (E.). – Database Desing Tool: an Expert System Approach. *In : Proceedings of the 11th Conference on Very Large DataBases*. – Stockholm, Sweden, 1985.

[48] Bouzeghoub (M.) et Métais (E.). – Semantic Modeling of Object Oriented Databases. *In : Proceedings of the International Conference on Very Large DataBases, VLDB'91*. – Barcelona, Spain, September 1991.

[49] Bouzeghoub (M.) et Simon (E.). – Integrity Specification and Enforcement in Databases. *Ingénierie des Systèmes d'Information*, Avril 1994. – Hermès-AFCET.

[50] Brodie (M.L.), Mylopoulos (J.) et Schmidt (J.W.) (édité par). – *On Conceptual Modelling - Perspectives from Artificial Intelligence, Databases and Programming Languages*. – Springer-Verlag, 1984, *Topics in Information Systems*.

[51] Brown (A.). – *Control Integration Through Message Passing in a Software Development Environment*. – Technical Report nCMU/SEI-92-TR35, Soft. Engineering Institute, 1992.

[52] Bry (F.). – Intensional updates: Abduction via Deductionj. *In : Proceedings of the International Conference on Logic Programming (ICLP'90)*. – Jerusalem, Israël, June 1990.

[53] Buneman (P.). – Data Types for Database Programming. *in [15]*, 1988, pp. 91–100.

[54] Butterworth (P.), Otis (A.) et Stein (J.). – The Gemstone Object Database Management System. *CACM*, vol. 34, n10, October 1991, pp. 64–77.

[55] Cagan (M.R.). – The HP Softbench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, June 1990, pp. 36–47.

[56] Canals (G.), Boudjlida (N.), Derniame (J.C.), Godart (C.) et Lonchamp (J.). – *ALF: A Framework for Building Process-Centered Software Engineering Environments*, chap. 7, pp. 153–187. – Research Studies Press, Taunton, Somerset England, 1994. In "Software Process Modeling and Technology", A. Finkelstein, J. Kramer and B.A. Nuseibeh, editors.

[57] Canals (G.), Charoy (F.), Godart (C.) et Molli (P.). – *P-Root & COO*: extending PCTE with new capabilities. *In : ICSE Workshop on Databases for Software Engineering*. – 1994.

[58] Cardelli (L.) et Wegner (P.). – On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, vol. 17, n4, December 1985, pp. 471–522.

[59] Carr (P.), Stevenson (R.), Alea (J.), Berthold (J.), Croucher (G.), Davis (M.), Dobbins (G.), Szarek (V.) et Webster (W.). – Implementation of a prototype CAIS environment. *Ada letters*, vol. 7, n2, March 1987.

[60] CASE88. – Proceedings of the Second International Workshop on Computer-Aided Software Engineering, Cambridge, MA, July 1988.

[61] Ceri (S.), Gottlob (G.) et Tanka (L.). – What you Always Wanted to Know about Datalog (and Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, n1, March 1989, pp. 146–166.

[62] Cerino (D.A.) et LaMonica (F.S.). – An overview of Rome Air Development Center's Software Life Cycle Support Environment. *In : Proceedings 5th Annual Knowledge-Based Software Assistant Conference*. Rome Air Development Center, pp. 1–12. – Syracuse, N.Y., September 1990.

[63] Chabrier (J.J.). – *Spécification et construction de systèmes orientés bases de données : Techniques et langages basés sur le concept de type abstrait*. – Nancy, France, Thèse de PhD, Université de Nancy 1, Octobre 1982. Doctorat ès-Sciences Mathématiques.

[64] Chabrier (J.J.), Boudjlida (N.), Hao (J.K.), Trousset (F.) et Jacquenet (F.). – *MASP Consistency Checker Requirements*. – Technical report, ALF Project, January 1990.

[65] Chakravarthy (U.S.), Grant (J.) et Minker (J.). – Foundations of Semantic Query Optimisation for Deductive Databases. *In : [146]*, pp. 241–242. – 1988.

[66] Charoy (F.) et al. – *PCIS Technical Study on Opportunities of Object-Oriented Technologies Applied to IPSEs*. – Issue 1.0, PCIS/TS/S5, October 1991.

[67] Charoy (F.) et Boudjlida (N.). – *From an Extended Entity-Relationship Data Model to an Object Oriented Data Model*. – Research Report n92-R-228, Nancy 1 University, CRIN, January 1992.

[68] Chen (P.P.). – The entity-relationship data model - Toward a unified view of data. *ACM - TODS*, vol. 1, n1, March 1976.

[69] Chimenti (D.), Gamboa (R.), Krishnamurthy (R.), Naqvi (S.), Tsur (S.) et Zaniolo (C.). – The *LDL* Prototype. *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, n 1, March 1990, pp. 76–90.

[70] Clifford (J.) et Croker (A.). – The Historical Relational Data Model (HRDM) and Algebra based on Lifespan. *In : Proceedings of the International Conference on Data Engineering*. pp. 528–537. – Los Angeles, CA, 1987.

[71] Codd (E.). – A relational Model for large shared databases. *CACM*, 1970 1970.

[72] Colton (M.). – *SYBASE System 10, the Foundation for Enterprise Client/Server Computing*. – Technical paper series, Sybase, Inc, 1994.

[73] Conradi (R.), Osjord (E.), Westby (P.H.) et Liu (C.). – Initial Software Process Management in EPOS. *Software Engineering Journal*, September 1991. – Special Issue on Software Environments and Factories.

[74] Dayal (U.), Buchmann (A.P.) et al. – The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD Record*, no1, March 1988.

[75] AFCET Comité Technique Bases de Données. – *Bases de Données Actives*. – Rapport technique, Paris, France, AFCET, Décembre 1993.

[76] AFCET Comité Technique Bases de Données. – *Ingénierie des Bases de Données : Migration, Intégration, Evolution (Database Engineering: Migration, Integration, Evolution)*. – Rapport technique, CNRS-Paris, AFCET, Septembre 1994.

[77] DEC. – DEC FUSE EnCASE Manual. – Digital Equipment Corporation, March 1993.

[78] Derniame (J.C.) et al. – *The MASP Concepts and their Rationales in the ALF Project*. – Technical Report nALF/NCY-JCD/WP-2/1/2-D3, ALF-Consortium, April 1990. Issue 1.1.

[79] Derniame (J.C.), Benali (K.), Boudjlida (N.), Godart (C.) et Lonchamp (J.). – Roles Cooperation through Software Process Instantiation. *In : Proceedings of the 6th International Software Process Workshop*, éd. par Press (IEEE Computer Society), pp. 77–82. – Hakodate, Japan, October 1990.

[80] Derniame (J.C.), Benali (K.), Boudjlida (N.), Godart (C.) et Lonchamp (J.). – Roles Cooperation through Software Process Instantiation. *In : [116]*, pp. 77–82. – 1990.

[81] O Deux et al. – The Story of O2. *IEEE Trans. on Knowledge and Data Eng.*, vol. 2, n1, March 1990, pp. 91–107.

[82] Collectif du projet inter PRC. – Gestion de l'évolutif et de l'incertain dans une base de connaissance. *In : Actes des 5èmes journées nationales d'intelligence artificielle*. PRC-GDR IA, CNRS, MESR, pp. 77–119. – Nancy, Février 1995.

[83] ECMA. – *A Reference Model for Frameworks of CASE Environments*. – Rapport technique n ECMA TR/55, European Computer Manufacturers Association, December 1990.

[84] Ehrig (H.), Kreowski (H.J.), Mahr (B.) et Padawitz (P.). – Algebraic Implementation of Abstract Data Types. *Theoretical Computer Science*, vol. 20, 1982, pp. 209–263.

[85] EIA. – CDIF, CASE Data Interchange Format Overview. – EIA/IS-106, March 1994.

[86] Fedchak (E.). – An Introduction to Software Engineering Environments. *In : Proceedings of the Computer Software and Applications Conference (COMPSAC)*. – Chicago, Illinois, October 1986.

[87] Feldman (S.I.). – Make, a program for maintaining computer programs. *Software Practice and Experience*, vol. 9, n4, April 1979.

[88] Finin (T.), Fritzson (R.) et McKay (D.). – A language and protocol to support intelligent agent interoperability. *In : Proceedings of the CE and CAS Washington'92 Conference*. – 1992.

[89] Finin (T.), Weber (J.), Wiederhold (G.), Genesereth (M.), Fritzson (R.), McKay (D.), McGuire (J.), Pelavin (R.) et Shapiro (S.). – Specification of the KQML Agent-Communication Language (plus Example Agent Policies and Architectures). – The DARPA Knowledge Sharing Initiative, External Interfaces Working Group (Draft version), February 1994.

[90] Finkelstein (A.), Kramer (J.) et Nuseibeh (B.A.) (édité par). – *Software Process Modeling and Technology*. – Taunton, Somerset England, Research Studies Press, 1994, *Advanced Software Development Series*.

[91] Frankel (R.). – *The ToolTalk Service*. – Technical report, SunSoft Inc., September 1991.

[92] Gallaire (H.), Minker (J.) et Nicolas (J.M.) (édité par). – *Advances in Database Theory*. – N.Y., Plenum Press, 1981volume 1.

[93] Gallaire (H.), Minker (J.) et Nicolas (J.M.) (édité par). – *Advances in Database Theory*. – N.Y., Plenum Press, 1984volume 2.

[94] Gallaire (H.), Minker (J.) et Nicolas (J.M.). – Logic and databases : a deductive approach. *ACM computing surveys*, vol. 16, n2, 1984.

[95] Gallo (F.), Minot (R.) et Thomas (I.). – The object management system of PCTE as a software engineering database management system. *ACM SIGPLAN Notices*, vol. 22, n1, 1986.

[96] Gardarin (G.) et Simon (E.). – Les systèmes de gestion de bases de données déductives. *TSI*, vol. 6, n5, 1987.

[97] Gardarin (G.) et Valduriez (P.). – *ESQL: an Extended SQL with Object and Deductive Capabilities*. – Research Report n1185, INRIA-Rocquencourt, March 1990.

[98] Geller (J.), Metha (A.), Perl (Y.), Neuhold (E.) et Sheth (A.). – Algorithms for Structural Schema Integration. *In : Proceedings of the 2nd Conference on Systems Integration*, éd. par Ng (P.A.), Ramamoorthy (C.V.), Seifert (L.C.) et Yeh (R.T.). pp. 604–614. – Morristown, N.J., June 1992.

[99] Genesereth (M.R.). – Knowledge Interchange Format. *In : [10]*, pp. 599–600. – Cambridge, MA, 1991. Position Paper.

[100] Gey (F.), Benson (W.) et Carthy (J.L. Mc). – IRDS Standards : A Superior Mechanism for CASE Information Repository and Exchange. *In : [60]*. – 1988.

[101] GIE EMERAUDE. – The Emeraude Environment, August 1987.

[102] Godart (C.). – Contribution à la modélisation des procédés de fabrication de logiciels: Support au travail coopératif. – Thèse de doctorat d'état, UHP Nancy 1, Janvier 1993.

[103] Godart (C.), Benali (K.), Boudjlida (N.), Charoy (F.) et Derniame (J.C.). – Les bases de données sur le chemin du génie du logiciel. *In : Actes des journées "Des bases de données aux bases de connaissances" - Sophia Antipolis*. – September 1987.

[104] Godart (C.), Canals (G.), Molli (P.), Lonchamp (J.) et Derniame (J.C.). – Extending Cooperating Transaction Model To Support Software Engineering Activities. *In : [152]*, pp. 184–192. – 1992.

[105] Goguen (J.A.), Thatcher (J.W.) et Wagner (E.G.). – An initial algebra approach to the specification, correctness and implementation of abstract data types. *in Current Trends in programming methodology*, vol. 4, 1978.

[106] Gruhn (V.). – *Validation and Verification of Software Process Models*. – Germany, Thèse de PhD, University of Dortmund, 1991.

[107] Guttag (J.V.) et Horning (J.J.). – The Algebraic Specification of Abstract Data Types. *Acta Informatica*, vol. 10, 1978, pp. 27–52.

[108] Guttag (J.V.), Horowitz (E.) et Musser (D.). – The Design of Data Types Specifications. *Current Trends in programming methodology*, vol. 4, 1978.

[109] Hainault (J.L.), Englebert (V.), Henrard (J.), Hick (J.M.) et Roland (D.). – Database Applications Evolution: the DB-MAIN Project. *In : "Ingénierie des Bases de Données : Migration, Intégration, Evolution"*, pp. 1–20. – CNRS-Paris, September 1994.

[110] Hammer (M.) et Leod (D. Mc). – Database description with SDM : a semantic database model. *ACM TODS*, vol. 6, n3, September 1981.

[111] Hao (J-K.), Trousset (F.) et Chabrier (J.J.). – Prototyping an Inconsistency Checking Tool for Software process Models. *In : Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*. – Capri, Italy, June 1992.

[112] Haton (J.P.). – Panorama des systèmes multi-agents. *In : Architectures avancées pour l'Intelligence Artificielle*, éd. par Haton (J.P.). – Paris, France, 1989.

[113] Haton (J.P.), Bouzid (N.), Haton (M.C.), Lâasri (B.), Lâasri (H.), Marquis (P.), Mondot (T.) et Napoli (A.). – *Le raisonnement en intelligence arificielle : Modèles, techniques et architectures pour les systèmes à bases de connaissances*. – Paris, InterEditions, 1991.

[114] Hull (R.) et King (R.). – Semantic Database Modeling : Survey, Applications and Research Issues. *ACM Computing Surveys*, vol. 19, n3, September 1987.

[115] Imielinski (T.). – Intelligent Query Answering in Rule Based Systems. *Journal of Logic Programming*, vol. 4, 1987, pp. 229–257. – also in [146].

[116] ISPW6. – "Support for the Software Process". – Proceedings of the 6th International Software Process Workshop, October 1990. Hakodate, Japan, Edited by IEEE Computer Society Press.

[117] Jaccheri (M.L.) et Conradi (R.). – Techniques for Process Model Evolution in EPOS. *IEEE Transactionse on Software Engineering*, vol. 19, n12, December 1993, pp. 1144–1156. – Special Issue on Process Evolution.

[118] Jamart (Ph.). – *Description of MASP/DL Denotational Semantics using SML*. – Technical Report nALF/UCL-PhJ/WP-3/5/3-D2, ALF-Consortium, November 1990. Issue 1.0.

[119] Kaiser (G.E.), Feller (P.H.) et Popovitch (S.S.). – Intelligent Assistance for software development and maintenance. *In : IEEE - Software*. – 1988.

[120] Katsumo (H.) et Mendelzon (A.O.). – On the Difference between Updating a Knowledge Base and Revisiting it. *In : [10]*, pp. 387–394. – Cambridge, MA, April 1991.

[121] Keller (A.M.) et Wilkins (M. Winslett). – On the use of an extended relational model to handle changing incomplete information. *IEEE Transaction on Software Engineering*, vol. SE-11, n7, July 1985, pp. 620–633.

[122] Kellner (M.I.) et Rombach (H.D.). – Comparisons of Software Process Descriptions. *In : [116]*, pp. 7–18. – 1990.

[123] Khammaci (T.), Boudjlida (N.) et Derniame (J.C.). – Un Modèle Objet-Relation pour la modélisation du procédé de développement de logiciel. *In : Actes des Troisièmes Journées Internationales "Le Génie Logiciel et ses Applications"*. – Toulouse - France, Decembre 1990.

[124] Kifer (M.), Lausen (G.) et Wu (J.). – *Logical Foundations of Object-Oriented and Frame-Based Languages*. – Technical Report n93/06, New York, Department of Computer Science SUNY at Stony Brook, June 1993.

[125] Kim (W.), Garza (J. F.), Ballou (N.) et Woelk (D.). – Architecture of the ORION Next-Generation Database System. *IEEE Trans. on Knowledge and Data Eng.*, vol. 2, n 1, March 1990, pp. 109–123.

[126] Konstantas (D.). – Object Oriented Interoperability. *In : ECOOP'93, 7th European Conference on Object Oriented Programming, Lecture Notes in Computer Science 707*, pp. 80–102. – Kaiserslautern, Germany, July 1993.

[127] Kowalski (R.A.). – Database Updates in the Event Calculus. *Journal of Logic Programming*, vol. 12, n1&2, January 1992, pp. 121–146.

[128] Kurbel (K.). – Integration-Oriented Data Reengineering Supporting Long-Term Information Systems Evolution and Business Process Reegineering. *In : in [153]*, pp. 466–475. – 1994.

[129] Lehman (M.M.). – Approach to a disciplined Development Process : the ISTAR Integrated Project Support Environment. *In : Proceedings of Int'l Workshop on the Software Process and Software Environments*. – March 1985.

[130] Lehman (M.M.). – Process models, process programs, programming support. *In : Proceedings of the 10th Int'l Conf. On Soft. Eng.* – April 1987.

[131] Liskov (B.H.) et Wing (J.M.). – Specifications and their Use in Defining Subtypes. *In : Proceedings of OOPSLA'93*, pp. 16–28. – 1993.

[132] Liskov (B.H.) et Zilles (S.N.). – Specification Techniques for Data Abstractions. *IEEE Transactions on Software Engineering*, vol. SE-1, n1, March 1975, pp. 7–19.

[133] Lohman (G. M.), Lindsay (B.), Pirahesh (H.) et Schiefer (K. B.). – Extensions to STARBURST: Objects, Types, Functions and Rules. *CACM*, vol. 34, n10, October 1991, pp. 94–109.

[134] Lonchamp (J.), Godart (C.) et Derniame (J.C.). – Les environnements intégrés de production de logiviels. *TSI*, vol. 11, n1, 1991, pp. 31–95.

[135] Maciaskek (L. A.). – AD/CYCLE Repository Manager from Object-Oriented Perspective. *ACM SIGSOFT, Software Engineering Notes*, vol. 16, n1, January 1991, pp. 49–53.

[136] Maier (D.), Stein (J.), Otis (A.) et Purdy (A.). – Development of an Object-Oriented DBMS. *In : Proceedinds OOPSLA Conf.*, éd. par ACM, pp. 472–481. – September 1986.

[137] Maindreville (C.de) et Simon (E.). – A Production Rule Based Approach to Deductive Databases. *In : 4th International Conference on Data Engineering, ICDE'88*. – Los Angeles, CA, February 1988.

[138] Maindreville (C.de) et Simon (E.). – Modelling Non Deterministic Queries and Updates in Deductive Databases. *In : 14th Conference on Very Large DataBases, VLDB'88*, éd. par bancilhon (F.) et DeWitt (D.J.), pp. 395–406. – Los Angeles, CA, August-September 1988.

[139] Manchanda (S.) et D.S.Warren. – A logic-based language for database updates. *In : Foundations of Deductive Databases and Logic Programming*, éd. par Minker (J.). pp. 363–394. – Los Altos, CA, 1988.

[140] Manna (Z.) et Waldinger (R.). – Synthesis: Dreams $\Longrightarrow$ Programs. *IEEE Trans. on Software Engineering*, vol. SE-5, n4, July 1979, pp. 294–328.

[141] Manna (Z.) et Waldinger (R.). – A Deductive Approach to Program Synthesis. *ACM TOPLAS*, vol. 2, n1, 1980, pp. 90–121.

[142] Markosian (L.), Newcomb (P.), Brand (R.) et Kitzmiller (T.). – Using an Enabling Technology to Reengineer Legacy Systems. *Communications of the ACM*, vol. 37, n5, May 1994, pp. 58–70.

[143] Martin (N.G.), Navathe (S.B.) et Ahmed (R.). – Dealing with Temporal Schema Anomalies in History Databases. *In : Proceedings of the Conference on Very Large DataBases*, éd. par Hammersley (P.), pp. 177–184. – Brighton, UK, September 1987.

[144] Masini (G.), Napoli (A.), Colnet (D.), Léonard (D.) et Tombre (K.). – *Object-Oriented Languages*. – Lyon, Academic Press, 1991.

[145] Matheus (C.J.), Chan (P.K.) et Piatetsky-Shapiro (G.). – Systems for Knowledge Discovery in Databases. *IEEE Trans. on Knowledge and Data Engineering*, vol. 5, n6, December 1993, pp. 903–912.

[146] Minker (J.) (édité par). – *Foundations of Deductive Databases and Logic Programming*. – Los Altos, CA, Morgan-Kaufmann, 1988.

[147] Mittermeir (R.T.), Klösch (R.R.) et Gall (H.C.). – Object Recovery from Procedural Systems for Changing the Architecture of Applications. *In : Proceedings of the International Conference on Systems Integration, ICSI'94*. pp. 148–157. – São Paulo, Brazil, August 1994.

[148] Moerkotte (G.) et Lockemann (P.C.). – Reactive Consistency Control in Deductive Databases. *ACM Transactions on Database Systems*, vol. 16, n4, December 1991, pp. 670–702.

[149] Morgenstern (M.). – The Role of Constraints in Databases, Expert Systems and Knowledge Representation. *In : Expert Database Systems*, éd. par Kerschberg (Larry). pp. 351–368. – The Benjamin Cummings Publishing Company, 1986.

[150] Naqvi (S.) et Krishnamurthy (R.). – Database updates in logic programming. *In : ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'88)*, pp. 251–262. – March 1988.

[151] Nebel (B.). – Belief Revision and Default Reasoning. *In : [10]*, pp. 417–428. – Cambridge, MA, April 1991.

[152] Ng (P.A.), Ramamoorthy (C.V.), Seifert (L.C.) et Yeh (R.T.) (édité par). – *Proceedings of the Second International Conference on Systems Integration, "Managing Large-Scale Integration in the 1990s"*. – Morristown, NJ, IEEE Comp. Society Press, June 1992.

[153] Ng (P.A.), Sobrinho (F Gattaz), Ramamoorthy (C.V.), Seifert (L.C.) et Yeh (R.T.) (édité par). – *Proceedings of the 3rd Intern'l Conf. on Systems Integration*. – São Paulo, Brazil, IEEE Comp. Society Press, August 1994.

[154] Nicolas (J.M.). – Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, vol. 18, n2, 1982, pp. 227–253.

[155] Oivo (M.) et Basili (V.R.). – Representing Software Engineering Models: The TAME Goal Oriented Approach. *IEEE Trans. on Soft. Eng.*, vol. 18, n10, October 1992, pp. 886–898.

[156] OMG. – *The Common Object Request Broker: Architecture and Specification*. – Technical Report n91.12.1, Revision 1.1, Object Management Group, December 1991.

[157] OMG. – *The Object Model Architecture Guide*. – Technical Report n91.11.1, Revision 2.0, Object Management Group, September 1992.

[158] Oquendo (F.S.). – *Une approche formelle des environnements de génie logiciel assisté par ordinateur : de la gestion des objets à l'autommaatisation des processus logiciels*. – Habilitation à diriger des recherches, Université Joseph Fourier, Grenoble 1, Janvier 1995.

[159] ORACLE7. – Présentation technique, December 1993. Nancy.

[160] Osterweil (L.). – Software Processes are Software Too. *In : Proceedings of the 9th Int'l Conf. on Soft. Eng.*, pp. 2–13. – Monterey, CA, March 1987.

[161] Osterweil (L.). – Automated support for the enactment of rigorously described software processes. *In : 4th Intern. Software Process Workshop*. – Moreonhamstead Devon UK, May 1988.

[162] Oszu (M. Tamer) et Valduriez (P.). – *Principles of Distributed Database Systems*. – Prentice Hall International, 1991.

[163] Pair (C.). – *Types Algébriques et sémantique algébrique des langages de programmation*. – Rapport interne n80-R-11, Nancy, France, CRIN, Février 1980.

[164] Peckham (J.) et Maryinski (F.). – Semantic Data Models. *ACM Computing Surveys*, vol. 20, n3, September 1988, pp. 153–189.

[165] Perrin (O.). – *Un Modèle d'Intégration d'Outils dans les Environnements de Développement de Logiciels (A Tool Integration Model in Software Engineering Environments)*. – Thèse de PhD, CRIN-Université Henri Poincaré Nancy 1, November 1994. (In French).

[166] Perrin (O.) et Boudjlida (N.). – Intégration d'outils dans les environnements de logiciels: l'aspect données. (Integrating Tools in Software Development Environments: the Data Perspective). *In : Proceedings of the 5th International Conference on Software Engineering and its Applications, Toulouse'92*, pp. 95–106. – Toulouse, France, December 1992. In French (92-R-229).

[167] Perrin (O.) et Boudjlida (N.). – Toward a Model for Persistent Data Integration. *In : Advanced Information Systems Engineering, 5th International Conference, CAiSE'93*, éd. par Rolland (C.), Bodart (F.) et Cauvet (C.). pp. 93–117. – Paris, June 1993.

[168] Perrin (O.) et O'Mullane (W.). – *Experiences with CORBA and Interoperability*. – Research Report n94-R-021, Nancy, France, CRIN-CNRS, April 1994.

[169] Perry (D.E.) et Kaiser (G.E.). – Models of software development environments. *In : Proceedings of the 10th Int'l Conf. on Soft. Eng.* – Singapoor, March 1988.

[170] Perry (D.E.) et Kaiser (G.E.). – Models of Software Development Environments. *IEEE Trans. on Software Engineering*, vol. 17, n3, March 1991, pp. 283–295.

[171] Petit (J.M.), Kouloumdjian (J.), Boulicaut (J.F.) et Toumani (F.). – Using Queries to Improve Database Reverse Engineering. *In : Proceedings of the 13th Intern'l Conf. on ER Approach*. – December 1994.

[172] Pirotte (A.), Delobel (C.) et Gottlob (G.) (édité par). – *Advances in Database Technology, Proceedings of the Third International Conference, EDBT'92*. – Vienna, Austria, Springer Verlag, LNCS 580, March 1992.

[173] Proch (K.). – *ORSEC: un Outil de Recherche de Spécifications Equivalentes par Comparaison d'exemples (An Example Comparison Tool for Equivalence of Specifications)*. – Nancy (F), Thèse de PhD, Université Nancy 1, 1982. (in French).

[174] Reiss (S.). – Connecting Tools Message Passing in the Field Environment. *IEEE Software*, July 1990, pp. 57–66.

[175] Reiter (R.). – Towards a Logical Reconstruction of Relational Database Theory. *In : [50]*, pp. 191–233. – 1984.

[176] Reiter (R.). – On Formalizing Database Updates: Preliminary Report. *In : [172]*, pp. 10–20. – 1992.

[177] Reiter (R.). – On Formalizing Database Updates: Preliminary Report. *In : Proceedings of the 3rd International Conference on Extending DataBase Technology*, pp. 10–20. – March 1992.

[178] Ridle (W.E.) et Williams (L.G.). – Modeling Software Development in the Large. *In : 4th Int'l Software Process Workshop*, éd. par Dowson. – Breckenridge, Colorado, November 1986.

[179] Schmidt (J.W.). – Some high-level language constructs for data of type relation. *ACM Transactions on Database Systems*, vol. 2, n3, September 1977, pp. 247–261.

[180] Sheth (A.P.) et Kashyap (V.). – So Far (Schematically) Yet So Near (Semantically). *In : Interoperable Database Systems*, éd. par D.K. Hsiao (E.J. Neuhold) et Sack-Davis (R). – North Holland, 1993.

[181] Sheth (A.P.) et Larson (J.A.). – Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. *ACM Computing Surveys*, vol. 22, n3, September 1990, pp. 183–236.

[182] Siegel (M.) et Madnick (S.). – A Metadata Approach to Resolving Semantic Conflicts. *In : Proceedings of the International Conference on Very Large Databases*. – Barcelona, Spain, September 1991.

[183] Skarra (A.H.) et Zdonik (S.B.). – The Management of Changing Types in an Object-Oriented Database. *SIGPLAN Notices*, vol. 21, n11, 1986, pp. 483–495. – Proceedings of OOPSLA'86.

[184] Snodgrass (R.) et Ahn (I.). – A Taxinomy of Time in Databases. *In : Proceedings of ACM SIGMOD International Conference on Management of Data*, éd. par Navathe (S.). pp. 236–246. – Austin, TX, May 1985.

[185] Soo (M.D.). – Bibliography on Temporal Databases. *ACM SIGMOD record*, vol. 20, n1, March 1991, pp. 14–23.

[186] Spaccapietra (S.) et Parent (C.). – View Integration: a Step Forward in Solving Structural Conflicts. *IEEE Transaction on Knowledge and Data Engineering*, 1993.

[187] Stonebraker (M.) et Kemnitz (G.). – The POSTGRES Next-Generation Database Management System. *CACM*, vol. 34, n10, October 1991, pp. 78–92.

[188] Stonebraker (M.), Rowe (L. A.) et Hirohama (M.). – The Implementation of POSTGRES. *IEEE Trans. on Knowledge and Data Eng.*, vol. 2, n1, March 1990, pp. 125–142.

[189] Storey (V.C.). – A Selective Survey of the Use of Artificial Intelligence for Database Design Systems. *Data and Knowledge Engineering*, vol. 11, 1993, pp. 61–102.

[190] Suzuki (M.), Iwai (A.) et Katayama (T.). – A Formal Model of Re-execution in Software Process. *In : Proceedings of the 2nd Internationaal Conference on the Software Process*. – Berlin, Germany, February 1993.

[191] Taylor (R.N.), Belz (F.C.), Clarke (L.), Osterweil (L.J.), Selby (R.W.), Wileden (J.C.), Wolf (A.) et Young (M.). – Foundations of the ARCADIA Environment Arcitecture. *In : Proceedings of the ACM SIGSOFT-SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. – Boston, 1988. Appeared also in Software Engineering Notes, 13(5), February 1989.

[192] Thieme (C.) et Siebes (A.). – An Approach to Schema Integration Based on Transformations and Behaviour. *In : Proceedings of the 6th Intern'l Conf. on Advanced Information Systems Engineering, CAiSE'94*, éd. par LNCS, pp. 297–310. – 1994.

[193] Thomas (I.). – The PCTE Initiative. – ESPRIT Technical Meeting, 1987.

[194] Thomas (I.). – PCTE Interfaces : Supporting Tools in Software Engineering Environments. *IEEE Software*, vol. 6, n6, November 1989, pp. 15–23.

[195] Thomas (I.). – Multiple Paradigm Descriptions of Software Process. *In : [116]*, pp. 205–206. – 1990.

[196] Thomas (I.), Boudier (G.), Gallo (F.) et Minot (R.). – An overview of PCTE and PCTE+. *In : Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. – November 1988.

[197] THOMSON. – *Specification du langage XRete (XRete Specification Language)*. – Technical report, THOMSON CSF, 1989. In French.

[198] Trousset (F.). – *Etudes de méthodes de contrôle de consistances basées sur des techniques de résolution (Study of Consistency Control Methods Based on Resolution Techniques)*. – Thèse de PhD, FRANCHE-COMTE University, Juin 1992. (In French).

[199] Tully (C.J.). – Prospects for Future Environments. *In : Proceedings of the 9th International Conference on Software Engineering*, pp. 340–341. – Monterey-CA, March-April 1987.

[200] Warboys (B.). – The IPSE 2.5 Project : Process Modelling as the basis for a Support Environment. *In : Proceedings of the International Conference on System Environment Development and Factories*. – Berlin - Western Germany, May 1989.

[201] Wasserman (A.I.). – Tool Integration in Software Engineering Environments. *In : Software Engineering Environments: International Workshop on Environments*. – Berlin, 1990.

[202] Wileden (J.C.) et Dowson (M.) (édité par). – *Proceedings of the International Workshop on the Software Process in Software Environments*. – Trabuco Canyon, CA, March 1985. Also as Software Engineering Notes, 11,4, August 1986.

[203] Winograd (T.). – Breaking the complexity barrier (again). – SIGPLAN SIGIR Interface Meeting on Programming Languages - Information Retrieval, 1973. November.