

THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

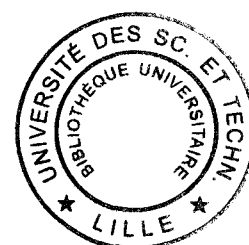
par

Laurent Debrauwer

Des vues aux contextes pour la structuration fonctionnelle de bases de données à objets en CROME

Thèse soutenue le 1 décembre 1998, devant la commission d'examen :

Président :	A. DERYCKE	TRIGONE
Directeurs de thèse :	J.M. GEIB	LIFL
	B. CARRE	LIFL
Rapporteurs :	J.P. GIRAUDIN	IMAG-LSR
	C. BOCKSENBAUM	LIRMM
	T. LIBOUREL	LIRMM
Examineurs :	A. NAPOLI	LORIA
	O. CARON	Université d'Artois



Remerciements

Je tiens à remercier Bernard Carré pour l'aide considérable qu'il m'a apportée au cours de toutes ces années de thèse. Le chemin conduisant à ce mémoire fut long à trouver et sans son soutien, celui-ci n'aurait jamais vu le jour. Je tiens à dire combien j'ai apprécié son enthousiasme, son exigence, sa modestie et l'ampleur de ses connaissances scientifiques. Les fréquentes réunions que nous avons tenues furent pour moi une source considérable d'enseignement.

Je remercie Alain Derycke pour m'avoir fait l'honneur de présider ce jury.

Je remercie Jean-Marc Geib pour m'avoir accueilli au sein de son équipe lorsque je suis arrivé au tout début, voulant faire une thèse tout en gardant mon statut d'ingénieur indépendant. Je suis très honoré de sa présence dans le jury.

Je remercie Jean-Pierre Giraudin, Claude Bocksbaum et Thérèse Libourel d'avoir accepté de rapporter ce travail, d'avoir étudié avec la plus grande attention ce mémoire et pour toutes les remarques qu'ils m'ont apportées.

Je remercie Amédéo Napoli d'avoir accepté d'être examinateur et pour l'intérêt porté à nos travaux.

Je remercie Olivier Caron pour l'intérêt qu'il a manifesté lorsque je suis venu présenter mes travaux à l'Université d'Artois. Qu'il soit remercié pour son aide dans la réalisation du prototype, pour ses conseils et sa participation au jury.

Je tiens à remercier Gilles Vanwormhoudt pour les nombreuses discussions que nous avons eues et pour ses nombreux conseils. Gilles m'a beaucoup aidé par les très nombreuses remarques pertinentes qu'il fit sur mes travaux.

Je remercie aussi Lenneke Dekker qui, malheureusement, nous a quittés peu de temps après avoir soutenu sa thèse.

Enfin, je remercie tous les membres de l'équipe GOAL ainsi que toutes les personnes du LIFL et du LabOGP. Que mes parents et mes amis soient aussi remerciés.

Table des matières

Introduction	11
Chapitre 1 - Descriptions multiples d'entités de bases de données	15
1.1. Introduction	15
1.2. Structuration en fonctions d'une organisation complexe	15
1.3. Informations communes et informations spécifiques aux fonctions	16
1.4. Besoin de prise en compte de la structuration en fonctions au sein de la base de données du SI	17
1.4.1. Prise en compte des fonctions de la conception à l'exploitation du SI	17
1.4.2. L'approche ANSI/X3/SPARC	17
1.4.3. La conception par intégration de vues	18
1.4.4. L'approche prônée en CROME	19
1.5. Orthogonalité entités/fonctions	20
1.5.1. Conception « classique » du schéma du système d'information	20
1.5.2. Les entités du domaine sont orthogonales aux fonctions	22
1.6. Prise en compte de la structuration en fonctions de l'organisation au sein des entités et classes du domaine	22
1.6.1. Descriptions multiples des entités du domaine	22
1.6.2. Instances détenant les informations fonctionnelles et descriptions multiples : notions d'extension et d'état fonctionnels	24
1.7. Partage des informations	26
1.7.1. Besoin de partage des informations	26
1.7.2. Problèmes issus du partage des informations	27
1.8. Cycle de vie de participation d'une instance et d'un couple d'instances à un ensemble de contextes fonctionnels	28
1.8.1. Cycle de vie de participation d'une instance à un ensemble de contextes	28
1.8.2. Généralisation à un couple d'instances	29
1.9. Intégration d'un composant « base de données »	31
Chapitre 2 - Les vues dans les bases de données à objets	33
2.1. Introduction	33
2.2. Vues « utilisateurs » lors de l'exploitation de la base de données	33
2.2.1. Besoins	33
2.2.2. Statut des classes-vues	34
2.2.3. Techniques par insertion des classes-vues dans la hiérarchie des classes	35
2.2.3.1. Définition des classes-vues (COCOON et MultiView)	35
2.2.3.1.1. Modèle de données	35
2.2.3.1.2. Algèbre « objet »	38
2.2.3.1.3. Exemples	41
2.2.3.2. Classification (COCOON et MultiView)	44
2.2.3.3. Schémas-vues (MultiView)	49
2.2.3.3.1. Définition	49
2.2.3.3.2. Fermeture du schéma-vue par rapport aux domaines des attributs	50
2.2.3.3.3. Validité du schéma-vue par rapport à la relation is-a	51
2.2.4. Techniques sans insertion des classes-vues dans la hiérarchie des classes	52
2.2.5. Autres travaux	55

2.2.6. Vues et descriptions multiples d'entités	56
2.2.6.1. Techniques par extraction lors de l'exploitation	56
2.2.6.2. Granularité fine des classes-vues	57
2.3. Intégration de vues	58
2.3.1. Besoins et objectifs	58
2.3.2. Méthode d'intégration	58
2.3.2.1. Aperçu de la méthode	58
2.3.2.2. Conflits lors de l'intégration	59
2.3.2.3. Obtention d'un référentiel commun et fusion des descriptions	60
2.3.3. Absence de structuration du schéma conceptuel issu de la conception vis-à-vis des différents utilisateurs	61
2.4. Bilan	63
Chapitre 3 - L'approche CROME dans le cadre des bases de données	65
3.1. Introduction	65
3.2. Choix du modèle « objet » de l'ODMG dans le cadre des bases de données	67
3.2.1. Modèle de l'ODMG	67
3.2.1.1. Caractéristiques importantes du modèle « objet » de l'ODMG	68
3.2.1.2. Langage d'interrogation OQL	71
3.2.2. Prise en compte des contraintes de cardinalités	72
3.2.3. Bilan sur le modèle choisi	73
3.3. Présentation des principes généraux de CROME	74
3.3.1. Rappel sur l'approche multi-intensionnelle	74
3.3.2. Référentiel d'entités orthogonales aux fonctions	74
3.3.3. Plans de description des entités	75
3.3.3.1. Plan de base	75
3.3.3.2. Plans fonctionnels	78
3.3.4. Héritage local dans un plan fonctionnel	81
3.3.5. Descriptions fonctionnelles	84
3.4. Interaction entre fonctions	87
3.4.1. Accès inter-plans	87
3.4.2. Attributs et relations du plan de base	91
3.5. Contraintes de participation aux fonctions	95
3.5.1. Prédicat de participation	96
3.5.2. Extension fonctionnelle d'une classe	97
3.5.3. Etat fonctionnel d'une instance	99
3.5.4. Héritage du prédicat de participation	101
3.5.5. Contextes	103
3.5.5.1. Contexte de base	103
3.5.5.2. Contextes fonctionnels	103
3.5.5.3. Contextes d'instanciation d'une classe	104
3.5.6. Dépendance entre contextes fonctionnels due aux accès inter-plans	106
3.5.6.1. Définition et conséquence sur les contraintes de participation	106
3.5.6.2. Restriction sur les prédicats de participation due à la dépendance provoquée par les accès inter-plans	109
3.5.7. Préservation de la structuration en classes du contexte de base dans les contextes fonctionnels	109
3.6. Contraintes d'intégrité	111
3.6.1. Type de contraintes étudiées	111
3.6.2. Contraintes d'intégrité référentielle	112
3.6.2.1. Contraintes d'intégrité référentielle au sein d'un contexte	113

3.6.2.1.1. Intégrité référentielle au sein du contexte de base	113
3.6.2.1.2. Intégrité référentielle au sein d'un contexte fonctionnel	114
3.6.2.2. Intégrité référentielle dans le cas de relations partagées	115
3.6.2.3. Restriction sur les prédicats de participation due au respect des CIR d'une relation partagée	118
3.6.3. Contraintes de cardinalité	118
3.6.3.1. Contraintes de cardinalité au sein d'un contexte	119
3.6.3.1.1. Contraintes de cardinalité au sein du contexte de base	119
3.6.3.1.2. Contraintes de cardinalité au sein d'un contexte fonctionnel	119
3.6.3.2. Contraintes de cardinalité de relations partagées	120
3.6.3.3. Restriction sur les prédicats de participation due au respect des contraintes de cardinalité minimale des relations partagées	128
3.6.4. Contraintes d'intégrité statiques sur les propriétés des classes au sein d'un contexte fonctionnel	128
3.6.4.1. Expression et localisation	128
3.6.4.2. Violation et validité des contraintes sur les propriétés	130
3.6.5. Contraintes de participation et contraintes d'intégrité	132
3.7. Propriétés d'un schéma de base de données conçu en CROME	134
3.7.1. Modularité interne des classes du domaine	134
3.7.2. Généricité du schéma de base	135
3.7.3. Interaction entre contextes fonctionnels	137
3.7.4. Les contextes fonctionnels ont un statut de schéma de classes	139
Chapitre 4 - Mises en œuvre	141
4.1. Introduction	141
4.2. Formulation des contextes fonctionnels en schémas-vues	141
4.2.1. Mise en œuvre par extraction	143
4.2.1.1. Définition des classes-vues relatives aux descriptions fonctionnelles	143
4.2.1.2. Classification des classes-vues	146
4.2.1.3. Réalisation des contextes par les schémas-vues de MultiView	149
4.2.2. Mise en œuvre par augmentation de capacité	150
4.2.2.1. Définition des classes-vues relatives aux descriptions fonctionnelles	150
4.2.2.2. Classification des classes-vues	153
4.2.2.3. Réalisation des contextes par les schémas-vues de MultiView	156
4.2.3. Réalisation des contraintes de participation des instances	157
4.2.4. Conclusion	159
4.3. Mise en œuvre dans les SGBD à objets couplés au langage Java	163
4.3.1. Utilisation du prototype CromeJava	164
4.3.1.1. Outil de génération de schémas en Java	164
4.3.1.2. Le browser de schémas	164
4.3.1.3. Prise en compte des entités locales aux fonctions	166
4.3.2. Génération des schémas en Java	166
4.3.2.1. Interfaces fonctionnelles et classes d'implantation	166
4.3.2.1.1. Principes	166
4.3.2.1.2. Interfaces fonctionnelles et de base	167
4.3.2.1.3. Classes d'implantation	168
4.3.2.2. Paquetages fonctionnels	169
4.3.2.3. Préservation de la structuration en classes issue de la conception en CROME au sein des paquetages générés	170

4.3.3. Gestion des extensions et des extensions fonctionnelles dans le cadre du SGBD à objets POET	172
4.3.3.1. Principales caractéristiques du couplage de POET avec Java	172
4.3.3.2. La classe « CromeExtent »	173
4.3.3.3. Calcul des extensions fonctionnelles	174
4.3.3.4. Gestion des exceptions	175
4.3.4. Conclusion	175
4.4. Bilan	176
Chapitre 5 - Extensions de l'approche CROME	177
5.1. Introduction	177
5.2. Cycle de vie de participation d'instances aux contextes	178
5.2.1. Notion de contexte et de dynamique d'instance en Farandole	178
5.2.1.1. Contexte sémantique (en Farandole 2)	178
5.2.1.2. Expression de la dynamique en Farandole	181
5.2.1.2.1. Farandole 1	181
5.2.1.2.2. Contextes dynamiques de Farandole 2	182
5.2.2. Prise en compte du cycle de vie fonctionnel des instances dans l'approche CROME	184
5.2.2.1. Interprétation du réseau de Petri représentant le cycle de vie fonctionnel d'une instance	184
5.2.2.2. Restrictions apportées au cycle de vie fonctionnel	184
5.2.2.3. Traduction en CROME du cycle de vie fonctionnel d'une instance	186
5.2.2.3.1. Traduction dans le cas de transitions d'alternance	189
5.2.2.3.2. Traduction dans le cas de transitions d'alternance et de dépendance	189
5.2.2.3.3. Cas des extrémités	192
5.2.2.4. Cycle de vie fonctionnel d'un couple d'instances	193
5.2.2.4.1. Définition de la notion de couple d'instances	193
5.2.2.4.2. Traduction en CROME du cycle de vie fonctionnel d'un couple d'instances	197
5.3. Intégration d'un composant « base de données » existant lors de la conception	198
5.3.1. Démarche d'intégration de composants « bases de données » existants lors de la conception	198
5.3.1.1. Besoins et objectifs	198
5.3.1.2. Problématique de l'intégration de composants : architecture d'une base de données fédérée	199
5.3.1.2.1. Problèmes spécifiques à l'intégration de composants : différences avec l'intégration de schémas-vues non peuplés	199
5.3.1.2.2. Architecture d'une base fédérée	200
5.3.1.3. Techniques d'intégration de composants existants	202
5.3.1.3.1. Approche par classes-vues préservant l'identité dans le cadre de COCOON	203
5.3.1.3.2. Approche par génération d'instances	205
5.3.2. Intégration d'un composant lors de la conception d'une base de données multi-fonctionnelle en CROME	206
5.3.2.1. Etablissement de la coréférence entre instances du composant et de la base multi-fonctionnelle	207
5.3.2.2. Intégration du composant	211
5.3.2.2.1. Statut de contexte fonctionnel pour le composant	211
5.3.2.2.2. Articulation du composant	213

5.3.2.3. Reconstitution de la structure de la base multi-fonctionnelle au sein du composant	214
5.4. Conclusion	219
Conclusion générale	221
Annexe A - SI d'une entreprise de fabrication et distribution	227
Annexe B - Interfaces destinées aux clients des contextes du SI	245
Annexe C - Modèle CROME dans le cadre des bases de données	255
Annexe D - Algorithme de calcul des extensions fonctionnelles	269
Bibliographie	271

Introduction

Dès que l'entreprise a dépassé le stade initial de la petite organisation peu structurée, sa croissance impose une certaine spécialisation des tâches et leur regroupement en *fonctions*. L'entreprise éprouve alors le besoin d'adopter une structure qui peut se définir comme « l'ensemble des fonctions et des relations déterminant formellement les missions que chaque unité de l'organisation doit accomplir, et les modes de collaboration entre ces unités » [Charpentier97]. La structuration fonctionnelle qui consiste à répartir les tâches et missions en fonction d'une spécialisation conférée à chaque acteur de l'entreprise est la plus fréquemment rencontrée.

Le principal inconvénient de cette structuration fonctionnelle est qu'elle engendre des cloisonnements et pose ainsi des problèmes de coordination et de communication transversale entre fonctions. Généralement, le dirigeant est le seul qui soit susceptible d'assurer la cohésion globale de l'organisation. Cependant, il arrive un seuil où la taille de l'entreprise ayant encore crû, il devient difficile pour le dirigeant d'assurer seul cette mission de cohésion. Il est alors nécessaire de formaliser des processus de gestion destinés à organiser, coordonner, animer et contrôler les différentes fonctions de la structure. L'information est à la base de tous ces processus de gestion. [Charpentier97] précise : « Elle (l'information) constitue aussi un des fondements essentiels de la cohérence de l'organisation. L'efficacité globale du système entreprise dépend en effet de la manière dont vont s'articuler ses différentes activités, dont vont se prendre les décisions et, en fin de compte, dont va se structurer l'information. »

L'information apparaît donc comme un support essentiel pour les organisations structurées. Son acquisition, son stockage, son traitement, son exploitation sont réalisés au sein de l'entreprise par le Système d'Information (SI). Le SI est donc l'un des piliers de l'organisation destinée à garantir son fonctionnement et sa cohérence. Il doit prendre en compte les fonctions de la structure : fournir aux fonctions de l'organisation les informations qui leur sont nécessaires est l'un des objectifs du SI d'une entreprise cités dans [Tardieu83]. On retrouve le besoin de structuration de l'information citée ci-dessus.

L'informatisation du SI doit également préserver cette prise en compte des fonctions de l'organisation. Le développement non contrôlé de l'informatisation, souvent accéléré par l'arrivée de l'informatique personnelle, a conduit à la mise en œuvre de solutions répondant aux besoins d'une fonction, voire même d'une partie de ses besoins. La réaction des entreprises est de mener maintenant une informatisation intégrée répondant localement aux besoins des fonctions. Cette réaction est notamment à l'origine des nombreux travaux sur la fédération de composants « bases de données » pré-existants. Un SI informatisé et intégré doit permettre à chaque fonction de manipuler les informations qui lui sont spécifiques ainsi que les informations communes à l'organisation.

Pour agir, communiquer et mémoriser, les acteurs de l'organisation ont besoin d'une interprétation commune des informations qu'ils utilisent [Tardieu83]. L'« Univers du Discours » est alors utilisé pour désigner les choses et les événements auxquels se réfère cette interprétation commune à l'organisation. [Tardieu83] précise aussi : « Un univers du discours est perçu comme contenant des objets réels ou abstraits que nous appellerons *entités* ». Ces entités de l'organisation forment son domaine. Elles sont communes à l'ensemble des fonctions de l'organisation qui deviennent ainsi autant de *contextes de participation* de ces entités. Le domaine peut ainsi être considéré comme un *référentiel* commun constitué d'entités

orthogonales aux fonctions. Ce référentiel peut alors servir de support pour assurer la transmission et le partage des informations entre fonctions, deux notions essentielles pour garantir la cohésion de l'organisation.

Le travail réalisé dans ce mémoire porte sur les bases de données qui sous-tendent le SI d'une entreprise. Notre objectif est de prendre en compte la structuration fonctionnelle de l'organisation sous-jacente au sein des bases de données afin qu'elles répondent aux objectifs du SI cités ci-dessus. Ce besoin de considérer les fonctions de l'organisation apparaît clairement dans les techniques de vues des bases de données. La nécessité d'élaborer et d'exploiter des schémas externes est assez ancienne : l'architecture ANSI/X3/SPARC [ANSI75] remonte quasiment aux débuts des travaux sur les bases de données relationnelles. Elle concerne essentiellement la phase d'exploitation des bases de données. Les méthodes de conception par intégration de vues ont fait leur apparition à la même époque (1978). Rappelons qu'elles ont pour objectif de construire le schéma conceptuel à partir d'un ensemble de schémas-vues d'utilisateurs. Elles s'appliquent principalement au cas des bases de données dont le schéma est trop important pour pouvoir être conçu directement, ce qui est souvent le cas du système d'information d'une organisation complexe.

Les concepteurs des premières bases de données à objets n'ont pas toujours pris en compte ce besoin. Pour la plupart, ces bases de données sont issues d'un langage de programmation à objets auquel elles ajoutent des notions spécifiques des bases de données, comme la persistance, la gestion des extensions et des transactions, la sécurité, un langage de requêtes, etc. C'est le cas de la très grande majorité des SGBD à objets commerciaux actuels. Cependant, il est apparu nécessaire de doter les bases de données à objets des techniques de vues à la façon des bases de données relationnelles. Ceci a donné lieu à de nombreux travaux sur lesquels nous aurons l'occasion de revenir. De même, des travaux ont été menés pour adapter les méthodes de conception par intégration de vues à l'approche « objet ». Nous aurons également l'occasion d'y revenir. Cependant, que les vues soient utilisées dans les bases de données relationnelles ou dans les bases de données à objets, la structuration fonctionnelle reste externe à la base.

Nous retenons l'orthogonalité entre entités et fonctions pour structurer de façon interne la base de données, ce qui se traduit, de façon homogène, par la structuration interne et fonctionnelle des entités. Nous faisons le choix de l'approche « objet » et décrivons chaque entité du domaine par une classe unique. Nous pouvons alors établir que cette orthogonalité nous amène aux problèmes suivants :

- conférer des descriptions multiples et systématisées aux entités relativement aux fonctions ;
- déterminer pour chaque fonction et chaque classe du domaine un ensemble d'instances de cette classe détenant des informations de cette fonction et donc y participant. Ceci conduit au besoin de pouvoir faire évoluer les multiples descriptions conférées à chaque instance ;
- partager et transmettre des informations entre fonctions et réaliser ce partage au sein des entités.

Nous proposons d'appliquer notre approche CROME pour répondre à ces problèmes dans le cadre des bases de données à objets. CROME est un support pour spécifier des schémas de bases de données en prônant une conception d'objets par fonctions. Dans ce mémoire, CROME est introduit au travers d'une extension du modèle « objet » de l'ODMG¹ basée sur les principes de l'approche multi-intensionnelle dont relève CROME.

1. Ce modèle a justement pour objectif la spécification à un niveau logique de bases de données à objets.

L'application de CROME conduit à systématiser, dès la conception, les multiples descriptions des entités par des plans. Le référentiel des entités est introduit par un plan de base sous la forme d'un schéma de classes. L'ensemble des descriptions des entités relatives à une fonction est détenu par un plan fonctionnel. Le partage et la transmission d'informations entre fonctions sont réalisés par des mécanismes internes aux entités. Ceux-ci permettent de spécifier le partage des informations communes à toutes les fonctions ainsi que la transmission d'informations spécifiques à une fonction vers une autre fonction. L'ensemble des instances d'une classe participant à une fonction est déterminé par un ensemble de contraintes de participation qui respecte les contraintes d'intégrité référentielle et de cardinalité. Pour une fonction, la spécification de ces différents éléments (plan de base, plan fonctionnel, mécanismes de partage, contraintes de participation) définit précisément le contexte fonctionnel correspondant.

Nous montrons alors comment cette structuration fonctionnelle de toutes les entités du domaine apporte une réponse aux problèmes posés par la granularité fine des vues. A l'exploitation, les contextes fonctionnels constituent les schémas pertinents pour les différents clients. Ils sont directement issus de la conception et n'ont pas à être élaborés comme dans le cas des techniques de vues classiques.

Le mémoire est organisé en cinq chapitres. Dans le premier chapitre, nous présentons l'ensemble de la problématique de prise en compte de la structuration de l'organisation. Après avoir confronté celle-ci avec les propositions de l'architecture ANSI/X3/SPARC et des méthodes de conception par intégration de vues, nous expliciterons l'orthogonalité des entités vis-à-vis des fonctions. Dans le cadre de l'approche « objet », cette orthogonalité nous amènera à expliciter la problématique introduite ci-dessus concernant la nécessité de conférer de multiples descriptions aux entités, de déterminer les ensembles d'instances d'une classe participant à une fonction ainsi que les besoins de partage d'informations entre fonctions. Nous étudierons aussi la problématique relative à deux extensions de l'approche CROME, à savoir :

- représenter le cycle de vie de participation d'une instance dans les différents contextes ;
- prendre en compte un composant « base de données » existant et déjà peuplé lors de la conception. Ce composant est issu du besoin de l'une des fonctions de l'organisation d'informatiser sa partie du système d'information avant qu'une informatisation globale du SI de l'entreprise soit mise en œuvre.

Dans le deuxième chapitre, nous examinons les techniques de vues des bases de données à objets. Dans un premier temps, nous nous focalisons sur les techniques de vues « utilisateurs » mises en œuvre lors de l'exploitation d'une base de données. Ensuite nous étudions la conception du schéma d'une base de données par intégration de vues.

L'approche CROME fait l'objet du troisième chapitre. Nous y explicitons et formalisons les différentes notions de l'approche qui ont été présentées ci-dessus. Nous illustrons l'approche au travers de l'exemple du SI d'une entreprise de fabrication et de distribution. L'extension du modèle de l'ODMG donne lieu à une extension de son langage de spécification « ODL », extension que nous désignons ensuite « CromeODL ». Il faut noter que nous abordons également l'étude des contraintes d'intégrité sur les propriétés des entités vis-à-vis de la systématisation des descriptions multiples d'entités et des contraintes de participation. Enfin, nous étudions les différentes propriétés de l'approche CROME dont le caractère modulaire des classes obtenues lors de la conception et le statut de schémas de classes des contextes fonctionnels.

Le chapitre 4 est consacré à deux mises en œuvre de l'approche CROME : une par les techniques de vues des SGBD à objets MultiView [Rundensteiner92a] et COCOON [Scholl94] et une dans le cadre des SGBD supportant Java et en particulier, du SGBD POET [POET97]. La première mise en œuvre exploite les propriétés avancées de MultiView et COCOON comme la préservation de l'identité, la classification des vues, les vues à augmentation de capacité et les schémas-vues. La seconde mise en œuvre utilise certaines caractéristiques du langage Java et a donné lieu à la réalisation d'un prototype. Ces deux mises en œuvre contribuent à la validation de l'approche CROME dans le cadre des bases de données.

Dans le dernier chapitre, nous étudions deux extensions de CROME. La première concerne l'expression du cycle de vie d'une instance dans les différents contextes. Après une présentation de la dynamique d'objets dans Farandole, nous montrons comment traduire ce cycle de vie à l'aide des caractéristiques de CROME, notamment des contraintes de participation. Puis, nous généralisons cette traduction à des couples d'instances issues de classes distinctes.

La seconde extension est une approche d'intégration d'un composant « base de données » relatif à l'une des fonctions de l'organisation. L'objectif de cette intégration est l'exploitation des informations détenues par ce composant par les autres fonctions. La principale originalité de cette approche est de réaliser l'intégration au sein des entités. La présentation de cette seconde extension est précédée d'une synthèse des techniques d'intégration de composants « base de données » existants et peuplés pour bâtir des bases fédérées.

Chapitre 1

Descriptions multiples d'entités de bases de données

1.1. Introduction

Dans ce premier chapitre, nous cherchons à montrer les besoins de descriptions multiples d'entités de bases de données sous-tendant le système d'information (SI) d'organisations complexes. Ces besoins proviennent de la structuration fonctionnelle de telles organisations : chaque fonction ne perçoit et manipule que les informations qui lui sont pertinentes au sein du SI de l'organisation, ce qui conduit à une prise en compte de la structuration de l'organisation dans la base de données correspondante. Nous montrerons les avantages à ce que cette prise en compte de la structuration se fasse directement au sein du schéma de la base de données et nous verrons que, dans le cadre de l'approche « objet », ceci conduit à décrire de façon multiple les entités de la base de données. La systématisation de ces descriptions multiples relativement aux fonctions conduit alors à la structuration du schéma de la base de données. Nous étudierons ensuite les besoins de partage d'informations entre fonctions et les problèmes qui en découlent. Nous examinerons la problématique du cycle de vie d'instances dans le cadre d'une base de données dont le schéma est structuré en fonctions. Enfin, nous présenterons la problématique de l'intégration d'un composant « base de données » existant lors de la conception du SI d'une organisation.

1.2. Structuration en fonctions d'une organisation complexe

Notre objectif est de nous intéresser à la conception du SI d'une entreprise dont l'organisation est structurée fonctionnellement [Charpentier97]. La structuration fonctionnelle consiste à répartir les activités et missions en fonction d'une spécialisation conférée aux différents acteurs de l'entreprise.

Prenons l'exemple d'une entreprise industrielle qui fabrique et distribue ses produits. Celle-ci est structurée en un grand nombre d'activités distinctes. Parmi celles-ci, nous pouvons citer la fabrication, le stockage de produits finis, le SAV, la direction, les activités commerciale et comptable.

Les fonctions sont des parties disjointes de l'organisation, chacune ayant sa propre activité dont elle est responsable. Ce non-recouvrement des fonctions est important pour répartir précisément les responsabilités afin que chaque acteur sache exactement jusqu'où il doit et peut intervenir relativement à sa fonction.

Dans le cas de l'exemple précédent, les activités sont réalisées par des parties spécifiques de l'entreprise que sont les fonctions commerciale, comptable, de fabrication, de stock, de SAV et de direction. Chaque fonction a pour mission d'assurer une activité de l'organisation et n'intervient pas directement dans les activités des autres fonctions¹.

L'ensemble des fonctions partage le domaine de l'organisation. Celui-ci est constitué des entités manipulées par l'organisation, appelées *entités du domaine*. Ici, il est constitué des

1. Un commercial ne conçoit pas de voitures et un ingénieur de conception ne recherche pas de clients.

produits, des clients, des fournisseurs, des bons de commande, des bons de livraison et des entrepôts de stockage. Ces différentes entités sont, par conséquent, manipulées dans les différentes fonctions ou contextes de l'organisation. La figure 1.1 schématise ce partage du domaine par les fonctions de l'organisation.

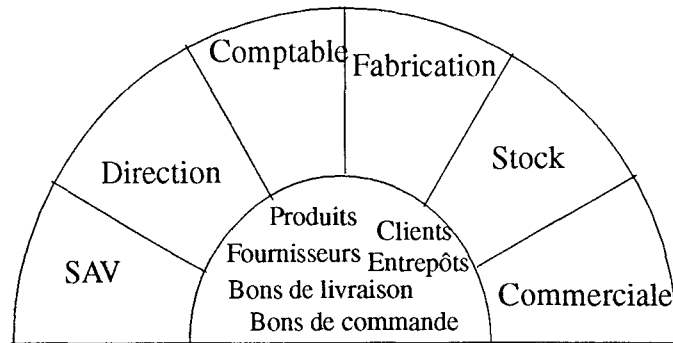


Figure 1.1. Domaine partagé par un ensemble de fonctions

Une organisation complexe ainsi structurée doit être considérée relativement à ses fonctions et non globalement. Son domaine n'est perçu que relativement à ses fonctions, provoquant le découpage de la perception de ce domaine en autant de perceptions que de fonctions [Brodie92]. Un client ne sera pas perçu de la même façon dans le contexte commercial que dans le contexte comptable. Dans le premier cas, on surveillera la date de sa dernière commande afin de pouvoir le relancer régulièrement pour l'inciter à passer à nouveau commande. Dans le second cas, on surveillera plutôt l'encours des factures non encore réglées pour relancer le règlement s'il n'est toujours pas intervenu.

1.3. Informations communes à l'organisation et informations spécifiques à chaque fonction

Le SI d'une organisation détient les informations concernant son domaine. Cependant, comme nous l'avons vu en 1.2, la perception du domaine est scindée en autant de perceptions que de fonctions. Ceci va conduire à une division des informations relatives au domaine. Seule une partie de ces informations est spécifique à une fonction. Nous appelons *informations fonctionnelles* la partie spécifique à une fonction des informations détenues par le SI d'une organisation complexe.

Dans notre exemple, les informations relatives à la fonction comptable sont le montant des factures restant à régler, la valeur des stocks dans un entrepôt, etc. Les informations relatives à la fonction commerciale sont les commandes en cours, les produits à promouvoir, etc. Ces informations sont spécifiques à chaque fonction. Les informations d'une fonction ne se recouvrent pas avec les informations d'une autre fonction, de par la nature disjointe des fonctions.

Le SI détient également des informations communes à l'organisation dans son ensemble et partagées par chaque fonction. Nous les appelons *informations communes*. Ces informations ne sont pas spécifiques à une fonction de l'organisation mais sont généralement relatives à des aspects propres aux entités du domaine. Le nom et l'adresse d'un client ne sont pas des

informations spécifiques à une fonction mais utilisées par l'ensemble des contextes manipulant l'entité « Client ». Par définition, les informations communes ne se recouvrent pas avec les informations fonctionnelles.

L'existence d'informations fonctionnelles relatives à une fonction est totalement liée à l'existence de cette fonction. Le SI d'une entreprise qui n'a pas de fonction « SAV » ne détiendra pas les informations fonctionnelles relatives à cette fonction, comme le coût de réparation d'un produit. Par ailleurs, chaque fonction est l'unique *fonction fournisseuse* de ses informations fonctionnelles.

A l'inverse, les informations communes existent indépendamment des fonctions.

1.4. Besoin de prise en compte de la structuration en fonctions au sein de la base de données du SI

1.4.1 Prise en compte des fonctions de la conception à l'exploitation du SI

Le fait que le SI d'une organisation complexe détienne les différentes informations fonctionnelles rend compte de la participation des fonctions à la conception et à l'exploitation de ce SI. Lors de la conception, une fonction fournit les éléments nécessaires pour décrire les informations fonctionnelles qui lui sont propres. Lors de l'exploitation, cette fonction manipule ses propres informations fonctionnelles détenues par le SI. Les interventions de l'organisation responsable d'un SI ne se font qu'au travers des fonctions de cette organisation, tant à la conception qu'à l'exploitation. A la conception, les fonctions sont les uniques fournisseurs des descriptions nécessaires pour former le schéma de la base de données sous-tendant le SI. Lors de l'exploitation, les clients du SI² appartiennent à une fonction précise de l'organisation et ne manipulent que les informations communes et les informations relatives à cette fonction. Il n'existe pas de clients hormis ceux des fonctions. Un tel client est appelé, par la suite, *client d'une fonction*.

Lors de l'exploitation, les clients d'une fonction manipulent ses informations fonctionnelles au travers des descriptions fournies par cette fonction lors de la conception. Ces descriptions forment une partie spécifique du schéma de la base de données du SI. Prendre en compte les fonctions dans le cadre d'une organisation complexe au sein du SI conduit donc à circonscrire les différentes parties du schéma relativement aux fonctions. Ceci nous conduit à un besoin de structuration du schéma. Ce besoin apparaît tant à la conception qu'à l'exploitation.

Par la suite, nous allons étudier comment l'approche ANSI/X3/SPARC (1.4.2) et l'approche de conception par intégration de vues (1.4.3) répondent à ce besoin de structuration. Puis nous introduirons l'approche « CROME » en 1.4.4.

1.4.2 L'approche ANSI/X3/SPARC

Dans le cadre des bases de données et dans certaines approches de conception comme [Batini92], la solution consiste en l'élaboration de schémas externes apposés sur le schéma d'une base de données et qui sont une extraction ad hoc des descriptions pertinentes pour une

2. Pour être précis, un client du SI peut être une personne ou un programme de type « application ».

fonction. Cette solution est prônée dans l'architecture à trois niveaux des bases de données³ proposée par l'ANSI/X3/SPARC [Ansi75, Gardarin93-p42, Ullman89-p7]. Cette architecture représentée à la figure 1.2 intègre trois niveaux de schémas : externe, conceptuel et interne. Le niveau interne correspond au niveau physique des bases de données et ne nous intéresse pas ici. Le niveau conceptuel est celui du schéma canonique de la base de données qui détient la totalité des descriptions. Au niveau externe sont introduits les différents schémas externes qui sont, dans le cas d'une organisation complexe, relatifs aux différentes fonctions.

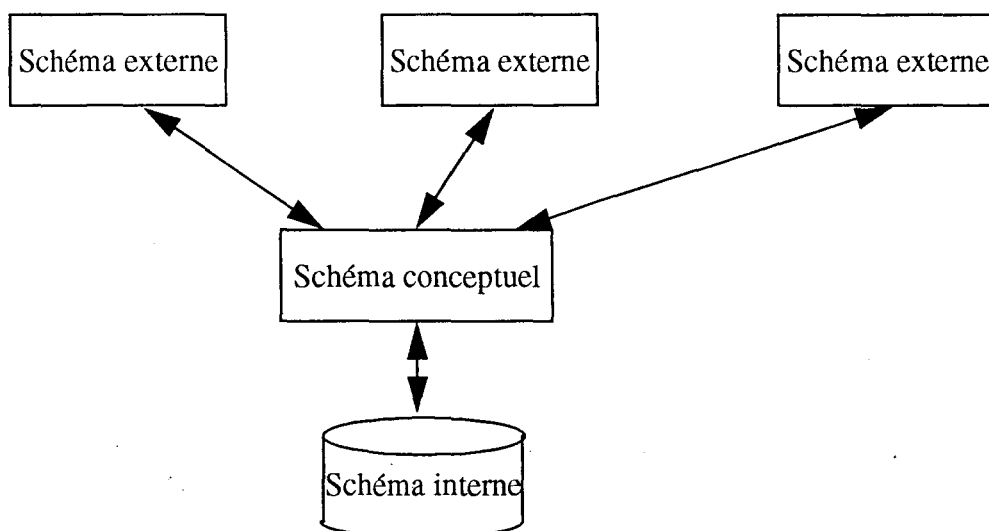


Figure 1.2. Les trois niveaux de l'architecture ANSI/X3/SPARC

Chaque fonction accède alors à la base de données au travers du schéma externe qui lui est associé et qui ne contient que les descriptions des informations communes et fonctionnelles correspondantes.

Les schémas externes peuvent être déterminés lors de la conception ou de l'exploitation. Ils sont construits à l'aide des techniques de vues lors de l'exploitation d'une base de données. La définition des vues inclut la définition des schémas externes ainsi que l'articulation entre ceux-ci et le schéma conceptuel. Nous reviendrons plus en détail sur les techniques de vues dans le chapitre 2 (section 2.2).

Cette solution d'un niveau externe de schéma préserve la conception d'un schéma canonique non structuré relativement aux fonctions, celle-ci n'apparaissant que dans l'articulation entre schémas externes et schéma conceptuel et en général au moment de l'exploitation.

1.4.3 La conception par intégration de vues

Cette approche prône la conception de vues relatives à des ensembles d'utilisateurs d'une base de données et détenant l'ensemble des descriptions nécessaires à ces utilisateurs. Elle est utilisée dans le cas de grosses base de données où la conception directe du schéma conceptuel dans sa globalité devient une tâche impossible. Chaque vue constitue un schéma conçu classiquement par ses utilisateurs et séparément des autres vues.

Chaque vue doit ensuite être prise en compte pour déterminer le schéma conceptuel et

3. principalement des bases de données relationnelles

global de la base de données. Ce dernier est construit par une méthode d'intégration des vues qui résout les différents problèmes relatifs à la conception séparée⁴. L'objectif de cette méthode est de supprimer la redondance et les incohérences issues des vues. Cependant, elle n'apporte pas une structuration du schéma en parties spécifiques à chaque ensemble d'utilisateurs. La correspondance entre descriptions et utilisateurs est perdue. La figure 1.3 montre les étapes de la conception par intégration de vues d'un schéma conceptuel. Cette approche est présentée de façon plus détaillée à la section 2.3 du chapitre 2.

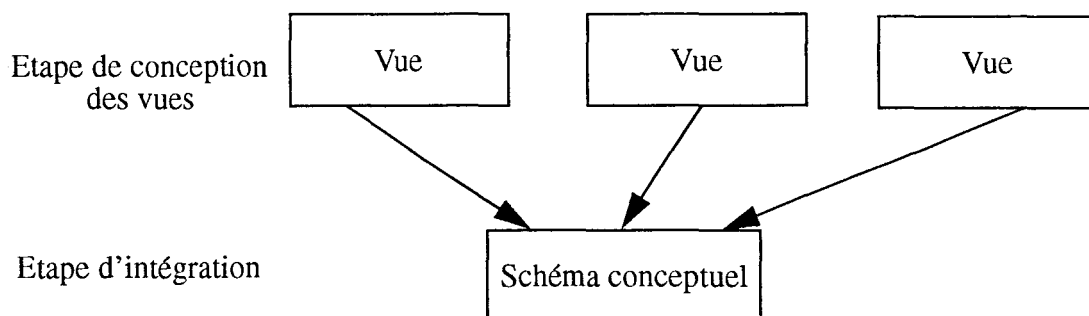


Figure 1.3. Conception par intégration de vues d'utilisateurs

Appliquées au cas d'une organisation complexe, les méthodes par intégration de vues [Batini92, Biskup86, Navathe86, Larson89, Spaccapietra92, Spaccapietra93] permettent, lors de la conception, de prendre en compte les descriptions issues des différentes fonctions pour former le schéma conceptuel. Ces méthodes rendent compte du statut de fournisseur de descriptions que détiennent les fonctions lors de la conception. Cependant la phase d'intégration débouche sur un schéma conceptuel canonique non structuré, ne reflétant pas la structuration en fonctions de l'organisation. De plus, la nécessité d'élaborer des schémas externes comme exposée en 1.4.2 reste entière, les schémas externes ne pouvant être obtenus directement de l'étape de conception des vues.

1.4.4 L'approche prônée en CROME

Nous proposons de rendre compte de la structuration en fonctions au sein même du schéma de la base de données dès sa conception en prenant pour objectif qu'il n'y ait pas lieu d'intégrer puis d'extraire des schémas externes. Cette structuration permet de circonscrire les parties du schéma décrivant les informations fonctionnelles relatives à chaque fonction. Nous appelons *plan fonctionnel* la partie du schéma spécifique à une fonction. La partie du schéma décrivant les informations communes est appelée *plan de base*. La figure 1.4 montre cette structuration en plans du schéma de la base de données de notre exemple.

Lors de la conception, chaque plan fonctionnel est formé par les descriptions fournies par sa fonction. Nous avons vu en 1.4.1 que, lors de l'exploitation, les clients d'une fonction manipulent ses informations fonctionnelles au travers des descriptions fournies par cette fonction lors de la conception. Par conséquent, ils exploitent le plan fonctionnel qui leur est associé. Ceci montre que la structuration du schéma de la base est préservée de la conception à l'exploitation.

Les plans fonctionnels sont tous disjoints, deux à deux. En effet, ils décrivent des

4. notamment les conflits de structure

informations fonctionnelles qui ne se recouvrent pas⁵. Par ailleurs, et pour des raisons similaires, les plans fonctionnels sont tous disjoints du plan de base.

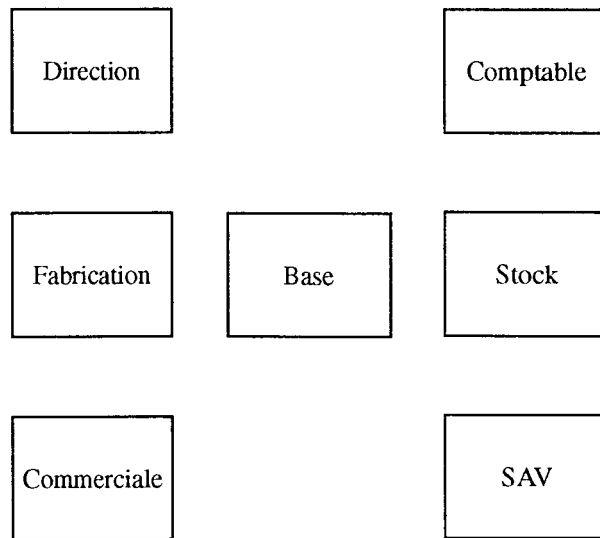


Figure 1.4. Schéma d'une base de données structurée en plans

La prise en compte de la structuration au sein du schéma a pour objectifs :

- structuration et modularité accrues du schéma ;
- descriptions pertinentes pour une fonction issues directement de la conception, sans qu'il soit nécessaire d'élaborer a posteriori des schémas externes ;
- aide à la conception qui peut se réaliser fonction par fonction, chaque fonction fournissant les descriptions du schéma nécessaires à ses besoins sans procéder à une intégration ;
- préservation de la correspondance entre fonctions et descriptions du schéma relatives à ces fonctions de la conception à l'exploitation.

1.5. Orthogonalité entités/fonctions

1.5.1 Conception « classique » du schéma du système d'information

Le SI d'une organisation a pour principal objectif la gestion des informations de son domaine. Pour remplir cet objectif, la grande majorité des approches de conception de bases de données proposent un modèle de données destiné à structurer le schéma pour représenter le plus fidèlement possible le domaine de l'organisation.

Les modèles de données cherchent à représenter cette réalité à l'aide de primitives de description dont les plus courantes sont :

- les entités ;
- les relations entre entités ;
- les attributs relatifs aux entités ou aux relations.

Ces trois primitives de description sont celles du modèle Entité-Association introduit dans [Chen76].

5. cf 1.3

Nous faisons le choix de l'approche « objet » où chaque entité du domaine est représentée par une seule classe. Les principales propriétés de l'approche « objet » sont les suivantes :

- toutes les descriptions sont centralisées au sein des classes. Il n'existe pas de construction descriptive en dehors des classes. Une relation entre entités est décrite de façon répartie par des attributs dans les classes correspondantes [ODMG97] ;
- les instances détiennent une identité indépendante de leurs valeurs. Cette identité permet de référencer et de partager des instances ;
- l'ensemble des classes forme une hiérarchie ordonnée par la relation d'héritage. Cette structuration permet de mieux capturer la sémantique du domaine.

Les classes associées aux entités du domaine sont appelées, par la suite, *classes du domaine*.

La figure 1.5 montre le schéma d'une base de données tel qu'il pourrait exister dans le cadre de notre entreprise de fabrication et de distribution. Ce schéma est structuré par le modèle de données « objet » et représenté à l'aide de la notation graphique employée dans [ODMG97]⁶. Ce schéma est une représentation plus ou moins fine du domaine.

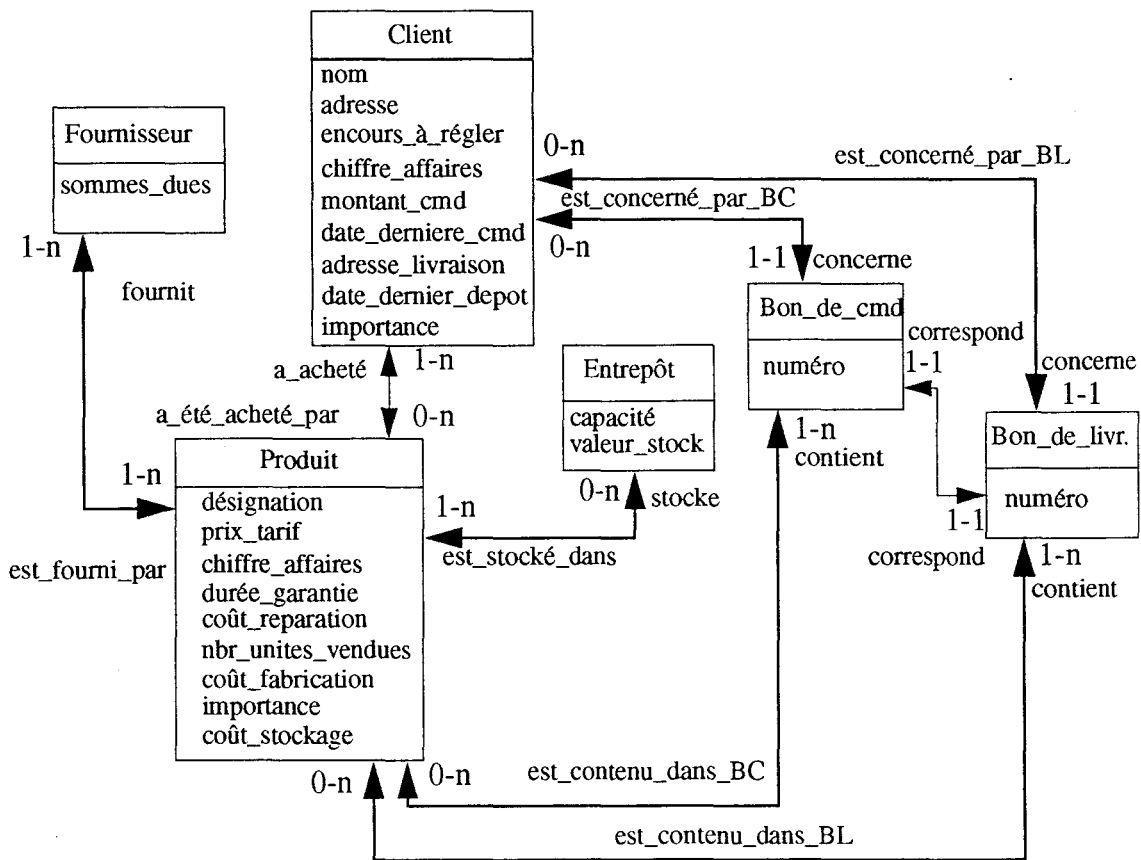


Figure 1.5. Schéma de la base de données qui sous-tend le SI d'une entreprise de fabrication et de distribution

6. étendu pour représenter les cardinalités des relations et les attributs des entités

1.5.2 Les entités du domaine sont orthogonales aux fonctions

Nous avons vu en 1.2 que les fonctions partagent le domaine de l'organisation. Chaque entité du domaine participe à l'ensemble (ou à un sous-ensemble) des fonctions. Dualement, les fonctions font participer l'ensemble (ou un sous-ensemble) des entités du domaine. Cette dualité peut être représentée par le diagramme de la figure 1.6 qui montre ainsi l'orthogonalité entre entités et fonctions. A chaque intersection, la présence d'un cercle montre la participation de l'entité à la fonction. Cette orthogonalité révèle le caractère multi-fonctionnel des entités du domaine.

Fonctions/Objets	Bons_de_cmd	Entrepôts	Bons_de_livr.	Fournisseurs	Produits	Clients
SAV		⊙		⊙	⊙	⊙
Direction				⊙	⊙	⊙
Comptable	⊙	⊙	⊙	⊙	⊙	⊙
Fabrication		⊙		⊙	⊙	⊙
Stock	⊙	⊙	⊙	⊙	⊙	⊙
Commercial	⊙		⊙	⊙	⊙	⊙

Figure 1.6. Orthogonalité entités du domaine/fonctions

Dans notre exemple, la fonction commerciale manipule les entités « Produits », « Clients », « Fournisseurs », « Bons de commande » et « Bons de livraison ». La fonction de stockage manipule les mêmes entités ainsi que l'entité « Entrepôts ». Il en est de même pour les autres fonctions qui font participer l'ensemble (ou un sous-ensemble) des entités du domaine. Dualement, l'entité « Entrepôts » participe aux fonctions « SAV », « Comptable », « Fabrication » et « Stock ».

Dans la suite, nous exploiterons chaque aspect de cette dualité ainsi ;

- la participation d'une entité à différentes fonctions va nous permettre de diffuser la structuration en fonctions du schéma au sein des classes associées à ces entités ;
- la participation de plusieurs entités à une fonction va se traduire par un *contexte fonctionnel* faisant intervenir l'ensemble de ces entités et de leurs relations.

Les besoins de partage d'informations entre fonctions ainsi que le caractère multi-fonctionnel des entités du domaine nécessitent une structuration de l'ensemble des contextes.

1.6. Prise en compte de la structuration en fonctions de l'organisation au sein des entités et classes du domaine

1.6.1 Descriptions multiples des entités du domaine

Un plan fonctionnel est constitué par un ensemble d'attributs et de relations entre les entités. Toutes les descriptions étant, dans le cadre de l'approche « objet », centralisées au sein

des classes, chaque plan fonctionnel est distribué au sein des classes des différentes entités. Les entités du domaine étant multi-fonctionnelles, elles détiennent une description pour chaque contexte fonctionnel où elles participent. L'orthogonalité entités/fonctions montre ainsi que la partie d'un plan fonctionnel relative à une entité est une partie de la description de cette entité.

Ceci peut être illustré par la figure 1.6 où chaque intersection entourée par un cercle représente une description de l'entité dans le contexte où elle participe. Nous appelons une telle description *partie fonctionnelle*. L'ensemble des parties fonctionnelles des entités du domaine relatives à un même contexte forme le plan fonctionnel correspondant. La description relative à une entité et commune aux fonctions est appelée *description de base*. L'ensemble des descriptions de base des entités du domaine forme le plan de base.

Les parties fonctionnelles sont disjointes. En effet, elles sont issues de plans fonctionnels, eux-mêmes disjoints. La description de base est également, pour les mêmes raisons, disjointes avec les parties fonctionnelles.

Toutes les descriptions d'une entité du domaine, description de base et parties fonctionnelles, sont détenues par la classe associée à cette entité.

Dans notre exemple, l'entité « Client » intervient, entre autres, dans les contextes comptable et « Stock ». La fonction comptable fournit, comme informations, l'encours des factures non encore réglées ainsi que le chiffre d'affaires total réalisé par ce client. La partie fonctionnelle correspondante est formée par les attributs : -

- encours_à_régler ;
- chiffre_affaires.

La fonction « Stock » fournit l'adresse de livraison du client. L'attribut « adresse_livraison » constitue la partie fonctionnelle correspondante.

Le besoin de structurer en fonctions le schéma de la base de données nous conduit à un problème de descriptions multiples de chaque entité relativement à l'ensemble des fonctions.

La figure 1.7 montre graphiquement le besoin de structuration de la description de l'entité « Client » relativement aux contextes fonctionnels où elle intervient. A droite de cette figure, l'entité est associée à plusieurs parties fonctionnelles disjointes.

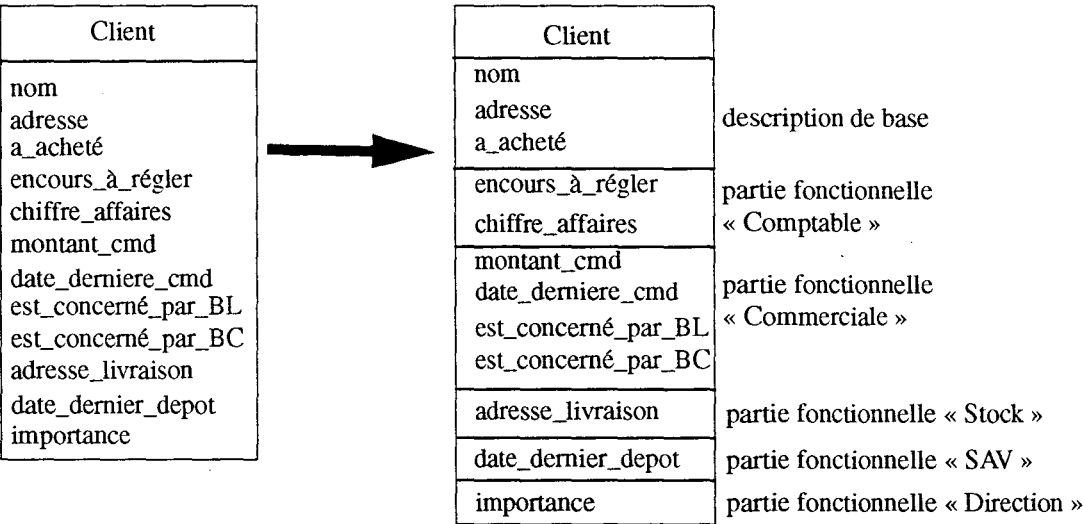


Figure 1.7. Besoin de structuration de la description de l'entité « Client »

Ce problème de descriptions multiples des entités a été abordé dans le langage ROME [Carré89] et dans le cadre des bases de données [Abiteboul91, Albano93, Al-Jadir93, Bertino92, Pernici90a, Scholl91, Rundensteiner92c] qui sont présentées au chapitre 2. Une approche systématisée par rapport aux fonctions est présentée dans le chapitre 3 consacrée à l'approche CROME et plus précisément à la section 3.3.

1.6.2 Instances détenant les informations fonctionnelles et descriptions multiples : notions d'extension et d'état fonctionnels

Les informations sont détenues par les instances des différentes classes du domaine. Chaque classe détermine une extension qui est l'ensemble de ses instances au sein de la base de données prise dans sa totalité.

Les informations relatives à une fonction sont détenues par un ensemble d'instances des différentes classes du domaine présentes dans le contexte fonctionnel correspondant. Plus précisément, pour chacune de ces instances, les informations relatives à la fonction sont décrites par la partie fonctionnelle correspondante. Pour une classe du domaine, l'ensemble de ses instances détenant des informations pour une fonction et donc y participant forme, pour celle-ci, son *extension fonctionnelle*. Chaque classe du domaine détermine une extension fonctionnelle dans chaque contexte fonctionnel où l'entité correspondant à cette classe intervient.

Chaque extension fonctionnelle d'une classe du domaine est un sous-ensemble de son extension. Ce sous-ensemble est obtenu par une sélection basée sur un prédicat⁷. Chacune des instances participant à l'extension fonctionnelle d'une classe est décrite pour le contexte fonctionnel correspondant, c'est-à-dire par la partie fonctionnelle correspondante.

Dans notre exemple, la fonction « SAV » ne s'intéresse qu'aux produits dont le prix est supérieur à 1000 F. Les autres produits ne sont pas réparables. L'extension fonctionnelle de la classe « Produits » dans le contexte fonctionnel « SAV » est un sous-ensemble de son extension complète.

Dualement, chaque instance ne participe qu'à un sous-ensemble des contextes fonctionnels. Ce sous-ensemble n'est pas fixe et peut évoluer dans le temps. Nous appelons ce sous-ensemble *état fonctionnel* de l'instance. L'état fonctionnel détermine l'ensemble des parties fonctionnelles qui décrivent une instance. Dans l'exemple précédent, une instance de « Produits » dont le prix est supérieur à 1000 F est décrite par la partie fonctionnelle « SAV ».

Le besoin de fournir à chaque fonction de l'organisation que les informations fonctionnelles qui lui sont propres nous conduit à un problème de descriptions multiples et évolutives des instances au travers d'une sélection de celles-ci pour chaque fonction. La possibilité de décrire de façon multiple et évolutive des instances au travers de sélections a été principalement abordée dans le cadre des vues des bases de données à objets [Abiteboul91, Al-Jadir93, Bertino92, Scholl91, Rundensteiner92c]. Une approche systématisée par les fonctions est présentée à la section 3.5 du chapitre 3.

Dans notre exemple, seules les instances de « Produits » dont le prix est supérieur à 1000 F participent au contexte « SAV ». La partie fonctionnelle pour le SAV contient les attributs « durée_garantie » et « cout_réparation ». Ces attributs n'ont évidemment de sens que

7. L'utilisation d'un prédicat pour réaliser des sélections est courante dans le cadre des bases de données.

pour les instances participant au contexte « SAV ». La figure 1.8 montre l'ensemble des parties fonctionnelles décrivant une instance des « Produits » participant au contexte « SAV » et l'ensemble des parties fonctionnelles décrivant une instance de « Produits » n'y participant pas. La figure montre aussi l'état des instances.

Remarque

La notion classique d'état d'une instance est généralement définie par l'ensemble (ou un sous-ensemble) des valeurs de ses attributs. Changer d'état consiste à changer les valeurs d'attributs. Ici, la notion est plus forte car le changement d'état fonctionnel d'une instance provoque un changement de l'ensemble des parties qui la décrivent et donc de sa structure. Cette notion d'état fonctionnel ne se substitue pas à celle « d'état classique » d'une instance.

Ayant fait le choix d'une sélection basée sur un prédicat sur les valeurs des différentes propriétés d'une instance, l'état fonctionnel de celle-ci est donc déterminé par son « état classique ».

produit1		
désignation : "GSM" prix_tarif : 3500 a_été_acheté_par : {client1,client2} est_fourni_par : {fournisseur1}	description de base	
chiffre_affaires : 2356200	partie fonctionnelle « Comptable »	
durée_garantie : 1 cout_réparation : 1500	partie fonctionnelle « SAV »	
nbr_unites_vendues : 748 est_contenu_dans_BC : {bc1,bc2} est_contenu_dans_BL : {bl1,bl2}	partie fonctionnelle « Commerciale »	
est_stocké_dans : {entrepot1} coût_stockage : 700	partie fonctionnelle « Stock »	instance de « Produit » participant à la fonction « SAV »
cout_fabrication : 2000	partie fonctionnelle « Fabrication »	
importance : "produit à developper"	partie fonctionnelle « Direction »	
produit2		
désignation : "téléphone fixe" prix_tarif : 250 a_été_acheté_par : {client2, ...} est_fourni_par : {fournisseur2}	description de base	
chiffre_affaires : 5127456	partie fonctionnelle « Comptable »	
nbr_unites_vendues : 22560 est_contenu_dans_BC : {bc11,...} est_contenu_dans_BL : {bl11,...}	partie fonctionnelle « Commerciale »	
est_stocké_dans : {entrepot10} coût_stockage : 350	partie fonctionnelle « Stock »	instance de « Produit » ne participant pas à la fonction « SAV »
cout_fabrication : 100	partie fonctionnelle « Fabrication »	
importance : "faible"	partie fonctionnelle « Direction »	

Figure 1.8. Parties fonctionnelles décrivant deux instances de « Produit » en fonction de leur état fonctionnel

1.7. Partage des informations

1.7.1 Besoin de partage des informations

Les clients d'une fonction exploitent et manipulent les informations fonctionnelles correspondant à cette fonction. Ils ont également besoin d'exploiter d'autres informations qui leur sont pertinentes, à savoir les informations communes mais aussi des informations fonctionnelles fournies par d'autres fonctions.

L'exploitation des informations communes et d'informations entre fonctions conduit à deux formes de partage qui se distinguent ainsi :

- les informations communes propres aux entités et existant indépendamment des fonctions. Elles sont implicitement partagées par toutes les fonctions. Par exemple, la désignation des produits est partagée et exploitée par les clients de toutes les fonctions ;
- les informations fonctionnelles partagées sont issues de l'unique fonction fournisseuse qui détermine leur existence (cf 1.3). Elles sont exploitées par un ensemble de *fonctions consommatrices* qui sont autant de clients internes de la fonction fournisseuse. Ce partage doit être spécifié explicitement pour chaque fonction consommatrice. Par exemple, les clients de la fonction « Fabrication » s'intéressent au nombre d'unités vendues de chaque produit, information fonctionnelle fournie et gérée par la fonction commerciale ainsi qu'au coût de stockage de chaque produit, information fonctionnelle fournie par la fonction « Stock ». La figure 1.9. montre l'ensemble des informations que détient une instance de la classe « Produit » et qui sont pertinentes pour les clients de la fonction « Fabrication ». Les informations fournies par d'autres fonctions sont indiquées en italique.

produit1	
désignation : "GSM" prix_tarif : 3500 a_été_acheté_par : {client1,client2} est_fourni_par : {fournisseur1}	
<i>nbr_unites_vendues : 748</i>	provenant de la fonction « Commerciale »
<i>coût_stockage : 50</i>	provenant de la fonction « Stock »
cout_fabrication : 2000	

Figure 1.9. Informations pertinentes détenues par une instance de « Produit » pour les clients de la fonction « Fabrication »

L'ensemble des informations communes et fonctionnelles est détenu par les différents plans. La figure 1.10 illustre ainsi l'ensemble des descriptions pertinentes pour les clients de la fonction « Fabrication ».

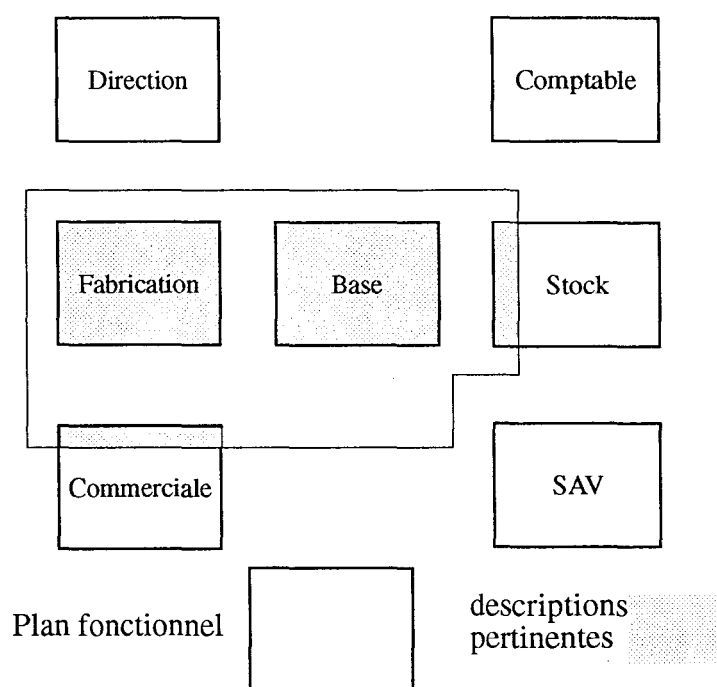


Figure 1.10. Ensemble des descriptions pertinentes pour les clients de la fonction « Fabrication »

Les plans sont diffusés au sein des classes des différentes entités, ce qui donne lieu à leur description de base et leurs parties fonctionnelles. Les deux formes de partage sont donc réalisées au sein des entités du domaine, respectant ainsi l'orthogonalité entités/fonctions. Ce besoin de partage nous conduit à un problème d'articulation au sein des classes du domaine entre la description de base et les différentes parties fonctionnelles et entre les différentes parties fonctionnelles elles-mêmes. Nous verrons au chapitre 3 (sections 3.3 et 3.4) les mécanismes destinés à prendre en compte ces articulations dans le cadre de CROME.

Pour une entité du domaine, les informations pertinentes pour les clients d'une fonction sont détenues par sa partie fonctionnelle, sa description de base et sont issues des fonctions fournisseuses.

1.7.2 Problèmes issus du partage des informations

Le partage des informations que nous avons vu en 1.7.1 et la notion d'extension fonctionnelle de chaque classe au sein des contextes introduite en 1.6.2 posent les problèmes suivants :

- Comment partager une relation inter-entités ? Quelles sont les contraintes sur les extrémités dans les contextes consommateurs ?
- Dans un contexte fonctionnel, les instances ne participant pas aux contextes fournisseurs des informations nécessaires pour ses clients sont-elles pertinentes pour celui-ci ou non ?
- Dans un contexte fournisseur, l'intégrité référentielle de chaque relation inter-entités

est vérifiée par rapport aux extensions fonctionnelles des classes. Cette vérification ne peut servir à garantir l'intégrité référentielle de ces relations dans les contextes consommateurs où les extensions fonctionnelles des classes peuvent être différentes. Comment garantir alors l'intégrité référentielle des relations partagées dans les contextes consommateurs ?

- La même problématique se pose pour les contraintes de cardinalité des relations inter-entités. Celles-ci sont vérifiées dans un contexte fournisseur par rapport aux extensions fonctionnelles des classes. Dans un contexte consommateur, les extensions fonctionnelles des classes étant différentes, il faut mettre en œuvre des mécanismes nouveaux pour garantir les contraintes de cardinalité des relations partagées.

Ces questions conduisent à des problèmes de contraintes d'intégrité originaux que nous étudions dans le chapitre 3 (sections 3.5 et 3.6).

1.8. Cycle de vie de participation d'une instance et d'un couple d'instances à un ensemble de contextes fonctionnels

1.8.1 Cycle de vie de participation d'une instance à un ensemble de contextes

Au sein de l'organisation, les différents contextes manipulent les entités du monde réel. Il arrive fréquemment que, pour assurer le fonctionnement de l'organisation, des entités soient manipulées par un premier contexte fonctionnel puis transmises à un second, etc. C'est, par exemple, le cas des formulaires transmis de fonction en fonction. Cette chaîne de participations successives d'une entité à différents contextes forme ainsi le cycle de vie de cette entité.

Un exemple de cycle de vie est celui du bon de livraison. Chaque bon de livraison est produit par la fonction commerciale. Puis lorsqu'il est prêt, il est transmis à la fonction « Stock »⁸. Enfin, lorsque l'ensemble des produits figurant sur le bon est livré au client correspondant, le bon est transmis à la comptabilité afin d'être comptabilisé. Après la comptabilisation, la fonction « Comptable » n'en garde ensuite plus trace.

Nous voulons rendre compte du cycle de vie de ces entités au sein de la base de données qui sous-tend le SI de l'organisation. Ceci se traduit par la prise en compte des participations successives d'une instance d'une classe du domaine à l'ensemble des contextes auxquels participe son référent⁹. Il s'agit donc de prendre en compte les différents états fonctionnels d'une instance au cours de son existence. La suite des états fonctionnels d'une instance d'une classe du domaine au cours de sa vie forme son *cycle de vie fonctionnel*.

La plupart des travaux relatifs [Rumbaugh95, Batini92] à l'expression du cycle de vie porte principalement sur les changements d'état de l'ensemble ou d'une partie de la base de données. Dans le cadre de l'approche « objet », il s'agit alors des changements d'état d'instances. Notre préoccupation ne concerne pas ici cette notion classique d'état mais celle d'état fonctionnel (cf 1.6.2).

Ramené à la classe « Bon_de_livraison », ceci signifie que chaque instance de cette classe est créée par le contexte commercial et y participe. Vient ensuite s'ajouter sa participation au contexte « Stock », puis au contexte comptable. Enfin, cette dernière participation est retirée.

8. qui assure aussi les livraisons

9. c'est-à-dire l'entité du monde réel que l'instance représente dans le système d'information.

Un cycle de vie fonctionnel d'une instance peut être représenté par un réseau de Petri [Peterson77] dont les places correspondent aux contextes fonctionnels. Un jeton dans l'une des places du réseau de Petri signifie la participation de l'instance au contexte correspondant. Les transitions correspondent à des événements provoquant un changement d'état fonctionnel. Cette représentation est proche de celle utilisée dans Farandole [Al-Jadir93, Al-Jadir94, Al-Jadir95, Falquet88] qui est présentée à la section 5.2.1.2 du chapitre 5.

La figure 1.11 montre le réseau de Petri relatif à la classe « Bon_de_livraison ». Celui-ci représente le cycle de vie fonctionnel d'une instance de « Bon_de_livraison ».

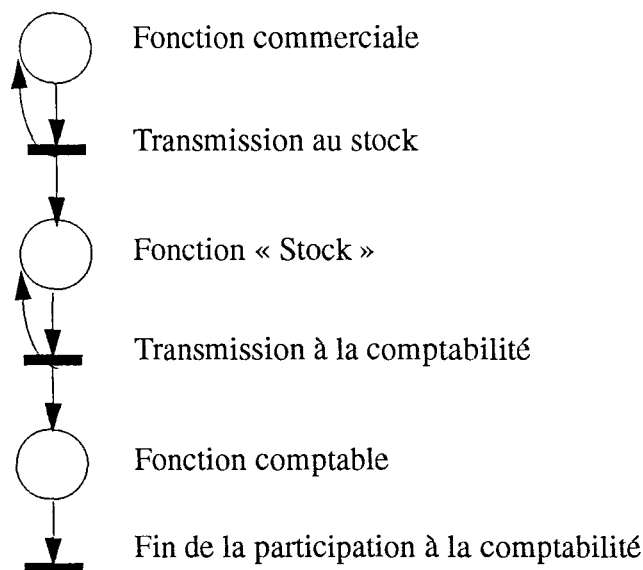


Figure 1.11. Réseau de Petri représentant le cycle de vie fonctionnel d'une instance de « Bon_de_livraison »

Si un réseau de Petri peut aider à représenter et à déterminer le cycle de vie fonctionnel des instances d'une classe du domaine, le problème consiste à fournir les mécanismes au sein de la base de données, et plus particulièrement au sein de cette classe pour que ce cycle de vie soit pris en compte. Nous présentons une solution basée sur les moyens de sélection des instances participant à un contexte dans la section 5.2.2 du chapitre 5.

1.8.2 Généralisation à un couple d'instances

Au sein de l'organisation, il existe des couples d'entités du monde réel qui participent ensemble successivement à un ensemble de contextes. Dans notre exemple, un bon de commande et un bon de livraison détiennent le même cycle de vie fonctionnel. Le bon de commande suit le bon de livraison. Il est reçu par la fonction commerciale, puis est transmis à la fonction « Stock » avec le bon de livraison. Lors de la livraison¹⁰, la présence du bon de commande permet de vérifier auprès du client que les produits livrés correspondent bien à ceux qu'il a commandés. Ensuite, bon de commande et de livraison sont transmis à la comptabilité où ils y participent jusqu'à ce qu'ils soient comptabilisés.

¹⁰. effectuée par la fonction « Stock »

Au sein de la base de données, la notion de cycle de vie fonctionnel peut être étendue à un couple d'instances. Un tel couple d'instances est connecté par une occurrence d'une relation définie entre leurs classes respectives. Il faut aussi que chaque instance du couple ait une partie ou la totalité de son cycle de vie fonctionnel qui soit commune avec le cycle de vie fonctionnel de l'autre instance.

Par exemple, les classes « Bon de commande » et « Bon de livraison » sont connectées par la relation « correspond » illustrée à la figure 1.5. Il est possible de former des couples d'instances des classes « Bon de commande » et « Bon de livraison » telles qu'il existe une occurrence de la relation « correspond » entre ces instances.

La partie commune du cycle de vie des instances devient le cycle de vie fonctionnel du couple d'instances. Cette partie commune est représentée par un unique réseau de Petri.

Les instances des classes « Bon de commande » et « Bon de livraison » connectées par la relation « correspond » ont le même cycle de vie fonctionnel. La figure 1.12 montre le cycle de vie fonctionnel du couple « Bon de commande/Bon de livraison » qui est le même que celui de chacune des instances des classes correspondantes.

Le cycle de vie d'un bon de commande peut être enrichi en le faisant participer aux contextes fonctionnels du client avant d'être transmis au fournisseur. Un bon de commande est créé par la fonction « Achats » du client puis est transmis à la fonction commerciale. Celle-ci crée le bon de livraison associée au bon de commande reçu. Puis le couple ainsi formé est transmis successivement aux fonctions « Stock » et comptable comme précédemment.

Dans cette organisation, le cycle de vie fonctionnel de la figure 1.12 n'est alors plus qu'une partie du cycle de vie fonctionnel d'un bon de commande, partie commune avec le bon de livraison qui lui est associé lorsqu'il vient à participer au contexte commercial.

A la problématique de prise en compte du cycle de vie fonctionnel des instances d'une classe du domaine par un mécanisme au sein de cette classe, vient s'ajouter la problématique de définition d'un couple d'instances telles qu'elles aient, en commun, une partie ou la totalité de leur cycle de vie fonctionnel. Cette problématique est abordée à la section 5.2.2.4 du chapitre 5.

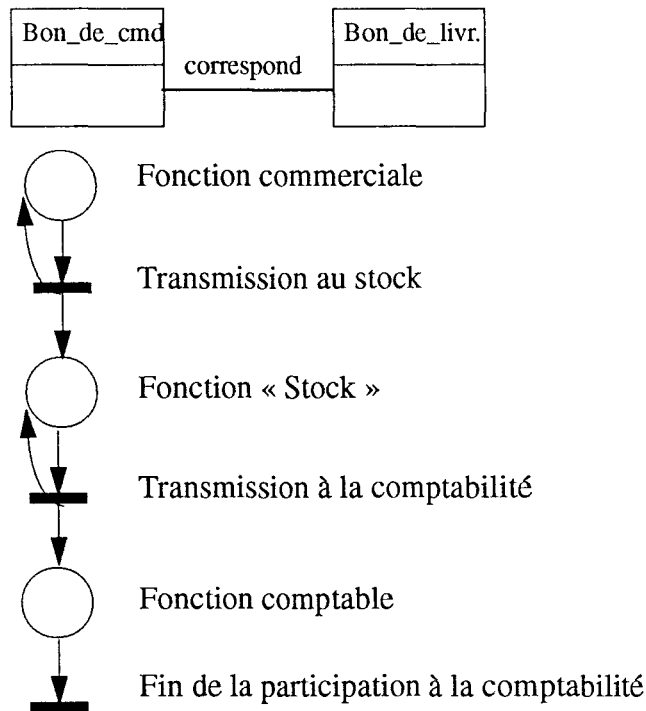


Figure 1.12. Réseau de Petri représentant le cycle de vie fonctionnel du couple d'instances « Bon de commande/Bon de livraison »

1.9. Intégration d'un composant « base de données »

La conception du SI d'une organisation complexe peut passer par l'intégration d'un composant « bases de données » existant¹¹. En effet, il est possible que certaines fonctions de cette organisation aient conçu un composant « base de données » avant que l'informatisation globale du SI de l'organisation soit envisagée. L'intégration effectuée lors de la conception du SI global a alors pour objectif de récupérer les informations fonctionnelles contenues dans un tel composant afin qu'elles puissent être partagées par les autres fonctions de l'organisation.

Les travaux relatifs à l'intégration de composants concernent principalement l'élaboration de bases de données fédérées encore appelées *fédérations* [Sheth90]¹². Une base de données fédérée ne détient pas d'informations propres. Elle offre à ses clients une vue sur l'ensemble des composants qu'elle intègre. Ceux-ci peuvent ainsi percevoir et manipuler, de façon transparente au travers du schéma de la fédération, l'ensemble des informations détenues par les différents composants. Si une fédération doit détenir des informations qui lui sont propres, celles-ci doivent être décrites par un composant spécifique appelé « composant auxiliaire » qui doit être intégré avec les autres composants (cf [Sheth90]).

Par rapport à l'élaboration d'une fédération, nous ne cherchons pas à obtenir un schéma uniquement par intégration de schémas de composants existants. Nous proposons de concevoir la base de données relative au SI de l'organisation en déterminant son référentiel constitué des entités du domaine. Le composant à intégrer étant relatif à une fonction, nous proposons

11. En procédant par itération, il est possible d'intégrer plusieurs composants successivement

12. voir aussi [Ahmed91, Boulanger97, Busse94, Kaul90, Klas90, Scholl93, Tresch94]

ensuite de partager ses informations au sein des entités du domaine comme nous l'avons vu en 1.7.1. Ceci suppose que nous fassions l'hypothèse que le composant est structuré relativement aux mêmes entités du domaine que la base de données en cours de conception. Cependant, ce partage est plus complexe que dans le cas d'une seule base de données à cause des deux problèmes suivants dus au fait que chaque base de données a été peuplée séparément :

- la nécessité d'établir une relation de coréférence entre les instances du composant et de la base en cours de conception ;
- la nécessité de mettre en œuvre un mécanisme d'articulation entre les deux bases pour permettre à une instance de la base d'accéder aux informations fonctionnelles détenues par son instance coréférente dans le composant.

L'intégration du composant est ainsi diffusée au sein des entités du domaine.

Il est également nécessaire de circonscrire les parties du schéma du composant détenant les informations spécifiques à sa fonction, c'est-à-dire de retrouver les parties fonctionnelles au sein des classes de ce schéma. Cette étape est nécessaire pour déterminer les partages possibles.

Cette problématique d'intégration d'un composant « base de données » existant est abordée à la section 5.3.2 du chapitre 5. Dans cette section, nous présentons également une approche d'intégration dans le cas où le schéma du composant n'est pas structuré avec les mêmes entités que celui de la base en cours de conception.

Chapitre 2

Les vues dans les bases de données à objets

2.1. Introduction

Dans le cadre des bases de données à objets [Bancilhon92, Bertino93, Cattell97, Kim90], nous examinons les différentes techniques de vues. D'une façon générale, une vue est une partie du schéma de la base pertinente pour un utilisateur ou un groupe d'utilisateurs. Dans la littérature, les techniques de vue relèvent soit de la phase de conception soit de la phase d'exploitation. Si elles relèvent de la phase de conception, il s'agit alors de **techniques d'intégration de vues** visant à l'obtention du schéma conceptuel de la base de données. Si les techniques relèvent de la phase d'exploitation, il s'agit alors de **techniques de vues « utilisateurs »**. Dans ce dernier cas, ces techniques procèdent à une extraction du schéma et des informations de la base : un utilisateur ou groupe d'utilisateurs n'accède au travers de la vue qu'à la partie des informations de la base qui lui est pertinente. Une telle vue correspond à la notion de schéma externe prônée par l'architecture ANSI/X3/SPARC. Nous étudierons également les solutions proposées par les vues aux problèmes abordés dans le chapitre 1 et notamment vis-à-vis du besoin de structuration du schéma (cf 1.4) et des entités (cf 1.6).

Dans la littérature, la granularité de la notion de vue n'est pas toujours la même. Certains travaux confèrent aux vues une granularité fine : celles des classes¹. Dans ce chapitre, ces vues sont appelées « classes-vues ». D'autres travaux confèrent aux vues la granularité des schémas. Dans ce chapitre, ces vues sont appelées « schémas-vues ».

Les deux approches de vues que nous examinons successivement sont :

- les vues « utilisateurs » lors de l'exploitation ;
- les techniques d'intégration de vues lors de la conception.

2.2. Vues « utilisateurs » lors de l'exploitation de la base de données

2.2.1 Besoins

Le besoin principal des vues « utilisateurs » provient de la nécessité de concevoir des schémas externes dans le cadre du modèle de l'architecture ANSI/X3/SPARC [ANSI75, Gardarin93- p42, Ullman89-p7] pour les bases de données relationnelles. Cette architecture déjà présentée en 1.4.2 et illustrée à la figure 2.1 intègre trois niveaux de schémas : externe, conceptuel et interne. Un schéma externe décrit l'ensemble des informations pertinentes pour un utilisateur ou un groupe d'utilisateurs de la base de données. Un schéma externe est une abstraction du schéma conceptuel. Cette notion de schéma externe nécessaire pour les bases de

1. Dans le modèle relationnel, la granularité des vues est alors celles des relations (cf [Gardarin93, p193])

données relationnelles s'avère également nécessaire pour les bases de données à objets.

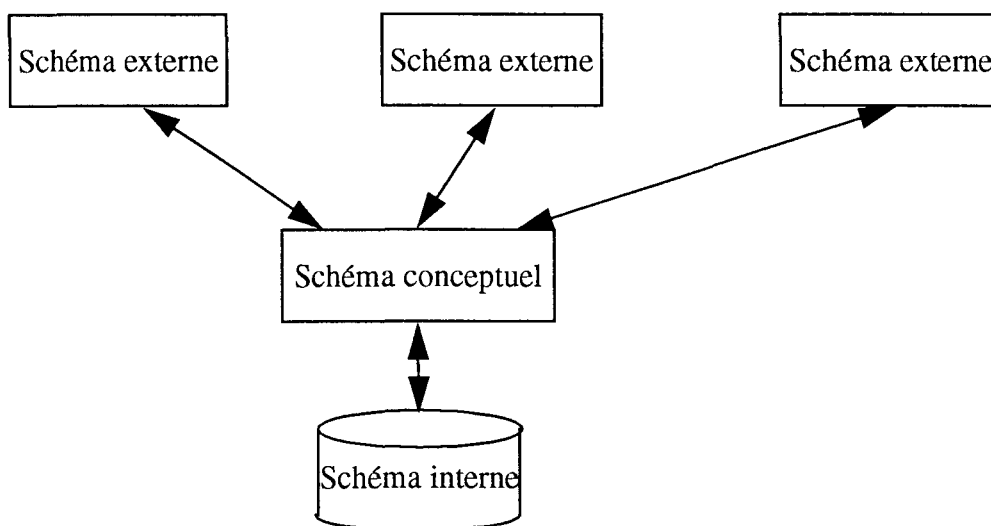


Figure 2.1. L'architecture ANSI/X3/SPARC

Les vues peuvent également répondre totalement ou partiellement à d'autres besoins dont les principaux sont [Bertino92, Chan95] :

- fournir un outil de rémanence des requêtes ;
- permettre la simulation d'évolution de schémas et fournir un support pour les versions de schémas ;
- gérer les autorisations d'accès ;
- l'intégration de bases de données existantes.

2.2.2 Statut des classes-vues

Dans le modèle relationnel, les vues² sont définies à l'aide d'une requête. Ces requêtes sont exprimées à l'aide des opérateurs de l'algèbre relationnelle définis dans [Codd70]. Une propriété essentielle de cette algèbre est que le résultat de ses différents opérateurs est une relation. Le résultat d'une requête généralement composée de plusieurs opérateurs est donc une relation. Par conséquent, les vues acquièrent le statut de relation. Si l'on ne considère pas les problèmes de mise à jour, les utilisateurs d'un ensemble de vues d'une base de données relationnelle peuvent les manipuler de la même façon que les relations du schéma conceptuel.

Dans le modèle « objet », les vues sont également définies à l'aide d'une requête. La transposition du résultat du modèle relationnel sur le statut des vues consiste à conférer un statut de classes aux vues. Une classe-vue est la réification d'une requête possédant une intension et une extension. Cependant, la plus grande richesse sémantique du modèle « objet » pose, en effet, de nombreux problèmes à résoudre, dont le plus complexe est l'insertion des classes-vues considérées comme classes dans la hiérarchie de généralisation/spécialisation. Nous en verrons en 2.2.3 les solutions proposées pour insérer les classes-vues au sein de la hiérarchie de généralisation/spécialisation des classes du schéma de la base. Puis nous

2. qui sont des relations « virtuelles »

étudierons en 2.2.4 les techniques de vues qui déterminent une nouvelle relation et une nouvelle hiérarchie pour les classes-vues.

2.2.3 Techniques par insertion des classes-vues dans la hiérarchie des classes

Notre étude sur les techniques de classes-vues par insertion dans la hiérarchie des classes est faite au travers des approches COCOON [Scholl91, Tresch92, Scholl93, Tresch93, Scholl94, Tresch94] et MultiView [Rundensteiner92a-Rundensteiner92d, Rundensteiner94, Ra95] qui nous semblent être les travaux les plus avancés sur ces techniques. Il convient de noter que le SGBD O₂ supporte également les classes-vues et procède, dans certains cas, à leur classification dans la hiérarchie des classes [Abiteboul91]. Il permet également de définir des schémas-vues [Souza Dos Santos94].

2.2.3.1 Définition des classes-vues (COCOON et MultiView)

Pour pouvoir donner la définition des classes-vues, il nous faut d'abord préciser les spécificités du modèle de données « objet » retenu dans le cadre de COCOON et de MultiView ainsi que l'algèbre « objet » sous-tendant les requêtes.

2.2.3.1.1 Modèle de données

Avant d'étudier plus en détail le modèle de données de MultiView et de COCOON, il convient de préciser que ceux-ci possèdent les propriétés suivantes :

- les objets ne sont pas encapsulés. L'accès à l'intension³ des classes est donc libre pour permettre l'expression des requêtes ;
- la relation d'héritage est une relation d'héritage multiple ;
- ces approches relèvent de l'approche multi-intensionnelle permettant de décrire un objet de façon multiple par plusieurs intensions ;
- une instance peut appartenir à plusieurs classes, même si celles-ci sont indépendantes⁴ ;
- la classification des instances dans les classes et dans les classes-vues est réalisée de façon automatique et dynamique. Elle est basée sur l'héritage multiple.

Par ailleurs, ces deux modèles ne distinguent pas syntaxiquement l'accès à un attribut de l'invocation d'une méthode. Ceux-ci s'écrivent sous la forme de l'invocation d'une fonction : Pour une instance *o*, *f(o)* désigne :

- si *f* est un attribut, l'accès à la valeur de cet attribut ;
- si *f* est une méthode, l'invocation de cette méthode.

Le modèle de données de MultiView [Rundensteiner92a]

Chaque classe possède un nom unique, un type (intension) et une extension. Le type d'une classe *C*, notée **type(C)** est un ensemble de propriétés qui sont des fonctions, notées **properties(type(C))** ou **properties(C)**. Une propriété peut détenir une valeur d'un type de données de base⁵, une référence vers une instance, un ensemble de références ou une méthode. Pour un type *t*, **domain_p(t)** est le domaine de la propriété *p* dans le type *t*. L'extension d'une classe est un ensemble d'instances noté **extension(C)**. TV est l'ensemble des types et V l'ensemble des classes.

Le modèle définit un ordre partiel sur les types « est un sous-type de », noté $\leq$. Soit deux

3. L'intensions est appelée « type » dans ces approches

4. Deux classes sont indépendantes s'il n'existe pas de relation d'héritage entre ces classes [Carré89]

5. un littéral au sens de l'ODMG [ODMG93, ODMG97] (cf 3.2.1)

types t_1 et t_2 , t_2 est un sous-type de t_1 , c'est-à-dire $t_2 \leq t_1$ si et seulement si $\mathbf{properties}(t_1) \subseteq \mathbf{properties}(t_2)$ et $\forall p \in \mathbf{properties}(t_2), \mathbf{domain}_p(t_2) \leq \mathbf{domain}_p(t_1)$

Soit deux types t_1 et t_2 , l'opérateur binaire $\vee$ définit un nouveau type t_3 par $t_3 = t_1 \vee t_2$ tel que $\mathbf{properties}(t_3) = \mathbf{properties}(t_1) \cup \mathbf{properties}(t_2)$ et $\forall p \in \mathbf{properties}(t_3), \mathbf{domain}_p(t_3) = \mathbf{domain}_p(t_1) \vee \mathbf{domain}_p(t_2)$. t_3 est le plus grand minorant de t_1 et de t_2 .

Soit deux types t_1 et t_2 , l'opérateur binaire $\wedge$ définit un nouveau type t_3 par $t_3 = t_1 \wedge t_2$ tel que $\mathbf{properties}(t_3) = \mathbf{properties}(t_1) \cap \mathbf{properties}(t_2)$ et $\forall p \in \mathbf{properties}(t_3), \mathbf{domain}_p(t_3) = \mathbf{domain}_p(t_1) \wedge \mathbf{domain}_p(t_2)$. t_3 est le plus petit majorant de t_1 et de t_2 .

La hiérarchie des types forme un graphe orienté acyclique (DAG) dont les sommets sont les types et dont les arcs correspondent à la relation de sous-type direct.

Soit deux classes C_1 et C_2 , C_1 est une sous-classe de C_2 , noté C_1 **is-a** C_2 si et seulement si $\mathbf{type}(C_1) \leq \mathbf{type}(C_2)$ et $\mathbf{extension}(C_1) \subseteq \mathbf{extension}(C_2)$ ⁶.

Soit deux classes C_1 et C_2 appartenant à V , C_1 est une sous-classe directe de C_2 dans V , noté C_1 **is-a^d** C_2 si et seulement si C_1 **is-a** C_2 et s'il n'existe pas C_3 appartenant à V telle que C_1 **is-a** C_3 et C_3 **is-a** C_2 .

La hiérarchie des classes est également un DAG dont le sommet est appelé « Objects ».

Modèle de données de COCOON [Scholl91, Scholl94]

Le modèle de données de COCOON diffère assez peu de celui de MultiView. Il introduit la notion de relations qui est proche de celle du modèle de l'ODMG [ODMG93, ODMG97] (cf 3.2.1) avec contrôle de l'intégrité référentielle. COCOON ne permet pas d'affiner le domaine d'une propriété. Il faut aussi remarquer qu'une propriété est en fait une fonction dont le domaine correspond au type d'introduction de la propriété et le codomaine à ce qui vient d'être appelé (abusivement) le domaine de la propriété. Par conséquence, l'espace des noms des fonctions est unique et l'homonymie est impossible.

Il faut aussi remarquer en COCOON que les types sont définis indépendamment des classes. Un objet peut être décrit par un (ou plusieurs) types sans appartenir à une classe. Il est alors instance de ce ou de ces types⁷. Il existe, par conséquent, une hiérarchie des types (basée sur la relation d'ordre partiel $\leq$) et une hiérarchie séparée des classes (basée sur la relation d'ordre partiel **is-a**).

COCOON dispose d'opérateurs pour manipuler les différents types qui peuvent être attribués aux instances. L'opérateur « **gain** » permet à une instance d'être décrite par un nouveau type indépendamment des types qui le décrivaient déjà. Cette opération procède à une modification de la structure de l'instance. De façon analogue, l'opérateur « **lose** » permet à une instance de ne plus être décrite par un type. Il n'existe aucun contrôle sur les types qui peuvent être ajoutés à ou retirés de la description d'une instance, à la différence de ce qui est fait dans le langage ROME[Carré89]. D'ailleurs, une instance peut ne plus être décrite par son type initial, celui qui la décrivait lors de l'instanciation.

Une classe détient, de façon classique, un type, une extension et un nom. Lors de la définition de la classe, il faut spécifier un type qui aura déjà été défini au sein du schéma⁸. La relation **is-a** entre classes est définie de la même façon que dans MultiView. Pour pouvoir

6. Ce qui signifie aussi que les instances d'une classe appartiennent à l'extension de ses surclasses à la différence de certains modèles de bases de données à objets (cf [Kim90, Kim95])

7. d'où une notion d'extension de type en COCOON que nous n'étudierons pas

appartenir à l'extension d'une classe, une instance doit être décrite par le type de la classe⁹. L'acte d'appartenance d'une instance à l'extension d'une classe peut se faire explicitement par l'opérateur « **add** ». Inversement, cette appartenance peut ensuite être perdue par l'opérateur « **remove** ». On parle alors de classes ayant un statut de classe de description¹⁰, au sens des logiques terminologiques [Dekker94, Borgida89].

L'extension d'une classe peut être déterminée par un prédicat. Celui-ci peut définir totalement l'extension de la classe, déterminant ainsi une condition nécessaire et suffisante d'appartenance. La classe détient alors un statut de classe de définition¹¹. Dans ce cas, le classificateur détermine automatiquement l'extension de la classe : chaque instance du type de la classe vérifiant le prédicat appartient automatiquement à l'extension de la classe. Cette classification est dynamique et donc toute mise à jour d'une instance peut provoquer sa reclassification (perte de l'appartenance à certaines classes et gain de l'appartenance à d'autres). Les classes de définition peuvent être considérées comme des classes-vues, de par la détermination automatique de leur extension.

Le prédicat peut aussi ne décrire qu'une condition nécessaire. Dans ce cas, la classe conserve un statut de classe de description, le prédicat étant alors une contrainte d'intégrité devant être vérifiée pour toute instance appartenant à l'extension de la classe.

L'exemple suivant illustre l'utilisation des types et des classes en COCOON.

```
define type personne isa object = nom: string,  
    nationalité: string,  
    age: integer;  
    adresse: string;
```

Le type « personne » est défini comme un sous-type du type « object » (le sommet de la hiérarchie des types de COCOON).

```
define class Personne: personne some Objects;
```

La classe « Personne » est définie comme sous-classe de la classe « Objects » et dont les membres sont instances du type personne. Le mot-clé « **some** » indique que la condition d'appartenance est une condition nécessaire, conférant à « Personne » un statut de classe de description.

```
define class Jeunes: personne some Personne where age < 30;
```

La classe « Jeunes » est définie comme sous-classe de la classe « Personne » et dont les membres sont instances du type « personne » et vérifiant que la valeur de la propriété « age » est inférieure à 30. La classe « Jeunes » détient également un statut de classe de description.

```
define class Belges: personne all Personne where nationalité='B';
```

La classe « Belges » est définie comme sous-classe de la classe « Personne » et dont les membres sont instances du type « personne » et vérifiant que la valeur de la propriété « nationalité » vaille 'B'. Le mot-clé « **all** » indique que la condition d'appartenance est une

8. Ce type pourra déjà avoir été utilisé comme type d'une autre classe. Un même type peut constituer l'intension de plusieurs classes.

9. elle peut être décrite par d'autres types.

10. Une classe de description détermine des conditions nécessaires d'appartenance. L'appartenance d'une instance doit donc être exprimée impérativement.

11. Une classe de définition détermine des conditions nécessaires et suffisantes d'appartenance. L'appartenance d'une instance est déterminée automatiquement.

condition nécessaire et suffisante. De ce fait toute instance du type « personne » dont la valeur de la propriété « nationalité » vaut 'B' sera automatiquement classifiée dans la classe « Belges ». De telles classes de définition existent également dans le système « F2 » [Al-Jadir95].

Dans MultiView, le statut des classes n'est pas explicité. Si l'on regarde précisément la relation de subsumption telle qu'elle est définie dans [Rundensteiner92c], celle-ci ne peut être utilisée que pour des classes de définition. Cependant, il nous semble que la hiérarchie des classes puisse contenir à la fois des classes de description et de définition.

2.2.3.1.2 Algèbre « objet »

Les approches COCOON et MultiView introduisent une algèbre « objet » à l'instar de l'algèbre relationnelle. Celle-ci sert de support pour l'expression des requêtes qui sont formées par composition d'opérateurs. Les opérateurs de cette algèbre sont ensemblistes. Ils sont basés sur des ensembles d'instances qui peuvent être des extensions ou d'autres ensembles comme le résultat de fonctions ou d'autres opérateurs de l'algèbre. Le résultat des opérateurs est également un ensemble d'instances de classes existantes, préservant ainsi leur identité. Il est alors possible d'effectuer des opérations de mise à jour sur les éléments du résultat des requêtes. Respecter cette préservation de l'identité impose à l'algèbre « objet » d'exclure le produit cartésien ou la jointure. Ce dernier opérateur peut cependant être remplacé par une combinaison d'opérateurs préservant l'identité comme expliqué dans [Scholl91].

COCOON

La syntaxe et la sémantique des principaux opérateurs de l'algèbre « objet » de COCOON sont données par la table 2.1. Les opérateurs de sélection (« select »), de projection (« project »), d'augmentation (« extend ») sont des opérateurs unaires dont l'utilisation répond à la syntaxe suivante :

define view résultat **as** opérateur[paramètres](opérande);

Les paramètres peuvent être un prédicat (« select »), une liste d'attributs existants (« project ») ou une liste de nouveaux attributs (« extend »).

Les opérateurs d'union (« union »), d'intersection (« intersect ») et de différence (« difference ») sont des opérateurs binaires dont l'utilisation répond à la syntaxe suivante :

define view résultat **as** opérande1 opérateur opérande2;

Ces opérateurs sont ceux du langage COOL-QL¹². Pour chaque opérateur, le résultat considéré comme une classe-vue est comparé aux opérandes vis-à-vis de la relation « is-a », c'est-à-dire que son type est comparé aux types des opérandes vis-à-vis de la relation de sous-type ($\leq$) et son extension est comparée aux extensions des opérandes vis-à-vis de la relation d'inclusion d'ensembles. Cette comparaison sera utile par la suite pour insérer les classes-vues dans la hiérarchie des classes.

12. le langage de requêtes de COCOON

Table 2.1: Sémantique des principaux opérateurs de l'algèbre « objet » de COCOON

Opérateur	Syntaxe et sémantique
Sélection d'un opérande E par un prédicat P	<u>Syntaxe</u> select[P](E) <u>Sémantique</u> $\text{type}(\text{résultat}) := \text{type}(E)$ $\text{extension}(\text{résultat}) := \{x \in \text{extension}(E) \mid P(x) \text{ soit vrai}\}$ <u>Comparaison résultat/opérande</u> $\text{type}(\text{résultat}) \leq \text{type}(E)$ $\text{extension}(\text{résultat}) \subseteq \text{extension}(E)$ Si E est une classe, alors résultat is-a E sinon le résultat n'est comparable avec aucune autre classe et est placé directement sous la classe « Objects ».
Projection d'un opérande E sur une liste de fonctions $f_1, f_2, \dots, f_n$	<u>Syntaxe</u> project[$f_1, f_2, \dots, f_n$](E) avec $\{f_1, f_2, \dots, f_n\} \subseteq \text{type}(E)$ <u>Sémantique</u> $\text{type}(\text{résultat}) := \{f_1, f_2, \dots, f_n\}$ $\text{extension}(\text{résultat}) := \text{extension}(E)$ <u>Comparaison résultat/opérande</u> $\text{type}(E) \leq \text{type}(\text{résultat})$ $\text{extension}(E) \subseteq \text{extension}(\text{résultat})$ Si E est une classe, alors E is-a résultat sinon le résultat n'est comparable avec aucune autre classe et est placé directement sous la classe « Objects ».
Opérateur d'augmentation définissant une ou plusieurs nouvelles fonctions $g_1, g_2, \dots, g_m$ aux éléments d'un opérande E	<u>Syntaxe</u> extend[$g_1 := \text{expr}_1, g_2 := \text{expr}_2, \dots, g_m := \text{expr}_m$](E) <u>Sémantique</u> $\text{type}(\text{résultat}) := \text{type}(E) \cup \{g_1, g_2, \dots, g_m\}$ $\text{extension}(\text{résultat}) := \text{extension}(E)$ <u>Comparaison résultat/opérande</u> $\text{type}(\text{résultat}) \leq \text{type}(E)$ $\text{extension}(\text{résultat}) \subseteq \text{extension}(E)$ Si E est une classe, alors résultat is-a E sinon le résultat n'est comparable avec aucune autre classe et est placé directement sous la classe « Objects ».

Table 2.1: Sémantique des principaux opérateurs de l'algèbre « objet » de COCOON

Opérateur	Syntaxe et sémantique
Union de E_1 et de E_2	<u>Syntaxe</u> E_1 union E_2 <u>Sémantique</u> $\text{type}(\text{résultat}) = \text{type}(E_1) \wedge \text{type}(E_2)$ $\text{extension}(\text{résultat}) := \text{extension}(E_1) \cup \text{extension}(E_2)$ <u>Comparaison résultat/opérande</u> $\text{type}(E_1) \leq \text{type}(\text{résultat})$ $\text{type}(E_2) \leq \text{type}(\text{résultat})$ $\text{extension}(E_1) \subseteq \text{extension}(\text{résultat})$ $\text{extension}(E_2) \subseteq \text{extension}(\text{résultat})$ Si E_1 est une classe, alors E_1 is-a résultat Si E_2 est une classe, alors E_2 is-a résultat sinon le résultat n'est comparable avec aucune classe de la hiérarchie
Intersection de E_1 et de E_2	<u>Syntaxe</u> E_1 intersect E_2 <u>Sémantique</u> $\text{type}(\text{résultat}) = \text{type}(E_1) \vee \text{type}(E_2)$ $\text{extension}(\text{résultat}) := \text{extension}(E_1) \cap \text{extension}(E_2)$ <u>Comparaison résultat/opérande</u> $\text{type}(\text{résultat}) \leq \text{type}(E_1)$ $\text{type}(\text{résultat}) \leq \text{type}(E_2)$ $\text{extension}(\text{résultat}) \subseteq \text{extension}(E_1)$ $\text{extension}(\text{résultat}) \subseteq \text{extension}(E_2)$ Si E_1 est une classe, alors résultat is-a E_1 Si E_2 est une classe, alors résultat is-a E_2 sinon le résultat n'est comparable avec aucune classe de la hiérarchie
Différence de E_1 et de E_2	<u>Syntaxe</u> E_1 difference E_2 <u>Sémantique</u> $\text{type}(\text{résultat}) := \text{type}(E_1)$ $\text{extension}(\text{résultat}) := \{x \in \text{extension}(E_1) \text{ et } x \notin \text{extension}(E_2)\}$ <u>Comparaison résultat/opérande</u> $\text{type}(\text{résultat}) \leq \text{type}(E_1)$ $\text{extension}(\text{résultat}) \subseteq \text{extension}(E_1)$ Si E_1 est une classe, alors résultat is-a E_1 sinon le résultat n'est comparable avec aucune classe de la hiérarchie

MultiView

MultiView introduit un opérateur « hide » à la place de l'opérateur « project » consistant à masquer les fonctions non pertinentes pour l'utilisateur de la classe-vue. L'opérateur d'augmentation¹³ permet d'ajouter non seulement des fonctions calculées mais également des

13. appelé « refine » dans MultiView

propriétés stockées, en augmentant la structure des instances [Ra95]. Notons aussi que tous les opérandes sont systématiquement des classes en MultiView. La table 2.1 donne la syntaxe et la sémantique des opérateurs « hide » et « refine » ainsi que le placement de leur résultat en tant que classe-vue par rapport aux opérandes.

Table 2.2: Sémantique des opérateurs « hide » et « refine » spécifiques à l’algèbre « objet » de MultiView

Opérateur	Syntaxe et sémantique
Masquage du type d’une classe C relativement à une liste de fonctions $f_1, f_2, \dots, f_n$	<u>Syntaxe</u> hide[$f_1, f_2, \dots, f_n$](C) avec $\{f_1, f_2, \dots, f_n\} \subseteq \text{type}(C)$ <u>Sémantique</u> $\text{type}(\text{résultat}) := \text{type}(C) - \{f_1, f_2, \dots, f_n\}$ $\text{extension}(\text{résultat}) := \text{extension}(C)$ <u>Comparaison résultat/opérande</u> $\text{type}(C) \leq \text{type}(\text{résultat})$ $\text{extension}(C) \subseteq \text{extension}(\text{résultat})$ C is-a résultat
Opérateur d’augmentation définissant une ou plusieurs nouvelles fonctions $g_1, g_2, \dots, g_m$ aux éléments d’une classe C. A la différence de COCOON, ces fonctions peuvent être des propriétés stockées modifiant la structure des instances. Elles peuvent également être une propriété héritée du type d’une autre classe (ou classe-vue).	<u>Syntaxe</u> Cas de l’augmentation par une fonction calculée : refine[$p := \text{expr}$](C) Cas de l’augmentation par une propriété « p » stockée : refine[$p < \text{domain}_p >$](C) Cas de l’augmentation par une propriété « p » héritée du type d’une autre classe (ou classe-vue) refine[$< \text{classe} > : p$](C) <u>Sémantique</u> $\text{type}(\text{résultat}) := \text{type}(C) \cup \{p\}$ $\text{extension}(\text{résultat}) := \text{extension}(C)$ <u>Comparaison résultat/opérande</u> $\text{type}(\text{résultat}) \leq \text{type}(C)$ $\text{extension}(\text{résultat}) \subseteq \text{extension}(C)$ résultat is-a C

Précision sur le statut des classes-vues

Les classes-vues sont définies à l’aide d’une requête basée sur les différents opérateurs de l’algèbre « objet ». Leur type et leur extension sont celles du résultat de la requête. L’appartenance d’une instance à une classe-vue est déterminée automatiquement par cette requête. La requête détermine l’ensemble des conditions nécessaires et suffisantes d’appartenance à l’extension de la classe-vue. Une classe-vue est donc une classe de définition. Par conséquent, elle peut être classifiée automatiquement dans la hiérarchie des classes du schéma. Cette classification sera étudiée en 2.2.3.2.

2.2.3.1.3 Exemples

Pour chacun des opérateurs ci-dessus (à l’exception de celui de l’application d’une fonction), nous donnons un exemple, en prenant pour opérandes des classes. Le schéma des classes qui vont servir d’opérandes pour les différents opérateurs est le suivant :

```
define type actif isa object = nom: string,
  nationalite: string,
  annee_de_naissance: integer,
```

```

    adresse: string,
    adresse_professionnelle: string;
define class Actifs: actif some Objects;
define type salarie isa actif = salaire: integer;
define class Salaries: salarie some Actifs;
define type prof_liberale isa actif = profession: string;
define class Prof_Liberales: prof_liberale some Actifs;
define type commerçant isa actif = type_commerce: string;
define class Commerçants: commerçant some Actifs;

```

Opérateur de sélection

```

define view SalariesLillois as select ['Lille' in adresse] (Salaries);

```

La figure 2.2 montre que la classe-vue « SalariesLillois » est une sous-classe de « Salaries ». En effet, le type de « SalariesLillois » est « actif » comme pour « Salaries » et l'extension de « SalariesLillois » est incluse dans l'extension de « Salaries ».

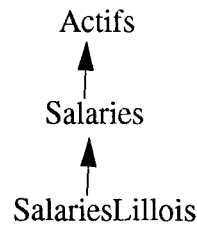


Figure 2.2. Le résultat de l'opérateur de sélection est une sous-classe

Opérateur de projection

```

define view Personnes as project[nom, annee_de_naissance, adresse](Actifs);

```

Le figure 2.3 montre que la classe-vue « Personnes » est une surclasse de « Actifs ». Son type est un surtype du type de « Actifs », c'est-à-dire « actif ». L'extension de la classe « Personnes » est la même extension que la classe « Actifs ».

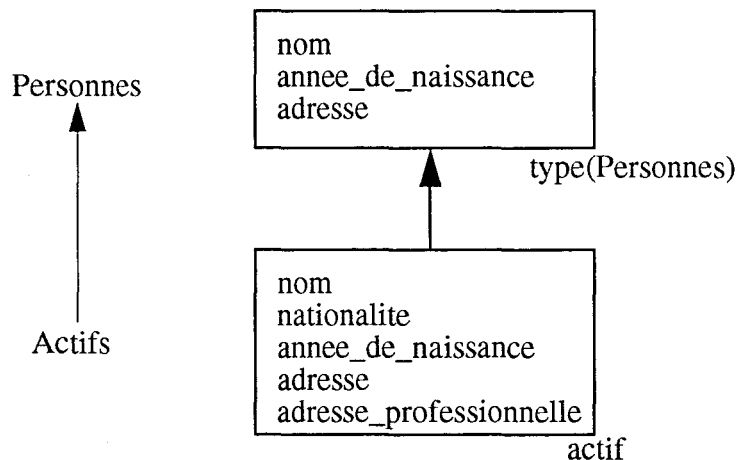


Figure 2.3. Le résultat de l'opérateur de projection est une surclasse

Opérateur d'augmentation

define view Personnes2 **as extend**[age = annee_courante() - annee_de_naissance](Personnes);

La figure 2.4 montre que la classe-vue « Personnes2 » est une sous-classe de « Personnes ». Son type est un sous-type de celui de « Personnes » et son extension est égale à celle de « Personnes ».

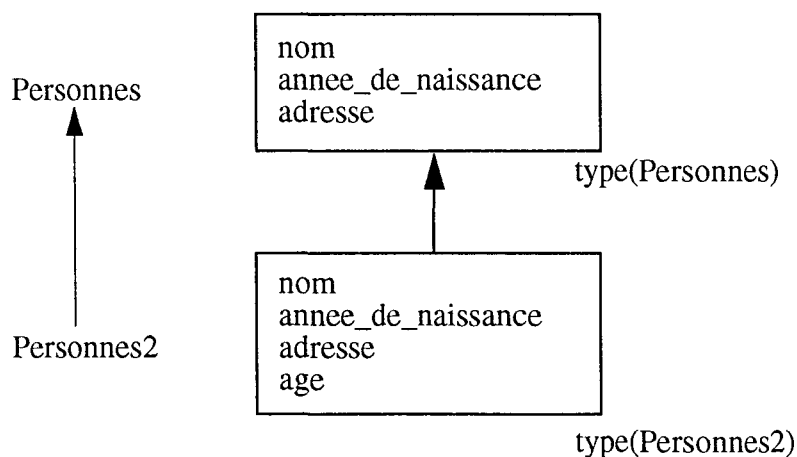


Figure 2.4. Le résultat de l'opérateur d'augmentation est une sous-classe

Opérateur d'union

define view NonSalaries **as** Prof_Liberales **union** Commerçants;

La figure 2.5 montre que la classe-vue « NonSalaries » est une surclasse des classes « Prof_Liberales » et « Commerçants ». Son extension est l'union des extensions des classes « Prof_Liberales » et « Commerçants ». Son type est le plus petit majorant des types de « Prof_Liberales » et « Commerçants ». Ici, dans le cadre de cet exemple, il s'avère qu'il est égal à « actif ».

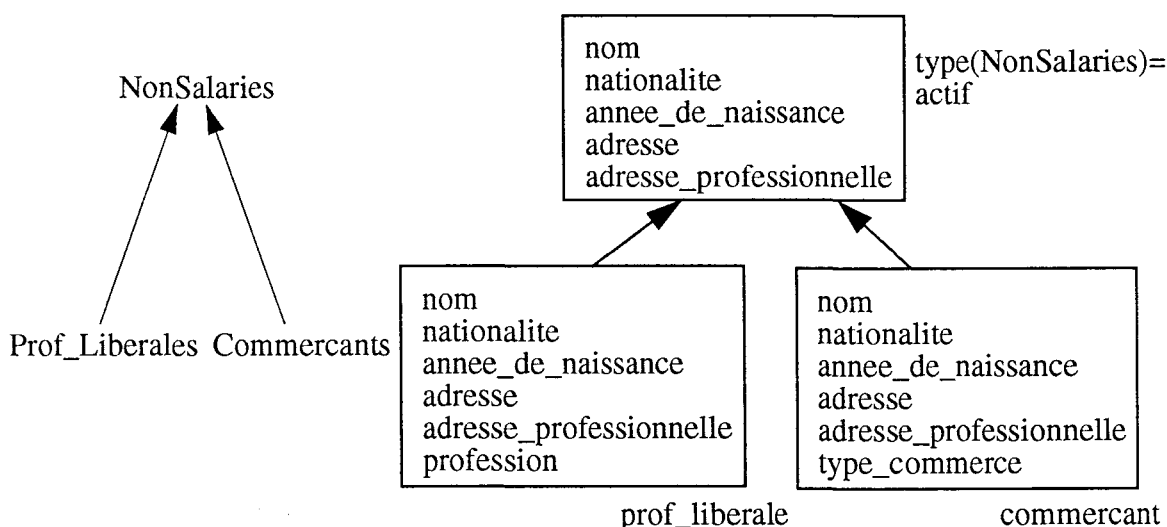


Figure 2.5. Le résultat de l'opérateur d'union est une surclasse des deux opérantes

Opérateur d'intersection

define view LiberauxSalaries **as** Prof_Liberales **intersect** Salaries;

La figure 2.6 montre que la classe « LiberauxSalaries » est une sous-classe des classes « Prof_Liberales » et « Salaries ». Son extension est l'intersection des deux classes « Prof_Liberales » et « Salaries » et son type est le plus grand minorant des types de ces deux classes.

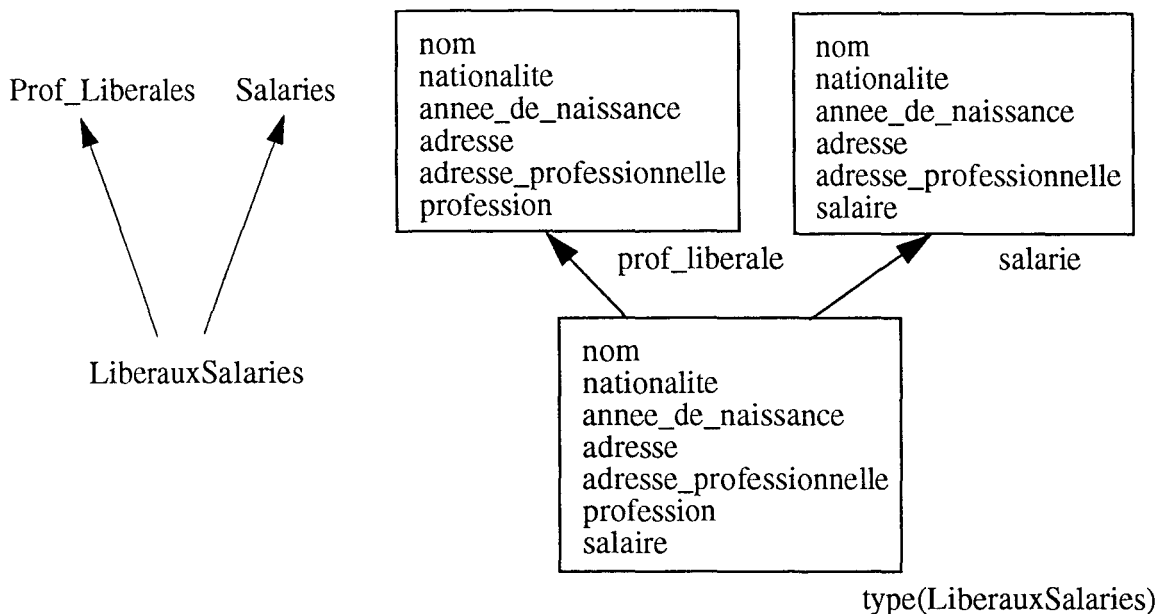


Figure 2.6. Le résultat de l'opérateur d'intersection est une sous-classe des deux opérandes

2.2.3.2 Classification (COCOON et MultiView)

Le statut de classe de définition étant conféré aux classes-vues, il est donc possible d'insérer celles-ci dans la hiérarchie des classes par classification. COCOON et MultiView propose ainsi une classification des classes-vues basée sur des approches d'insertion d'un type dans un treillis¹⁴ [Missikoff89] ainsi que sur la classification des concepts définis dans les logiques terminologiques [Borgida89]. Cette classification de la classe-vue se fait en deux temps :

- insertion du type de la classe-vue dans la hiérarchie des types ;
- insertion de la classe-vue dans la hiérarchie des classes.

Si l'insertion d'un type dans une hiérarchie de types est assez bien maîtrisée¹⁵, il n'en va pas de même pour l'insertion de la classe-vue. En effet, celle-ci requiert des comparaisons d'extension, ce qui se traduit ici par des comparaisons de prédicats¹⁶ qui sont, d'une manière générale, indécidables¹⁷. Cette classification est donc nécessairement incomplète et n'est donc pas systématiquement choisie. La figure 2.7 illustre à la fois le cas où la comparaison des

14. pouvant nécessiter une restructuration de la hiérarchie de types avant l'insertion

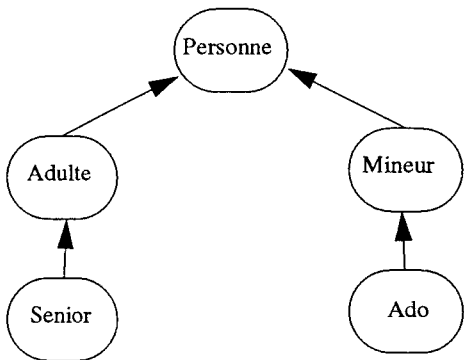
15. comparer des structures est une opération décidable et d'un coût acceptable

16. pour les classes-vues définies par une sélection

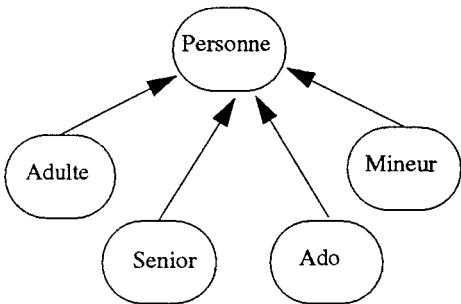
17. Surtout avec la possibilité d'utiliser dans les prédicats des fonctions calculées

extensions des classes-vues est réalisée et le cas où elle ne l'est pas. Dans ce dernier cas, nous obtenons une classification incomplète car celle-ci n'est réalisée que relativement aux types.

```
define type personne =nom: string,  
age: integer;  
  
define class Personne : Personne some Objects;  
define view Adulte as select[age>=18](Personne);  
define view Senior as select[age>=60](Personne);  
define view Mineur as select[age<18](Personne);  
define view Ado as select[(age<18) and (age>=13)](Personne);
```



Classification complète des classes-vues (comparaison des extensions)



Classification incomplète des classes-vues (pas de comparaison des extensions)

Figure 2.7. Classification et non-classification de l’extension des classes-vues

Pour COCOON puis pour MultiView, nous donnons maintenant les caractéristiques principales de leur algorithme de classification. Le mécanisme de MultiView sera étudié de façon plus fine car il nous servira de support pour la mise en œuvre de CROME en MultiView au chapitre 4.

COCOON

L’algorithme de classification introduit dans le prototype de COCOON est décrit dans [Nguyen91]. Celui-ci est essentiellement basé sur le placement de la classe-vue par rapport à son ou ses opérandes. Il prend en compte la distinction entre classes de description et classes de définition. Il procède à la comparaison des prédicats pour les classes-vues de sélection mais

en se restreignant à une forme limitée mais comparables de prédicats. La forme des prédicats comparables est précisée dans [Nguyen91]. Cette comparaison de prédicats de forme restreinte permet à l'algorithme de classification de COCOON de fournir le premier résultat de la figure 2.7.

Par contre, cet algorithme ne procède pas à une restructuration de la hiérarchie des types avant insertion du type de la classe-vue, ce qui permettrait de garantir que chaque propriété soit introduite par un seul type¹⁸ et d'assurer ainsi une meilleure cohérence de la hiérarchie. Ce besoin de restructuration est cependant abordé dans [Scholl91].

MultiView

L'algorithme de classification de MultiView est décrit dans [Rundensteiner94] et de façon plus précise dans [Rundensteiner92c]. Cet algorithme ne procède pas à la comparaison des prédicats aboutissant donc à une classification incomplète des classes-vues, notamment celles de sélection. Par conséquent, dans le cas de la figure 2.7, l'algorithme de classification de MultiView fournit le second résultat.

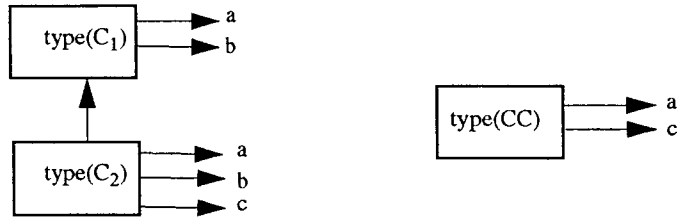
L'algorithme de classification consiste d'abord en une préparation de la hiérarchie des types afin que de pouvoir insérer correctement le type de la vue dans celle-ci. Cette préparation consiste en la création de nouveaux types dans la hiérarchie. Pour chacun de ces nouveaux types, une classe est créée dans la hiérarchie des classes afin de la préparer à son tour. Enfin, l'algorithme procède au placement de la classe-vue dans la hiérarchie des classes ainsi préparée.

Nous détaillons maintenant ces trois étapes.

Préparation de la hiérarchie des types

Le besoin de préparation de la hiérarchie des types provient de la volonté d'imposer dans celle-ci qu'il n'y ait qu'un seul type qui introduise chaque propriété (ou fonction) des objets. En effet, sans préparation, l'insertion directe de certaines classes-vues provoque un manque de respect de cet énoncé, comme l'illustre la figure 2.8. En effet, la classe-vue « CC » peut être classifiée comme classe indépendante de « C_1 » et de « C_2 », mais alors les propriétés « a » et « c » seront introduites dans deux types distincts, à savoir « type(C_1) » et « type(CC) » pour « a » et « type(C_2) » et « type(CC) » pour « c ». La préparation de la hiérarchie des types vise à permettre l'insertion d'une telle classe-vue en faisant en sorte que les propriétés « a » et « c » soient, chacune, introduites par un seul type.

18. Nous reviendrons de façon plus détaillée sur ce besoin de restructuration lors de l'étude de l'insertion des types dans le cadre de MultiView



define view CC as hide[b](C2);

Figure 2.8. Classification et respect de l'introduction unique de chaque propriété

L'introduction de chaque propriété par un type unique peut être formalisée par la notion de fermeture de la hiérarchie de types relativement à l'opérateur « $\wedge$ » :

Une hiérarchie de types est fermée par rapport à l'opérateur « $\wedge$ » si et seulement si, quels que soient deux types T_1 et T_2 de cette hiérarchie, il existe un type T_3 appartenant également à cette hiérarchie tel que $T_3 = T_1 \wedge T_2$, c'est-à-dire que le plus petit majorant de chaque paire de types soit présent¹⁹.

Si l'on reprend le cas de la figure 2.8, on peut se rendre compte que la hiérarchie formée des types de « C_1 », « C_2 » et « CC » n'est pas fermée par rapport à « $\wedge$ ». En effet, $\text{type}(C_1) \wedge \text{type}(CC) = \{a\}$ qui est un type inexistant dans la hiérarchie. L'insertion du type de la classe-vue sans préparation a rendu la hiérarchie non fermée.

Pour garantir cette fermeture après insertion du type de la classe-vue, la hiérarchie des types est préparée par insertion de tous les plus petits majorants entre le type de la classe-vue et un type de la hiérarchie existant avant l'insertion. C'est-à-dire que, pour classer une classe-vue CC , les types suivants sont créés et insérés

$\forall T \in TV, T \wedge \text{type}(CC)$ est inséré dans « TV » s'il n'y est déjà pas présent.

Un tel majorant est aisé à insérer, $T \wedge \text{type}(CC)$ est un surtype de T . L'insertion de ces majorants permet de garantir que la hiérarchie des types reste fermée par rapport à « $\wedge$ » après l'insertion du type de la classe-vue.

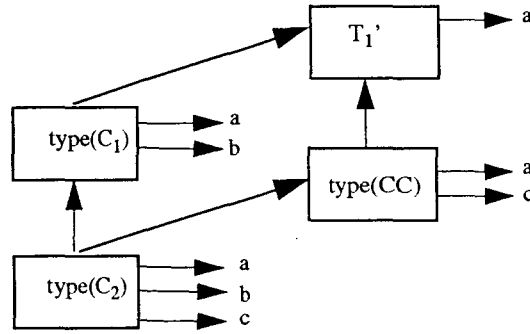
Par exemple, l'insertion de ces majorants dans le cas de la figure 2.8 donne la hiérarchie illustrée à la figure 2.9. Les deux majorants à créer et insérer sont :

$\text{type}(C_1) \wedge \text{type}(CC) = \{a\} = T_1'$

$\text{type}(C_2) \wedge \text{type}(CC) = \{a, c\} = \text{type}(CC)$

La hiérarchie de la figure 2.9 est bien fermée par rapport à l'opérateur « $\wedge$ ».

19. La hiérarchie de types est alors un demi-treillis supérieur



define view CC as hide[b](C2);

Figure 2.9. Préparation de la hiérarchie des types

Dans [Rundensteiner94], est exposée une technique d'optimisation pour diminuer le nombre de majorants à créer. Nous ne présentons pas cette technique ici.

Préparation de la hiérarchie des classes

Pour chaque type créé et inséré dans la hiérarchie des types, est créée et insérée dans la hiérarchie des classes une classe C pour l'insertion d'une classe-vue CC :

$\forall C \in V$, est créée et insérée, si elle n'est pas déjà présente, la classe C' telle que :

$$\text{type}(C') = \text{type}(C) \wedge \text{type}(CC)$$

$$\text{extension}(C') = \text{extension}(C) \cup \text{extension}(CC)$$

L'extension d'une classe ainsi créée est telle que cette classe est une surclasse à la fois de la classe de la hiérarchie dont elle est issue et de la classe-vue à insérer, respectant la hiérarchie des types. Dans le cas de l'exemple introduit à la figure 2.8, la hiérarchie des classes après préparation est isomorphe à la hiérarchie des types illustrée à la figure 2.9. Les deux classes créées sont les suivantes :

$$\text{type}(C_1') = \text{type}(C_1) \wedge \text{type}(CC) = \{a\}$$

$$\text{extension}(C_1') = \text{extension}(C_1) \cup \text{extension}(CC)$$

La classe « C_1' » est une surclasse de « C_1 » et de « CC ». La seconde classe à créer est, en fait, égale à la classe-vue « CC » :

$$\text{type}(C_2) \wedge \text{type}(CC) = \{a, c\} = \text{type}(CC)$$

$$\text{extension}(C_2) \cup \text{extension}(CC) = \text{extension}(CC) \text{ car } \text{extension}(CC) = \text{extension}(C_2)^{20}.$$

Insertion de la classe-vue dans la hiérarchie des classes

Une fois la hiérarchie des types et la hiérarchie des classes préparées, l'insertion de la classe-vue dans la hiérarchie des classes est possible en respectant la fermeture de la hiérarchie des types vis-à-vis de l'opérateur « $\wedge$ ». Celle-ci se fait par comparaisons successives basées sur la relation de subsomption définie ainsi :

$$\text{subsume}(C1, C2) = \text{vrai si } C2 \text{ is-a } C1$$

20. cf la sémantique de l'opérateur **hide**.

subsume(C1,C2) = faux sinon.

La subsomption ne peut cependant être déterminée ainsi qu'entre classes de définition, ce qui n'est explicité ni dans [Rundensteiner92c] ni dans [Rundensteiner94].

2.2.3.3 Schémas-vues (MultiView)

MultiView possède un mécanisme de construction de schémas-vues. Un schéma-vue est un ensemble de classes et de classes-vues nécessaires pour un utilisateur ou un groupe d'utilisateurs de la base de données. Celui-ci ne perçoit et ne manipule ainsi que les informations pertinentes à son utilisation de la base de données. Ces informations sont alors décrites par les différentes classes et classes-vues du schéma-vue. Un schéma-vue correspond exactement au niveau externe du modèle de l'architecture ANSI/X3/SPARC (cf figure 2.1).

Nous retenons pour cette étude MultiView mais celle-ci n'est pas la seule base de données à objets à offrir les schémas-vues (voir aussi O_2 dans [Souza Dos Santos94]). A noter que les schémas-vues sont inexistant dans COCOON.

2.2.3.3.1 Définition

Le schéma global d'une base de données est un graphe dirigé acyclique noté $GS = (V,E)$ où V est l'ensemble des classes et E un ensemble fini d'arcs orientés correspondant à la relation binaire **is-a^d** sur $V \times V$.

Le schéma de base est un schéma noté $BS=(BV,BE)$ contenant toutes les classes de base ainsi que les relations **is-a^d** dans cet ensemble. Les classes de base sont les classes dont le type est défini explicitement²¹. Le schéma de base correspond au schéma conceptuel de l'architecture ANSI/X3/SPARC.

Le schéma global « GS » est l'union du schéma de base « BS » et de toutes les classes-vues ainsi que les relations **is-a^d** dans cet ensemble²². Il correspond, en quelque sorte, à l'union de tous les éléments du schéma conceptuel et des schémas externes de l'architecture ANSI/X3/SPARC.

Un sous-graphe du schéma global est un schéma-vue noté $VS=(VV,VE)$ à la condition qu'il vérifie les propriétés suivantes :

1. VS possède un nom unique noté $\langle VS \rangle$;
2. $VV \subseteq V$;
3. $VE \subseteq \text{fermeture-transitive}(E)$.

La troisième propriété signifie que la relation **is-a** entre les classes du schéma-vue est dérivable de celle existante dans GS et ne contient pas de nouvelles occurrences.

Un schéma-vue est construit à l'aide d'un ensemble de commandes permettant de sélectionner des classes ou des classes-vues du schéma global.

Les principales commandes pour définir et pour modifier des schémas-vues sont :

- DEFINE-VIEW $\langle \text{schema-view-name} \rangle$;
- MODIFY-VIEW $\langle \text{schema-view-name} \rangle$;
- ADD-CLASS($\langle \text{class-name} \rangle$);
- ADD-CLASS-DAG($\langle \text{class-name} \rangle$);
- REMOVE-CLASS($\langle \text{class-name} \rangle$);

21. à l'opposé des classes-vues

22. les classes-vues sont insérées par classification dans le schéma global.

• REMOVE-CLASS-DAG(<class-name>);

« DEFINE-VIEW » et « MODIFY-VIEW » permettent respectivement de définir et de modifier un schéma-vue à l'aide des commandes « ADD-CLASS », « ADD-CLASS-DAG », « REMOVE-CLASS » et « REMOVE-CLASS-DAG ». La suite de commandes de définition ou de modification doit se terminer par « END-VIEW ».

« ADD-CLASS » et « REMOVE-CLASS » permettent respectivement d'ajouter et de supprimer une classe ou une classe-vue à un schéma-vue

« ADD-CLASS-DAG » et « REMOVE-CLASS-DAG » permettent respectivement d'ajouter et de supprimer une classe ou une classe-vue et toutes ses sous-classes à un schéma-vue.

Exemple

Nous reprenons l'exemple issu de 2.2.3.1.3. Notre objectif est de créer un schéma-vue contenant les classes « Prof_Liberales » et « Commerçants » et les classes-vues « NonSalaries » et « Personnes ». Le schéma-vue NON-SALARIES est alors défini ainsi :

```
DEFINE-VIEW NON-SALARIES;
ADD-CLASS(Prof_Liberales);
ADD-CLASS(Commerçants);
ADD-CLASS(NonSalaries);
ADD-CLASS(Personnes);
END-VIEW;
```

2.2.3.3.2 Fermeture du schéma-vue par rapport aux domaines des attributs

Le schéma-vue est construit à partir d'une sélection explicite de classes et de classes-vues du schéma global. La propriété de fermeture de ce schéma consiste à s'assurer que les classes dont le type est domaine de l'une des propriétés de l'une des classes du schéma soient présentes dans ledit schéma. La fermeture concerne les propriétés dont le domaine est un type autre qu'un littéral.

Pour donner une définition plus formelle d'un schéma-vue fermé, nous commençons par définir la fonction « Uses ». « Uses » est une fonction de $VV \rightarrow 2^{VV}$ définie par :

Soit $C \in VV$, $Uses(C) = \{C' \in VV \mid \exists p \in \text{properties}(C) \text{ et } \text{domain}_p(C) = \text{type}(C')\}$

Uses est également définie de $2^{VV} \rightarrow 2^{VV}$ par :

Soit $S \in 2^{VV}$, $Uses(S) = \bigcup_{C \in S} Uses(C)$

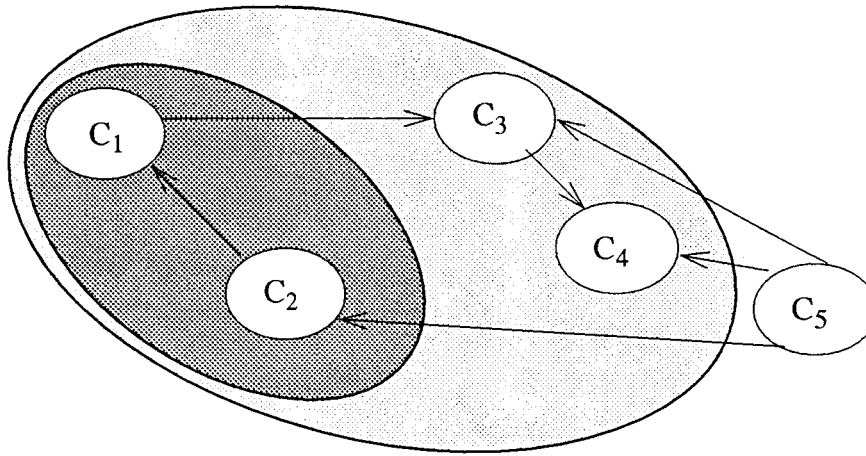
$Uses^*$ est ensuite définie par :

$Uses^*(S) = \bigcup_{j=1, |S|} Uses^j(S)$ où pour $j > 1$ $Uses^j(S) = Uses^{j-1}(Uses(S))$ et $Uses^1(S) = Uses(S)$

VV est fermé si et seulement si $VV = Uses^*(VV) \cup VV$

MultiView vérifie que le schéma-vue construit par l'utilisateur est fermé. Si ce n'est pas le cas, il calcule $Uses^*(VV)$ afin de lui adjoindre les classes et classes-vues nécessaires pour assurer la fermeture de VV . Dans [Rundensteiner92b], il est démontré que l'ensemble $MIN = Uses^*(VV) - VV$ est l'ensemble minimal des classes à adjoindre à VV pour assurer sa fermeture. Un algorithme est également décrit pour calculer de façon efficace cet ensemble.

La figure 2.10 illustre un exemple de schéma-vue non fermé ainsi que l'ensemble MIN , ensemble minimal des classes nécessaires pour le fermer.



Chaque flèche représente une propriété dont le domaine est le type de la classe pointée

-  Schéma-vue initial $VS=\{C_1, C_2\}$ non fermé
L'ensemble minimal des classes à adjoindre est $MIN = \{C_3, C_4\}$
-  Schéma-vue $VS=\{C_1, C_2\}$ fermé après adjonction de MIN

Figure 2.10. Exemple de schéma-vue non fermé et de l'ensemble MIN des classes à lui adjoindre pour assurer sa fermeture

2.2.3.3.3 Validité du schéma-vue par rapport à la relation *is-a*

L'utilisateur détermine explicitement les classes et classes-vues qu'il choisit d'intégrer dans le schéma-vue. MultiView construit alors automatiquement les liens de la relation *is-a* entre les classes et classes-vues sélectionnées à l'aide d'un algorithme décrit dans [Rundensteiner92d]. Cet algorithme génère un schéma-vue valide par rapport à la relation *is-a*. Un schéma-vue est valide s'il est complet, minimal et consistant, trois propriétés que nous définissons maintenant.

Complétude

Soit un schéma-vue $VS=(VV, VE)$ défini sur le schéma global $GS=(V, E)$. Soit deux classes C_1 et C_2 appartenant à VV . Un arc $\langle C_1, C_2 \rangle$ de VE représentant la relation C_1 *is-a*^d C_2 est requis pour VS si et seulement si C_1 *is-a* C_2 dans GS et s'il n'existe pas C_3 appartenant à VV telle que C_1 *is-a* C_3 et C_3 *is-a* C_2 dans GS . VS est complet si et seulement si VE contient tous les arcs requis pour VS .

Minimalité

Soit un schéma-vue $VS=(VV, VE)$ défini sur le schéma global $GS=(V, E)$. Soit deux classes C_1 et C_2 appartenant à VV . Un arc $\langle C_1, C_2 \rangle$ de VE représentant la relation C_1 *is-a*^d C_2 est redondant dans VS si et seulement si il existe C_3 appartenant à VV telle que C_1 *is-a* C_3 et C_3 *is-a* C_2 dans GS . VS est minimal si et seulement si VE ne contient aucun arc redondant dans VS .

Consistance

Soit un schéma-vue $VS=(VV, VE)$ défini sur le schéma global $GS=(V, E)$. Soit deux classes C_1 et C_2 appartenant à VV . Un arc $\langle C_1, C_2 \rangle$ de VE représentant la relation C_1 *is-a*^d C_2

est incompatible pour VS si et seulement si C_1 is-a C_2 est faux dans GS. VS est consistant si et seulement si VE ne contient aucun arc incompatible pour VS.

La figure 2.11 illustre un exemple de schéma global et de schémas-vues non valides.

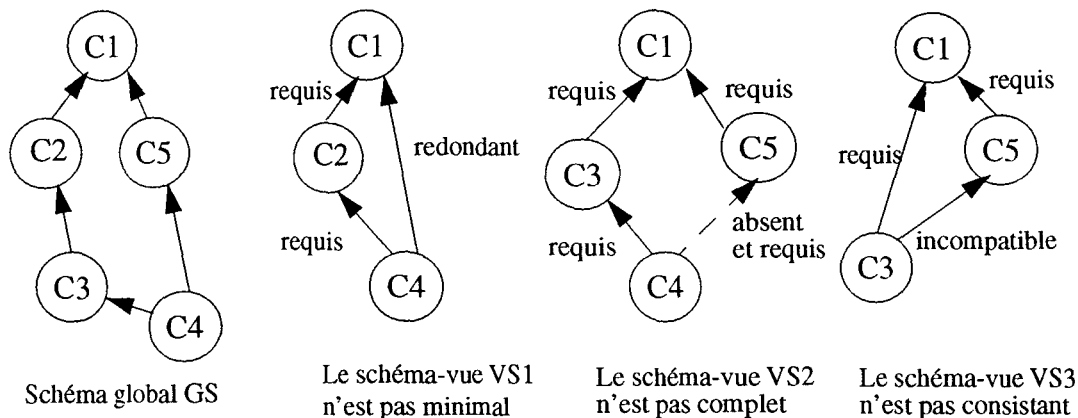


Figure 2.11. Exemple de schémas-vues non valides

Par ailleurs, dans [Rundensteiner92b], est aussi démontrée l'indépendance des schémas-vues. L'indépendance des schémas-vues se définit par le fait que les caractéristiques et la sémantique du schéma de base et des schémas-vues existants ne sont pas modifiées lors de la définition d'un nouveau schéma-vue. Ce résultat rend possible la construction et la modification de schémas-vues de façon incrémentale.

2.2.4 Techniques sans insertion des classes-vues dans la hiérarchie des classes

Il existe de nombreuses techniques qui n'insèrent pas les classes-vues dans la hiérarchie de généralisation/spécialisation des classes du schéma conceptuel. Les classes-vues restent définies par une requête. Elles possèdent un type calculé de la même façon qu'en 2.2.3 et une extension peuplée par les instances de leur(s) classe(s) de base²³. Les classes-vues sont alors liées à leur(s) classe(s) de base par une relation spécifique appelé « relation de dérivation » [Bertino92]. La relation de dérivation ne fait pas partie de l'approche « objet » [Kim95]. Elle assure la correspondance entre le type et l'extension de la classe-vue et le type et l'extension de la ou des classes de base, ce qui est nécessaire lors des requêtes et des mises à jour effectuées sur la classe-vue. Cette relation est le support d'une nouvelle hiérarchie appelée « hiérarchie de dérivation » [Bertino92].

Parmi les raisons pour ne pas insérer les classes-vues dans la hiérarchie de généralisation/spécialisation, citons :

- complexité accrue de la hiérarchie de généralisation/spécialisation des classes dans le schéma global alors que la présence de deux hiérarchies, une pour les classes et une pour les classes-vues, est plus simple à affronter [Bertino92] ;
- le fait que pour pouvoir correctement assurer cette insertion, il faille d'une façon générale que

23. classes servant d'opérandes dans la requête de définition de la vue

- l'extension d'une classe inclue l'extension de toutes ses sous-classes [Kim95]²⁴ ;
- la difficulté de faire évoluer un schéma incluant des classes-vues [Kim95].

Pour notre étude sur ces techniques de vues, nous étudions les travaux de « NO² »²⁵, Bertino et Kim qui nous semblent de bons représentants de l'ensemble des travaux sur ces techniques.

Travaux de NO²[Geppert93]. Bertino[Bertino92] et Kim [Kim95]

Ces trois approches utilisent des requêtes pour déterminer le type et l'extension des classes-vues à l'aide d'un langage similaire à OQL [ODMG93, ODMG97]. La table 2.3 montre les différentes caractéristiques des classes-vues dans ces trois approches. Dans cette table est fait mention de l'existence d'une relation d'héritage entre classes-vues. Sa sémantique est la même que la relation d'héritage entre classes, mis à part le fait qu'elle s'exprime entre des classes-vues. Cette relation sous-tend alors une hiérarchie similaire à celle existante entre classes mais totalement indépendante de cette dernière. Elle est déterminée explicitement dans [Bertino92].

Table 2.3: Caractéristiques principales des trois approches

	Hiérarchie d'héritage entre classes- vues	Génération d'identités propres	Préservation des instances des classes	Mises à jour
NO ²	non	non	obligatoire	oui
Bertino	oui	oui sauf si les instances ne doivent pas être référencées et si elles n'ont pas d'attributs stockés	non	non
Kim	oui	oui si l'opérateur de définition de la classe-vue ne préserve pas l'identité (cas de la jointure)	oui si l'opérateur de définition de la classe-vue préserve l'identité	oui si celle-ci n'est pas ambiguë (comme dans le modèle relationnel)

Exemple

Nous reprenons l'exemple développé en 2.2.3.1.3 avec le même schéma de base.

24. Il convient de remarquer que cette remarque est propre à Kim qui défend l'idée que l'extension d'une classe ne doit pas inclure l'extension de ses sous-classes. Ceci est contraire à la sémantique des classes dans les langages à objets (ne fût-ce que pour respecter le polymorphisme d'inclusion) mais est principalement retenu par Kim pour des raisons d'optimisation.

25. Il faut noter que les classes-vues sont appelées « derived types » dans NO²

Les classes-vues « SalariesLillois », « Personne » et « Personne2 » peuvent être définies ainsi à l’aide du langage défini dans [Bertino92] :

```
create-view SalariesLillois
view-query s; s/Salaries; s.adresse = 'Lille'

create-view Personnes
view-query p.nom,p.annee_de_naissance,p.adresse; p/Actifs

create-view Personnes2
superviews Personnes
Methods (age() → numeric)

age() → numeric
Begin
return (annee_courante() - annee_de_naissance)
End
```

La hiérarchie des classes et des classes-vues est montrée à la figure 2.12.

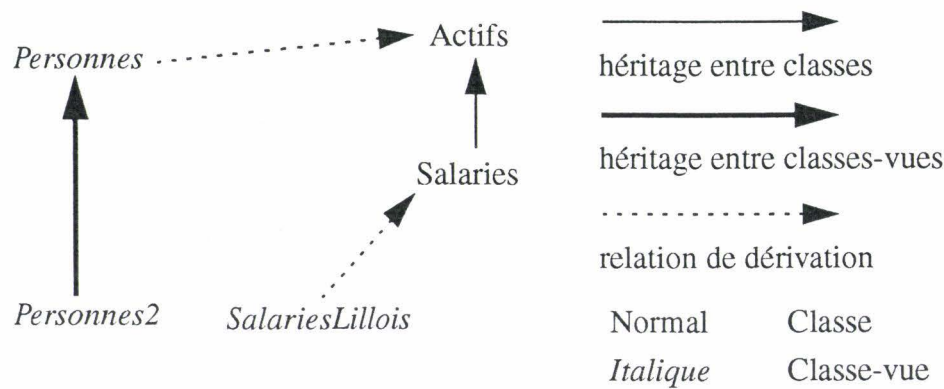


Figure 2.12. Hiérarchies des classes et des classes-vues

Remarques

• Il convient de remarquer que ces trois approches offrent la possibilité d’avoir pour domaine d’un attribut d’une classe-vue, soit une classe-vue soit une classe. La même relation de composition s’applique alors entre classes-vues et classes. [Kim95] interdit d’avoir une classe-vue pour domaine d’un attribut d’une classe. Bien que rien ne soit spécifié pour les deux autres approches, la même règle semble s’appliquer également. Par ailleurs, [Kim95] interdit d’utiliser des classes-vues non peuplées d’instances de classes²⁶ comme domaine d’attribut. Ceci ne pose pas de problème dans [Bertino92] où les instances des classes-vues se voient doter d’une identité propre dans le cas où il est nécessaire de les référencer. Dans [Geppert93], la question ne se pose pas étant donné que les classes-vues ne peuvent être peuplées que

26. C’est, par exemple, le cas d’une classe-vue issue d’un opérateur de jointure

d'instances existantes de classes ;

- Dans [Geppert93] (NO²), le besoin d'adapter la relation de composition issue de deux classes à leurs classes-vues respectives est montré ainsi que la solution proposée. La figure 2.13 illustre un exemple d'une telle adaptation. L'attribut « employees » est remplacé dans la classe-vue « ComputerDepartment » par un attribut calculé qui fait une sélection avec l'extension de la classe-vue « ComputerEmployee » ;

```
DEFINE OBJECT TYPE Employee =  
  TUPLE (name: string, address: string, skill: string, salary: integer).  
DEFINE OBJECT TYPE Department =  
  TUPLE (name: string, address: string, employees:SET(Employee)).  
DEFINE DERIVED TYPE ComputerEmployee  
  FROM emp IN Employee  
  AS SELECT * WHERE skill = 'Computer'  
DEFINE DERIVED TYPE ComputerDepartment  
  FROM dept IN Department  
  AS SELECT [name: dept.name, address: dept.address]  
  ADD employees:(SELECT emp FROM emp IN ComputerEmployee WHERE emp IN dept.employees)
```

Figure 2.13. Exemple d'adaptation de la relation de composition entre classes-vues

- Les trois approches permettent d'ajouter dans une classe-vue des attributs calculés et/ou des méthodes. Dans [Bertino92], il est également possible d'ajouter de nouveaux attributs stockés²⁷. Dans ce cas, les instances de la classe-vue sont nécessairement dotées d'une identité propre ;

- NO² offre aussi la possibilité de définir des schémas-vues constitués de classes et de classes-vues. Il doit être fermé relativement à la relation de composition comme dans MultiView (cf 2.2.3.3.2) ;

- L'approche de Kim est implantée dans la base de données UniSQL/X dont le modèle de données est un modèle mixte relationnel et « objet » y compris pour les classes-vues.

2.2.5 Autres travaux

Il existe de nombreux autres travaux sur les vues dans le cadre des bases de données à objets. Nous pouvons citer :

FUGUE [Heiler90] : Un schéma-vue est constitué d'un ensemble de types dérivés associés à un ensemble d'instances. Un type dérivé est formé de fonctions²⁸ nouvelles qui sont basées ou non sur les fonctions de son type de base²⁹. Les types d'un schéma-vue ne sont pas insérés dans la hiérarchie des types du schéma de base. En plus de la fermeture d'un schéma-vue par rapport aux domaines des fonctions, FUGUE introduit la fermeture de valeur (value-closure). Celle-ci signifie qu'il n'est pas possible de faire disparaître une instance d'un

27. cf les classes-vues à augmentation de capacité de MultiView [Ra95]

28. FUGUE possède un modèle fonctionnel comme COCOON.

29. qui est inclus dans le schéma de base

schéma-vue par une mise à jour de ses attributs par un client du schéma-vue.

VODAK [Kaul90, Klas90] : Une classe-vue peut être définie à partir de plusieurs opérateurs : spécialisation, agrégation, regroupement, généralisation de rôle ou de catégorie. Une classe-vue peut hériter de méthodes de sa classe de base ou en définir de nouvelles. Les classes-vues sont principalement utilisées pour intégrer des bases de données sémantiquement hétérogènes.

[Tanaka88] : Les classes-vues sont déterminées par une sélection qui se réalise au travers d'un prédicat. Leur classification doit être réalisée par l'utilisateur. Un schéma-vue doit être fermé par rapport aux valeurs des attributs des différentes classes et classes-vues du schéma-vue.

[Dobrovnik94] : Il s'agit ici d'ajouter le support des classes-vues au modèle de l'ODMG-93 [ODMG93]. Les classes-vues sont introduites au moyen de types et de classes dérivés. Un type dérivé est obtenu à partir d'un seul type de base par une projection et une augmentation de méthodes. Il ne peut avoir au plus qu'un seul surtype qui doit être aussi un type dérivé. Deux types de base indépendants ne peuvent avoir par définition des types dérivés dont l'un serait surtype de l'autre. Les schémas-vues sont constitués de types dérivés et de classes dérivées.

[Mamou91] : Une classe-vue est constituée par un couple $\langle V, P \rangle$. « V » est la définition de la classe-vue, requête sur un schéma. « P » est la présentation constituée par un type.

[Busse94] : Comme [Mamou91], les classes-vues disposent d'un type introduit par une interface spécifiée ici en « ODL » (cf [ODMG93, ODMG97]) et d'une définition introduite pour chaque attribut et relation, sous forme d'une requête. L'identité des instances d'une classe-vue est déterminée lors de leur introduction dans l'extension de la classe-vue.

Dans le cadre des bases de données à objets déductives, nous pouvons citer [Urban89] et [Ye95]. [Urban89] propose de constituer des classes-vues à partir de prédicats formulés en logique du premier ordre et de les insérer dans la hiérarchie des classes à l'aide de contraintes également exprimées en logique du premier ordre. [Ye95] utilise une base de données déductive ayant pour modèle ERC+ [Parent89]. Les règles de dérivation sont utilisées pour définir les classes-vues. Il propose un algorithme pour déterminer automatiquement les attributs des classes-vues qui forment une clef ainsi qu'un mécanisme de classification des classes-vues.

Nous pouvons encore citer, sans les détailler, [Dayal89] et [Shilling89] ainsi que [Albano93], [Carré89], [Pernici90a, Pernici90b], [Richardson91] et [Wierenga91] qui sont des travaux sur les rôles.

2.2.6 Vues et descriptions multiples d'entités

2.2.6.1 Techniques par extraction lors de l'exploitation

Au travers de cette synthèse, nous pouvons déterminer que les vues sont des techniques qui procèdent principalement par extraction lors de l'exploitation (cf 1.4.2), pour les types (intensions) comme pour les extensions.

Les vues interviennent lors de l'exploitation. Les classes-vues sont basées sur le langage de requêtes. De ce fait, nous pouvons les considérer comme des requêtes rémanentes³⁰. Lors de l'exploitation, un utilisateur décidera des requêtes qui déterminent les informations qui lui

sont pertinentes. Les classes-vues sont des réifications de ces requêtes d'exploitation, les rendant ainsi persistantes. Les schémas-vues sont des ensembles de classes et de classes-vues spécifiés explicitement lors de l'exploitation. Par ailleurs, l'introduction et le retrait de classes-vues et de schémas-vues n'affecte en rien le schéma issu de la conception [Rundensteiner92b]. Ceci montre également que les vues relèvent d'une phase d'exploitation ultérieure à la conception et dont cette dernière est indépendante.

Les vues procèdent par extraction. La projection (ou le masquage) retient une partie du type. Une classe-vue par projection procure ainsi, à ses utilisateurs, une partie du type de la classe, sorte de point de vue sur la description des instances de cette classe. Ce point de vue est déterminé librement³¹ à partir de l'ensemble des propriétés de la classe, indépendamment de critères issus de la phase de conception comme la hiérarchie d'héritage des classes. La sélection extrait une partie de l'extension de la classe. Elle est réalisée au travers d'un prédicat lui aussi entièrement déterminé lors de l'exploitation. Un schéma-vue procède également par extraction puisqu'il ne contient qu'un sous-ensemble des classes et des classes-vues de la base.

Les classes-vues par augmentation de capacité sont basées sur un opérateur du langage de requêtes étendu pour prendre en compte l'introduction de nouvelles propriétés, d'une façon similaire à celle du langage de définition du schéma. Elles constituent donc une exception à ce que nous venons dire sur le fait que les vues procèdent par extraction, même si les classes-vues par augmentation sont utilisées postérieurement à la conception principalement pour offrir des mécanismes d'évolution de schéma [Ra95]. Elles peuvent néanmoins constituer un support de descriptions multiples d'entités sans procéder par extraction, en intégrant leur définition à la phase de conception (cf la section 4.2.2 du chapitre 4).

2.2.6.2 Granularité fine des classes-vues

Par rapport aux besoins de descriptions multiples d'un ensemble d'entités du domaine relativement à une fonction exprimés en 1.6.1, les classes-vues ne constituent pas une approche systématisante. Chaque classe-vue est introduite et définie sur sa classe de base indépendamment des autres classes et classes-vues.

Lorsqu'ils existent, les schémas-vues sont une sélection de classes et de classes-vues sans guide particulier. Une même classe ou classe-vue peut participer à plusieurs schémas-vues. Plusieurs classes-vues issues d'une même classe peuvent participer au même schéma-vue. Ceci montre bien que les schémas-vues ne correspondent pas à une systématisation des descriptions d'entités relativement à une fonction. Il s'agit, en fait, d'une notion aussi souple que les classes-vues et utilisée lors de l'exploitation.

Cette absence de systématisation d'un ensemble de classes-vues comme ensemble de descriptions d'entités relativement à une fonction oblige à réaliser « manuellement » certaines adaptations. Un exemple est celui de l'adaptation de la relation composition entre deux classes dans leurs classes-vues respectives (cf figure 2.13). Cette adaptation peut être réalisée de façon automatique dans un cadre de descriptions d'entités systématisées pour une fonction (cf 3.6.2 et 3.6.3).

30. ce qui est souvent cité comme l'un des objectifs des classes-vues [Bertino92]

31. L'utilisateur ou l'administrateur choisit explicitement les propriétés qu'il décide d'inclure dans le type de la classe-vue.

2.3. Intégration de vues

2.3.1 Besoins et objectifs

L'intégration de vues relève de la phase de conception d'une base de données. La conception du schéma conceptuel d'une petite base de données peut être réalisée directement et globalement. Mais, dans le cas de base de données de taille importante, cette tâche devient impossible. Concevoir directement un seul schéma répondant à l'ensemble des besoins de tous les utilisateurs de la base n'est plus envisageable.

L'intégration de vues a pour objectif de répondre à ce problème en construisant le schéma conceptuel à partir d'un ensemble de schémas-vues d'utilisateurs. Chaque schéma-vue d'utilisateurs est un schéma détenant l'ensemble des descriptions répondant à leurs besoins. Chaque ensemble d'utilisateurs, clients de la base, construit son propre schéma-vue, ce qui reste une tâche envisageable.

Tous les schémas-vues sont ensuite intégrés pour former le schéma conceptuel. La phase d'intégration supprime les incohérences, les erreurs et les redondances issues de ces schémas-vues.

L'intégration de schémas-vues peut être menée entièrement à la main mais lorsque le nombre de schémas-vues est important, cette tâche devient quasiment impossible. Les méthodes d'intégration de vues offrent un support d'automatisation ou plutôt de semi-automatisation de cette tâche. La tâche de la personne menant l'intégration se réduit alors à la résolution de conflits qui ne peuvent être traités automatiquement. Ceci consiste généralement à fournir des assertions sémantiques permettant de résoudre les conflits.

Les travaux sur l'intégration de vues sont très nombreux, étant donné qu'il s'agit d'une problématique assez ancienne (1978)³². La plupart des travaux ont été réalisés dans le modèle de données Entité-Association ([Batini92]) ou dans le modèle relationnel ([Biskup86]). Les travaux de Larson, Navathe et Elmasri ([Navathe86, Larson89]³³) sont réalisés dans le modèle « ERC » et ceux de Spaccapietra ([Spaccapietra92, Spaccapietra93]) dans le modèle « ERC+ » [Parent89] plus proche du modèle « objet ».

Les démarches d'intégration de vues sont également employées pour mener l'intégration des schémas de bases de données peuplées dans le but de les fédérer. Nous reviendrons sur les différences entre intégration de vues et fédération dans le chapitre 5 (section 5.3.1).

2.3.2 Méthode d'intégration

2.3.2.1 Aperçu de la méthode

La méthode d'intégration intervient après l'élaboration des différents schémas-vues d'utilisateurs. Elle consiste à intégrer plusieurs schémas en un seul schéma conceptuel. Le haut de la figure 2.14 illustre ce processus. Il est possible de vouloir intégrer, en même temps, plusieurs schémas-vues. Comme il n'est pas rare d'avoir des dizaines, voire des centaines de schémas-vues à intégrer, la plupart des approches proposent d'intégrer deux schémas-vues et

32. Nous ne nous sommes donc basés que sur quelques travaux.

33. [Larson89] propose une méthode de détection des entités similaires dans chaque composant basée sur la sémantique des attributs et notamment des clefs.

d'itérer l'intégration entre le résultat obtenu précédemment et un autre schéma-vue. Cette démarche par itération est illustrée au bas de la figure 2.14.

Par la suite, nous étudierons différents problèmes de l'intégration pour deux schémas-vues.

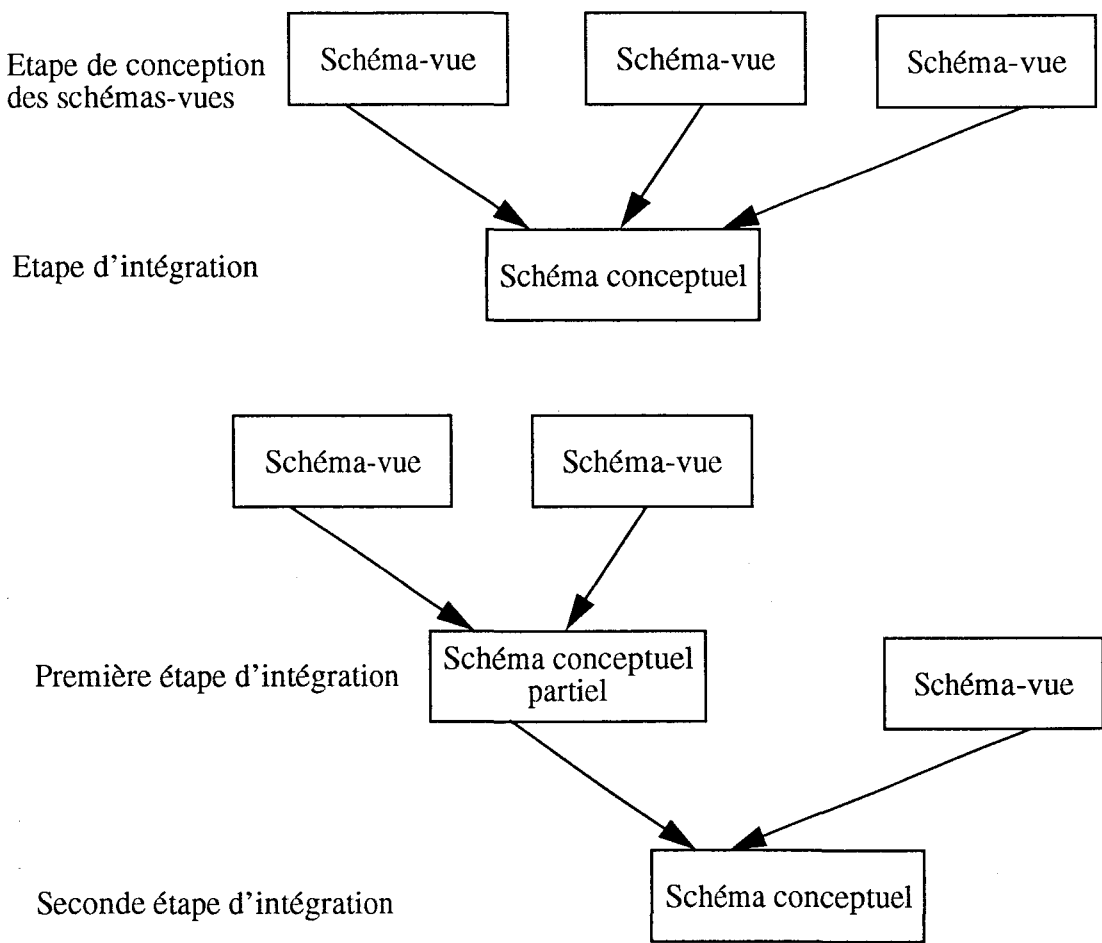


Figure 2.14. Conception par intégration de schémas-vues d'utilisateurs

2.3.2.2 Conflits lors de l'intégration

Lors de l'intégration de schémas-vues, plusieurs types de conflits peuvent apparaître. Ces conflits sont dus à l'hétérogénéité des schémas-vues, elle-même ayant pour cause la conception autonome de chaque schéma. Les trois principaux conflits rencontrés lors de l'intégration sont les conflits de nom, les conflits sémantiques et les conflits structurels [Spaccapietra93].

Conflits de nom

Les conflits de nom proviennent soit d'un problème de synonymie, soit d'un problème d'homonymie. La synonymie se produit quand une même construction est désignée différemment dans les deux schémas-vues. L'homonymie se produit quand deux constructions distinctes sont désignées identiquement dans les deux schémas-vues. La découverte de ces conflits passe par une bonne connaissance de la sémantique contenue dans chaque schéma. Leur résolution se fait par la fourniture d'assertions précisant les synonymes et les homonymes.

Conflits sémantiques

Les conflits sémantiques apparaissent dans les approches basées sur le modèle « objet » ou le modèle « ERC+ ». Dans ces approches, une classe de l'un des schémas-vues peut être sémantiquement proche d'une classe de l'autre schéma-vue sans être totalement équivalente. Par exemple, une classe d'étudiants en informatique n'est pas équivalente à une classe d'étudiants. Ces deux classes ne peuvent pas donner lieu à une classe unique dans le schéma intégré. La classe d'étudiants en informatique est une sous-classe de la classe d'étudiants. Ici, encore, la découverte de ces conflits passe par une bonne connaissance de la sémantique et leur résolution par la fourniture d'assertions précisant l'ordre des classes.

Conflits structurels

Un conflit structurel se produit lorsqu'une partie du monde réel n'est pas représentée par le même ensemble de structures dans chaque schéma-vue. Par exemple, dans le cadre du modèle Entités-Associations, un objet du monde réel est représenté par une entité dans un schéma et par un attribut d'une entité dans un autre schéma. D'autres incompatibilités peuvent se produire :

- dans le cadre du modèle « objet », une classe « A » est une sous-classe d'une classe « B » dans un schéma alors que la classe « B » est une sous-classe de la classe « A » dans l'autre schéma ;
- les cardinalités d'une même association sont distinctes dans les deux schémas.

[García-Solaco96] contient une étude beaucoup plus complète de ces types de conflits. Il faut noter que tous les types de conflits précités y sont classifiés comme problématique d'hétérogénéité sémantique.

2.3.2.3 Obtention d'un référentiel commun et fusion des descriptions

La résolution des conflits étudiés précédemment aboutit à la constitution d'un référentiel commun entre les deux schémas-vues. Ce référentiel contient les entités communes aux deux schémas-vues. Dans le cadre du modèle « objet », les classes correspondant à ces entités sont organisées selon la relation de spécialisation et de généralisation au sein de ce référentiel. Nous n'étudions pas en détail le processus de résolution des conflits ni celui d'élaboration du référentiel commun qui est présenté dans [Spaccapietra92], [Spaccapietra93] et [Navathe86]. La semi-automatisation de l'intégration est utilisée principalement ici pour déterminer le référentiel commun à l'aide des assertions nécessaires. La figure 2.15 montre un exemple où l'intégration se fait automatiquement grâce à une assertion. Cet exemple montre aussi que, dans le cas où une entité est représentée par une classe dans un schéma et par un ou plusieurs attributs dans un autre, la méthode opte pour la représentation par une classe dans le référentiel commun. Dans le cas de l'exemple de la figure 2.15, il s'agit de l'entité « Auteur » qui n'est perçue dans le schéma-vue logistique qu'au travers d'attributs de l'entité « Livre ».

Le référentiel commun déterminé, la fusion des descriptions des deux schémas-vues est alors possible. Les descriptions correspondant à une même entité sont fusionnées en prenant l'union des attributs [Batini92]. Chaque entité est ainsi dotée des descriptions nécessaires pour les utilisateurs des deux schémas-vues. Cette fusion peut également être réalisée automatiquement. Ceci est également illustré à la figure 2.15 pour les deux entités « Auteur » et « Livre » en utilisant le formalisme d'OMT [Rumbaugh95].

Cette nécessité de devoir construire un référentiel commun à partir des schémas-vues provient principalement des conflits de structure. Chaque schéma ayant sa propre structuration,

il doit être confronté aux autres pour pouvoir déterminer la structuration commune.

Lors d’une phase d’intégration, le nombre de conflits de structure à résoudre est généralement d’autant plus important que le nombre de constructions présentes dans le modèle de données est important. Le modèle relationnel ou le modèle « objet » qui ne détiennent qu’une seule construction³⁴ présentent ainsi un avantage pour la conception de grosses bases de données par intégration de vues (cf [Saltor91]).

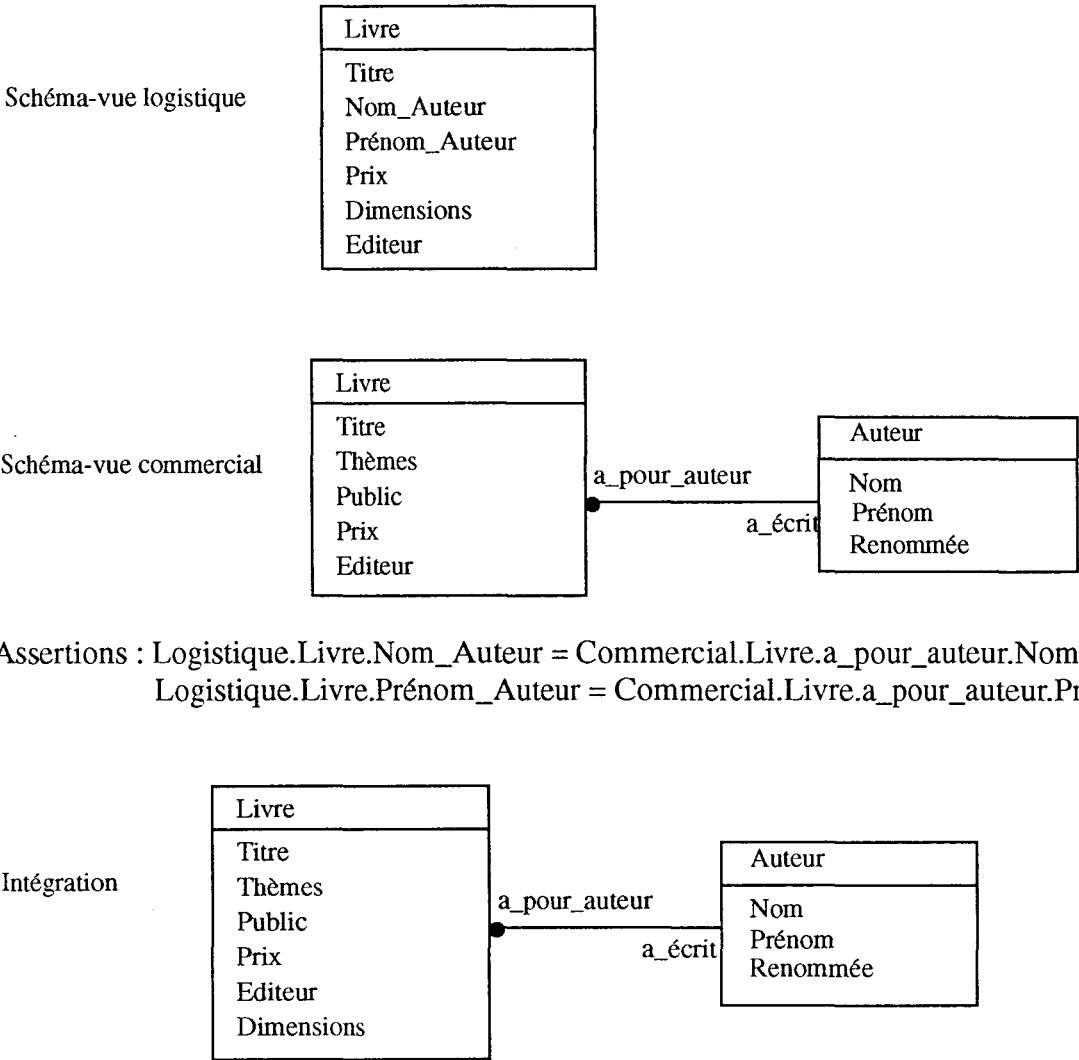


Figure 2.15. Intégration dans le cas d’un conflit de structure

2.3.3 Absence de structuration du schéma conceptuel issu de la conception vis-à-vis des différents utilisateurs

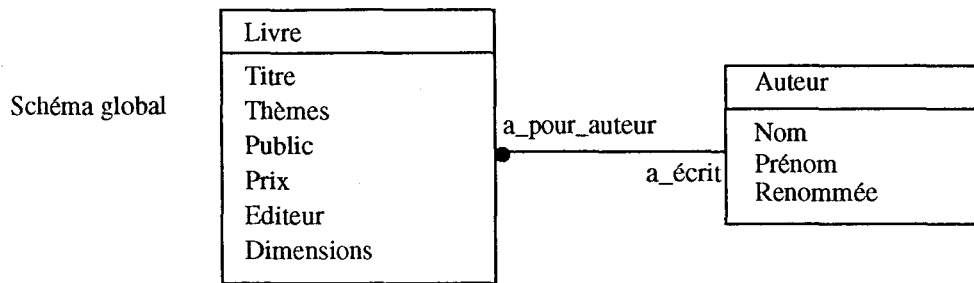
Les méthodes de conception par intégration de vues se basent sur les descriptions pertinentes des différents utilisateurs pour élaborer le schéma conceptuel. L’utilisation de ces

34. respectivement la relation pour le modèle relationnel et la classe pour le modèle « objet »

descriptions d'utilisateurs a pour but de rendre possible la conception du schéma conceptuel de grosses bases de données où la conception directe serait impossible à cause de la taille de ce schéma. La figure 2.15 illustre bien le fait que l'intégration de vues soit basée sur les descriptions pertinentes des différents utilisateurs qui sont ici la logistique et la fonction commerciale.

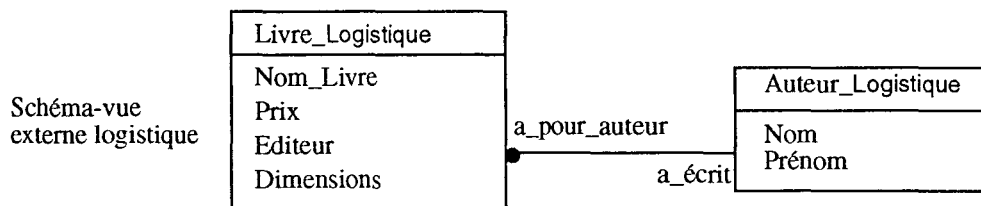
L'objectif de l'intégration de vues n'est donc pas de prendre en compte les descriptions des utilisateurs pour construire un schéma conceptuel structuré relativement à ces différents utilisateurs. Le schéma conceptuel issu d'une intégration par vues ne diffère pas dans sa forme de celui issu d'une conception directe. Les spécificités relatives aux vues d'utilisateurs sont perdues à l'issue de la conception par intégration. L'intégration de vues ne répond donc pas aux besoins de structuration exprimés en 1.4 dans le cas d'une organisation complexe.

Lors de l'exploitation, la problématique de réalisation des schémas-vues externes reste entière. La figure 2.16 montre qu'il faut alors élaborer ces schémas-vues externes d'exploitation pour les utilisateurs de la base de données (formalisme d'OMT). Cette élaboration est identique à ce qu'elle serait dans le cas d'une conception directe du schéma conceptuel alors que les utilisateurs sont ceux qui fournissent les différents schémas-vues lors de la conception. En effet, les différentes descriptions qui composent le schéma conceptuel ne détiennent pas leur spécificité vis-à-vis des utilisateurs, spécificité qui était présente dans les différents schémas-vues de conception.



```

DEFINE-VIEW Logistique;
define view Livre_Logistique as hide[Thèmes, Public](Livre);
define view Auteur_Logistique as hide[Renommée](Auteur);
ADD-CLASS(Livre_Logistique, Auteur_Logistique);
END-VIEW;
  
```



```

DEFINE-VIEW Commercial;
define view Livre_Commercial as hide[Dimensions](Livre);
ADD-CLASS(Livre_Commercial, Auteur);
END-VIEW;
  
```

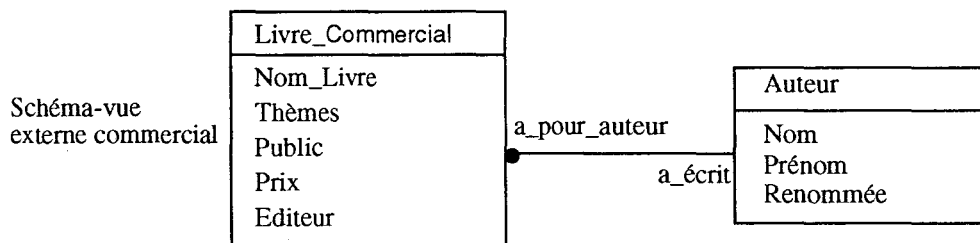


Figure 2.16. Schémas-vues construits lors de l'exploitation

2.4. Bilan

Dans ce chapitre, nous avons étudié différentes techniques de vues dans les bases de données à objets tant à l'exploitation (2.2) qu'à la conception (2.3).

Nous avons remarqué en 2.2.6 que :

- les vues « utilisateurs » relèvent de la phase d'exploitation ;
- les vues « utilisateurs » sont des extractions spécifiées lors de l'exploitation^{35 36} ;

35. tant pour les classes-vues que pour les schémas-vues

36. A l'exception toutefois des classes-vues par augmentation de capacité mais dont l'objectif n'est pas le support des schémas externes de l'architecture ANSI/X3/SPARC

- chaque classe-vue est introduite avec une granularité fine, indépendamment des autres. Chaque schéma-vue d'exploitation est constitué par une sélection libre de classes et de classes-vues. Ainsi, il apparaît clairement que les schémas-vues d'exploitation ne correspondent pas forcément à une systématisation des descriptions d'entités relativement à une fonction.

Ensuite, nous avons remarqué en 2.3.3 que le schéma issu d'une conception par vues ne diffère pas dans sa forme d'un schéma issu d'une conception classique. Les différentes descriptions qui le composent ne détiennent pas leur spécificité vis-à-vis des utilisateurs, spécificité qui était présente dans les différents schémas-vues de conception. Lors de l'exploitation, la problématique de l'élaboration des schémas-vues externes reste entière alors que les utilisateurs de ces schémas-vues externes sont ceux qui fournissent les différents schémas-vues de conception.

Les techniques de vues n'apportent donc pas de réponse au besoin de structuration du schéma conceptuel de la base de données relativement aux fonctions dès sa conception sans qu'il y ait lieu d'intégrer puis d'extraire des schémas externes. Pour y répondre, nous proposons dans le chapitre 3 une approche systématisée de conception des multiples descriptions des entités d'une base de données.

Chapitre 3

L'approche CROME dans le cadre des bases de données

3.1. Introduction

Dans ce chapitre, nous présentons l'approche CROME [Carré96, Debrauwer97, Vanwormhoudt97a-b, Vanwormhoudt99] (contextes en ROME [Carré89, Carré90a-b, Carré91]) dans le cadre des bases de données à objets¹. Cette approche constitue une solution à la problématique abordée dans le premier chapitre.

L'approche CROME se situe à un niveau logique de spécification et de conception. Nous l'introduisons à l'aide du modèle « objet » de l'ODMG qui nous sert de support pour établir des spécifications à un niveau logique². De façon plus précise, nous utilisons le langage de définition de schémas de bases de données « ODL ». Une spécification décrite en ODL est indépendante d'un SGBD précis. Lors de l'implantation, cette spécification peut être traduite dans le langage de définition des données du SGBD cible³.

La section 3.2 explicite les aspects importants du modèle de l'ODMG.

Nous abordons CROME dans le cadre des bases de données où nous mettons la priorité sur l'information. Par conséquent, nous nous situons dans le cadre d'un modèle d'objets serveurs d'information dont l'encapsulation [Snyder93] est peu justifiée⁴, leur structure étant directement accessible par les clients de la base de données et leur cohérence garantie par des contraintes d'intégrité. Par ailleurs, nous nous intéressons ici exclusivement à l'aspect « données » de ces objets sans prendre en compte les aspects relatifs aux traitements⁵. Nous retenons les caractéristiques du modèle « objet » et plus précisément l'identité [Kent91, Khoshafian86] et l'héritage.

Notre principale problématique est la prise en compte des fonctions de l'organisation au sein du schéma de la base de données, conduisant à une structuration de ce schéma relativement à ces fonctions. Nous avons vu au premier chapitre que l'approche CROME prône la structuration du schéma dès sa conception sans qu'il y ait lieu d'intégrer puis d'extraire des schémas externes. Dans ce même chapitre, nous avons également abordé l'ensemble des problèmes relatifs à ce besoin de structuration. Nous verrons les réponses apportées par CROME dans les sections 3.3 à 3.6. dont les principaux points sont les suivants :

- la structuration de la description des entités en *parties fonctionnelles*, structuration systématisée par les *plans fonctionnels*. Cette structuration est explicitée dans la section 3.3 ;
- l'*articulation entre le plan de base et les plans fonctionnels*, articulation réalisée au sein de chaque classe du domaine. Cette articulation est également décrite dans la section 3.3 ;

1. Rappelons que CROME est également abordé dans le cadre de la programmation (cf [Carré96, Vanwormhoudt97a-b, Vanwormhoudt99]).

2. L'approche est suffisamment générale pour ne pas imposer le langage ROME comme support, contrairement à ce que pourrait supposer son nom.

3. qui est souvent, dans le cas des SGBD à objets, un langage de programmation.

4. voir aussi à ce sujet les langages à objets de représentation des connaissances [Dekker92, Dekker94]

5. Il est également possible d'introduire des accesseurs.

- le partage des informations entre *fonctions fournisseuses et consommatrices* et les modalités de fourniture des informations communes décrites par le plan de base. Ceci fait l'objet de la section 3.4 ;

- les *contraintes de participation des instances* aux fonctions. Celles-ci sont introduites dans la section 3.5 où nous déterminons également les notions d'*état fonctionnel* et d'*extension fonctionnelle* ;

- la notion de *contexte fonctionnel* détenant l'ensemble des descriptions et des informations de la fonction pertinentes pour ses clients. Cette notion est également introduite dans la section 3.5 ;

- la notion de *dépendance entre contextes* pour une classe du domaine, définie dans la section 3.5. Cette notion conduit à une nouvelle contrainte de participation, ce qui nous amènera à enrichir une première fois la notion d'état fonctionnel ;

- les *contraintes d'intégrité référentielle et de cardinalité* des relations inter-entités partagées. La section 3.6 explicitera les conséquences de celles-ci dans les contextes consommateurs à la fois pour les occurrences des relations et les instances des classes du domaine. Nous obtiendrons un second enrichissement de la notion d'état fonctionnel ;

- la *généricité des relations partagées* et notamment des relations du plan de base. La section 3.7 explicite le caractère générique des relations d'héritage et des relations inter-entités partagées. Notons que le respect des contraintes d'intégrité référentielle et de cardinalité de ces dernières dans les contextes consommateurs est à la base de leur généricité ;

Enfin, la section 3.7 explicite les propriétés d'un schéma d'une base de données conçu en CROME. En plus de l'aspect générique des relations partagées évoqué ci-dessus, cette section montre le caractère modulaire des classes obtenues lors de la conception. Elle explicite l'interaction entre contextes et montre qu'un contexte fonctionnel est un schéma de classes qui peut être exploité comme tel par les clients de la fonction correspondante. Dans un objectif de comparaison, ces propriétés sont également étudiées dans le cadre de l'approche par les vues présentée au chapitre 2.

L'ensemble des sections s'appuie sur l'exemple de la base de données d'une entreprise de fabrication et de distribution introduit au premier chapitre et que nous avons étendu. Son schéma est décrit à l'annexe A.

3.2. Choix du modèle « objet » de l'ODMG dans le cadre des bases de données

3.2.1 Modèle de l'ODMG

Nous présentons le modèle de l'ODMG [ODMG93, Kim93, ODMG97] pour les bases de données à objets. Celui-ci repose principalement sur :

- un modèle « objet »,
- un langage de spécification de schémas de bases de données (ODL),
- un langage d'interrogation de bases de données (OQL).

Un aspect important du modèle de l'ODMG est qu'il n'impose pas de langage standard d'implantation. Celle-ci peut donc être réalisée dans un langage à objets tel que C++, Java ou Smalltalk. La spécification du schéma de la base en ODL est alors traduite dans un tel langage cible lors de l'implantation. C'est lors de l'implantation que le corps des méthodes des objets est réalisé. Pour faciliter ces phases de traduction, l'ODMG propose des couplages (« bindings ») avec les langages C++, Java et Smalltalk. Un couplage décrit la façon de traduire dans le langage cible les spécifications écrites en ODL. Il décrit aussi la façon de manipuler les objets (créer, détruire et mettre à jour leurs instances) et d'invoquer OQL depuis ce langage d'implantation.

Dans le modèle de l'ODMG, les objets sont décrits par des types. Chaque type possède une implantation et une spécification externe (interface) qui décrit les caractéristiques des objets destinées à ses clients. La spécification en ODL du schéma d'une base de données consiste à déterminer les interfaces des types des objets de la base. La syntaxe du langage de est défini par un surensemble de la syntaxe du langage IDL défini par l'OMG [Orfali96].

Dans ce chapitre, nous baserons notre approche sur le modèle « objet » de l'ODMG au travers du langage ODL. Nous nous limiterons à des objets dont la structure est décrite par une spécification d'interface ODL dépourvue de spécifications de méthodes. Par ailleurs, OQL nous servira pour la définition des contraintes d'intégrité et comme langage d'interrogation. Nous rappelons brièvement les caractéristiques les plus importantes de ce modèle et du langage OQL.

Nous retenons le modèle de l'ODMG 2.0 [ODMG97]. Celui-ci introduit principalement la séparation entre les notions d'interface et de classe ainsi qu'un couplage avec le langage Java. Ces deux notions sont assez proches mais se distinguent par certains points. Comme en Java, l'héritage multiple n'est possible que pour les interfaces mais pas pour les classes. Une classe peut être instanciable et peut donc disposer d'une extension⁶. Une interface ne détient pas d'extension. Nous n'utiliserons dans ce mémoire que les interfaces. Le fait que leur extension ne soit pas nommée pose un problème au niveau du langage de requêtes. Pour y remédier, nous faisons le choix d'utiliser en OQL le nom de l'interface pour désigner son extension⁷.

6. L'extension d'une classe doit être nommée explicitement pour la rendre persistante

7. A ce sujet, les concepteurs du SGBD POET qui font partie de l'ODMG ont fait un autre choix : ils utilisent en OQL le nom de l'interface suffixé de la chaîne « Extent » (cf [POET97]).

3.2.1.1 Caractéristiques importantes du modèle « objet » de l'ODMG

Hierarchie des types

Le modèle s'appuie sur deux primitives de représentation :

- les littéraux qui ne possèdent pas d'identifiant propre. Ils sont décrits par un type de littéraux. C'est, par exemple, le cas des entiers, chaînes de caractères mais aussi des collections ou des structures (« struct ») ;
- les objets qui possèdent une identité propre. Ceux-ci sont décrits par un type d'objets qui peut être soit un nouveau type défini en langage ODL, soit un type prédéfini du modèle. Parmi les types prédéfinis, on trouve principalement les collections.

La hiérarchie (simplifiée) des types est donnée par la figure 3.1. On peut y remarquer que :

- Les types en caractères italiques sont des types abstraits, tandis que les autres sont des types concrets instanciables ;
- « Immutable » est utilisé pour qualifier les types de littéraux « collection » et « structure ». En effet, les littéraux ne détiennent pas d'identité mais une valeur qui est, dans le cas de collections, un ensemble de valeurs et dans le cas de structures, un n-uplet de valeurs. En d'autres termes, les littéraux ne possèdent pas d'état ;
- Cette dernière remarque permet également d'expliquer les différences entre les types de littéraux « collection » (set<>,...) et les types d'objets « collection » (Set<>,...). Dans le premier cas, ce sont des valeurs sans identité, alors que dans le second cas, ce sont des objets possédant une identité invariante de leur état qui peut évoluer au cours du temps ;
- Les types de littéraux « collection » et « structure » ainsi que les types d'objets « collection » sont en fait des constructeurs de types. Par exemple, Set<> peut être paramétré par le type integer pour définir le type d'objets Set<integer>⁸ ;
- « Atomic_object » est un type abstrait d'objets dont les sous-types concrets sont des types d'objets définissables par l'utilisateur en ODL.

8. Il est ainsi possible de créer des ensembles de littéraux ou d'objets. Ces ensembles peuvent être indifféremment de la nature de leurs éléments des littéraux ou des objets.

```

Literal_type
  Atomic_literal
    long
    integer
    float
    boolean
    char
    string
  Immutable_Collection_literal
    set<>
    bag<>
    list<>
    array<>
    dictionary<>
  Immutable_Structured_literal
    struct<>
    date
Object_type
  Atomic_object
  Collection_object
    Set<>
    Bag<>
    List<>
    Array<>
    Dictionary<>

```

type concret (caractères normaux)

type abstrait (caractères italiques)

Figure 3.1. Hiérarchie simplifiée des types du modèle de l'ODMG

Définition d'un type d'objets

La définition de l'interface d'un type d'objets par l'utilisateur se fait grâce à la forme syntaxique ODL suivante [ODMG97] (simplifiée ici) :

```

interface <type_name> : <supertype_name> [, <supertype_name> ] * ;
{
  key[s] <key_list>;
  [<attribute_definition>]*
  [<relationship_definition>]*
}

```

<attribute_definition> est définie par :

```
attribute <domain_type> <attribute_name>;
```

<relationship_definition> est définie par :

```
relationship <target_type> <attr_name> inverse <target_type> :: <inverse_attr_name>;
```

Les définitions d'attributs et de relations peuvent être mélangées dans la définition de l'interface du type. Le type de l'attribut relatif à l'extrémité d'une relation peut être soit le type de l'autre extrémité soit un type de collection paramétré par le type de l'autre extrémité.

Nous avons omis de mettre la syntaxe de spécifications des méthodes étant donné que nous ne les prenons pas en compte.

La figure 3.2 montre deux exemples de définition de type d'objets. Remarquons que ceux-ci sont implicitement des sous-types de « Atomic_object ».

```

interface Part_externe
{
    key nom,telephone;
    attribute string nom;
    attribute string adresse;
    attribute string telephone;
}

interface Client: Part_externe
{
    relationship set<Produit> a_achete inverse Produit::a_ete_achete_par;
    attribute integer montant_cmd;
    attribute date date_derniere_cmd;
    relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
    relationship set<Bon_de_commande> est_concerne_par_BC inverse Bon_de_commande::concerne;
}

```

Figure 3.2. Exemples de définition de type d'objets

Le type « Client » hérite du type « Part_externe ». La description complète de son interface est donc déterminée par l'ensemble des attributs définis dans l'interface de « Part_externe » et de « Client ».

La notion de clef est analogue à celle du modèle relationnel. Les clefs sont constituées par un ou plusieurs attributs dont la ou les valeurs identifient de façon unique chaque instance. Chaque instance doit ainsi posséder une clef dont la valeur est différente de celle des autres instances. Ceci peut être utile pour expliciter que certains attributs correspondent à des « identifiants » du monde réel comme par exemple, le code d'un produit ou encore le couple (nom,téléphone) d'un partenaire de la société.

La clause **relationship** permet de définir l'extrémité d'une relation. Elle consiste en un attribut dont le domaine est le type de l'autre extrémité⁹ et en la spécification de l'attribut inverse défini à l'autre extrémité. Pour exemple, nous donnons l'autre extrémité de la relation « a_acheté/a_été_acheté_par » entre les types « Client » et « Produit » à la figure 3.3.

```

interface Produit
{
    attribute string designation;
    attribute integer prix_tarif;
    relationship set<Client> a_ete_achete_par inverse Client::a_achete;
    ...
}

```

Figure 3.3. Extrémité de la relation « a_acheté/a_été_acheté_par » entre les types « Client » et « Produit » au sein du type « Produit »

Cette relation est donc définie par deux extrémités introduites par chaque type. Une relation ne possède pas de nom et ses occurrences n'ont pas le statut d'objet possédant une identité¹⁰. Notons qu'ici, les relations ne sont pas une construction nouvelle apportée au modèle « objet »¹¹.

9. ou un type de collection paramétré par le type de l'autre extrémité

10. Les occurrences d'une relation ne possèdent pas d'état et ne peuvent être mises à jour. Chaque occurrence est soit ajoutée, soit retirée de l'ensemble des occurrences d'une relation.

Les cardinalités possibles d'une relation à l'une de ses extrémités sont soit 0-1 soit 0-n. Dans le premier cas, l'extrémité est définie par un attribut de type atomique. Dans le second cas, elle est définie par un attribut de type « collection » (de littéraux ou d'objets). Nous ne retenons du modèle de l'ODMG que les spécifications de relations dont les extrémités sont typées par un type de littéraux « collection ».

Par ailleurs, le modèle « objet » prévoit la gestion de l'intégrité référentielle pour les relations. Si une instance participant à une relation est détruite, toutes les occurrences de la relation relatives à cette instance sont automatiquement supprimées. Par exemple, si on supprime une instance de « Produit », alors les occurrences de la relation « a_acheté/a_été_acheté_par » relatives à ce produit sont automatiquement supprimées. Il n'est donc plus possible pour un client de référencer cette instance inexistante.

Ceci nous amène à deux remarques :

- La suppression des instances est réalisée par une opération explicite de l'utilisateur (le « delete »). Cette façon de faire est classique dans les bases de données, à l'inverse de ce qui se fait en programmation (Smalltalk [Golberg83], Eiffel [Meyer88]) où une instance est supprimée lorsqu'elle n'est plus référencée. Dans une base de données, il peut exister des instances pertinentes et non référencées auxquelles on accède par le langage d'interrogation [Bloom87]. Il est donc important de pouvoir détruire explicitement de telles instances et de ne pouvoir le faire qu'explicitement.

- Il existe également la possibilité, dans le modèle de l'ODMG, de définir des attributs qui ne sont pas à des extrémités de relation et dont le type est un type d'objets. Ce genre d'attribut permet à un type d'objets d'en référencer un autre par un lien unidirectionnel sans garantie d'intégrité référentielle. Par la suite, nous n'utiliserons pas cette possibilité de l'ODMG et imposerons que tout lien entre deux types d'objets soit systématiquement réalisé par une relation entre ces deux types.

3.2.1.2 Langage d'interrogation OQL

OQL est le langage d'interrogation supportant le modèle « objet » de l'ODMG. Il est issu de SQL. Il permet un accès aux instances de la base sous une forme déclarative (l'utilisateur ne doit pas préciser comment est calculée une requête). Les requêtes sont exprimées sur le schéma de la base au travers de sa description en ODL. La plupart des primitives du langage s'applique à des collections.

Nous ne présentons pas en détail OQL. Le lecteur intéressé pourra se référer à [ODMG97].

OQL permet d'interroger une base de données à partir des extensions des types d'objets qui la composent. Si l'on reprend le schéma donné en 3.2.1.1 comportant les types « Part_externe », « Client » et « Produit », on peut écrire en OQL les requêtes suivantes :

```
select x from x in Client where montant_cmd > 10000;
```

Cette requête renvoie un ensemble de clients dont le montant des commandes en cours dépasse 10000 F. Le résultat est du type set<Client>.

```
select x.nom from x in Client where montant_cmd > 10000;
```

Cette requête renvoie un ensemble de littéraux (chaînes de caractères) qui sont les noms des clients dont les commandes dépassent 10000 F. Le résultat est du type set<string>.

11. à la différence de [Rumbaugh87, Rumbaugh96] ou du modèle « Entité-Association » [Chen76]

```
select struct(nom:x.nom,adresse:x.adresse) from x in Client where montant_cmd > 10000;
```

Cette requête renvoie un ensemble de littéraux qui sont les noms et adresses des clients dont le montant des commandes en cours dépasse 10000 F. Le résultat de la requête a pour type :

```
set < struct
{
  string nom;
  string adresse;
}>
```

Enfin, Client est une requête qui renvoie l'extension du type « Client » et a donc pour type `set<Client>`.

Il convient de remarquer qu'OQL permet d'exprimer des requêtes dont le résultat n'est pas nécessairement un ensemble d'objets mais peut aussi être un ensemble de littéraux.

La notion de vue est limitée à la définition de requêtes rémanentes auxquelles on attribue un nom. Une telle vue ne se voit pas conférer le statut de classe-vue comme dans les systèmes de bases de données étudiés à la section 2.2 du chapitre 2.

Un autre point important en OQL est la présence dans le langage des quantificateurs universel et existentiel. Ils permettent de définir les expressions de chemin qui traversent des relations dont les extrémités sont typées par des ensembles. Ils nous permettront également d'utiliser OQL pour décrire des contraintes d'intégrité statiques sur les propriétés des entités.

Un exemple d'utilisation du quantificateur universel est le suivant :

```
select x from x in Client where (for all p in x.a_achete: p.prix_tarif > 1000);
```

Ceci consiste à sélectionner tous les clients n'ayant acheté que des produits dont le prix est strictement supérieur à 1000 F. L'exemple suivant basé sur le quantificateur existentiel permet de ne sélectionner que les clients ayant acheté au moins un produit dont le prix est strictement supérieur à 1000 F.

```
select x from x in Client where (exists p in x.a_achete: p.prix_tarif > 1000);
```

3.2.2 Prise en compte des contraintes de cardinalités

Dans la suite, nous avons besoin de pouvoir définir plus finement les cardinalités aux extrémités des relations que le permet ODL. Nous n'avons retenu pour notre modèle que les relations du modèle de l'ODMG dont les attributs des extrémités sont des types de littéraux « ensembles ». Ainsi, par défaut, la cardinalité des extrémités est 0-n, soit une cardinalité non contrainte. Nous proposons d'enrichir ODL pour pouvoir spécifier la cardinalité minimale et maximale lors de la définition d'une relation [Morejon94, Rumbaugh95].

Syntaxiquement, la définition d'une extrémité de relation en ODL se présente maintenant ainsi :

```
relationship <target_type>[[card_min[...card_max]]] <attr_name> inverse <target_type>::<inverse_attr_name>;
```

Par la suite, c'est cette définition syntaxique enrichie qui définit la clause `<relationship_definition>`.

La clause introduisant les cardinalités permet de spécifier la cardinalité minimale et maximale. Cette dernière peut ne pas être spécifiée si l'utilisateur ne désire pas de cardinalité maximale. S'il désire une cardinalité maximale mais pas de cardinalité minimale, il faut spécifier 0 comme cardinalité minimale.

Par exemple, la relation entre un bon de commande et un bon de livraison est une relation 1-1 à chaque extrémité. Dans l'interface de « Bon de livraison », on définit cette cardinalité ainsi :

```
relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
```

La relation entre « Client » et « Produit » a pour cardinalités 1-n à l'extrémité « Client ». Dans l'interface de « Client », on définit cette cardinalité ainsi :

```
relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
```

3.2.3 Bilan sur le modèle choisi

Nous retenons le modèle « objet » de l'ODMG 2.0. Au sein de ce modèle, nous n'utilisons que les interfaces pour définir des types d'objets. Nous ne retenons pas les spécifications de méthodes ni les attributs dont le type est un type d'objets. Tout lien entre deux types d'objets se fait au travers d'une relation entre ces deux types.

Par ailleurs, nous typons systématiquement chaque extrémité de relation par un type de littéraux « ensemble » (set<>). Enfin, nous enrichissons pour nos besoins ODL pour pouvoir définir des cardinalités minimales et maximales.

Par la suite, nous utiliserons plutôt le terme de « classe » à la place de celui de « type d'objets ».

3.3. Présentation des principes généraux de CROME

CROME est un cadre de conception d'entités complexes intervenant dans de multiples contextes. Il permet d'élaborer, de façon modulaire et systématisée, les différentes descriptions de ces entités relativement aux fonctions où elles participent. Nous présentons ici les principes généraux dans le cadre de l'application de CROME aux bases de données, c'est-à-dire dans un cadre d'objets essentiellement serveurs d'information. Nous allons d'abord nous intéresser à proposer une solution à la problématique décrite en 1.6.1, c'est-à-dire le problème des descriptions multiples et systématisées d'entités. CROME constitue alors une approche pour concevoir la description de base et les parties fonctionnelles d'entités intervenant dans plusieurs fonctions.

3.3.1 Rappel sur l'approche multi-intensionnelle

Les contraintes de l'approche « objet » classique ne permettent pas de satisfaire aux besoins de conception et de représentation d'objets détenant plusieurs descriptions, correspondant à des points de vue différents sur leur unique référent (cf les concepts coréférentiels de [Ferber89]).

L'approche multi-intensionnelle permet d'associer plusieurs intensions à une entité pour répondre à ce besoin de descriptions multiples. Ces intensions peuvent être détenues par différentes classes décrivant cette entité ou par une seule classe dont l'intension complète est alors elle-même fragmentée.

A l'instar de certains modèles « objet » nécessaires pour supporter correctement les vues dans les bases de données (cf 2.2 et plus précisément 2.2.3.1.1), CROME relève également de l'approche multi-intensionnelle.

3.3.2 Référentiel d'entités orthogonales aux fonctions

Comme explicité au chapitre 1 (1.5.2), les entités du domaine sont orthogonales aux différentes fonctions de l'organisation et forment un référentiel commun à ces dernières.

Chaque fonction fait participer l'ensemble (ou un sous-ensemble) des entités du domaine. Dualement, chaque entité participe à l'ensemble (ou un sous-ensemble) des fonctions de l'organisation.

Dans le cadre de notre exemple, cette dualité est explicitée à la figure 3.4. Celle-ci montre l'orthogonalité entre les entités du domaine et les différentes fonctions. Les entités introduites dans cette figure correspondent à celles pour lesquelles la définition des classes correspondantes figure à l'annexe A.

L'approche CROME prône l'hypothèse que le schéma de la base que l'on cherche à concevoir est structuré relativement aux fonctions de l'organisation. Par conséquent, cette notion de référentiel commun aux fonctions et constitué des entités du domaine est centrale dans la démarche.

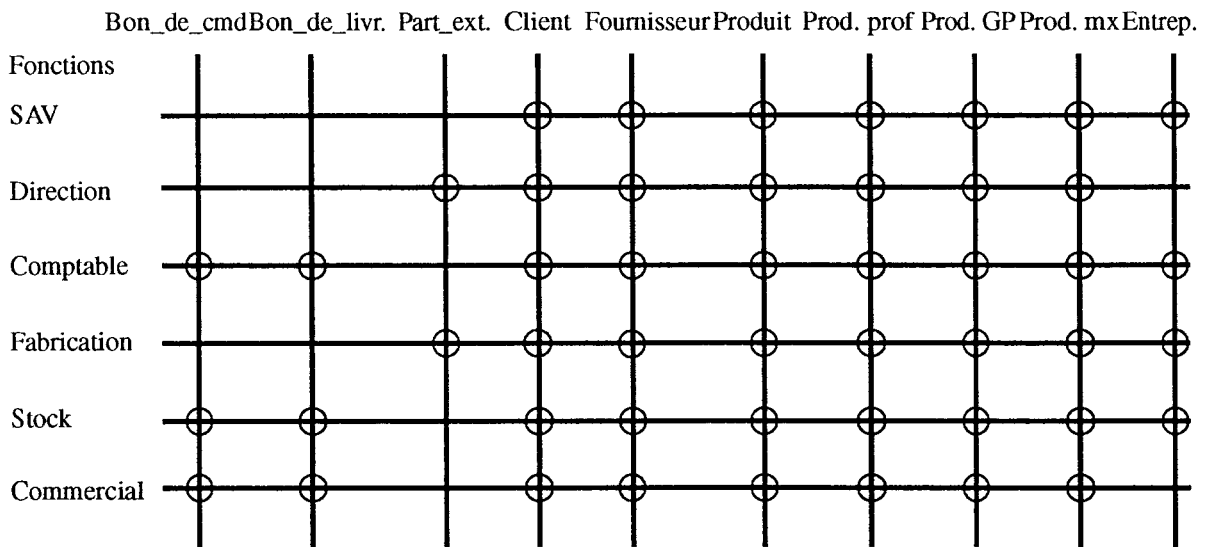


Figure 3.4. Orthogonalité entités du domaine/fonctions

3.3.3 Plans de description des entités

La conception en CROME est fondée sur une description par plans des entités. Le plan de base détient les descriptions de base des entités du référentiel alors que chaque plan fonctionnel détient les parties fonctionnelles correspondantes. Cette notion de plan transversal aux entités a pour but d'expliciter la systématisation des descriptions relativement aux fonctions, sans être une nouvelle construction du modèle « objet ».

3.3.3.1 Plan de base

Nous introduisons la notion de plan de base qui détient la structuration du référentiel : les entités y sont introduites par des classes organisées par la relation d'héritage. Chacune de ces classes détient la description de base commune aux différentes fonctions. Cette structuration et ces descriptions communes issues du plan de base sont partagées par les différentes fonctions au travers des entités du domaine. CROME n'impose pas de contraintes sur le contenu de ce plan. Celui-ci peut fournir une représentation plus ou moins riche du domaine, en fonction du niveau de partage requis. Au minimum, ce plan n'introduit que les classes relatives aux entités du référentiel sans leur conférer de descriptions partagées. Pour être précis, remarquons bien que le plan de base ne détient pas les extensions des classes mais uniquement leurs descriptions de base. En effet, un plan ne détient que des descriptions.

Forme syntaxique employée pour les classes introduites par le plan de base

Les interfaces de ces classes peuvent être définies directement grâce à la forme syntaxique « interface » du langage ODL présentée en 3.2.1.1. Cette interface constitue la description de base des classes du domaine.

Exemple

Le plan de base du SI de l'annexe A est représenté à la figure 3.5 qui en donne une représentation graphique du plan de base à l'aide de la notation graphique employée dans [ODMG97] à laquelle nous avons ajouté la spécification des cardinalités minimales et maximales des relations inter-entités. Dans cette notation, une classe est représentée par un

rectangle, les relations d'héritage par une flèche épaisse noire allant de la sous-classe à la surclasse et les relations inter-entités par une flèche bi-directionnelle.

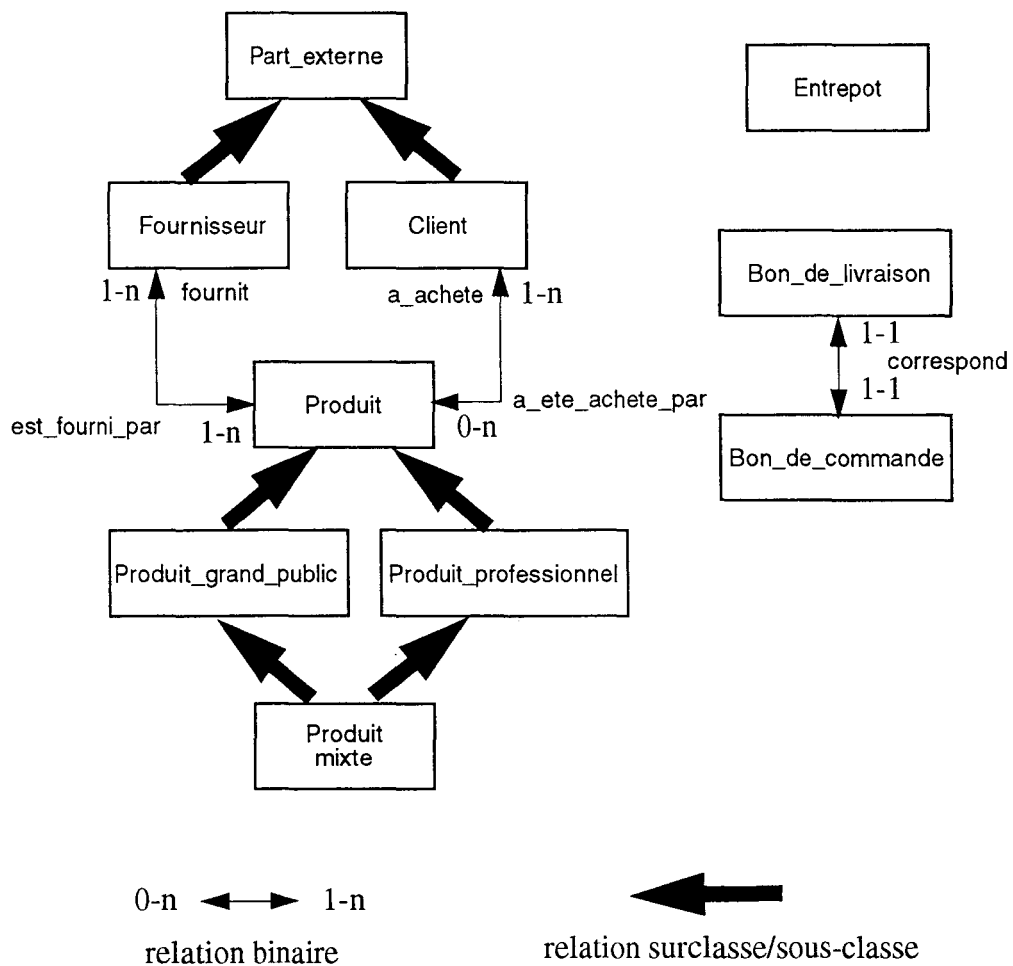


Figure 3.5. Représentation graphique du plan de base

La figure 3.6 donne une représentation du plan de base à l'aide du langage ODL. Les différentes entités du domaine sont introduites par des classes qui déterminent leur description commune à l'ensemble des fonctions. Les relations d'héritage entre les classes sont également déterminées par le plan de base, ainsi que les relations binaires qu'il introduit.

```

interface Part_externe
{
    attribute string nom;
    attribute string adresse;
    attribute string telephone;
}

interface Client: Part_externe
{
    relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
}

interface Fournisseur: Part_externe
{
    relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
}

interface Produit
{
    attribute string designation;
    attribute integer prix_tarif;
    relationship set<Client> a_ete_achete_par inverse Client::a_achete;
    relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
}

interface Produit_grand_public: Produit
{
    attribute integer reference_grand_public;
}

interface Produit_professionnel: Produit
{
    attribute string reference_professionnelle;
}

interface Produit_mixte: Produit_grand_public, Produit_professionnel
{
    attribute string reglementation_specifique;
}

interface Entrepot
{
    attribute string nom;
    attribute string adresse;
    attribute integer capacite;
}

interface Bon_de_livraison
{
    attribute integer numero;
    attribute string progression;
    relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
}

interface Bon_de_commande
{
    attribute string numero;
    relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
}

```

Figure 3.6. Plan de base décrit en ODL (étendu pour définir les cardinalités

3.3.3.2 Plans fonctionnels

Les plans fonctionnels sont apposés sur le plan de base pour enrichir les entités du domaine introduites par ce dernier plan. Chaque plan fonctionnel détient les parties des classes relatives à sa fonction. Ces parties fonctionnelles ont un rôle purement intensionnel qui est de décrire les informations fonctionnelles pour une entité. Sémantiquement, à la différence de sous-classes, les parties fonctionnelles n'introduisent pas de nouvelles entités et n'augmentent pas les extensions des classes issues du plan de base. Un plan fonctionnel préserve, par conséquence, les entités du référentiel.

Il existe des fonctions qui ne font pas participer certaines entités. Pour ces entités, le plan de telles fonctions n'apporte aucun enrichissement descriptif.

Remarques

- Toute relation doit être introduite dans un plan, c'est-à-dire que les attributs correspondant à ses extrémités doivent être introduits par le même plan, qu'il soit de base ou fonctionnel.

- Une relation inter-entités introduite dans le plan de base doit être partagée totalement dans une fonction¹². De ce fait, si une telle relation est partagée par une fonction, il faut que les entités situées aux deux extrémités de la relation participent à cette fonction. Par conséquent, les entités situées aux deux extrémités d'une relation introduite dans le plan de base doivent participer aux mêmes fonctions et leurs classes doivent donc être enrichies par les plans fonctionnels correspondants.

Par exemple, les relations entre les entités « Produit » et « Fournisseur », entre les entités « Produit » et « Client » et entre les entités « Bon_de_livraison » et « Bon_de_commande » sont introduites dans le plan de base. Il est facile de vérifier sur la figure 3.4 que ces entités mises en relation dans le plan de base participent aux mêmes fonctions.

Forme syntaxique employée pour les parties fonctionnelles

Nous introduisons une nouvelle forme syntaxique d'incrément d'interface en enrichissant ODL pour nos besoins. Celle-ci se présente ainsi :

```
function <function_name> interface increment <type_name>
{
  function part:
  {
    [<attribute_definition>]*
    [<relationship_definition>]*
  }
}
```

<function_name> est le nom de la fonction associée à la partie.

<type_name> est le nom de la classe dont on veut augmenter l'interface. Il s'agit du nom introduit dans le plan de base par la forme « interface ».

La forme « **function** <function_name> **interface increment** <type_name> » rend compte du fait qu'il s'agit d'augmenter l'interface d'une classe pour une fonction par une partie fonctionnelle.

12. pour respecter le fait que l'on partage la relation avec sa sémantique complète

Exemple

Le plan fonctionnel commercial enrichit les entités « Client », « Fournisseur », « Produit », « Bon_de_livraison » et « Bon_de_commande » et confère à leurs classes des parties fonctionnelles définies par le plan décrit à la figure 3.7. Ces parties fonctionnelles contiennent des attributs ainsi que des relations inter-entités.

```
function Commercial interface increment Client
{
  function part:
  {
    attribute integer montant_cmd;
    attribute date date_derniere_cmd;
    relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
    relationship set<Bon_de_commande> est_concerne_par_BC inverse Bon_de_commande::concerne;
  }
}

function Commercial interface increment Fournisseur
{
  function part:
  {
    attribute string contact_commercial;
  }
}

function Commercial interface increment Produit
{
  function part:
  {
    attribute integer nbr_unites_vendues;
    relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
    relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
  }
}

function Commercial interface increment Bon_de_livraison
{
  function part:
  {
    attribute date date_emission;
    relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BL;
    relationship set<Produit>[1] contient inverse Produit::est_contenu_dans_BL;
  }
}

function Commercial interface increment Bon_de_commande
{
  function part:
  {
    attribute date date_reception;
    relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BC;
    relationship set<Produit>[1] contient inverse Produit::est_contenu_dans_BC;
  }
}
```

Figure 3.7. Plan fonctionnel commercial décrit en ODL étendu

Le plan fonctionnel commercial est également représenté graphiquement à la figure 3.8 où les parties fonctionnelles sont représentées comme des classes mais avec un encadré pointillé.

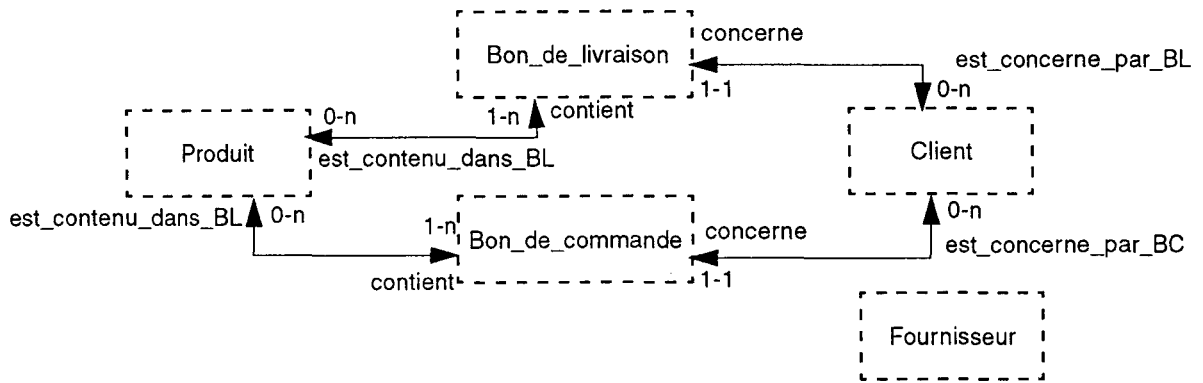


Figure 3.8. Représentation graphique du plan fonctionnel commercial

Définition formelle des descriptions de base et des parties fonctionnelles

Soit $\mathcal{C}$ l'ensemble des classes relatives aux entités du domaine et $\mathcal{A}$ l'ensemble de tous les attributs de toutes les classes appartenant à $\mathcal{C}$ ¹³. L'ensemble $\mathcal{A}$ contient également les attributs correspondant à des extrémités des relations.

L'intension d'une classe est une fonction qui associe à cette classe un ensemble d'attributs décrivant ses instances :

$$\begin{aligned} \text{intension: } \mathcal{C} &\rightarrow 2^{\mathcal{A}} \\ C &\rightarrow \{a_1, a_2, \dots, a_n\} \end{aligned}$$

Soit $\mathcal{F}$ un ensemble de symboles désignant les différentes fonctions et le symbole **base** se rapportant au plan de base.

Soit RF la relation d'équivalence qui associe deux attributs de l'ensemble $\mathcal{A}$ s'ils sont définis par le même plan.

$$\begin{aligned} \text{RF: } \mathcal{A} \times \mathcal{A} &\rightarrow \{0, 1\} \\ (a, b) &\rightarrow 1 \text{ si } a \text{ et } b \text{ sont définis par le même plan} \\ (a, b) &\rightarrow 0 \text{ sinon} \end{aligned}$$

Soit $\{\mathcal{D}_f, \forall f \in \mathcal{F}\}$ l'ensemble quotient¹⁴ $\mathcal{A}/\text{RF}$. $\mathcal{A}/\text{RF}$ détermine une partition de $\mathcal{A}$. $\mathcal{D}_{\text{base}}$ est, par convention, la classe d'équivalence des attributs du plan de base.

La description de base d'une classe $\mathcal{D}_{\text{base}}(C)$ est définie par la fonction suivante :

$$\begin{aligned} \mathcal{D}_{\text{base}}: \mathcal{C} &\rightarrow 2^{\mathcal{A}} \\ C &\rightarrow \mathcal{D}_{\text{base}} \cap \text{intension}(C) \end{aligned}$$

Une partie fonctionnelle d'une classe $\text{PF}_f(C)$ ($f \in \mathcal{F} - \{\text{base}\}$) est définie par la fonction suivante :

$$\begin{aligned} \text{PF}_f: \mathcal{C} &\rightarrow 2^{\mathcal{A}} \\ C &\rightarrow \mathcal{D}_f \cap \text{intension}(C) \end{aligned}$$

L'intersection illustre l'orthogonalité entre fonctions et entités.

13. c'est-à-dire $\forall a \in \mathcal{A}, \exists C \in \mathcal{C}$ tel que $a \in \text{intension}(C)$

14. ensemble des classes d'équivalence

Propriété 3.1 : l'intension d'une classe est partitionnée par $\mathcal{F}$

Cette propriété peut être formulée ainsi : $\text{intension}(C) = D_{\text{base}}(C) \cup_{f \in \mathcal{F} - \{\text{base}\}} PF_f(C)$

En effet :

$$\begin{aligned} D_{\text{base}}(C) \cup_{f \in \mathcal{F} - \{\text{base}\}} PF_f(C) &= (\mathcal{D}_{\text{base}} \cap \text{intension}(C)) \cup_{f \in \mathcal{F} - \{\text{base}\}} (\mathcal{D}_f \cap \text{intension}(C)) \\ &= (\mathcal{D}_{\text{base}} \cup_{f \in \mathcal{F} - \{\text{base}\}} \mathcal{D}_f) \cap \text{intension}(C) \\ &= \mathcal{A} \cap \text{intension}(C) \\ &= \text{intension}(C) \end{aligned}$$

Définition formelle du plan de base et d'un plan fonctionnel

Le plan de base est défini par :

$$P_{\text{base}} = \{D_{\text{base}}(C), C \in \mathcal{C}\}$$

Un plan fonctionnel est défini par :

$$P_f (f \in \mathcal{F} - \{\text{base}\}) = \{PF_f(C), C \in \mathcal{C} \wedge (PF_f(C) \neq \emptyset)\}$$

3.3.4 Héritage local dans un plan fonctionnel

Les plans fonctionnels préservent la structuration en classes du plan de base et en particulier, les relations d'héritage. Ainsi, au sein d'un plan fonctionnel, la partie fonctionnelle qui enrichit une classe du plan de base est héritée localement dans ses sous-classes. La relation d'héritage entre deux classes est ainsi contextualisée au sein d'un plan fonctionnel, portant localement sur des intensions (les parties fonctionnelles) spécifiques à ce plan.

Une conséquence de cet héritage local est qu'une sous-classe d'une classe enrichie dans un plan fonctionnel est également enrichie dans ce plan. Elle l'est soit explicitement par une partie fonctionnelle spécifique, soit implicitement par l'héritage de la partie fonctionnelle de sa surclasse. Cet enrichissement implicite provoque une participation implicite de l'entité correspondante dans la fonction correspondant au plan.

Exemple

La figure 3.9 montre un exemple d'héritage local au sein du plan fonctionnel « Fabrication ». Les entités « Part_externe », « Fournisseur », « Produit », « Produit_grand_public », « Produit_professionnel » et « Entrepôt » y sont explicitement enrichies.

```

function Fabrication interface increment Part_externe
{
  function part:
  {
    attribute string contact_technique;
  }
}
function Fabrication interface increment Fournisseur
{
  function part:
  {
    attribute boolean fournit_seulement_lot_complet_de_pieces;
  }
}
function Fabrication interface increment Produit
{
  function part:
  {
    attribute integer cout_fabrication;
  }
}
function Fabrication interface increment Produit_grand_public
{
  function part:
  {
    attribute integer nbr_chaines_fabrication;
  }
}
function Fabrication interface increment Produit_professionnel
{
  function part:
  {
    attribute integer duree_fabrication;
  }
}
function Fabrication interface increment Entrepot
{
  function part:
  {
    attribute integer distance_usine;
  }
}

```

Figure 3.9. Plan fonctionnel « Fabrication »

La contextualisation des relations d'héritage entre ces classes issues du plan de base fait que la partie fonctionnelle de « Produit » est héritée localement dans les parties fonctionnelles de « Produit_grand_public » et de « Produit_professionnel » comme le montre la figure 3.10. Les flèches épaisses grises illustrent cette contextualisation des relations d'héritage issues du plan de base au sein du plan fonctionnel.

Par ailleurs, le plan fonctionnel « Fabrication » n'enrichit pas explicitement l'intension de la classe « Client ». Cependant, celle-ci est enrichie par l'héritage de la partie fonctionnelle de « Part_externe », c'est-à-dire de l'attribut « contact_technique ». L'entité « Client » participe ainsi implicitement à la fonction « Fabrication ».

Il en va de même pour la classe « Produit_mixte » dont l'intension n'est enrichie que par

l'héritage des parties fonctionnelles de « Produit_grand_public » et de « Produit_professionnel ».

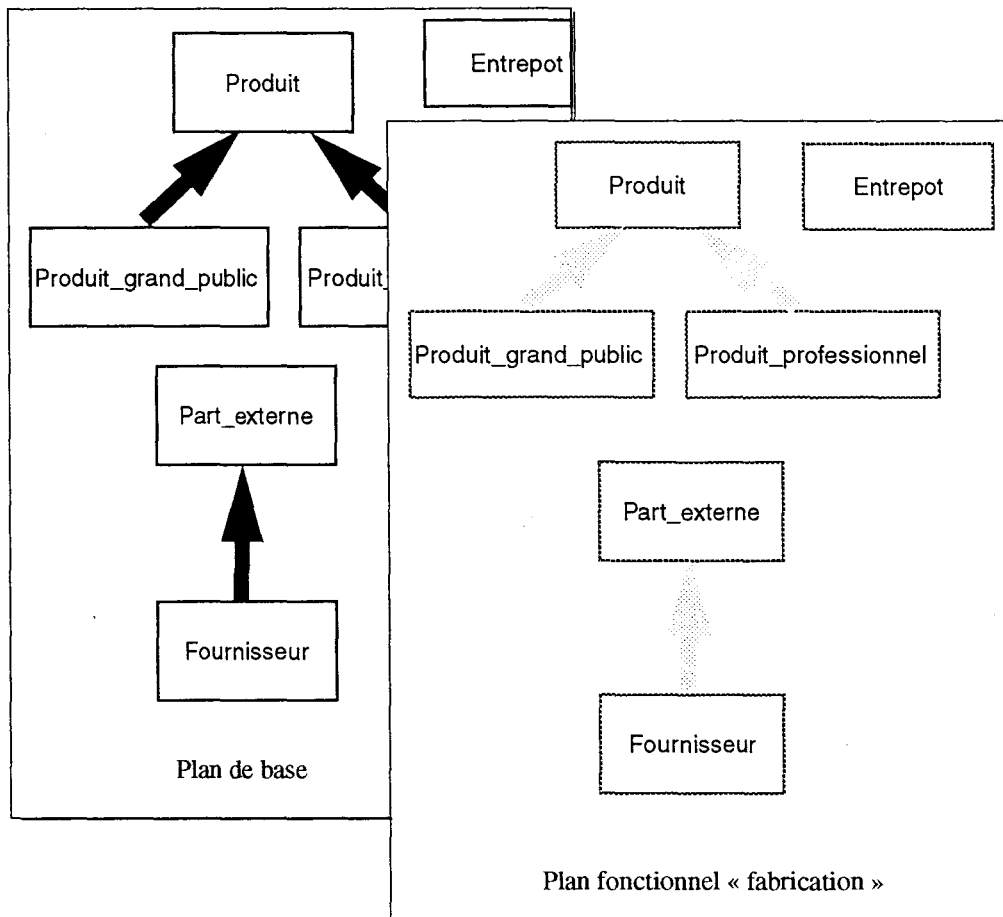


Figure 3.10. Héritage local dans le plan fonctionnel « Fabrication »

Propriété 3.2 : *Au sein d'un plan, la partie fonctionnelle d'une classe est héritée par ses sous-classes.*

La relation d'héritage entre intensions est définie ainsi :

Soit i et $i' \in 2^A$, $i \leq i' \Leftrightarrow i \supseteq i'$

La relation « sous-classe » entre deux classes implique l'héritage de leurs intensions respectives :

Soit $(C, C') \in \mathcal{C}^2$,

C sous-classe de $C' \Rightarrow \text{intension}(C) \leq \text{intension}(C')$

La relation d'héritage entre parties fonctionnelles peut être prouvée ainsi :

Soit $(C, C') \in \mathcal{C}^2$, C sous-classe de C' et soit $f \in \mathcal{F} - \{\text{base}\}$

C sous-classe de C' implique que $\text{intension}(C) \leq \text{intension}(C')$

Ceci est vrai à une intersection près.

Donc $\mathcal{D}_f \cap \text{intension}(C) \leq \mathcal{D}_f \cap \text{intension}(C')$ et donc $\text{PF}_f(C) \leq \text{PF}_f(C')$

3.3.5 Descriptions fonctionnelles

Une description fonctionnelle d'une classe est définie par l'union de l'une de ses parties fonctionnelles et de sa description de base. Sémantiquement, une description fonctionnelle d'une classe entretient une relation d'héritage avec la description de base correspondante. Cette relation assure l'articulation entre la description de base et une partie fonctionnelle. Elle permet ainsi le partage par toutes les fonctions des informations communes (cf 1.7.1). Ce partage est réalisé au sein des entités du domaine.

Notons que cet héritage ne permet pas d'affiner les propriétés introduites par la description de base. Dans notre cadre, l'affinement de propriétés consiste principalement en la restriction du type des attributs. Comme ces descriptions communes sont partagées par l'ensemble des fonctions, une restriction du type d'un attribut lors de l'héritage dans un plan fonctionnel provoquerait la même restriction dans les autres plans.

Exemple

Supposons que, dans la fonction commerciale, l'extrémité de la relation « fournit/est_fourni_par » puisse être affinée dans la partie fonctionnelle « Fournisseur » de sorte à ce que les fournisseurs livrent exclusivement des produits professionnels :

```
relationship set<Produit_professionnel>[1] fournit inverse Produit_professionnel::est_fourni_par;
```

Une telle restriction s'appliquerait systématiquement et il serait alors impossible, pour un fournisseur, de fournir des produits « grand public » sans violer cette restriction dans la fonction commerciale. D'autre part, dans le cadre du modèle de l'ODMG, cette spécialisation obligerait, dans la fonction commerciale, à modifier le lieu d'introduction de l'autre extrémité dans la sous-classe « Produit_professionnel ». Ceci semble difficilement compatible avec le statut de relation partagée entre deux entités par l'ensemble des fonctions.

Interfaces « ODL » pour les clients d'une fonction

Une description fonctionnelle contient les attributs et extrémités de relation qui sont présentés aux clients associés à la fonction correspondante. Les clients d'une fonction ne perçoivent les entités du domaine qu'au travers des descriptions fonctionnelles. De ce fait, une description fonctionnelle peut être représentée par la forme syntaxique « interface » d'ODL à des fins d'exploitation comme support pour exprimer des requêtes sans se préoccuper de la façon dont une telle description est obtenue. L'ensemble des interfaces correspondant aux différentes descriptions fonctionnelles sont présentées aux clients ainsi que les interfaces de toutes les surclasses d'une entité participant à la fonction, même si les entités relatives à ces surclasses ne participent pas à cette fonction. L'interface d'une surclasse, dans un cas de non-participation, correspond uniquement à sa description de base. Ceci permet de présenter l'ensemble des propriétés d'une classe présentes dans une fonction, y compris les propriétés introduites par ses surclasses.

Remarque

L'interface ODL d'une description fonctionnelle est générée automatiquement à partir de l'interface de la description de base et de l'incrément d'interface de la partie fonctionnelle.

Exemple

La figure 3.11 montre sous la forme syntaxique « interface » d'ODL les descriptions fonctionnelles relatives à la fonction commerciale.

```

interface Part_externe
{
    attribute string nom;
    attribute string adresse;
    attribute string telephone;
}

interface Client: Part_externe
{
    relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
    attribute integer montant_cmd;
    attribute date date_derniere_cmd;
    relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
    relationship set<Bon_de_commande> est_concerne_par_BC inverse Bon_de_commande::concerne;
}

interface Fournisseur: Part_externe
{
    relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
    attribute string contact_commercial;
}

interface Produit
{
    attribute string designation;
    attribute integer prix_tarif;
    relationship set<Client> a_ete_achete_par inverse Client::a_achete;
    relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
    attribute integer nbr_unites_vendues;
    relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
    relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
}

interface Produit_grand_public: Produit
{
    attribute integer reference_grand_public;
}

interface Produit_professionnel: Produit
{
    attribute string reference_professionnelle;
}

interface Produit_mixte: Produit_grand_public, Produit_professionnel
{
    attribute string reglementation_specifique;
}

interface Bon_de_livraison
{
    attribute integer numero;
    attribute string progression;
    relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
    attribute date date_emission;
    relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BL;
    relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BL;
}

interface Bon_de_commande
{
    attribute string numero;
    relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
    attribute date date_reception;
    relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BC;
    relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BC;
}

```

Figure 3.11. Descriptions fonctionnelles relatives à la fonction commerciale

Ces descriptions sont obtenues par union des parties fonctionnelles et des descriptions de base des classes dont l'entité participe à cette fonction.

Cet ensemble d'interfaces peut servir de support pour l'expression des requêtes issues des clients de la fonction commerciale. On note la présence d'une interface relative à « Part_externe » qui ne participe pas à la fonction commerciale. Ceci est dû à la participation de « Client » et de « Fournisseur ». Cette interface relative à « Part_externe » correspond à sa description de base.

Définition formelle d'une description fonctionnelle

Une description fonctionnelle d'une classe $DF_f(C)$ ($f \in \mathcal{F} - \{\mathbf{base}\}$) est définie par la fonction suivante :

$$\begin{aligned} DF_f: \quad \mathcal{C} &\rightarrow \mathcal{Z}^A \\ C &\rightarrow (\mathcal{D}_{\mathbf{base}} \cup \mathcal{D}_f) \cap \text{intension}(C) \end{aligned}$$

Propriété 3.3 : *Toute description fonctionnelle d'une classe hérite de la description de base de cette même classe.*

Cette propriété s'écrit par conséquent :

$$\forall f \in \mathcal{F} - \{\mathbf{base}\}, \forall C \in \mathcal{C}, DF_f(C) \leq D_{\mathbf{base}}(C)$$

En effet :

$$\begin{aligned} \forall f \in \mathcal{F} - \{\mathbf{base}\}, \forall C \in \mathcal{C} \\ DF_f(C) &= (\mathcal{D}_{\mathbf{base}} \cup \mathcal{D}_f) \cap \text{intension}(C) \\ &= (\mathcal{D}_{\mathbf{base}} \cap \text{intension}(C)) \cup (\mathcal{D}_f \cap \text{intension}(C)) \\ &= D_{\mathbf{base}}(C) \cup PF_f(C) \end{aligned}$$

Par conséquent, $DF_f(C) \leq D_{\mathbf{base}}(C)$

3.4. Interaction entre fonctions

Nous abordons le problème du partage des informations en prenant en compte la structuration en fonctions que nous avons présentée en 3.3. Les informations décrites par un plan fonctionnel sont purement spécifiques à la fonction correspondante et sont donc, par essence, fournies et consommées par les clients de cette fonction. Il existe aussi deux cas où des fonctions consomment les informations fournies par d'autres (cf 1.7.1) :

- le partage d'informations fonctionnelles entre fonctions. Ce partage donne lieu à une transmission de ces informations d'une fonction fournisseuse à une ou plusieurs fonctions consommatrices ;
- l'exploitation d'informations communes décrites par le plan de base.

Dans le second cas, si les descriptions de base sont partagées par l'ensemble des fonctions, les valeurs des attributs et des relations détenues par ces descriptions sont nécessairement fournies par les clients d'une ou plusieurs fonctions. En effet, il n'existe pas de clients de la base de données hormis ceux associés aux fonctions. Pour chaque attribut et relation des descriptions de base, il existe donc une ou plusieurs fonctions qui en déterminent les valeurs.

Nous proposons une solution pour chacun de ces deux cas de partage d'informations entre fonctions. Nous verrons qu'elles apportent la possibilité aux clients d'une fonction de distinguer les attributs et relations pour lesquels ils sont fournisseurs des valeurs, de ceux dont ils consomment les valeurs.

3.4.1 Accès inter-plans

Les accès inter-plans ont pour objectif de répondre au problème du partage d'informations fonctionnelles. Ce partage se fait dans un sens précis : de la fonction fournisseuse à celle consommatrice. Il donne lieu à un partage de descriptions entre fonctions au sein des classes du domaine et prend la forme d'un accès inter-plans. Un accès inter-plans enrichit les descriptions relatives à la fonction consommatrice avec des descriptions partagées issues de la fonction fournisseuse. Lors de l'exploitation, il offre un support aux clients de la fonction consommatrice pour que ceux-ci puissent exploiter les informations relatives aux descriptions partagées. En quelque sorte, il confère à la fonction consommatrice un statut de client interne¹⁵ : celle-ci exploite des descriptions et des informations issues d'une autre fonction.

Deux types d'accès doivent être distingués : celui relatif aux attributs et celui relatif aux relations.

Accès inter-plans d'attributs

Un accès inter-plans d'attributs spécifie un attribut d'une partie fonctionnelle dont la valeur est exploitée dans une autre fonction. Il ne peut s'agir d'un attribut d'une description de base car un tel attribut est, par définition, partagé par l'ensemble des fonctions. Chaque accès inter-plans est défini au sein d'une classe et est associé à la partie fonctionnelle de la fonction consommatrice.

15. Notons qu'un accès inter-plans relève de la conception. Il ne s'agit pas d'accéder à des informations ou à des données comme on le ferait à l'aide d'un langage de requêtes lors de l'exploitation.

Accès inter-plans de relations

Un accès inter-plans de relations spécifie une relation dont les occurrences sont exploitées par une autre fonction. Comme les relations sont décrites par deux extrémités, un accès inter-plans de relations se traduit par deux accès relatifs à ces deux extrémités. Ceux-ci doivent être présents simultanément pour respecter le fait qu'on accède à une relation et donc à sa sémantique complète.

Forme syntaxique employée pour les accès inter-plans

Les accès inter-plans sont définis dans la fonction consommatrice. La forme syntaxique d'un incrément d'interface est enrichie pour les accès inter-plans. Elle permet ainsi de définir simultanément la partie fonctionnelle et les accès inter-plans. Celle-ci se présente ainsi :

```
function <function_name> interface increment <type_name>
{
  [
    access function <function_name>:
    {
      [<attribute_definition>]*
      [<relationship_definition>]*
    }
  ]*
  function part:
  {
    [<attribute_definition>]*
    [<relationship_definition>]*
  }
}
```

Le symbole <function_name> qui suit la clause **access function** est le nom de la fonction qui fournit les attributs et relations dont la fonction consommatrice partage les descriptions. Une clause **access function** est définie pour chaque fonction fournisseuse. Elle contient l'ensemble des accès inter-plans dont cette fonction est fournisseuse. Chacun de ceux-ci est défini en reprenant à l'identique la déclaration initiale de l'attribut ou de l'extrémité de relation dans la partie fonctionnelle.

Exemple

La figure 3.12 montre les accès inter-plans des classes « Produit » et « Entrepôt » dans la fonction « SAV ».

```

function SAV interface increment Produit
{
  access function Fabrication:
  {
    attribute integer cout_fabrication;
  }
  access function Stock:
  {
    attribute integer cout_achat;
    relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
  }
  function part:
  ...
}

function SAV interface increment Entrepot
{
  access function Stock:
  {
    relationship set<Produit> stocke inverse Produit::est_stocke_dans;
  }
  function part:
  {
    attribute integer distance_centre_SAV;
  }
}

```

Figure 3.12. Accès inter-plans de « Produit » et « Entrepôt » dans la fonction « SAV »

Pour « Produit », il existe un accès vers la partie fonctionnelle « Fabrication » pour l'attribut « cout_fabrication » et deux accès vers la partie fonctionnelle « Stock » pour l'attribut « cout_achat » et pour la relation « est_stocke_dans/stocke ». L'autre extrémité de cette relation donne également lieu à un accès inter-plans dans la classe « Entrepôt ».

Prise en compte des accès inter-plans dans les interfaces générées pour les clients d'une fonction

Nous avons vu en 3.3.5 qu'une description fonctionnelle peut être représentée par la forme syntaxique « interface » d'ODL à des fins d'exploitation et non de conception. Nous proposons d'inclure dans l'interface destinée aux clients d'une fonction les accès inter-plans définis par cette dernière. Ceux-ci sont représentés par des déclarations d'attributs ou d'extrémités de relation préfixées par le mot clé **read-only**. Ce mot-clé a été introduit dans [ODMG97] pour les attributs et nous l'étendons aux extrémités de relation. Il convient de remarquer que cette solution ne permet pas aux clients d'une fonction de connaître les fournisseurs des accès inter-plans mais simplement de les exploiter¹⁶.

Exemple

La figure 3.13 donne l'interface pour les clients de la fonction « SAV » de la classe « Entrepôt ». L'accès inter-plans donne lieu à un préfixe **read-only**.

16. Ceci est suffisant pour les besoins de ces clients.

```

interface Entrepot
{
  attribute string nom;
  attribute string adresse;
  attribute integer capacite;
  attribute integer distance_centre_SAV;
  read-only relationship set<Produit> stocke inverse Produit::est_stocke_dans;
}

```

Figure 3.13. Interface pour les clients de la fonction « SAV » de la classe « Entrepôt »

Remarques

- Un accès inter-plans vers un attribut ou une extrémité d'une relation est obligatoirement réalisé dans la classe d'introduction de cet attribut ou de cette extrémité. Il ne peut avoir lieu dans l'une des sous-classes. Ceci se justifie aisément dans le cas des relations. Supposons que, dans la fonction « SAV », l'accès inter-plans relatif à la relation « est_stocké_dans/stocke » soit introduit non pas dans la classe « Produit » mais dans sa sous-classe « Produit_Professionnel ». Ceci ne serait pas consistant avec l'accès inter-plans situé dans « Entrepôt » et qui désigne « Produit » comme autre extrémité de la relation. Un tel accès inter-plans ne respecterait pas dans la fonction « SAV » la sémantique de la relation « est_stocké_dans/stocke » telle qu'elle est définie dans la fonction « Stock ». Nous généralisons cette règle à tous les attributs pour des raisons d'homogénéité.

- Les accès inter-plans relatifs à une classe du domaine sont hérités dans ses sous-classes, similairement aux parties fonctionnelles. Cet héritage est local comme celui des parties fonctionnelles au sein du plan fonctionnel.

Par exemple, dans la fonction « SAV », les accès inter-plans relatifs à la classe « Produit » sont hérités localement dans les sous-classes « Produit_grand_public », « Produit_professionnel » et « Produit_mixte » de la même façon qu'est héritée localement la partie fonctionnelle de « Produit ».

Définition formelle d'un accès inter-plans

$\text{Acc}_f(C)$ est défini comme l'ensemble des accès aux attributs d'une classe C fournis par les autres fonctions que f .

$$\begin{aligned}
 \text{Acc}_f(C): \quad \mathcal{C} &\rightarrow 2^{\mathcal{A} \times (\mathcal{I} - \{\text{base}\})} \\
 C &\rightarrow \{(a, ff), a \in \text{PF}_{ff}(C) \wedge \\
 &\quad \neg (\exists SC \in \mathcal{C}, (\text{PF}_{ff}(C) \leq \text{PF}_{ff}(SC)) \wedge (a \in \text{PF}_{ff}(SC)))\}
 \end{aligned}$$

Nous définissons $\mathcal{R}$ comme l'ensemble des relations entre deux classes de $\mathcal{C}$. Un élément R de $\mathcal{R}$ peut être représenté par le couple $((C, a), (C', b))$ tel que $(C, C') \in \mathcal{C}^2$ et $(a, b) \in \text{intension}(C) \times \text{intension}(C')$. a et b sont les attributs des extrémités de R . C et C' sont respectivement les classes d'introduction des extrémités a et b , c'est-à-dire que :

$$\neg (\exists SC \in \mathcal{C}, (\text{intension}(C) \leq \text{intension}(SC)) \wedge (a \in \text{intension}(SC)))$$

$$\neg (\exists SC' \in \mathcal{C}, (\text{intension}(C') \leq \text{intension}(SC')) \wedge (b \in \text{intension}(SC')))$$

Une relation est obligatoirement introduite par un plan, ce qui se traduit par :

$$a \in D_{\text{base}}(C) \wedge b \in D_{\text{base}}(C') \text{ ou}$$

$\exists f \in \mathcal{F}\text{-}\{\mathbf{base}\}$ tel que $a \in PF_f(C) \wedge b \in PF_f(C')$

Si $a \in D_{base}(C) \wedge b \in D_{base}(C')$, alors $PF_f(C) \neq \emptyset \Leftrightarrow PF_f(C') \neq \emptyset$ car les entités situées aux deux extrémités d'une relation du plan de base sont enrichies par les mêmes plans fonctionnels (cf 3.3.3.2).

Soit une relation $R \in \mathcal{R}$ et représentée par $((C,a),(C',b))$ avec $a \in PF_{ff}(C)$ et $b \in PF_{ff}(C')$. Le fait que R soit rendue accessible à la fonction f , se traduit par $(a,ff) \in Acc_f(C)$ et $(b,ff) \in Acc_f(C')$.

Héritage des accès inter-plans d'une classe par ses sous-classes

Les sous-classes d'une classe héritent au sein de chaque fonction de ses accès inter-plans

$\forall (C,C') \in \mathcal{C}^2, \forall f \in \mathcal{F}\text{-}\{\mathbf{base}\}, C \text{ sous-classe de } C' \Rightarrow Acc_f(C) \supseteq Acc_f(C')$

3.4.2 Attributs et relations du plan de base

Nous voulons expliciter les fonctions qui fournissent les valeurs des attributs et des relations du plan de base. Nous proposons d'utiliser une spécification de fourniture lors de l'apposition d'un plan fonctionnel sur le plan de base. Cette solution est plus modulaire et évolutive que celle qui consisterait à déterminer la spécification de fourniture au sein du plan de base. Ceci imposerait de connaître les plans fonctionnels lors de la conception du plan de base.

Cette spécification se traduit par une modalité optionnelle de « fourniture de valeur » relative à chaque attribut et extrémité de relations lors de l'héritage entre une description de base et une description fonctionnelle. Si cette modalité est précisée, la fonction fournit la valeur de l'attribut ou de l'extrémité de la description de base. Dans le cas contraire, la fonction se limite à exploiter cette valeur.

Forme syntaxique employée pour les modalités de fourniture

La forme syntaxique d'incrément d'interface est enrichie pour définir les attributs et extrémités de fonction héritées dont la fonction fournit les valeurs. Celle-ci se présente ainsi :

```
function <function_name> interface increment <type_name>
{
[
access function <function_name>:
{
[<attribute_definition>]*
[<relationship_definition>]*
}
]*
[
write base part:
{
[<attribute_definition>]*
[<relationship_definition>]*
}
]
function part:
{
[<attribute_definition>]*
[<relationship_definition>]*
}
}
}
```

La clause **write base part** permet de préciser les attributs et extrémités de relation de la description de base dont la fonction fournit les valeurs.

Exemple

A la figure 3.14 se trouvent les incréments d'interface des classes « Client » et « Produit » relatifs à la fonction commerciale. Il convient de remarquer qu'il n'existe pas de modalités de fourniture de propriétés de base pour la classe « Part_Externe ».

```
function Commercial interface increment Client
{
    access function Comptable:
    {
        attribute integer encours_a_regler;
    }
    write base part:
    {
        attribute string nom;
        attribute string adresse;
        attribute string telephone;
        relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
    }
    function part:
    {
        attribute integer montant_cmd;
        attribute date date_derniere_cmd;
        relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
        relationship set<Bon_de_commande> est_concerne_par_BC inverse Bon_de_commande::concerne;
    }
}

function Commercial interface increment Produit
{
    access function Comptable:
    {
        attribute integer chiffre_affaires;
    }
    access function Direction:
    {
        attribute integer reduction_maximale;
    }
    write base part:
    {
        attribute integer prix_tarif;
        relationship set<Client> a_ete_achete_par inverse Client::a_achete;
    }
    function part:
    {
        attribute integer nbr_unites_vendues;
        relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
        relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
    }
}
```

Figure 3.14. Accès inter-plans, modalités de fourniture des propriétés de base et parties fonctionnelles des classes « Client » et « Produit » dans la fonction commerciale

Prise en compte des modalités de fourniture dans les interfaces générées pour les clients d'une fonction

Une description fonctionnelle (cf 3.3.5) peut être représentée par la forme syntaxique « interface » d'ODL à des fins d'exploitation. Pour rendre compte dans cette interface des modalités de fourniture, nous proposons de préfixer par le mot-clé **read-only** les spécifications d'attributs ou de relations relatives aux propriétés de base dont la fonction ne fournit pas la valeur. Ainsi, les clients peuvent déterminer les propriétés dont ils fournissent les valeurs.

La figure 3.15 donne les interfaces des classes « Client » et « Produit » destinées aux clients de la fonction commerciale. On y trouve aussi une interface relative à « Part_externe » surclasse de « Client » (cf 3.3.5). Dans « Client », l'attribut préfixé par **read-only** est relatif à un accès inter-plans. Dans « Produit », l'attribut « désignation » et l'extrémité de la relation « est_fourni_par/fournit » sont issus du plan de base mais leurs valeurs ne sont pas fournies par les clients de la fonction commerciale. De ce fait, leur définition est préfixée par **read-only**.

```
interface Part_externe
{
    read-only attribute string nom;
    read-only attribute string adresse;
    read-only attribute string telephone;
}

interface Client: Part_externe
{
    attribute string nom;
    attribute string adresse;
    attribute string telephone;
    relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
    attribute integer montant_cmd;
    attribute date date_derniere_cmd;
    relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
    relationship set<Bon_de_commande> est_concerne_par_BC inverse Bon_de_commande::concerne;
    read-only attribute integer encours_a_regler;
}

interface Produit
{
    read-only attribute string designation;
    attribute integer prix_tarif;
    relationship set<Client> a_ete_achete_par inverse Client::a_achete;
    read-only relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
    attribute integer nbr_unites_vendues;
    relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
    relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
    read-only attribute integer chiffre_affaires;
    read-only attribute integer reduction_maximale;
}
```

*Figure 3.15. Interface pour les clients de la fonction commerciale des classes
« Client » et « Produit »*

Remarques

• Les deux extrémités d'une relation doivent être héritées avec la même modalité dans un plan fonctionnel. Une modalité de fourniture s'applique totalement à une relation : un utilisateur peut ajouter ou enlever des occurrences de la relation. Ajouter ou retirer une occurrence s'applique aux deux extrémités.

Les clients de la fonction commerciale fournissent les occurrences de la relation du plan de base « a_achetée/a_été_acheté_par » entre les entités « Client » et « Produit ». Chaque extrémité de cette relation est incluse dans les clauses **write base part** des deux incréments d'interface relatifs à cette même fonction.

• Une modalité de « fourniture de valeur » pour un attribut ou une extrémité de relation ne doit pas nécessairement intervenir dans leur classe d'introduction. C'est le cas dans l'exemple des figures 3.14 et 3.15 où la fonction commerciale ne fournit les attributs « nom », « adresse » et « telephone » que pour la sous-classe « Client » de « Part_Externe ». Ceci explique que ces attributs soient préfixés par **read-only** que dans l'interface de « Part_Externe » et qu'ils ne le soient pas dans l'interface de « Client ».

Supposons qu'une modalité de « fourniture de valeur » d'une relation soit introduite, pour chacune de ses extrémités, dans une sous-classe de leur classe d'introduction dans le plan de base. Les clients de la fonction ne pourront alors créer des occurrences qu'entre les instances de ces deux sous-classes.

• Une modalité de « fourniture de valeur » s'applique à toutes les instances d'une classe dans une fonction, y compris les instances de ses sous-classes. Elle est héritée dans ses sous-classes au sein de la fonction où elle s'applique.

Les clients de la fonction commerciale fournissent les valeurs de l'attribut « prix_tarif » de la description de base de « Produit ». Ils le font également pour ce même attribut pour les entités « Produit_grand_public », « Produit_professionnel » et « Produit_mixte »¹⁷.

Définition formelle des modalités de fourniture d'attributs de base

$WB_f(C)$ est défini comme l'ensemble des attributs de la description de base d'une classe dont la valeur est fournie par une fonction.

$$WB_f(C): \mathcal{C} \rightarrow 2^A$$

$$C \rightarrow \{a_1, a_2, \dots, a_n \text{ tels que } \forall i \in 1..n, a_i \in D_{\text{base}}(C)\}$$

Si les occurrences d'une relation du plan de base $R \in \mathcal{R}$ et représentée par $((C, a), (C', b))$ avec $a \in D_{\text{base}}(C)$ et $b \in D_{\text{base}}(C')$ sont fournies par la fonction f à partir des classes CC et CC' respectivement sous-classes de C et C' alors $a \in WB_f(CC)$ et $b \in WB_f(CC')$. Ceci signifie que les clients de la fonction f ne peuvent créer d'occurrences de R qu'entre des instances de CC et CC' .

Héritage des modalités de fourniture d'une classe par ses sous-classes

Les sous-classes d'une classe héritent dans chaque fonction de ses modalités de fourniture de propriétés du plan de base.

$$\forall (C, C') \in \mathcal{C}^2, \forall f \in \mathcal{F} - \{\mathbf{base}\}, C \text{ sous-classe de } C' \Rightarrow WB_f(C) \supseteq WB_f(C')$$

17. Rappelons que ces trois entités participent également à la fonction commerciale car leur classe y est enrichie implicitement par héritage de la partie fonctionnelle de « Produit » (cf 3.3.4).

3.5. Contraintes de participation aux fonctions

Après avoir étudié les aspects relatifs aux intensions de CROME, nous nous intéressons maintenant aux aspects relatifs aux extensions dont nous avons introduit la problématique en 1.6.2. Nous proposons une solution à cette problématique dans le cadre de CROME.

Il s'agit de former les extensions fonctionnelles de chaque classe du domaine. Une première solution consisterait à utiliser les classes-vues de sélection présentées dans le chapitre 2. Une classe-vue définie par sélection permettrait de déterminer l'ensemble des instances d'une classe participant à une fonction, la sélection se faisant par un prédicat. Cette classe-vue serait introduite localement à la fonction et ne serait présentée qu'aux clients de celle-ci. L'extension de la classe-vue serait alors égale à l'extension fonctionnelle introduite en 1.6.2.

Par exemple, dans la fonction « Fabrication », seuls les produits dont le coût de stockage est supérieur à 300 F y participent. La figure 3.16 montre la partie fonctionnelle et les accès inter-plans de la classe « Produit » dans la fonction « Fabrication ». La classe-vue « Produit-Fabrication » qui sélectionne les produits y est représentée graphiquement.

```
function Fabrication interface increment Produit
{
  access function Commercial:
  {
    attribute integer nbr_unites_vendues;
  }
  access function Stock:
  {
    attribute integer cout_stockage;
    relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
  }
  function part:
  {
    attribute integer cout_fabrication;
  }
}
```

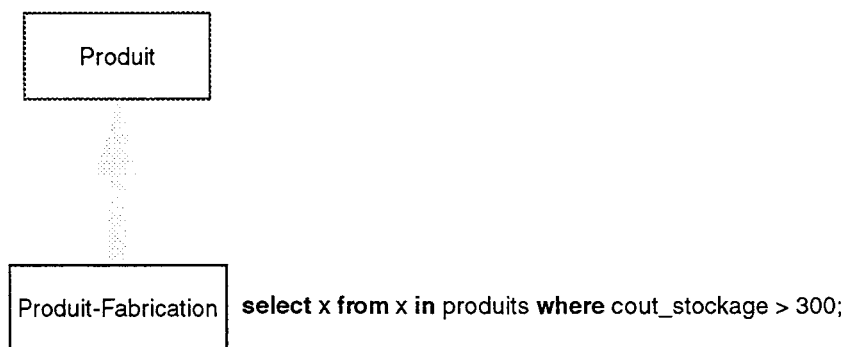


Figure 3.16. Sélection des instances de la classe « Produit » dans la fonction « Fabrication » au travers d'une classe-vue

Cette solution présente principalement l'inconvénient de ne pas être interne à la classe du domaine, offrant la possibilité d'avoir plusieurs classes-vues définies par sélection pour une même classe. Ceci est contradictoire avec la notion unique d'extension fonctionnelle (cf 1.6.2). D'autre part, les parties fonctionnelles sont liées à la classe. La sélection par une (sous-)classe-vue impose donc l'enrichissement systématique de toutes les instances de la classe, avant que la sélection se fasse. Enfin, les entités du domaine concernées par une telle sélection participent à la fonction et leurs classes présentent donc toutes leurs instances aux clients.

En fait, cette solution est une sélection d'instances au sein d'une fonction et non pas une sélection des instances de chaque classe pour former les extensions fonctionnelles. Nous proposons par la suite de réaliser cette sélection au sein de la classe du domaine, l'enrichissement descriptif de chaque instance étant alors lié à l'ensemble des fonctions auxquelles elle participe.

3.5.1 Prédicat de participation

Comme pour les classes-vues de sélection [Bertino92, Dayal89, Geppert93, Rundensteiner92a, Scholl91, Souza Dos santos94, Tanaka88] et les « predicate classes » de [Chambers93], nous proposons d'utiliser un prédicat pour déterminer la participation d'une instance d'une classe à une fonction. Ce prédicat s'applique sur les valeurs des attributs et/ou des relations de l'entité concernée. Nous utilisons les possibilités d'expression d'un prédicat fournies par le langage OQL. Un prédicat se présente alors sous la forme d'une requête OQL dont le résultat est une valeur logique (booléenne). Lorsque ce prédicat est vérifié pour une instance, celle-ci participe alors à la fonction¹⁸. Ce prédicat, appelé prédicat de participation, constitue (pour l'instant) la seule contrainte de participation.

Le prédicat de participation est défini au sein d'une classe et est associé à la partie fonctionnelle correspondante. Cette définition au sein de la classe rend compte du fait qu'il s'agit d'une caractéristique de l'entité associée et que cette caractéristique est unique pour chaque fonction où participe cette entité. Lorsque le prédicat de participation est vérifié pour une instance, celle-ci est enrichie de la partie fonctionnelle correspondante. Notons la présence systématique d'un prédicat de participation pour chaque fonction où une classe est enrichie. Dans le cas d'une participation systématique de toutes les instances d'une classe à une fonction, le prédicat est une tautologie.

Un prédicat de participation ne s'exprime que sur les propriétés héritées du plan de base ou définies dans une autre fonction. En effet, une instance qui ne participe pas à une fonction ne détient pas la partie fonctionnelle correspondante. Par conséquent, le prédicat de participation ne peut s'exprimer sur des propriétés d'une partie fonctionnelle dont il détermine l'existence. L'utilisation de propriétés définies dans une autre fonction se fait au travers d'accès inter-plans explicites comme nous l'avons vu en 3.4.1.

D'une façon générale, le prédicat de participation ne peut comporter de propriétés dont il détermine la valeur ou l'existence. Il ne peut donc porter sur l'extension fonctionnelle qu'il détermine ou, comme nous venons de le voir, sur les propriétés des parties fonctionnelles correspondantes. Nous verrons par la suite qu'il existe d'autres restrictions.

Remarquons aussi que le prédicat peut porter sur des propriétés d'autres entités s'il

18. nous verrons par la suite que d'autres conditions viennent s'ajouter à cette vérification.

comporte des expressions de chemin sous-tendues par les relations entre ces entités. Ces propriétés doivent être incluses dans les descriptions fonctionnelles ou les accès inter-plans correspondant à la même fonction.

Exemples

Nous reprenons l'exemple de la sélection des instances de la classe « Produit » dans la fonction « Fabrication ». Le prédicat de participation étant défini au sein d'une classe, l'absence de préfixe pour une propriété signifie qu'il s'agit d'une propriété de l'instance dont on détermine la participation :

`cout_stockage > 300`

Ce prédicat est correct. Il contient un accès vers un attribut de la partie fonctionnelle « Stock » par un accès inter-plans

`prix_tarif > 1000`

Ce prédicat est correct. Il utilise un attribut de la description de base de « Produit »

`for all entrepot in est_stocke_dans: entrepot.distance_usine < 50`

Ce prédicat est correct. Il consiste à sélectionner les produits stockés exclusivement dans des entrepôts situés à moins de 50 km de l'usine de fabrication. « est_stocke_dans » est un attribut de la partie fonctionnelle « Stock » pour lequel existe un accès inter-plans. « distance_usine » est un attribut de la partie fonctionnelle « Fabrication » de la classe « Entrepot »

`for all entrepot in Entrepot: entrepot.distance_usine < 50`

Ce prédicat est correct quoique étrange. La participation de chaque produit est conditionnée par le fait que tous les entrepôts soient situés à moins de 50 km de l'usine de fabrication, ce qui est indépendant en fait de chaque produit.

`true`

Ce prédicat est correct. C'est une tautologie qui assure que toutes les instances de « Produit » participent à la fonction « Fabrication ».

`cout_fabrication > 1000`

Ce prédicat n'est pas correct. « cout_fabrication » est un attribut de la partie fonctionnelle « Fabrication » de la classe « Produit ».

`cout_livraison > 200`

Ce prédicat n'est pas correct. « cout_livraison » est un attribut de la partie fonctionnelle « Stock » de la classe « Produit » (cf annexe A) pour lequel il n'existe pas d'accès inter-plans dans la fonction « Fabrication ».

`for all entrepot in est_stocke_dans: entrepot.valeur_stock < 1000`

Ce prédicat n'est pas correct. « valeur_stock » est un attribut de la partie fonctionnelle « Comptable » de la classe « Entrepot » (cf annexe A) pour lequel il n'existe pas d'accès inter-plans dans la fonction « Fabrication ».

3.5.2 Extension fonctionnelle d'une classe

La notion d'extension fonctionnelle d'une classe a été introduite en 1.6.2 comme l'ensemble de ses instances participant à une fonction. La contrainte de participation à une fonction étant déterminée par un prédicat de participation, l'extension fonctionnelle se définit maintenant comme l'ensemble des instances d'une classe vérifiant celui-ci. Elle est bien sûr

égale à l'extension de la classe dans le cas où le prédicat de participation est une tautologie.

Remarquons qu'il s'agit d'une notion propre à une classe du domaine et non pas à une (sous-)classe-vue. Pour chaque fonction où elle intervient, une classe du domaine ne détient qu'une seule extension fonctionnelle.

Forme syntaxique employée pour introduire les prédicats de participation

La forme syntaxique utilisée pour définir un incrément d'interface est enrichie pour inclure le prédicat de participation. Celle-ci se présente maintenant ainsi :

```
function <function_name> interface increment <type_name>
{
  [
    access function <function_name>:
    {
      [<attribute_definition>]*
      [<relationship_definition>]*
    }
  ]*
  [ function extent condition: <predicate_expression>; ]
  [
    write base part:
    {
      [<attribute_definition>]*
      [<relationship_definition>]*
    }
  ]
  function part:
  {
    [<attribute_definition>]*
    [<relationship_definition>]*
  }
}
```

La clause **function extent condition** permet de préciser le prédicat de participation des instances de la classe <type_name> à la fonction <function_name>. <predicate_expression> est une forme OQL définissant une requête dont le résultat est une valeur logique (booléenne). Les attributs et extrémités de relation non préfixés au sein de cette requête sont relatifs à la classe désignée par <type_name>.

Exemple

La figure 3.17 montre l'incrément d'interface des produits pour la fonction « Fabrication ». La présence du prédicat de participation fait que l'extension fonctionnelle de la classe « Produit » dans la fonction « Fabrication » ne contient que les instances de l'extension de la classe « Produit » dont le coût de stockage est supérieur à 300 F.

```

function Fabrication interface increment Produit
{
  access function Commercial:
  {
    attribute integer nbr_unites_vendues;
  }
  access function Stock:
  {
    attribute integer cout_stockage;
    relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
  }
  function extent condition: cout_stockage > 300;
  function part:
  {
    attribute integer cout_fabrication;
  }
}

```

Figure 3.17. Partie fonctionnelle, accès inter-plans et prédicat de participation des instances de la classe « Produit » à la fonction « Fabrication »

Remarques

- L'ajout de cette clause ne change pas les interfaces d'exploitation destinées aux clients des fonctions. En effet, le prédicat de sélection n'est pas présenté à ces derniers.
- Il n'est pas possible dans le prédicat de sélection d'utiliser des accès inter-plans qui ne soient pas présentés aux clients. Le rendre possible nécessiterait la possibilité de définir des accès inter-plans encapsulés à l'aide d'une seconde clause **access function**. Un accès inter-plans encapsulé d'une relation à une extrémité devrait également l'être à l'autre extrémité. Un tel choix nous paraît trop complexe pour répondre aux besoins.

3.5.3 Etat fonctionnel d'une instance

Nous avons introduit la notion d'état fonctionnel en 1.6.2 comme l'ensemble des fonctions auxquelles participe une instance. Pour chaque fonction où sa classe d'instanciation est enrichie, la participation d'une instance est décidée par un prédicat. Une instance participe donc à l'ensemble des fonctions pour lesquelles sa classe d'instanciation est enrichie et pour lesquelles elle vérifie le prédicat de participation qui est associé à sa classe.

L'état fonctionnel d'une instance détermine l'ensemble des parties fonctionnelles dont elle est dotée. La participation d'une instance à une fonction provoque l'existence d'une partie fonctionnelle correspondante que détient alors ladite instance.

Exemple

La fonction « Fabrication » ne fait participer que les produits dont le coût de stockage est supérieur à 300 F. La présence de la fonction « Fabrication » dans l'état fonctionnel d'une instance de « Produit » nécessite que son coût de stockage soit supérieur à 300 F. Dans ce cas, cette instance est munie de la partie fonctionnelle relative à la fonction « Fabrication ».

La figure 3.18 illustre les deux cas de participation et de non participation d'un produit à la fonction « Fabrication ».

produit1	
désignation : "GSM" prix_tarif : 3500 a_été_acheté_par : {client1,client2} est_fourni_par : {fournisseur1}	instance participant à la fonction « Fabrication » son état fonctionnel vaut {Stock,Fabrication,...}
est_stocké_dans : {entrepot1} coût_stockage : 350	description de base
cout_fabrication : 2000	partie fonctionnelle « Stock »
	partie fonctionnelle « Fabrication »
	autres parties fonctionnelles

produit2	
désignation : "batterie GSM" prix_tarif : 450 a_été_acheté_par : {client7,client8} est_fourni_par : {fournisseur1}	instance ne participant pas à la fonction « Fabrication » son état fonctionnel vaut {Stock,...}
est_stocké_dans : {entrepot12} coût_stockage : 50	description de base
	partie fonctionnelle « Stock »
	autres parties fonctionnelles

Figure 3.18. Parties fonctionnelles décrivant une instance de « Produit » participant à la fonction « Fabrication » et une instance de « Produit » n’y participant pas

Définition formelle de l’extension fonctionnelle et de l’état fonctionnel

Soit $\mathcal{O}$ l’ensemble de toutes les instances de toutes les classes du domaine.

L’extension d’une classe est une fonction qui associe à cette classe l’ensemble de ses instances :

$$\text{extension: } \mathcal{C} \rightarrow 2^{\mathcal{O}}$$

$$C \rightarrow \{o_1, o_2, \dots, o_n\}$$

La relation « sous-classe » entre deux classes est définie par :

$$\text{Soit } (C, C') \in \mathcal{C}^2,$$

$$C \text{ sous-classe de } C' \Leftrightarrow \text{intension}(C) \leq \text{intension}(C') \wedge \text{extension}(C) \subseteq \text{extension}(C')$$

La relation « sous-classe-directe » entre deux classes est définie par

$$\text{Soit } (C, C') \in \mathcal{C}^2,$$

$$C \text{ sous-classe-directe de } C' \Leftrightarrow C \text{ sous-classe de } C' \wedge \neg(\exists C'' \in \mathcal{C} \text{ telle que } C \text{ sous-classe de } C'' \wedge C'' \text{ sous-classe de } C')$$

L’extension stricte d’une classe est une fonction qui associe à cette classe l’ensemble des instances qui lui sont propres, à l’exclusion des instances de ses sous-classes :

$$\begin{aligned} \text{extension}_{\text{stricte}}: \quad & \mathcal{C} \rightarrow 2^{\mathcal{O}} \\ & C \rightarrow \{o_1, o_2, \dots, o_p\} \end{aligned}$$

L'extension peut être définie récursivement à partir de l'extension stricte :

$$\begin{aligned} \text{extension}(C) &= \text{extension}_{\text{stricte}}(C) \cup \bigcup_{SC \in \text{sous-classe-directe}(C)} \text{extension}(SC) \\ \text{sous-classe-directe}(C) &\text{ est l'ensemble des sous-classes directes de } C. \end{aligned}$$

La fonction « classe » associe à un élément de $\mathcal{O}$ sa classe d'instanciation

$$\begin{aligned} \text{classe}: \quad & \mathcal{O} \rightarrow \mathcal{C} \\ & o \rightarrow C \text{ telle que } o \in \text{extension}_{\text{stricte}}(C) \end{aligned}$$

L'état fonctionnel d'une instance est défini par la fonction suivante :

$$\begin{aligned} \mathcal{EF}: \quad & \mathcal{O} \rightarrow 2^{\mathcal{F}-\{\text{base}\}} \\ & o \rightarrow \{f \in \mathcal{F}-\{\text{base}\} \text{ telle que :} \\ & \quad \text{PF}_f(\text{classe}(o)) \neq \emptyset \text{ et} \\ & \quad \text{Pred}_f(\text{classe}(o)) \text{ soit vérifié pour } o\} \end{aligned}$$

Cette définition montre bien que c'est la classe d'instanciation qui détermine l'état fonctionnel d'une instance.

L'extension fonctionnelle d'une classe $\text{EF}_f(C)$ ($f \in \mathcal{F}-\{\text{base}\}$ et $\text{PF}_f(C) \neq \emptyset$) est définie par la fonction suivante :

$$\begin{aligned} \text{EF}_f: \quad & \mathcal{C} \rightarrow 2^{\mathcal{O}} \\ & C \rightarrow \{o \in \text{extension}(C) \text{ tel que } f \in \mathcal{EF}(o)\} \end{aligned}$$

Propriété 3.4 : *L'extension fonctionnelle d'une sous-classe d'une classe est incluse dans l'extension fonctionnelle de cette classe.*

Cette propriété s'écrit :

$$\begin{aligned} \forall (C, C') \in \mathcal{C}^2, \forall f \in \mathcal{F}-\{\text{base}\} \text{ avec } \text{PF}_f(C') \neq \emptyset \\ C \text{ sous-classe de } C' \Rightarrow \text{EF}_f(C) \subseteq \text{EF}_f(C') \end{aligned}$$

En effet, on peut écrire :

$$\text{EF}_f(C) = \{o \in \text{extension}(C) \text{ tel que } f \in \mathcal{EF}(o)\} \text{ car on sait que } \text{PF}_f(C) \neq \emptyset \text{ vu que } \text{PF}_f(C) \leq \text{PF}_f(C') \text{ et que } \text{PF}_f(C') \neq \emptyset.$$

De même, on peut écrire :

$$\begin{aligned} \text{EF}_f(C') &= \{o \in \text{extension}(C') \text{ tel que } f \in \mathcal{EF}(o)\} \\ \text{Comme } \text{extension}(C) &\subseteq \text{extension}(C'), \text{ si } o \in \text{EF}_f(C) \text{ alors } o \in \text{EF}_f(C') \\ \text{Par conséquent, } \text{EF}_f(C) &\subseteq \text{EF}_f(C') \end{aligned}$$

3.5.4 Héritage du prédicat de participation

Les prédicats de participation des classes « Produit », « Produit_grand_public » et « Produit_professionnel » dans la fonction « Fabrication » sont donnés à la figure 3.19. Rappelons que les classes « Produit_professionnel » et « Produit_grand_public » sont des sous-classes directes de « Produit ».

```

function Fabrication interface increment Produit
{
  ...
  function extent condition: cout_stockage > 300;
  function part:
  {
    attribute integer cout_fabrication;
  }
}

function Fabrication interface increment Produit_grand_public
{
  function extent condition: nbr_unites_vendues > 20000;
  function part:
  {
    attribute integer nbr_chaines_fabrication;
  }
}

function Fabrication interface increment Produit_professionnel
{
  function extent condition: nbr_unites_vendues > 2000;
  function part:
  {
    attribute integer duree_fabrication;
  }
}

```

Figure 3.19. Prédicats de participation à la fonction « Fabrication » des instances des classes « Produit », « Produit_grand_public » et « Produit_professionnel »

La définition de l'extension fonctionnelle peut poser un problème de cohérence. Supposons que « p » soit une instance de « Produit_professionnel » dont l'attribut « nbr_unites_vendues » est valué 2500 et dont l'attribut « cout_stockage » est valué 100. « p » participe à la fonction « Fabrication » car elle vérifie le prédicat de participation de sa classe d'instanciation. Par conséquent, elle appartient à l'extension fonctionnelle de « Produit_professionnel » et de « Produit »¹⁹ dans la fonction « Fabrication ». La participation à l'extension fonctionnelle de « Produit » n'est pas cohérente car « p » ne vérifie pas le prédicat de participation de cette classe. En tant qu'instance de « Produit », « p » doit en vérifier la contrainte de participation.

Il convient donc d'ajouter une règle « d'héritage du prédicat de participation » et que l'on peut décrire ainsi :

Règle d'héritage du prédicat de participation

Le prédicat de participation d'une classe à une fonction est la conjonction de tous les prédicats de participation à cette même fonction de ses surclasses et du prédicat de participation qui lui est propre. Ceci permet de garantir que les instances d'une classe vérifient le prédicat de participation de toutes ses surclasses.

19. En effet, « p » appartient à l'extension de « Produit » et la fonction « Fabrication » appartient à l'état fonctionnel de « p ».

Dans l'exemple précédent, le fait d'appliquer cette règle fait que « p » ne participe pas à la fonction « Fabrication ». La classe « Produit » introduit une contrainte générique à l'ensemble des produits qui est d'avoir un coût de stockage supérieur à 300 F. Ensuite, la classe « Produit_professionnel » affine cette contrainte en ajoutant le fait que le produit doit en plus avoir été vendu à plus de 2000 unités.

Par ailleurs, bien que sa classe n'y soit pas explicitement enrichie, l'entité « Produit_mixte » intervient dans la fonction « Fabrication »²⁰. Par conséquent, l'extension fonctionnelle de « Produit_mixte » est déterminée par l'ensemble de ses instances qui vérifient les contraintes de « Produit_professionnel », de « Produit_grand_public » et de « Produit ». Ceci se traduit par la conjonction des prédicats de participation, ce qui donne la contrainte qu'un produit mixte doit avoir un coût de stockage supérieur à 300 F et avoir été vendu à plus de 20000 unités pour participer²¹.

3.5.5 Contextes

Les contextes ont pour but de prendre en compte les extensions fonctionnelles des classes. Ils viennent ainsi compléter les plans qui détiennent exclusivement les descriptions. A l'instar de ces derniers, les contextes ne constituent pas une nouvelle construction du modèle « objet ».

3.5.5.1 Contexte de base

Le contexte de base est défini par l'ensemble des descriptions de base auxquelles on associe les extensions des classes du domaine. Le contexte de base détient les classes introduites par le plan de base, ce dernier ne détenant en fait que leur description de base (cf 3.3.3.1).

Il est correct que le contexte de base soit muni des extensions complètes des classes du domaine. En effet, chaque instance de ces classes dispose (au moins) d'une description de base et donc participe au contexte de base.

Définition formelle du contexte de base

Le contexte de base est défini formellement de la façon suivante :

$$C_{\text{base}} = \{(D_{\text{base}}(C), \text{extension}(C)), C \in \mathcal{C}\}$$

3.5.5.2 Contextes fonctionnels

Un contexte fonctionnel est formé par l'ensemble des descriptions fonctionnelles (modalités de fourniture des valeurs des propriétés de base incluses), accès inter-plans et extensions fonctionnelles des classes intervenant dans la fonction correspondante. Il détient ainsi la totalité des descriptions de la fonction nécessaires à ses clients²². Un contexte fonctionnel constitue donc une unité de présentation aux clients comme les schémas-vues étudiés en 2.2.3.3. Il décrit les informations communes et fonctionnelles pertinentes de l'organisation afin que les clients de la fonction correspondante puissent les manipuler.

20. la classe « Produit_mixte » est une sous-classe de « Produit_professionnel » et « Produit_grand_public ».

21. Un produit « grand public » n'est fabriqué que s'il a été au moins vendu à plus de 20000 unités même s'il est également vendu à des professionnels.

22. On parle ici des clients propres à la fonction et non pas d'une autre fonction qui peut également être considérée comme un client interne au travers d'accès inter-plans (cf 3.4.1).

Utilisation des interfaces générées pour les clients d'une fonction

Les clients d'une fonction peuvent utiliser OQL pour exprimer des requêtes sur un contexte fonctionnel. Les formes syntaxiques « interface » d'ODL correspondant, pour l'exploitation, aux descriptions fonctionnelles et aux accès inter-plans de chaque classe leur sont présentées à cette fin.

Remarque

Nous avons vu en 3.3.5 la nécessité d'inclure les interfaces des surclasses d'une classe présente dans un contexte, même si les entités relatives à ces surclasses n'interviennent pas dans ledit contexte. Ces interfaces ont pour seul objectif la présentation de l'ensemble des propriétés des instances de cette classe. L'inclusion de ces interfaces ne signifie donc pas que la manipulation des extensions de ces surclasses soit possible. Cet aspect ne peut malheureusement pas être explicitement indiqué en ODL.

Exemple

L'annexe B décrit l'ensemble des contextes fonctionnels de notre exemple, tels qu'ils peuvent être exploités. Ces descriptions peuvent servir de support pour les clients des différentes fonctions pour l'expression de requêtes écrites en OQL.

Précision relative au prédicat de participation

Dans un prédicat de participation, le nom de l'extension d'une classe désigne son extension fonctionnelle et non son extension complète. Pour compléter les exemples donnés en 3.5.1, le prédicat suivant utilisé pour sélectionner les instances de « Produit » dans la fonction « Fabrication » est incorrect :

for all produit in Produit: produit.cout_stockage > 300

En effet, pour connaître le résultat du prédicat, il faut connaître l'extension fonctionnelle de « Produit » dans la fonction « Fabrication ». Or c'est ce que le prédicat cherche à déterminer. De plus, notons qu'il s'agirait, de toute façon, d'un prédicat dont la valeur serait indépendante de chaque produit, pris un à un.

Définition formelle d'un contexte fonctionnel

Un contexte fonctionnel est défini formellement de la façon suivante :

$$CF_f (f \in \mathcal{F} - \{\text{base}\}) = \{(DF_f(C), WB_f(C), Acc_f(C), EF_f(C)), C \in \mathcal{C} \wedge (PF_f(C) \neq \emptyset)\}$$

3.5.5.3 Contextes d'instanciation d'une classe

Pour une classe du domaine, il existe un ou plusieurs contextes dont les clients peuvent instancier cette classe ainsi que détruire ses instances. Les contextes d'instanciation d'une classe correspondent aux fonctions fournisseuses des instances de cette classe. Les autres fonctions où cette classe est enrichie consomment et exploitent ces instances (ou un sous-ensemble).

Les contextes d'instanciation d'une classe font nécessairement participer toutes les instances de celle-ci. En effet, d'une part, ils font au moins participer les instances qu'ils fournissent et qui leur sont, par essence, pertinentes. D'autre part, un mécanisme permettant dans un contexte d'instanciation de filtrer les instances fournies par les autres contextes d'instanciation nécessiterait des moyens de sélection plus complexes que l'utilisation d'un prédicat. Ceci signifie donc que, dès qu'une classe est instanciée par l'un de ses contextes, la nouvelle instance participe à l'ensemble des contextes d'instanciation. Tout contexte

d'instanciation d'une classe a donc pour prédicat de participation une tautologie.

Forme syntaxique utilisée pour spécifier les contextes d'instanciation d'une classe

La forme syntaxique employée pour les incréments d'interface est augmentée de la clause **instanciation function** pour spécifier un contexte d'instanciation. Une classe ne détenant pas de contexte d'instanciation est une classe abstraite.

Exemple

Le contexte fonctionnel « Stock » est un contexte d'instanciation pour la classe « Produit_grand_public ». La clause **instanciation function** est donc spécifiée au sein de l'incrément d'interface de cette classe comme l'illustre la figure 3.20 où est également montrée l'incrément d'interface de sa surclasse « Produit » dans le même contexte fonctionnel. Remarquons qu'aucun prédicat de participation n'est explicitement défini ni pour la classe « Produit_grand_public » ni pour sa surclasse « Produit ».

```
function Stock interface increment Produit
{
  access function Commercial:
  {
    relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
    relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
  }
  write base part:
  {
    attribute string designation;
    relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
  }
  function part:
  {
    attribute integer cout_achat;
    attribute integer cout_stockage;
    attribute integer cout_livraison;
    attribute integer quantite_stockee;
    relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
  }
}
function Stock interface increment Produit_grand_public
{
  instanciation function;
  function part:
  {
    attribute string reglementation_stockage;
  }
}
```

*Figure 3.20. Le contexte fonctionnel « Stock » est spécifié comme contexte d'instanciation de la classe « Produit_grand_public » par la clause **instanciation function***

Les classes « Part_Externe » et « Produit » sont deux classes abstraites. Il est possible de vérifier à l'annexe A qu'elles ne détiennent pas de contextes d'instanciation.

Remarques

- Un contexte d'instanciation d'une classe est aussi un contexte d'instanciation pour ses sous-classes. En effet, il est cohérent que les clients d'un contexte qui peuvent instancier une classe puissent aussi instancier ses sous-classes pour créer toutes les instances possibles de cette classe. Par conséquent, il n'est pas possible d'introduire pour ces sous-classes de contraintes de participation pour ce contexte. Le contexte fonctionnel « Stock » est aussi un contexte d'instanciation pour la classe « Produit_mixte », sous-classe des classes « Produit_grand_public » et de « .Produit_professionnel » pour lesquelles le contexte fonctionnel « Stock » est un contexte d'instanciation.

- Il n'existe pas en ODL de possibilité au niveau d'une interface de spécifier le fait que la classe relative à cette interface puisse ou non être instanciée. Par conséquent, dans les interfaces générées pour les clients d'une fonction, nous ne pouvons pas indiquer explicitement les classes qu'ils peuvent ou non instancier.

3.5.6 Dépendance entre contextes fonctionnels due aux accès inter-plans

3.5.6.1 Définition et conséquence sur les contraintes de participation

Nous introduisons ici la notion de dépendance entre contextes relativement à une classe du domaine due à l'existence d'accès inter-plans au sein de cette classe.

Les accès inter-plans au sein d'une classe créent une dépendance entre les contextes correspondants pour cette classe. En effet, pour pouvoir détenir les informations issues d'un accès inter-plans dans son contexte consommateur, une instance doit participer au contexte fournisseur de cet accès. La participation d'une instance au contexte consommateur d'un accès inter-plans nécessite la participation de cette instance au contexte fournisseur de cet accès. Par conséquent, pour une classe du domaine, tout contexte consommateur d'un accès inter-plans dépend du contexte fournisseur de cet accès inter-plans²³.

Cette dépendance entre contextes provoquée par les accès inter-plans crée de nouvelles contraintes de participation dans chaque contexte consommateur. Ces contraintes viennent s'ajouter au prédicat de participation. Elles sont prises en compte dans l'évaluation des extensions fonctionnelles.

Remarque

Si un contexte est dépendant par un accès inter-plans d'un autre pour une classe, alors il l'est également pour toutes ses sous-classes. Ceci est dû à l'héritage de cet accès inter-plans dans les sous-classes.

Prise en compte des dépendances dues aux accès inter-plans dans la définition formelle de l'état fonctionnel

L'état fonctionnel d'une instance est maintenant défini par la fonction suivante :

23. voir aussi [Falquet88] pour les dépendances inter-contextes

$\mathcal{EF}$: $O \rightarrow 2^{\mathcal{F}-\{\mathbf{base}\}}$
 $o \rightarrow \{f \in \mathcal{F}-\{\mathbf{base}\} \text{ telle que}$

- $PF_f(\text{classe}(o)) \neq \emptyset$ et
- $Pred_f(\text{classe}(o))$ soit vérifié pour o et
- $\forall ff \in \mathcal{F}-\{\mathbf{base}\}$ telle que

 $\exists a \in PF_{ff}(\text{classe}(o)) \text{ avec } (a, ff) \in Acc_f(\text{classe}(o)), ff \in \mathcal{EF}(o)\}$

Cette définition apporte la certitude que les accès inter-plans puissent être correctement exploités.

Exemple

La figure 3.21 donne les accès inter-plans vers le contexte « Fabrication » et le prédicat de participation de la classe « Produit » dans le contexte « SAV ». Elle donne aussi, pour rappel, le prédicat de participation de la classe « Produit » dans le contexte « Fabrication ».

L'accès inter-plans relatif à l'attribut « cout_fabrication » fait que seuls les produits participant au contexte « Fabrication » participent au contexte « SAV ». Pour qu'un produit participe au contexte « SAV », il faut donc que son coût de stockage soit supérieur à 300 F et que son prix soit supérieur à 1000 F. Les autres accès inter-plans, non spécifiés dans la figure 3.21, ont pour contextes fournisseurs, des contextes dans lesquels toutes les instances de « Produit » participent.

Notons que cette dépendance est justifiée dans le cadre de notre exemple par le fait que seuls les produits fabriqués sont pris en charge par la fonction « SAV ».

```
function Fabrication interface increment Produit
{
...
function extent condition: cout_stockage > 300;
function part:
{
attribute integer cout_fabrication;
}
}

function SAV interface increment Produit
{
access function Fabrication:
{
attribute integer cout_fabrication;
}
...
function extent condition: prix_tarif > 1000;
function part:
{
attribute integer cout_reparation;
attribute integer duree_garantie;
}
}
```

Figure 3.21. Dépendance du contexte « SAV » vis-à-vis du contexte « Fabrication » pour la classe « Produit » due à l'accès inter-plans relatif au coût de fabrication

Par ailleurs, ces nouvelles contraintes de participation ne posent pas de problème de cohérence au sein des extensions fonctionnelles comme nous avons pu le voir pour le prédicat de participation (cf 3.5.4). En effet, les accès inter-plans d'une classe sont hérités localement dans ses sous-classes. Par conséquent, en vérifiant les contraintes de participation relatives aux accès inter-plans (pour un contexte) de sa classe d'instanciation, une instance vérifie les contraintes de participation relatives aux accès inter-plans (pour le même contexte) de ses surclasses.

Les classes « Produit_grand_public », « Produit_professionnel » et « Produit_mixte » héritent dans le contexte « SAV » de l'accès inter-plans relatif à l'attribut « cout_fabrication ». Par conséquent, leurs instances qui participent au contexte « SAV » participent également au contexte « Fabrication ». Donc toutes les instances appartenant à l'extension fonctionnelle de la classe « Produit » dans le contexte « SAV » participent au contexte « Fabrication », même si elles sont instances directes d'une sous-classe de « Produit ».

Remarque concernant les contextes d'instanciation

L'extension fonctionnelle d'une classe doit être égale à son extension dans ses contextes d'instanciation (cf 3.5.5.3). S'il existe des accès inter-plans pour cette classe dans un contexte d'instanciation, ils doivent avoir pour contextes fournisseurs, des contextes qui vérifient aussi l'égalité entre extension fonctionnelle et extension, c'est-à-dire qui n'introduisent pas de contraintes de participation.

Le contexte « Stock » est un contexte d'instanciation pour la classe « Produit_grand_public » (cf 3.5.5.3). Pour cette classe, il n'existe pas d'accès inter-plans dans le contexte « Stock » autres que ceux hérités de « Produit ». Il faut donc s'intéresser aux accès inter-plans de « Produit » pour déterminer si ceux provoquent une contrainte de participation des instances de la classe « Produit_grand_public ».

Pour la classe « Produit », il existe des accès inter-plans vers le contexte commercial (cf figure 3.22).

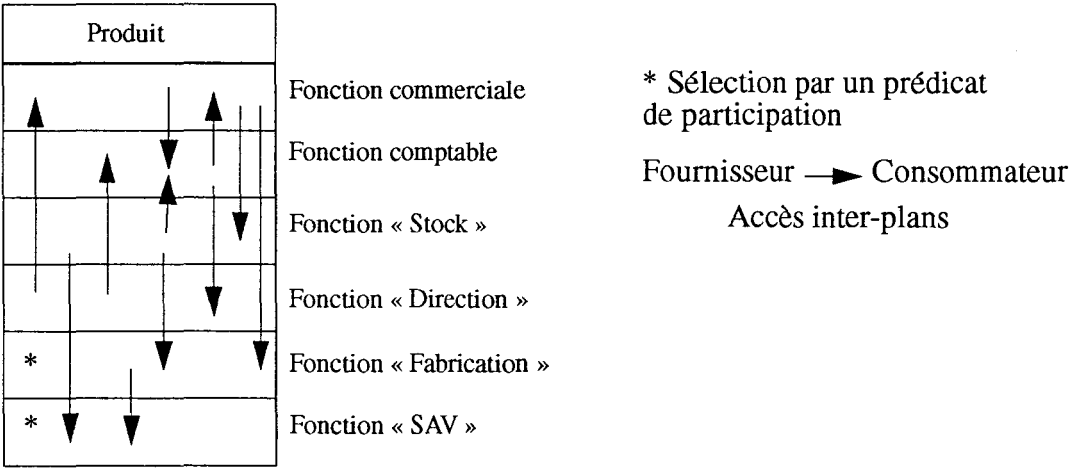


Figure 3.22. Accès inter-plans et prédicats de sélection au sein de la classe « Produit »

Le contexte commercial ne fait pas de sélection par un prédicat et introduit des accès inter-plans vers les contextes comptable et « Direction ». Ces derniers ne font pas non plus de sélection par un prédicat et n'introduisent pas de nouveaux contextes fournisseurs d'accès

inter-plans. Par conséquent, il n'existe pas dans le contexte « Stock » de contraintes de participation pour la classe « Produit » et il en est donc de même pour la classe « Produit_grand_public ».

3.5.6.2 Restriction sur les prédicats de participation due à la dépendance provoquée par les accès inter-plans

Le prédicat de participation des instances d'une classe à un contexte ne peut comporter de propriétés dont il détermine la valeur ou l'existence (cf 3.5.1). La dépendance due aux accès inter-plans va apporter une restriction supplémentaire à l'écriture des prédicats de participation. Cette restriction est qu'un prédicat de participation ne peut accéder à des propriétés d'une partie fonctionnelle relative à un contexte dépendant par d'autres accès inter-plans du contexte du prédicat de participation.

Le prédicat de participation de la classe « Produit » au contexte « Fabrication » contient un attribut de la partie fonctionnelle relative au contexte « Stock », en l'occurrence « cout_stockage » comme l'illustre la figure 3.23. La figure 3.22 nous montre qu'il n'existe pas d'accès inter-plans tel que le contexte « Stock » accède à une propriété de la partie fonctionnelle relative au contexte « Fabrication ». De ce fait, le prédicat de participation de la classe « Produit » à la fonction « Fabrication » est correct. Si un tel accès avait existé, le prédicat de participation n'aurait pas été correct. En effet, l'attribut « cout_stockage » n'aurait alors existé pour une instance que si celle-ci avait participé au contexte « Stock » et donc, du fait de l'accès inter-plans, au contexte « Fabrication ». Le prédicat aurait donc été basé sur un attribut dont il détermine l'existence.

```
function Fabrication interface increment Produit
{
...
access function Stock:
{
attribute integer cout_stockage;
...
}
function extent condition: cout_stockage > 300;
...
}
```

Figure 3.23. Prédicat de participation des instances de la classe « Produit » à la fonction « Fabrication » basé sur un accès inter-plans vers le contexte « Stock »

3.5.7 Préservation de la structuration en classes du contexte de base dans les contextes fonctionnels

Nous avons vu que les parties fonctionnelles d'une classe sont héritées, au sein du plan correspondant, dans ses sous-classes. Ainsi la structuration en classes du plan et du contexte de base est préservée au sein des plans fonctionnels.

Au sein d'un contexte fonctionnel, il convient de remarquer que :

- la description fonctionnelle d'une classe du domaine est héritée dans ses sous-classes ;
- les modalités de fourniture des valeurs des propriétés de la description de base d'une classe

du domaine sont héritées dans ses sous-classes ;

- les accès inter-plans d'une classe du domaine sont hérités dans ses sous-classes ;
- enfin, l'extension fonctionnelle d'une classe du domaine est incluse dans l'extension fonctionnelle de ses surclasses.

Les trois dernières propriétés ont déjà été établies. La première propriété s'obtient facilement en considérant qu'une description fonctionnelle est l'union de la description de base et de la partie fonctionnelle correspondante. Ces deux dernières étant héritées dans les sous-classes d'une classe, respectivement au sein du plan de base et du plan fonctionnel, leur union est donc héritée dans ces sous-classes.

Ainsi, la structuration en classes du plan et du contexte de base est également préservée au sein des contextes fonctionnels.

3.6. Contraintes d'intégrité

Nous nous proposons d'étudier les contraintes d'intégrité dans le cadre de CROME. Les contraintes d'intégrité sont utilisées pour assurer la cohérence de l'état des instances. En effet, les propriétés de ces instances ne sont pas encapsulées et la cohérence de l'état n'est pas garantie par des méthodes d'accès. Toute transaction conduisant à un état de la base violant une ou plusieurs contraintes est rejetée. Pour éviter ce rejet, un traitement compensatoire peut être enclenché avant la fin de la transaction. Le traitement compensatoire doit alors effectuer les mises à jour nécessaires pour revenir à un état cohérent vérifiant les contraintes d'intégrité.

Dans les systèmes de bases de données, les contraintes d'intégrité sont traditionnellement exprimées dans le schéma global et sont vérifiées pour toutes les instances de la base de données, c'est-à-dire pour toutes les instances de l'extension de chaque classe. Il n'est pas possible de définir de contraintes locales à un schéma-vue de la base qui ne s'appliqueraient qu'aux instances des extensions des classes-vues incluses dans ce schéma²⁴. Par ailleurs, un client d'un schéma-vue ne peut appréhender l'ensemble des contraintes d'intégrité exprimées dans le schéma global. En effet, il ne détient au travers du schéma-vue qu'une partie des descriptions contenues dans le schéma global.

La structuration en contextes d'une base de données en CROME nous incite à proposer de prendre en compte les contraintes à un niveau local au sein des contextes fonctionnels. Une telle contrainte est exprimée sur l'ensemble des descriptions présentes dans le contexte où elle est localisée. Par conséquent, elle peut être appréhendée par les clients de ce contexte. Par ailleurs, la localisation d'une contrainte dans un contexte fait que celle-ci ne s'applique qu'aux instances participant à ce contexte.

3.6.1 Type de contraintes étudiées

Dans le cadre des bases de données à objets, il existe de nombreuses formes de contraintes d'intégrité (cf [Bauzer Medeiros91, Bouaziz95, Delannoy94, Morejon94, Nassif91]). Cependant, les contraintes, que l'on rencontre le plus fréquemment et auxquelles nous nous limitons, sont :

- les contraintes d'intégrité référentielle (CIR),
- les contraintes de cardinalité,
- les contraintes d'intégrité statiques sur les propriétés des instances.

Parmi les contraintes étudiées dans le cadre des bases de données à objets que nous n'abordons pas, citons les contraintes d'intégrité dynamiques (cf [Delannoy94]) dont l'étude nécessiterait l'introduction d'une logique temporelle et les contraintes relatives à la relation d'agrégation (cf [Bertino93-p36]).

Les contraintes d'intégrité référentielle sont directement prises en compte dans le modèle de l'ODMG. A la différence des deux autres, elles n'ont pas à être spécifiées explicitement. Les contraintes de cardinalité s'appliquent aux extrémités des relations qui sont des entités et peuvent donc être exprimées au sein des classes correspondantes. Ces deux types de contraintes sont propres aux relations. Elles doivent donc être vérifiées au sein de tous les contextes qui définissent et exploitent des relations.

24. Les extensions des classes-vues sont souvent des sous-ensembles des classes du schéma de base obtenus par l'opérateur de sélection.

Des relations pouvant être définies dans le contexte de base, il y a donc aussi lieu de vérifier les contraintes d'intégrité référentielle et de cardinalité au sein de ce contexte. Par ailleurs, les relations peuvent être partagées entre les contextes, soit par héritage si elles sont définies dans le contexte de base, soit par accès inter-plans si elles sont définies dans l'un des contextes fonctionnels. Ce partage est accompagné du partage des contraintes d'intégrité référentielle et de cardinalité de la relation²⁵ dans les contextes consommateurs. Ces contraintes doivent donc également être vérifiées dans chaque contexte consommateur mais avec des extensions fonctionnelles qui peuvent être différentes de celles du contexte introduisant la relation. Ainsi, ce partage donne à ces contraintes une portée qui dépasse celle du contexte d'introduction.

Pour les contraintes sur les propriétés, nous nous limitons aux contraintes exprimables au sein des classes et qui s'appliquent individuellement à leurs instances. En effet, nous nous situons dans le cadre du modèle « objet » qui prône la centralisation des descriptions au sein des classes et nous voulons respecter cette centralisation pour l'expression des contraintes. Cette restriction interdit l'écriture d'une contrainte inter-entités s'il n'existe pas une relation entre elles qui permette l'expression de la contrainte dans l'une des classes des entités. Par ailleurs, nous ne prenons en compte que les contraintes sur les propriétés définies et partagées au sein d'un contexte fonctionnel. Ainsi, elles ne s'appliquent qu'aux instances participant à ce contexte et qui sont enrichies de la partie fonctionnelle correspondante. Ces contraintes peuvent donc s'appliquer aux propriétés introduites par ce contexte.

Le partage des descriptions du plan de base et les accès inter-plans permettent la définition de contraintes sur des propriétés d'autres contextes qui peuvent ainsi assurer une cohérence globale de la base. De telles contraintes ne peuvent s'appuyer que sur les accès inter-plans explicitement définis²⁶ dans un contexte. Enfin, il est intéressant de localiser les contraintes dans les contextes fournisseurs des valeurs d'une ou plusieurs propriétés présentes dans ces contraintes. En effet, ces contextes sont les seuls à pouvoir provoquer une violation de telles contraintes.

Nous verrons qu'il est possible d'exprimer les contraintes statiques sur les propriétés à l'aide de requêtes exprimées en OQL comme c'est le cas pour les prédicats de participation.

3.6.2 Contraintes d'intégrité référentielle

Le modèle « objet » de l'ODMG supporte la gestion de l'intégrité référentielle pour les relations inter-entités comme nous l'avons expliqué en 3.2.1.1. Comme nous avons retenu ce modèle, nous bénéficions ainsi de cette gestion qui permet d'assurer que les contraintes d'intégrité référentielle relatives à l'ensemble des relations inter-entités soient vérifiées. Cette gestion consiste, lors de la suppression d'une instance, à supprimer toutes les occurrences des relations ayant cette instance pour extrémité. Il ne peut ainsi exister de références vers des instances inexistantes dans la base de données. Cependant, il convient d'étudier plus finement cette gestion dans le cadre de CROME. En CROME, en plus du cas de la suppression d'instance, nous devons aussi prendre en compte le fait qu'une partie seulement des instances de la base peuvent participer à un contexte fonctionnel.

25. Ces contraintes font partie de la sémantique de la relation, ce qui justifie leur partage.

26. pour la raison que nous avons citée ci-dessus, à savoir que les clients d'un contexte doivent pouvoir appréhender ses contraintes.

3.6.2.1 Contraintes d'intégrité référentielle au sein d'un contexte

Dans un premier temps, nous nous intéressons à la gestion des CIR au sein d'un contexte et correspondant à des relations inter-entités définies dans ce contexte. Nous analysons les différences causes qui peuvent produire la violation des CIR et proposons des solutions pour assurer leur respect. Ces solutions incluent la solution du modèle « objet » de l'ODMG.

3.6.2.1.1 Intégrité référentielle au sein du contexte de base

Le contexte de base détient les extensions complètes des classes du domaine. Seule une suppression d'une instance de l'une de ces classes peut provoquer la violation d'une ou plusieurs CIR relatives à des relations définies dans le plan de base. Le traitement compensatoire consiste à supprimer les occurrences des relations dont cette instance est l'une des extrémités. Ce traitement permet de respecter à nouveau les CIR au sein du contexte de base. La solution du modèle « objet » de l'ODMG s'applique donc totalement pour le contexte de base.

Exemple

Prenons le cas de la relation du plan de base « a_acheté/a_été_acheté_par » entre les entités « Client » et « Produit » (cf figure 3.6 à la page 3.77). Supposons qu'il existe un client « c » ayant acheté un produit « p ». Supposons ensuite que cette instance « p » soit supprimée. L'occurrence entre « c » et « p » est alors également supprimée afin d'assurer l'intégrité référentielle.

Définition formelle de l'extension d'une relation

Nous définissons l'extension d'une relation par l'ensemble de ses occurrences. L'extension d'une relation est un sous-ensemble du produit cartésien de l'extension des classes situées à ses extrémités :

extension: $\mathcal{R} \rightarrow 2^{O \times O}$

$R \rightarrow \{(o_1, o_1'), (o_1, o_2'), (o_2, o_3'), \dots, (o_n, o_m')\}$ telle que

- $\{o_1, o_2, \dots, o_n\} \subseteq \text{extension}(C)$
- $\{o_1', o_2', o_3', \dots, o_m'\} \subseteq \text{extension}(C')$

Par conséquent :

$\text{extension}(R) \subseteq \text{extension}(C) \times \text{extension}(C')$

Cette inclusion garantit l'intégrité référentielle, traduit ici par le fait que tout couple de la relation appartient au produit cartésien des extensions des deux classes.

Notons bien que la notion d'extension de relation n'est utilisée ici que pour la formalisation. L'utilisateur, client d'une fonction, ne manipule explicitement que les extensions fonctionnelles des classes. Les relations sont manipulées notamment à l'aide d'expressions de chemin en OQL.

Vérification de l'intégrité référentielle pour une relation du plan de base

Soit $R \in \mathcal{R}$ une relation du plan de base représentée par le couple $((C, a), (C', b))$ tel que $(C, C') \in \mathcal{C}^2$ et $(a, b) \in D_{\text{base}}(C) \times D_{\text{base}}(C')$

L'intégrité référentielle doit être à nouveau vérifiée après une opération dont l'action est de supprimer une instance o de C pour laquelle il existe des occurrences de R l'ayant pour extrémité.

Nous préfixons par « old » un ensemble ou une propriété pour désigner sa valeur avant la suppression. De même, nous préfixons par « new » la valeur d'un ensemble ou d'une propriété après le traitement compensatoire enclenché par l'opération provoquant la violation de la contrainte.

Soit $o \in \text{extension}(C)$. La suppression de o a pour conséquence :
 $\text{new extension}(C) = \text{old extension}(C) - \{o\}$

La gestion de l'intégrité référentielle au sein du contexte de base provoque la suppression de toutes les occurrences de R ayant o pour extrémité :

$\text{new extension}(R) = \text{old extension}(R) - \{(u,v) \in \text{old extension}(R) \text{ tel que } u = o\}$

Enfin, l'extension de C' reste inchangée :

$\text{new extension}(C') = \text{old extension}(C')$

L'inclusion suivante est donc vérifiée et par conséquent, l'intégrité référentielle de R :

$\text{new extension}(R) \subseteq \text{new extension}(C) \times \text{new extension}(C')$

3.6.2.1.2 Intégrité référentielle au sein d'un contexte fonctionnel

Au sein d'un contexte fonctionnel, l'intégrité référentielle pour une relation locale consiste à vérifier l'absence d'occurrences de cette relation dont l'une des extrémités serait inexistante ou ne participerait pas à ce contexte. Il s'agit donc de vérifier que les extrémités de chaque occurrence appartiennent à l'extension fonctionnelle de leur classe. L'intégrité fonctionnelle peut être violée dans le cas d'une opération dont la conséquence est la suppression ou le retrait de participation d'une instance au contexte. Il faut alors un traitement compensatoire consistant à supprimer les occurrences des relations ayant cette instance comme extrémité. Ce traitement compensatoire permet de garantir que les CIR soient à nouveau respectées au sein du contexte à la suite de l'opération provoquant leur violation.

Exemple

Prenons le cas de la relation « est_stocke_dans/stocke » entre les entités « Produit » et « Entrepôt » et introduite dans le plan fonctionnel « Stock » (cf figure 3.24). Supposons qu'il existe un produit « p » stocké dans un entrepôt « e ». Supposons ensuite que « p » soit supprimée. L'occurrence entre « p » et « e » est alors également supprimée afin d'assurer l'intégrité référentielle au sein du contexte fonctionnel « Stock ».

```

function Stock interface increment Produit
{
...
function part:
{
...
relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
}
}

function Stock interface increment Entrepot
{
...
function part:
{
relationship set<Produit> stocke inverse Produit::est_stocke_dans;
}
}

```

*Figure 3.24. Relation « est_stocké_dans/stocke » locale au plan fonctionnel
« Stock » impliquant un respect des CIR au sein du contexte « Stock ».*

Vérification de l'intégrité référentielle pour une relation d'un plan fonctionnel

Soit $R \in \mathcal{K}$ une relation introduite dans un plan fonctionnel P_f ($f \in \mathcal{F} - \{\text{base}\}$) représentée par le couple $((C,a),(C',b))$ tel que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in PF_f(C) \times PF_f(C')$

R vérifie au sein du contexte fonctionnel CF_f l'inclusion suivante :

$$\text{extension}(R) \subseteq EF_f(C) \times EF_f(C')$$

Cette inclusion garantit l'intégrité référentielle au sein de CF_f , traduite ici par le fait que tout couple de la relation appartient au produit cartésien des extensions fonctionnelles des deux classes.

Soit $o \in EF_f(C)$. La suppression de o ou le retrait de participation de o au contexte CF_f a pour conséquence sa suppression de l'extension fonctionnelle de C :

$$\text{new } EF_f(C) = \text{old } EF_f(C) - \{o\}$$

La gestion de l'intégrité référentielle au sein du contexte fonctionnel CF_f provoque la suppression de toutes les occurrences de R ayant o pour extrémité :

$$\text{new extension}(R) = \text{old extension}(R) - \{(u,v) \in \text{old extension}(R) \text{ tel que } u = o\}$$

Enfin, l'extension fonctionnelle de C' reste inchangée :

$$\text{new } EF_f(C') = \text{old } EF_f(C')$$

L'inclusion suivante est par conséquent vérifiée après la suppression, et donc l'intégrité référentielle de R au sein de CF_f :

$$\text{new extension}(R) \subseteq \text{new } EF_f(C) \times \text{new } EF_f(C')$$

3.6.2.2 Intégrité référentielle dans le cas de relations partagées

L'intégrité référentielle d'une relation partagée peut être vérifiée dans le contexte fournisseur de cette relation sans l'être dans le ou les contextes consommateurs. Ceci est dû au fait que les extensions fonctionnelles des classes situées aux extrémités dans les contextes

consommateurs peuvent être distinctes des extensions fonctionnelles des mêmes classes dans les contextes fournisseurs. Le partage peut avoir pour cause l'héritage d'une relation définie dans le plan de base ou un accès inter-plans entre deux contextes. Dans tous les cas, le contexte consommateur ne peut être le contexte de base.

La vérification de l'intégrité référentielle dans les contextes consommateurs peut être mise en œuvre en n'y exploitant que les occurrences dont les deux extrémités participent à ces contextes. Le partage de la relation est alors restreint par les extensions fonctionnelles des extrémités dans chaque contexte consommateur.

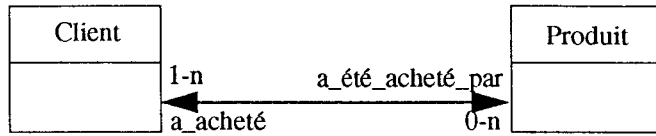
Cette solution basée sur la restriction aux extensions fonctionnelles des extrémités est valable que la relation soit partagée par héritage du plan de base ou par accès inter-plans. La modalité de partage est sans conséquence sur cette solution.

Exemple

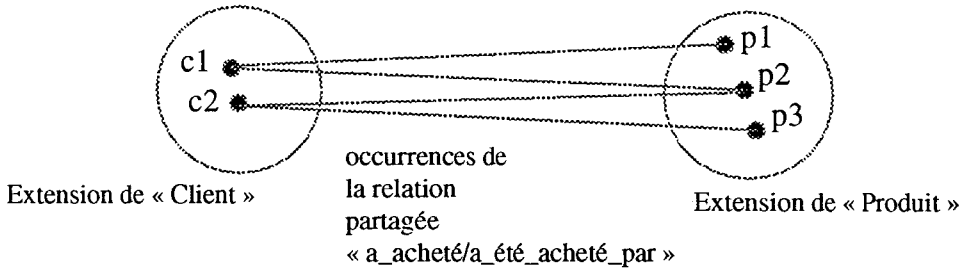
La relation du plan de base « a_acheté/a_été_acheté_par » entre les entités « Client » et « Produit » (cf figure 3.6 à la page 3.77) est partagée, entre autres, dans le contexte « Fabrication ». Le contexte « Fabrication » sélectionne les instances de la classe « Produit » par un prédicat de participation (cf figure 3.25). Les occurrences de la relation « a_acheté/a_été_acheté_par » ayant pour extrémité une instance de « Produit » ne participant pas au contexte « Fabrication » n'y sont pas exploitées. Ceci permet de garantir l'intégrité référentielle de la relation partagée au sein du contexte « Fabrication » comme l'illustre la figure 3.26.

```
function Fabrication interface increment Produit
{
  access function Commercial:
  {
    attribute integer nbr_unites_vendues;
  }
  access function Stock:
  {
    attribute integer cout_stockage;
    relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
  }
  function extent condition: cout_stockage > 300;
  function part:
  {
    attribute integer cout_fabrication;
  }
}
```

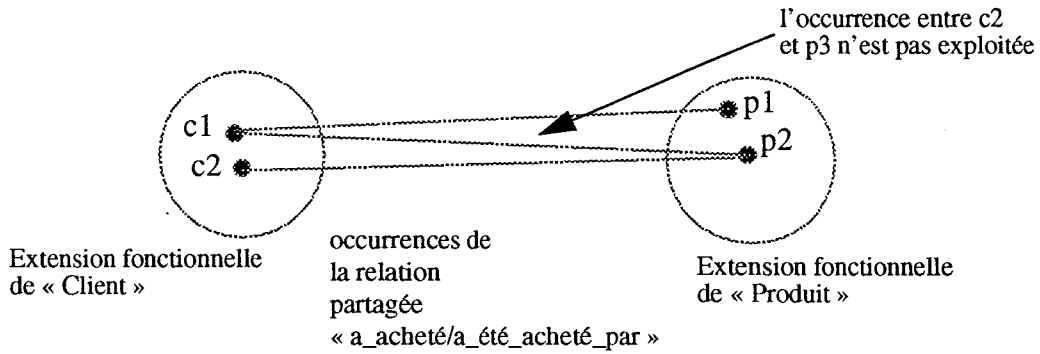
Figure 3.25. incrément de description de la classe « Produit » dans le contexte « Fabrication »



Relation partagée « a_acheté/a_été_acheté_par » dans le plan de base



Extensions des classes « Client » et « Produit »



Extensions fonctionnelles des classes « Client » et « Produit » dans la fonction « Fabrication »

Figure 3.26. Respect de l'intégrité référentielle de la relation partagée « a_acheté/a_été_acheté_par » dans la fonction « Fabrication »

Définition formelle de l'extension fonctionnelle d'une relation

Soit $R \in \mathcal{R}$ une relation représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in \text{intension}(C) \times \text{intension}(C')$. R peut être définie dans le plan de base ou dans un plan fonctionnel.

L'extension fonctionnelle $EF_f(R)$ ($f \in \mathcal{F} - \{\text{base}\}$ telle que le contexte CF_f soit fournisseur ou consommateur de la relation R) est définie par ;

$$EF_f: \quad \mathcal{R} \rightarrow 2^{O \times O} \\ R \rightarrow \text{extension}(R) \cap (EF_f(C) \times EF_f(C'))$$

L'extension fonctionnelle est la restriction de l'extension d'une R dans le contexte CF_f aux seules occurrences dont les deux extrémités participent à CF_f .

Remarque :

Si R est une relation locale au contexte CF_f et y vérifiant l'intégrité référentielle, son extension fonctionnelle dans CF_f est égale à son extension :

$EF_f(R) = \text{extension}(R) \cap (EF_f(C) \times EF_f(C')) = \text{extension}(R)$ car $\text{extension}(R) \subseteq EF_f(C) \times EF_f(C')$, R vérifiant l'intégrité référentielle dans le contexte CF_f .

Utilisation de l'extension fonctionnelle d'une relation partagée pour vérifier son intégrité référentielle dans les contextes consommateurs

Soit $R \in \mathcal{R}$ une relation représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in \text{intension}(C) \times \text{intension}(C')$. R peut être définie dans le plan de base ou dans un plan fonctionnel.

La vérification de l'intégrité référentielle de R dans CF_f ($f \in \mathcal{F} - \{\text{base}\}$) et telle que le contexte CF_f consomme la relation R se fait en considérant l'extension fonctionnelle $EF_f(R)$ à la place de son extension. Ainsi, on a la garantie qu'il n'existe pas d'occurrence de R ayant une extrémité absente de CF_f .

3.6.2.3 Restriction sur les prédicats de participation due au respect des CIR d'une relation partagée

Nous complétons les restrictions sur le prédicat de participation que nous avons déjà explicitées²⁷. Rappelons (cf 3.5.1) que le prédicat de participation des instances d'une classe à une fonction ne peut comporter de propriétés dont il détermine la valeur ou l'existence. Un prédicat de participation d'une classe peut comporter une relation partagée²⁸ à la condition que l'utilisation de celle-ci n'oblige pas à connaître l'extension fonctionnelle de cette classe pour déterminer le résultat du prédicat. L'utilisation d'une expression de chemin dans le prédicat de participation peut commencer mais ne doit donc pas passer ou se terminer par la classe de ce prédicat.

Nous reprenons l'exemple de prédicats de participation possibles pour la classe « Produit » dans le contexte « Fabrication ». Le prédicat suivant est correct. Il comporte la relation partagée « stocke/est_stocké_dans » entre les entités « Entrepôt » et « Produit » et à laquelle le contexte « Fabrication » accède depuis le contexte « Stock » au travers d'un accès inter-plans (cf figure 3.25).

for all entrepot **in** est_stocke_dans: entrepot.distance_usine < 50

Ce prédicat est correct car il n'est pas nécessaire de connaître l'extension fonctionnelle de « Produit » dans le contexte « Fabrication » pour évaluer ce prédicat pour chacune des instances de « Produit ».

Par contre, le prédicat suivant est incorrect car il nécessite la connaissance de l'extension fonctionnelle de « Produit » dans le contexte « Fabrication » pour être évalué.

for all entrepot **in** Entrepot: **for all** produit **in** entrepot.stocke: produit.prix_tarif > 50

En effet, « entrepot.stocke » est ici réduit aux seuls produits appartenant à l'extension fonctionnelle de « Produit » dans le contexte « Fabrication ».

3.6.3 Contraintes de cardinalité

Chaque relation peut déterminer des contraintes de cardinalité sur les classes situées à ses extrémités. Elles expriment le nombre minimal et maximal de participation à la relation

27. en 3.5.1, 3.5.5.2 et 3.5.6.2

28. Il ne peut comporter de relations locales au plan d'une fonction pour laquelle il détermine l'extension fonctionnelle de l'une des classes du domaine.

que chaque instance de ces classes doit vérifier (cf 3.2.2).

3.6.3.1 Contraintes de cardinalité au sein d'un contexte

Dans un premier temps, nous nous intéressons aux contraintes de cardinalité [Morejon94] au sein d'un contexte et correspondant à des relations définies dans ce contexte. Nous étudions les différences causes qui peuvent produire la violation des contraintes de cardinalité et proposons des solutions pour assurer leur respect.

3.6.3.1.1 Contraintes de cardinalité au sein du contexte de base

Les contraintes de cardinalité relatives à une relation du plan de base s'appliquent aux extensions complètes des classes. Les types d'opération qui peuvent être à l'origine d'une violation d'une contrainte de cardinalité sont multiples : ajout ou suppression d'une occurrence de relation entre deux instances ou encore suppression d'une instance (ce dernier cas provoquant la suppression d'une occurrence de relation pour respecter les CIR). La violation peut provoquer soit un rejet de la transaction qui englobe l'opération, soit l'invocation d'un traitement compensatoire. Il n'existe pas de traitement compensatoire automatique à la différence des CIR. L'utilisateur peut être amené à ajouter ou supprimer des occurrences de relation pour faire en sorte que les contraintes de cardinalité soient à nouveau respectées.

La violation est due à une opération issue de l'un des contextes fournisseurs de la relation. Si le choix d'un traitement compensatoire est fait, ce traitement est généralement réalisé dans le même contexte fournisseur.

Exemple

La relation « fournit/est_fourni_par » introduite dans le plan de base et entre les entités « Fournisseur » et « Produit » (cf figure 3.6 à la page 3.77) détermine une cardinalité minimale de un à chaque extrémité. Ainsi toute instance de « Fournisseur » doit être en relation avec au moins une instance de « Produit ». Il en est de même pour chaque instance de « Produit ».

Formalisation des contraintes de cardinalité dans le contexte de base

Soit $R \in \mathcal{R}$ une relation du plan de base représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in D_{\text{base}}(C) \times D_{\text{base}}(C')$.

$\text{card-min}_C(R)$ est la cardinalité minimale de R pour l'extrémité C .

$\text{card-max}_C(R)$ est la cardinalité maximale de R pour l'extrémité C .

$\text{card-min}_{C'}(R)$ est la cardinalité minimale de R pour l'extrémité C' .

$\text{card-max}_{C'}(R)$ est la cardinalité maximale de R pour l'extrémité C' .

La notation $|E|$ désigne le cardinal de l'ensemble E . La vérification des contraintes de cardinalité se formalise par la vérification des inégalités suivantes :

$$\forall o \in \text{extension}(C), \text{card-min}_C(R) \leq |\{(u,v) \in \text{extension}(R) \text{ tel que } u=o\}| \leq \text{card-max}_C(R)$$

$$\forall o' \in \text{extension}(C'), \text{card-min}_{C'}(R) \leq |\{(u,v) \in \text{extension}(R) \text{ tel que } v=o'\}| \leq \text{card-max}_{C'}(R)$$

3.6.3.1.2 Contraintes de cardinalité au sein d'un contexte fonctionnel

Les contraintes de cardinalité relatives à une relation localisée dans un plan fonctionnel s'appliquent aux extensions fonctionnelles des classes. Les types d'opération à l'origine d'une violation d'une contrainte de cardinalité sont les mêmes que dans le cas d'une relation du plan de base auxquels s'ajoute le retrait de participation d'une instance à un contexte fonctionnel (ce dernier cas provoquant la suppression d'une occurrence de relation pour respecter les CIR). Comme dans le cas d'une relation du plan de base, un rejet de la transaction qui englobe

l'opération ou un traitement compensatoire explicite peuvent être réalisés.

La relation étant localisée dans un contexte fonctionnel, les clients de ce contexte en sont les seuls fournisseurs et donc il leur incombe de réaliser le traitement compensatoire.

Exemple

La relation « est_stocke_dans/stocke » entre les entités « Produit » et « Entrepot » introduite dans le plan fonctionnel « Stock » (cf figure 3.24 à la page 3.115) introduit une cardinalité minimale de un pour « Produit ». Dans le contexte fonctionnel « Stock », toute instance de la classe « Produit »²⁹ doit être en relation avec une instance de la classe « Entrepot ». Le non-respect de cette contrainte impose la réalisation d'un traitement compensatoire par les clients du contexte « Stock ».

Formalisation des contraintes de cardinalité d'une relation localisée dans un contexte fonctionnel

Soit $R \in \mathcal{R}$ une relation localisée d'un plan fonctionnel représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in PF_f(C) \times PF_f(C')$.

$\text{card-min}_C(R)$, $\text{card-max}_C(R)$, $\text{card-min}_{C'}(R)$, $\text{card-max}_{C'}(R)$ sont définies comme précédemment.

La vérification des contraintes de cardinalité est formalisée par la vérification des inégalités suivantes :

$$\forall o \in EF_f(C), \text{card-min}_C(R) \leq |\{(u,v) \in \text{extension}(R) \text{ tel que } u=o\}| \leq \text{card-max}_C(R)$$

$$\forall o' \in EF_f(C'), \text{card-min}_{C'}(R) \leq |\{(u,v) \in \text{extension}(R) \text{ tel que } v=o'\}| \leq \text{card-max}_{C'}(R)$$

3.6.3.2 Contraintes de cardinalité de relations partagées

Comme nous l'avons vu en 3.6.2.2 pour l'intégrité référentielle, les contraintes de cardinalité peuvent être respectées dans le contexte où est localisée une relation sans l'être dans ceux qui sont consommateurs de cette relation. De façon plus précise, c'est la cardinalité minimale d'une relation partagée qui peut ne pas être vérifiée dans un contexte consommateur. Ceci est dû au fait qu'un sous-ensemble des occurrences d'une relation participe à un contexte consommateur afin de respecter les CIR.

Nous nous plaçons dans le cas d'une relation dont toutes les contraintes de cardinalité sont vérifiées dans le contexte où elle est localisée. Si, dans un contexte consommateur, le nombre de participation d'une instance d'une classe située à une extrémité de cette relation est inférieur à la cardinalité minimale, il n'est pas possible d'invoquer un traitement compensatoire ou de rejeter la transaction qui a conduit à cette violation. En effet, cette contrainte de cardinalité reste vérifiée dans le contexte où est localisée la relation. Par conséquent, la seule façon d'arriver à la vérification de la contrainte de cardinalité dans le contexte consommateur est d'exclure de celui-ci l'instance qui est la source de la violation.

Notons enfin que cette solution est valable que la relation soit partagée par héritage du plan de base ou par accès inter-plans. La modalité de partage est sans conséquence sur cette solution.

29. c'est-à-dire, toute instance de « Produit » appartenant à l'extension fonctionnelle de « Produit » dans le contexte « Stock »

Exemple

La relation du plan de base « a_acheté/a_été_acheté_par » entre les entités « Client » et « Produit » (cf figure 3.6 à la page 3.77) est partagée, entre autres, dans le contexte « Fabrication ». Cette relation détermine dans le plan de base une cardinalité minimale de un pour « Client ». Le contexte « Fabrication » sélectionne les instances de la classe « Produit » par un prédicat de participation (cf figure 3.25 à la page 3.116). La figure 3.27 montre le cas d'un client (« c2 ») n'ayant acheté que des produits (« p2 » et « p3 ») qui ne participent au contexte « Fabrication ». Pour respecter la contrainte de cardinalité minimale, cette instance de « Client » ne participe pas au contexte « Fabrication ».

La non-participation d'un tel client dans le contexte « Fabrication » est justifiée par le fait que la sémantique d'un client inclut notamment le fait qu'il ait acheté au moins un produit. Un client n'ayant acheté aucun produit pertinent pour la fonction « Fabrication » n'est pas considéré comme un client pertinent pour cette fonction.

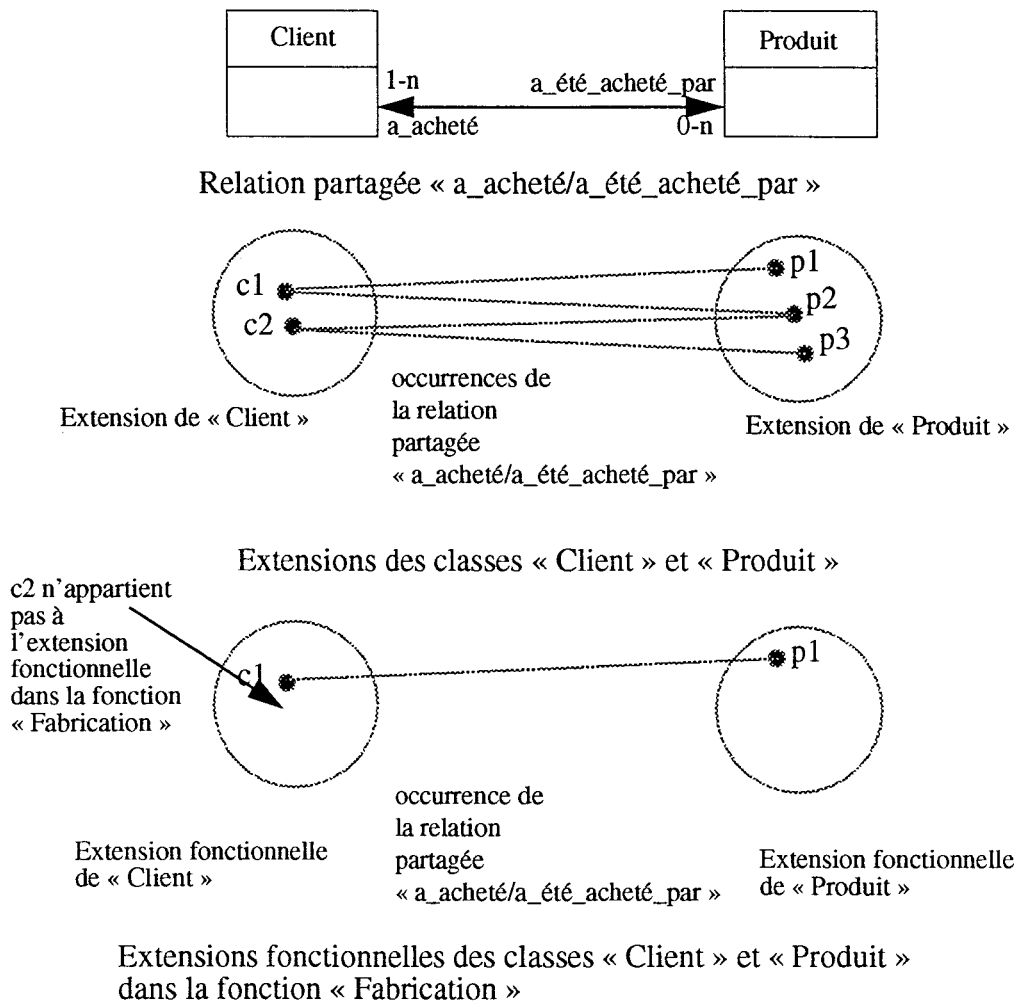


Figure 3.27. Respect des contraintes de cardinalité de la relation partagée « a_acheté/a_été_acheté_par » dans la fonction « Fabrication »

Formalisation des contraintes de cardinalité d'une relation dans un contexte consommateur

Soit $R \in \mathcal{R}$ une relation représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in \text{intension}(C) \times \text{intension}(C')$. R peut être définie dans le plan de base ou dans un plan

fonctionnel.

$\text{card-min}_C(R)$, $\text{card-max}_C(R)$, $\text{card-min}_{C'}(R)$, $\text{card-max}_{C'}(R)$ sont définies comme précédemment.

La vérification des contraintes de cardinalité dans un contexte fonctionnel CF_f ($f \in \mathcal{F} - \{\mathbf{base}\}$) et telle que le contexte CF_f consomme la relation R , se formalise par la vérification des inégalités suivantes :

$$\begin{aligned} \forall o \in EF_f(C), \text{card-min}_C(R) &\leq |\{(u,v) \in EF_f(R) \text{ tel que } u=o\}| \leq \text{card-max}_C(R) \\ \forall o' \in EF_f(C'), \text{card-min}_{C'}(R) &\leq |\{(u,v) \in EF_f(R) \text{ tel que } v=o'\}| \leq \text{card-max}_{C'}(R) \end{aligned}$$

Remarquons que, dans le cas d'une relation définie dans un plan fonctionnel, ces inégalités sont les mêmes que nous avons introduites ci-dessus (en 3.6.3.1.2). En effet, l'extension fonctionnelle d'une relation est égale à son extension au sein du contexte où elle est localisée.

La violation des inégalités pour une instance de $EF_f(C)$ ou $EF_f(C')$ dans CF_f conduit donc à l'exclusion de l'instance concernée de l'extension fonctionnelle. On obtient ainsi une nouvelle contrainte de participation à un contexte fonctionnel pour chaque instance d'une classe du domaine qui est la vérification des contraintes de cardinalité des relations partagées dans ce contexte.

Prise en compte des contraintes de cardinalité des relations partagées dans la définition formelle de l'état fonctionnel

L'état fonctionnel d'une instance est maintenant défini par la fonction suivante :

$$\begin{aligned} \mathcal{EF}: \quad O &\rightarrow 2^{\mathcal{F} - \{\mathbf{base}\}} \\ o &\rightarrow \{f \in \mathcal{F} - \{\mathbf{base}\} \text{ telle que} \\ &\quad \bullet \text{PF}_f(\text{classe}(o)) \neq \emptyset \\ &\quad \bullet \text{Pred}_f(\text{classe}(o)) \text{ soit vérifié pour } o \\ &\quad \bullet \forall ff \in \mathcal{F} - \{\mathbf{base}\} \text{ telle que} \\ &\quad \exists a \in \text{PF}_{ff}(\text{classe}(o)) \text{ tel que } (a, ff) \in \text{Acc}_f(\text{classe}(o)), \\ &\quad ff \in \mathcal{EF}(o) \\ &\quad \bullet \forall R \in \mathcal{R} \text{ représentée par } ((C,a),(C',b)) \text{ telle que } (C,C') \in \mathcal{C}^2 \wedge \\ &\quad \text{classe}(o) \text{ sous-classe de } C \wedge \\ &\quad (a,b) \in D_{\text{base}}(C) \times D_{\text{base}}(C'), \\ &\quad \text{card-min}_C(R) \leq |\{(u,v) \in EF_f(R) \text{ tel que } u = o\}| \\ &\quad \bullet \forall R \in \mathcal{R} \text{ représentée par } ((C,a),(C',b)) \text{ telle que } (C,C') \in \mathcal{C}^2 \wedge \\ &\quad \text{classe}(o) \text{ sous-classe de } C \wedge \\ &\quad \exists ff \in \mathcal{F} - \{f, \mathbf{base}\}, (a,b) \in \text{PF}_{ff}(C) \times \text{PF}_{ff}(C') \wedge \\ &\quad (a, ff) \in \text{Acc}_f(C), (b, ff) \in \text{Acc}_f(C'), \\ &\quad \text{card-min}_{C'}(R) \leq |\{(u,v) \in EF_f(R) \text{ tel que } u = o\}| \\ &\quad \} \end{aligned}$$

Les trois premières conditions ont déjà été explicitées : enrichissement de la classe dans le contexte fonctionnel, vérification par l'instance du prédicat de participation et participation aux contextes fournisseurs des accès inter-plans.

La quatrième condition concerne la vérification de la cardinalité minimale des ensembles d'occurrences des relations héritées du plan de base dont o est l'une des extrémités. Ces

ensembles sont bien sûr restreints aux extensions fonctionnelles des relations dans la fonction f . La cinquième condition est identique à la quatrième mais porte sur les relations partagées par accès inter-plans.

Cette définition présente l'inconvénient d'utiliser l'extension fonctionnelle $EF_f(R)$ de relations R ayant la classe de o comme extrémité pour déterminer si f appartient à $\mathcal{EF}(o)$. En effet, $EF_f(R)$ est définie à l'aide de $EF_f(\text{classe}(o))$ ³⁰, elle-même définie par l'appartenance de f à $\mathcal{EF}(o)$. Il est facile d'écrire à la place :

$\mathcal{EF}$: $O \rightarrow 2^{\mathcal{F}-\{\mathbf{base}\}}$
 $o \rightarrow \{f \in \mathcal{F}-\{\mathbf{base}\} \text{ telle que}$

- $PF_f(\text{classe}(o)) \neq \emptyset$
- $Pred_f(\text{classe}(o))$ soit vérifié pour o
- $\forall ff \in \mathcal{F}-\{\mathbf{base}\} \text{ telle que}$
 $\exists a \in PF_{ff}(\text{classe}(o)) \text{ tel que } (a, ff) \in Acc_f(\text{classe}(o)),$
 $ff \in \mathcal{EF}(o)$
- $\forall R \in \mathcal{R} \text{ représentée par } ((C, a), (C', b)) \text{ telle que } (C, C') \in \mathcal{C}^2 \wedge$
 $\text{classe}(o) \text{ sous-classe de } C \wedge$
 $(a, b) \in D_{\mathbf{base}}(C) \times D_{\mathbf{base}}(C'),$
 $\text{card-min}_C(R) \leq | \{ (u, v) \in \text{extension}(R) \text{ tel que } u = o \wedge f \in \mathcal{EF}(v) \} |$
- $\forall R \in \mathcal{R} \text{ représentée par } ((C, a), (C', b)) \text{ telle que } (C, C') \in \mathcal{C}^2 \wedge$
 $\text{classe}(o) \text{ sous-classe de } C \wedge$
 $\exists ff \in \mathcal{F}-\{f, \mathbf{base}\}, (a, b) \in PF_{ff}(C) \times PF_{ff}(C') \wedge$
 $(a, ff) \in Acc_f(C), (b, ff) \in Acc_f(C'),$
 $\text{card-min}_C(R) \leq | \{ (u, v) \in \text{extension}(R) \text{ tel que } u = o \wedge f \in \mathcal{EF}(v) \} |$

 $\}$

Ces nouvelles contraintes de participation ne posent pas de problème de cohérence au sein des extensions fonctionnelles comme nous avons pu le voir pour le prédicat de participation ce qui nous avait contraint à instaurer une règle d'héritage (cf 3.5.4).

En effet, comme pour les accès inter-plans (cf 3.5.6.1), les extrémités de relations partagées d'une classe sont héritées dans ses sous-classes. S'il s'agit d'une relation du plan de base, cet héritage se fait dans ce plan et est ensuite diffusé dans les plans fonctionnels. S'il s'agit d'une relation entre deux classes partagée au travers d'un accès inter-plans, celle-ci est héritée dans leurs sous-classes par l'héritage des accès inter-plans. Par conséquent, en vérifiant les contraintes de participation relatives aux relations partagées (pour un contexte) de sa classe d'instanciation, une instance vérifie les contraintes de participation relatives aux relations partagées (pour le même contexte) de ses surclasses.

Basé sur cette dernière définition de l'état fonctionnel, un algorithme de calcul de l'extension fonctionnelle de chaque classe du domaine dans chaque contexte fonctionnel est présenté à l'annexe D.

Exemple

Comme l'illustre la figure 3.28, les instances de « Produit » sont sélectionnées dans le contexte « SAV » par le prédicat de participation relatif à ce contexte mais aussi par celui du

30. ou de $EF_f(C)$ avec C surclasse de $\text{classe}(o)$

contexte « Fabrication » du fait de l'existence d'un accès inter-plans dont le contexte fournisseur est « Fabrication » (cf 3.5.6.1). Il existe d'autres contextes fournisseurs d'accès inter-plans mais ceux-ci ne sélectionnent pas les instances de « Produit ».

```

interface Fournisseur: Part_externe
{
    relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
}
interface Produit
{
    attribute string designation;
    attribute integer prix_tarif;
    relationship set<Client> a_ete_achete_par inverse Client::a_achete;
    relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
}

function Fabrication interface increment Produit
{
    ...
    function extent condition: cout_stockage > 300;
    function part:
    {
        attribute integer cout_fabrication;
    }
}

function SAV interface increment Produit
{
    access function Fabrication:
    {
        attribute integer cout_fabrication;
    }
    ...
    function extent condition: prix_tarif > 1000;
    function part:
    {
        attribute integer cout_reparation;
        attribute integer duree_garantie;
    }
}

function SAV interface increment Fournisseur
{
    access function Fabrication:
    {
        attribute boolean fournit_seulement_lot_complet_de_pieces;
    }
    function extent condition: not fournit_seulement_lot_complet_de_pieces;
    function part:
    {
        attribute integer supplement_pieces_detachees;
    }
}

```

Figure 3.28. Ensemble des descriptions où sont localisées les caractéristiques provoquant une sélection des instances de « Produit » dans le contexte « SAV »

Dans le contexte « SAV », seuls les fournisseurs livrant des pièces détachées à l'unité (et pas uniquement en lot) y participent (cf figure 3.28). Le partage de la relation « fournit/est_fourni_par » introduite dans le plan de base (cf figure 3.28) entre les entités « Fournisseur » et « Produit » va provoquer une nouvelle contrainte de participation pour les

instances de « Produit ». En effet, cette relation détermine une contrainte de cardinalité minimale de un pour « Produit ». Cette contrainte vient s'ajouter à la vérification du prédicat de participation et à la participation au contexte « Fabrication ».

Par conséquent, pour qu'un produit participe au contexte « SAV », il faut donc que son coût de stockage soit supérieur à 300 F, que son prix soit supérieur à 1000 F et qu'au moins un de ses fournisseurs livre des pièces détachées à l'unité.

La relation « fournit/est_fourni_par » détermine également une contrainte de cardinalité minimale de un pour « Fournisseur ». Par conséquent, seuls les fournisseurs livrant au moins un produit participant au contexte « SAV » y participent également.

Remarque concernant les contextes d'instanciation

Nous rappelons que les contextes d'instanciation d'une classe font nécessairement participer toutes les instances de celle-ci (cf 3.5.5.3). Ils ne doivent donc pas instaurer de contraintes de participation sur ces instances.

Chaque contexte d'instanciation d'une classe détenant une ou plusieurs extrémités de relations du plan de base avec une cardinalité minimale supérieure ou égale à un, doit être fournisseur de ces relations. En effet, il doit lors de l'instanciation et de l'initialisation de l'instance, faire en sorte que cette ou ces contraintes de cardinalité minimale soient respectées. Un autre contexte fournisseur de ces relations ne peut pas rendre leur cardinalité minimale respectée car alors, l'instance nouvellement créée n'y participerait pas. Ceci permet aussi de garantir la participation de la nouvelle instance dans le contexte d'instanciation sans qu'elle y soit exclue pour non respect de la cardinalité minimale d'une relation partagée.

D'autre part, nous interdisons la possibilité que le contexte d'instanciation d'une classe soit consommateur au travers d'accès inter-plans d'une relation localisée dans un autre contexte fonctionnel, relation ayant cette classe pour extrémité avec une cardinalité minimale supérieure ou égale à un. En effet, un tel partage créerait une contrainte de participation pour chaque instance nouvellement créée que les clients du contexte d'instance ne pourraient valider, conduisant alors à l'exclusion de cette instance.

Ces deux conditions viennent s'ajouter au fait que le prédicat de participation doit être une tautologie et que, s'il existe des accès inter-plans pour cette classe dans un contexte d'instanciation, ils doivent avoir pour contextes fournisseurs, des contextes qui vérifient aussi l'égalité entre extension fonctionnelle et extension, c'est-à-dire n'introduisant pas de contraintes de participation (cf 3.5.5.3 et 3.5.6.1)

Exemple

Le contexte « Stock » est un contexte d'instanciation pour la classe « Produit_grand_public ». Il respecte les conditions vis-à-vis du prédicat de participation³¹ et des accès inter-plans³². Il n'existe pas pour la classe « Produit_grand_public » de relations partagées autres que celles héritées de la classe « Produit ». Celles-ci sont visibles à la figure 3.29.

La relation du plan de base « fournit/est_fourni_par » entre les entités « Fournisseur » et

31. iAucun prédicat de participation n'est explicitement défini pour pour la classe « Produit_grand_public » ni pour sa surclasse « Produit » (cf 3.5.5.3)

32. Les contextes fournisseurs des accès inter-plans exploités par le contexte « Stock » de la classe « Produit » n'introduisent pas de contraintes de participation (cf 3.5.6.1).

« Produit » détermine une cardinalité minimale de un pour « Produit ». Le contexte « Stock » est fournisseur des valeurs de cette relation (figure 3.29). Par ailleurs, les autres relations partagées dans le contexte « Stock » et dont « Produit » est à l'une des extrémités n'ont pas de cardinalité minimale pour « Produit ». Ceci confirme définitivement que le contexte « Stock » puisse être un contexte d'instanciation pour la classe « Produit_grand_public ».³³

```

interface Produit
{
    attribute string designation;
    attribute integer prix_tarif;
    relationship set<Client> a_ete_achete_par inverse Client::a_achete;
    relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
}

function Stock interface increment Produit
{
    access function Commercial:
    {
        relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
        relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
    }
    write base part:
    {
        attribute string designation;
        relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
    }
    function part:
    {
        ...
    }
}

function Stock interface increment Produit_grand_public
{
    instanciation function;
    function part:
    {
        attribute string reglementation_stockage;
    }
}

```

Figure 3.29. Ensemble des descriptions où sont localisées les relations partagées de « Produit_grand_public » dans le contexte « Stock »

Remarques concernant les relations définies dans le plan de base

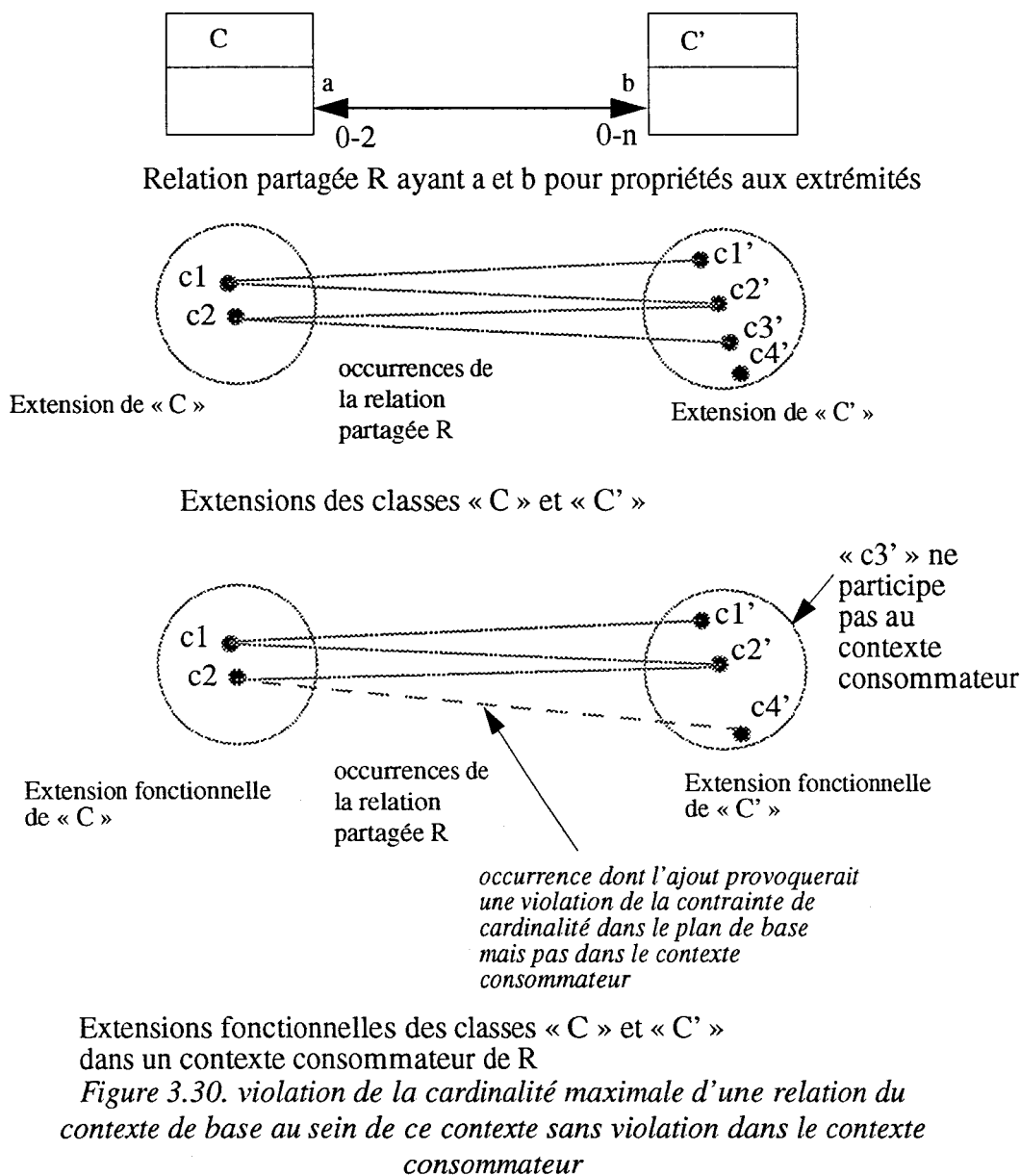
- Dans le cas de la violation d'une contrainte de cardinalité minimale d'une relation dans le contexte de base, seul un rejet de la transaction peut être envisagé. En effet, une relation du plan de base est une relation partagée dans ses contextes fournisseurs. Par conséquent, l'instance dont le nombre de participation à la relation est devenu insuffisant est maintenant exclue du contexte fournisseur. Ceci a pour effet d'interdire un traitement compensatoire.

- Dans le cas de la violation d'une contrainte de cardinalité maximale d'une relation du plan de base par un client d'un contexte fournisseur de cette relation et sélectionnant les

33. Pour les mêmes raisons, le contexte « Stock » peut être un contexte d'instanciation pour la classe « Produit_professionnel ».

instances de la classe correspondant à cette contrainte de cardinalité maximale, il se peut que cette cardinalité maximale reste respectée dans ce contexte à cause de la sélection. Ici la contrainte est vérifiée dans le contexte consommateur sans l'être dans le contexte de base à l'opposé de la situation décrite au début de 3.6.3.2. Cependant, il s'agit d'une violation d'une contrainte de cardinalité au sein du contexte de base (cf 3.6.3.1.1) et qui doit faire l'objet d'un rejet ou d'un traitement compensatoire par les clients du contexte ayant provoqué cette violation.

La figure 3.30 illustre ce cas. La relation « R » introduit une cardinalité maximale de 2 pour la classe « C ». Dans le contexte de base, cette cardinalité maximale est atteinte pour l'instance « c2 » mais n'est pas atteinte pour dans le contexte consommateur de « R » du fait de la non-participation de l'instance « c3' » à ce contexte. Un client du contexte consommateur qui ajouterait une occurrence de « R » entre « c2 » et « c4' » provoquerait une violation de la contrainte de cardinalité maximale au sein du contexte de base sans en provoquer dans le contexte fonctionnel consommateur.



3.6.3.3 Restriction sur les prédicats de participation due au respect des contraintes de cardinalité minimale des relations partagées

Nous finissons de compléter les restrictions sur le prédicat de participation que nous avons déjà explicitées³⁴. Nous rappelons (cf 3.5.1) que le prédicat de participation des instances d'une classe à un contexte ne peut comporter de propriétés dont il détermine la valeur ou l'existence. Nous avons vu que le prédicat de participation d'une classe peut comporter une relation partagée à la condition que l'utilisation de celle-ci n'oblige pas à connaître l'extension fonctionnelle de cette classe pour déterminer le résultat du prédicat (cf 3.6.2.3).

Si le prédicat de participation d'une première classe utilise une seconde classe dont les instances sont sélectionnées du fait d'une relation partagée entre les deux classes³⁵, celui-ci n'est pas correct. En effet, pour déterminer les instances de cette seconde classe³⁶, il faut alors connaître l'extension fonctionnelle de la première classe, ce qui rend le prédicat incorrect.

Nous reprenons l'exemple de prédicats de participation possibles pour la classe « Produit » dans le contexte « Fabrication ». Le prédicat suivant déjà donné en 3.5.1 et en 3.6.2.3 est correct. Il comporte la relation partagée « stocke/est_stocké_dans » entre « Entrepôt » et « Produit » et à laquelle le contexte « Fabrication » accède depuis le contexte « Stock » par un accès inter-plans.

for all entrepot in est_stocke_dans: entrepot.distance_usine < 50

Ce prédicat est correct car les instances de la classe « Entrepôt » ne sont pas sélectionnées dans le contexte « Fabrication » par une relation partagée entre « Entrepôt » et « Produit ». En effet, la seule relation partagée « stocke/est_stocké_dans » entre ces deux classes ne détermine pas de cardinalité minimale pour « Entrepôt ».

Par contre, le prédicat suivant est incorrect

for all fournisseur in est_fourni_par: not fournit_seulement_lot_complet_de_pieces;

En effet, les instances de « Fournisseur » sont sélectionnées dans le contexte « Fabrication » par la relation partagée du plan de base « fournit/est_fourni_par » entre « Fournisseur » et « Produit » qui détermine une cardinalité minimale de un pour « Fournisseur ».

3.6.4 Contraintes d'intégrité statiques sur les propriétés des classes au sein d'un contexte fonctionnel

3.6.4.1 Expression et localisation

Les contraintes d'intégrité statiques sur les propriétés sont exprimées par un prédicat. Ce prédicat est exprimé à l'aide d'une requête OQL dont le résultat est logique (booléen).

La grande majorité des travaux relatifs aux contraintes d'intégrité sur les propriétés est basée sur l'utilisation des prédicats du premier ordre, généralement étendus par des fonctions d'agrégation comme la somme, la moyenne, etc. Nous reprenons cette expression des contraintes sur les propriétés par des prédicats que nous limitons au sein des classes et qui sont

34. en 3.5.1, 3.5.5.2, 3.5.6.2 et 3.6.2.3.

35. dont la cardinalité minimale pour la seconde classe est supérieure à un

36. que ce soit en désignant explicitement l'extension fonctionnelle de la seconde classe ou au travers d'une expression de chemin

appliquées individuellement aux instances. En effet, nous nous situons dans le cadre du modèle « objet » qui prône la centralisation des descriptions au sein des classes. De tels prédicats s'appliquent aux valeurs des attributs et/ou des relations des instances. Comme pour les prédicats de participation, nous utilisons les possibilités d'expression d'un prédicat fournies par OQL en imposant que celui-ci ne soit exprimé que sur les attributs et relations de l'instance. Un prédicat se présente alors sous la forme d'une requête OQL dont le résultat est une valeur logique (booléenne) et dont l'unique point de départ est l'instance à laquelle on veut appliquer la contrainte.

Les contraintes sur les propriétés sont toujours locales aux contextes fonctionnels. Elles s'appliquent aux instances de la classe participant à ce contexte.

Pour chaque contrainte sur les propriétés d'une classe, nous associons un prédicat défini au sein de cette classe et associé à la partie fonctionnelle correspondante. Cette définition au sein de la classe rend compte du fait qu'il s'agit d'une caractéristique de l'entité associée et que cette caractéristique est spécifique à un contexte où intervient cette entité.

Pour respecter la localité à un contexte fonctionnel, un prédicat de contrainte ne peut être exprimé que sur les propriétés de la description fonctionnelle et les accès inter-plans de la classe concernée. Remarquons aussi qu'un prédicat peut porter sur des valeurs de propriétés d'autres classes s'il comporte des expressions de chemin sous-tendues par les relations inter-entités. Ces propriétés doivent être incluses dans les descriptions fonctionnelles et les accès inter-plans de ces autres classes relatives au même contexte.

Les prédicats de contrainte d'une classe relatifs à un contexte fonctionnel sont hérités localement dans ses sous-classes.

Forme syntaxique employée pour définir les prédicats relatifs aux contraintes sur les propriétés

La forme syntaxique utilisée pour définir un incrément d'interface d'une classe est enrichie pour inclure les prédicats relatifs aux contraintes sur les propriétés. La clause **function integrity constraint** permet de spécifier un ou plusieurs prédicats correspondant à des contraintes d'intégrité.

Celle-ci se présente maintenant ainsi :

```
function <function_name> interface increment <type_name>
{
[
access function <function_name>:
{
[<attribute_definition>]*
[<relationship_definition>]*
}
]*
[ function extent condition: <predicate_expression>; ]
[ instantiation function; ]
[
write base part:
{
[<attribute_definition>]*
[<relationship_definition>]*
}
]
function part:
{
[<attribute_definition>]*
[<relationship_definition>]*
}
```

```

    }
  [
    function integrity constraint:
    {
      [<predicate_expression>]*
    }
  ]
}

```

Exemple

La figure 3.31 illustre un exemple de contrainte d'intégrité relative aux instances de la classe « Client » dans le contexte comptable. Tout instance de « Client » doit vérifier dans le contexte comptable que son chiffre d'affaires est supérieur au montant qui lui reste encore à régler.

```

function Comptable interface increment Client
{
  ...
  function part:
  {
    attribute integer encours_a_regler;
    attribute integer chiffre_affaires;
  }
  function integrity constraint:
  {
    encours_a_regler < chiffre_affaires;
  }
}

```

Figure 3.31. Contrainte d'intégrité de la classe « Client » dans la fonction comptable

3.6.4.2 Violation et validité des contraintes sur les propriétés

Les contraintes d'intégrité sur les propriétés peuvent être violées par des opérations qui ont lieu dans d'autres contextes. Il suffit que le prédicat correspondant contienne des accès inter-plans ou des propriétés du plan de base dont les valeurs sont fournies par d'autres contextes. De telles opérations ne violent pas nécessairement les contraintes du contexte où elles ont lieu. Le rejet des transactions englobant ces opérations ne serait pas justifiable vis-à-vis des clients de ce contexte et ne peut donc être envisagé. Un traitement compensatoire dans le contexte où la contrainte est violée est donc l'unique possibilité.

Notons que ce déclenchement d'un traitement compensatoire dans un contexte dû à une opération dans un autre contexte constitue une nouvelle forme d'interaction entre contextes.

Exemple

La figure 3.32 montre la contrainte d'intégrité relative à la classe « Produit » dans le contexte comptable. Cette contrainte impose que le chiffre d'affaires d'un produit soit supérieur au produit du nombre d'unités vendues par le prix unitaire au tarif et par la remise maximale autorisée. Elle impose aussi que ce chiffre d'affaires soit inférieur au produit du nombre d'unités vendues par le prix unitaire au tarif. Seul le chiffre d'affaires est une propriété du plan comptable. Les valeurs des autres propriétés sont fournies par d'autres contextes. Ainsi, une opération issue par les clients du contexte commercial, augmentant le nombre

d'unités vendues d'un produit et provoquant la violation de cette contrainte ne peut conduire au rejet de sa transaction englobante. Un traitement compensatoire au sein du contexte comptable visant à augmenter le chiffre d'affaires du même produit doit être invoqué.

```

function Comptable interface increment Produit
{
  access function Commercial:
  {
    relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
    relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
    attribute integer nbr_unites_vendues;
  }
  access function Stock:
  {
    attribute integer quantite_stockee;
  }
  access function Direction:
  {
    attribute integer reduction_maximale;
  }
  function part:
  {
    attribute integer chiffre_affaires;
    attribute integer marge_moyenne;
  }
  function integrity constraint:
  {
    (chiffre_affaires >= nbr_unites_vendues*prix_tarif*reduction_maximale) AND
    (chiffre_affaires <= nbr_unites_vendues*prix_tarif);
  }
}

```

Figure 3.32. Contrainte d'intégrité de la classe « Produit » dans la fonction comptable

Cette obligation d'invocation d'un traitement compensatoire crée des restrictions sur l'expression des contraintes d'intégrité. La limitation du prédicat aux accès inter-plans et aux propriétés des descriptions fonctionnelles n'est pas suffisante. Il faut que ce dernier soit écrit de sorte à ce qu'en cas de violation, les clients de la fonction puissent toujours réaliser un traitement compensatoire qui permette de rendre ce prédicat à nouveau vérifié. Un tel prédicat garantit la validité d'une contrainte locale d'intégrité sur les propriétés.

Nous disons qu'une contrainte est valide si et seulement si, en cas de violation, les clients de la fonction peuvent toujours réaliser un traitement compensatoire qui la rende à nouveau vérifiée.

Exemples

- La contrainte « encours_a_regler < chiffre_affaires » relative à la classe « Client » dans le contexte comptable (cf figure 3.31) n'est exprimée que sur des attributs de la partie fonctionnelle comptable de « Client ». Elle est par conséquent valide.
- La contrainte relative à la classe « Produit » dans le contexte comptable et définie à la figure 3.32 est valide comme nous avons pu le montrer ci-dessus. Il est toujours possible pour les clients de la fonction comptable d'ajuster le chiffre d'affaires entre les deux bornes. En

effet, le nombre d'unités vendues et le tarif doivent être strictement positifs et la réduction maximale est comprise entre 0,5 et 1³⁷. De ce fait, la borne inférieure imposée au chiffre d'affaires est nécessairement inférieure à la borne supérieure imposée.

- Si, dans le contexte comptable, le prédicat de contrainte relatif à « Produit » était « `quantite_stockee < nbr_unites_vendues` », il constituerait une contrainte invalide. En effet, les clients de la fonction comptable n'ont aucun moyen de rendre un tel prédicat vérifié.

- Le contexte commercial introduit le prédicat de contrainte relatif à « Produit » suivant (voir annexe A) :

`prix_tarif > 0`

Celui-ci est correct car le contexte commercial fournit les valeurs de « `prix_tarif` » qui est un attribut de la description de base de « Produit ».

- Si le contexte comptable introduisait ce dernier prédicat pour la classe « Produit », celui-ci constituerait une contrainte invalide car le contexte comptable ne fournit pas les valeurs de « `prix_tarif` ».

La validité d'une contrainte dépend de nombreux facteurs : l'utilisation d'accès inter-plans, de propriétés de descriptions de base dont les valeurs ne sont pas fournies par la fonction et des opérateurs employés au sein du prédicat. La détermination des règles qui permettraient d'obtenir la validité d'une contrainte est difficile et mériterait une étude en soi.

3.6.5 Contraintes de participation et contraintes d'intégrité

Nous nous intéressons ici aux contraintes d'intégrité qui ne se traduisent pas par des contraintes de participation. C'est-à-dire que nous ne prenons pas en compte les contraintes relatives aux relations partagées dans leur contexte consommateur.

En CROME, une instance ne doit vérifier les contraintes d'un contexte que si elle y participe. C'est vrai pour les contraintes relatives aux relations locales à ce contexte ainsi que pour les contraintes sur les propriétés définies ou exploitées dans ce contexte. Ce principe s'applique également aux contraintes relatives aux relations du contexte de base dont on peut considérer que la contrainte de participation est une tautologie.

Ainsi, si une opération provoque, dans un contexte, à la fois une violation des contraintes d'intégrité pour une instance et la non-participation de cette instance à ce contexte, cette violation n'a plus de sens. Elle ne donne pas lieu à un traitement compensatoire, ni à un rejet de la transaction englobante.

Une instance qui ne participait pas à un contexte et qui vient à en vérifier toutes les contraintes de participation doit alors également en vérifier les contraintes d'intégrité. L'opération qui a provoqué la participation de cette instance peut conduire, au sein du nouveau contexte de participation, au déclenchement d'un traitement compensatoire pour que ces contraintes d'intégrité soient vérifiées.

Cette participation préalable au(x) contexte(s) introduisant les contraintes d'intégrité fait que la combinaison de toutes les contraintes de la base ne s'obtient pas directement par une conjonction, mais par une conjonction de chaque contrainte conditionnée par les contraintes de

37. Voir pour cela les contraintes d'intégrité relative à « Produit » dans les contextes commercial et de « Direction » à l'annexe A.

participation du contexte. Au sein d'un contexte, la combinaison des contraintes est obtenue par une conjonction.

Formalisation de la combinaison des contraintes d'intégrité d'une base de données structurée en CROME

On note $CI_{C,f}$ ($C \in \mathcal{C}$, $f \in \mathcal{F} - \{\mathbf{base}\}$) le prédicat qui représente la conjonction de toutes les contraintes d'intégrité du contexte fonctionnel CF_f pour les instances de la classe C ³⁸. $CI_{C,base}$ représente la conjonction de toutes les contraintes d'intégrité du contexte de base pour les instances de la classe C

Soit CI_C le prédicat qui représente la combinaison de toutes les contraintes des instances de la classe C vis-à-vis de l'ensemble des contextes.

La combinaison des contraintes d'intégrité peut s'écrire ainsi :

$\forall o \in \text{extension}(C),$

$$CI_C(o) = CI_{C,base}(o) \wedge_{f \in \mathcal{F} - \{\mathbf{base}\}} ((f \in \mathcal{E}\mathcal{F}(o)) \Rightarrow CI_{C,f}(o))$$

Ainsi, si $f \in \mathcal{E}\mathcal{F}(o)$ est faux, $((f \in \mathcal{E}\mathcal{F}(o)) \Rightarrow CI_{C,f}(o))$ est vrai sans qu'il faille vérifier la valeur des contraintes d'intégrité pour ce contexte.

38. Rappelons que toutes les contraintes d'intégrité sont exprimées au sein des classes.

3.7. Propriétés d'un schéma de base de données conçu en CROME

CROME permet de structurer le schéma d'une base de données relativement aux fonctions de l'organisation. Cette structuration se traduit par l'existence des plans et contextes fonctionnels, tant à la conception qu'à l'exploitation. Dans le cadre de notre exemple, l'annexe A montre la prise en compte de cette structuration lors de la conception, chaque plan et contexte étant spécifié séparément des autres. L'annexe B montre pour chaque fonction les interfaces destinées à ses clients qui sont générées automatiquement.

La prise en compte de la structuration en fonctions dès la conception permet de concevoir le schéma « fonction par fonction ». Lors de la conception, chaque fonction détermine et fournit les descriptions du schéma qui lui sont nécessaires, c'est-à-dire que chaque fonction détermine et fournit le plan et le contexte qui lui sont propres.

A partir de cette propriété de structuration en fonctions du schéma de la conception à l'exploitation, nous pouvons déterminer les autres propriétés suivantes du schéma que nous allons maintenant étudier :

- modularité interne des classes du domaine ;
- généricité des relations partagées et généricité du schéma de base ;
- interaction entre contextes fonctionnels ;
- statut de schéma de classes des contextes fonctionnels.

3.7.1 Modularité interne des classes du domaine

Nous avons montré l'existence d'un référentiel commun aux fonctions constitué des entités du domaine, ce qui a eu pour effet de diffuser les descriptions relatives à chaque fonction au sein de chaque classe du domaine. Dans le cadre de notre exemple, l'annexe A montre cette diffusion : pour une fonction et pour une classe, il existe un incrément de description d'interface spécifique³⁹.

De l'existence de ce référentiel commun et de la structuration du schéma résulte donc une structuration interne des classes du domaine traduite par l'existence de plusieurs incréments de description. Pour une classe, chacun de ses incréments est déterminé lors de la conception du plan et du contexte relatifs à une fonction, en même temps que les incréments des autres classes⁴⁰. Par conséquent, chaque incrément est conçu séparément des autres et constitue donc un module au sein de la classe. Ceci conduit donc à l'existence d'une modularité interne à chaque classe du domaine.

Les accès inter-plans relèvent de l'articulation entre parties fonctionnelles et ne provoquent pas d'interférence dans la conception des parties fonctionnelles, préservant la possibilité de conception séparée. Ils correspondent à une interaction entre modules basée sur leur interface et ne remettent pas en cause la modularité interne des classes du domaine.

Exemple

Chaque incrément de description d'interface de la classe « Client » est déterminé indépendamment des autres lors de la conception successive des descriptions relatives aux fonctions commerciale, « Stock », comptable, « SAV » et « Direction ». Chaque accès inter-

39. moyennant la participation de l'entité correspondante à la fonction

40. de par la conception « fonction par fonction » du schéma

plans introduit ne remet pas en cause la conception des éléments de description relatifs à la fonction fournisseuse de l'accès et préserve donc la conception par modules de la classe.

Remarque

L'indépendance de conception peut conduire à des problèmes d'homonymie entre des propriétés de différentes parties fonctionnelles. Le problème se pose plus principalement lors de l'exploitation au travers d'un accès inter-plans d'une propriété homonyme à l'une des propriétés de la partie fonctionnelle.

Cette problématique de l'homonymie a déjà été abordée dans le cadre de l'héritage multiple [Carré89, Ducournau95]. Certaines solutions pourraient être reprises ici : renommage, préfixage ou points de vue de sélection mais relativement à des fonctions et non à des classes.

Exemple

La fonction commerciale introduit l'attribut « contact_commercial » dans la partie fonctionnelle de « Fournisseur ». La fonction « Fabrication » introduit l'attribut « contact_technique » dans la partie fonctionnelle de « Part_externe », surclasse de « Fournisseur ». Supposons maintenant que ces deux attributs aient le même nom à savoir « contact ». Dans ce cas, l'introduction dans un contexte relatif à une fonction tierce d'un accès inter-plans pour chaque attribut « contact » conduirait à un problème d'homonymie lors de l'exploitation et qui pourrait être résolu, par exemple, par un préfixage sur les noms des fonctions.

En OQL, une requête exprimée par un client de cette fonction tierce ayant pour objectif de retrouver les fournisseurs tels que leur contact technique soit égal à leur contact commercial pourrait s'écrire moyennant une extension d'OQL relative au préfixage .

```
select f from f in Fournisseur where f.Commercial:contact = f.Fabrication:contact;
```

Notons qu'il y a lieu d'utiliser un symbole distinct du point pour éviter la confusion avec les expressions de chemin, par exemple les deux points.

Modularité interne des classes dans le cadre de l'approche par vues

L'approche par vues n'apporte pas de modularité interne aux classes. Dans le cas de la conception par intégration de vues, chaque schéma est conçu indépendamment. Mais la phase d'intégration ne cherche pas à préserver la structuration issue des vues, brisant toute possibilité de modularité du schéma et donc des classes. On peut donc au plus parler de conception « fonction par fonction » mais pas de conception modulaire.

3.7.2 Généricité du schéma de base

Nous définissons le schéma de base en CROME comme le schéma du contexte de base. Pour obtenir sa généricité par rapport aux fonctions, nous devons établir la généricité des relations d'héritage et des relations inter-entités qu'il contient et qui sont partagées par l'ensemble des fonctions.

Nous proposons d'étudier pour chaque sorte de relation les modalités de son partage et le caractère générique que celui-ci lui confère.

Relations d'héritage

Nous avons vu en 3.3.4 que les plans fonctionnels préservent les relations d'héritage

issues du plan de base. La partie fonctionnelle d'une classe hérite des parties fonctionnelles de ses surclasses au sein du plan correspondant. Au sein d'un plan, la relation d'héritage entre deux classes ne concerne donc que les descriptions détenues par ce plan. Plus généralement, au sein d'un contexte fonctionnel, la relation d'héritage entre deux classes porte sur :

- les parties fonctionnelles ;
- les modalités de fourniture des propriétés du plan de base ;
- les accès inter-plans ;
- les prédicats de participation.

Ainsi, au sein d'un contexte fonctionnel, même si elle associe les mêmes deux classes que dans le plan de base, une relation d'héritage concerne des aspects purement locaux à ce contexte.

Du point de vue des extensions, au sein d'un contexte, les relations d'héritage ne doivent être considérées que relativement aux extensions fonctionnelles⁴¹. Ceci apporte un aspect supplémentaire à la contextualisation de ces relations au sein des contextes.

Tout ceci confère un aspect générique aux relations d'héritage du plan de base relativement aux fonctions qui les adaptent aux aspects locaux des fonctions.

Relations inter-entités

Dans le contexte de base, une relation inter-entités détermine un ensemble d'occurrences entre les instances des deux classes. Dans un contexte fonctionnel, seules les occurrences dont chaque extrémité appartient à l'extension fonctionnelle de sa classe ont un sens⁴². Cette restriction de l'ensemble des occurrences d'une relation rend compte de son partage qui se fait par une adaptation aux extensions locales d'un contexte.

Par ailleurs, les contraintes de cardinalité d'une relation du plan de base sont également prises en compte dans un contexte fonctionnel relativement aux occurrences de la relation présentes dans le contexte. Elles interviennent alors en tant que contraintes de participation dans le calcul des extensions fonctionnelles des classes situées aux extrémités d'une relation (cf 3.6.3.2).

Par conséquent, les relations inter-entités du plan de base sont génériques relativement aux différentes fonctions.

Notons que la genericité s'applique également à une relation inter-entités définie dans un plan fonctionnel. Celle-ci est adaptée dans les contextes qui l'exploitent au travers d'un accès inter-plans, ce qui lui confère également un aspect générique vis-à-vis ses contextes consommateurs.

Généricité des relations dans les schémas-vues dans le cadre de l'approche par vues

Une relation entre deux classes ne peut être contextualisée dans un schéma-vue. Soit les deux classes participent à ce schéma-vue et la relation d'héritage entre elles en fait autant, soit l'une d'elles ou les deux ne participent pas à ce schéma-vue et il en va de même pour la relation d'héritage. Ceci est également valable pour les autres relations entre les classes. Il n'y

41. Nous avons vu en 3.5.3 que l'extension fonctionnelle d'une sous-classe est incluse dans l'extension fonctionnelle de sa surclasse, ce qui est conforme à la sémantique de la relation d'héritage.

42. Ceci permet de vérifier l'intégrité référentielle de la relation dans les contextes fonctionnels (cf 3.6.2.2)

a donc pas de possibilité d'avoir de relations génériques.

Nous avons étudié dans le chapitre 2 (2.2.4) la situation où les deux classes d'une relation de composition ne participent pas à un schéma-vue mais où chacune y est, en quelque sorte, représentée par une classe-vue obtenue par application d'un ou plusieurs opérateurs depuis elle-même⁴³. Dans ce cas, la relation entre les deux classes doit être adaptée entre les deux classes-vues de façon ad hoc. Cette adaptation est généralisable aux relations inter-entités.

A la différence de CROME, le schéma de base ne possède aucune généricité vis-à-vis des différents schémas-vues.

3.7.3 Interaction entre contextes fonctionnels

Nous avons déterminé en 3.7.1 la modularité interne des classes du domaine. Cette modularité porte sur un ensemble des caractéristiques propres aux classes : descriptions, extensions et contraintes d'intégrité⁴⁴. Ces caractéristiques servent de support pour les différentes formes d'interaction entre contextes fonctionnels que nous avons présentées : transmission d'informations par partage de descriptions, évaluation des extensions fonctionnelles dans les contextes consommateurs et gestion de contraintes d'intégrité. La modularité interne des classes du domaine nous offre ainsi le moyen de décrire ces interactions entre contextes au sein des classes.

Nous proposons d'examiner ces différentes formes d'interaction au sein des entités et classes du domaine en montrant à chaque fois les caractéristiques et mécanismes utilisés.

Transmission d'informations par partage de descriptions

Il existe deux formes de partage de descriptions entre contextes. Tout d'abord, au sein de chaque classe, la description de base est partagée par l'ensemble des contextes. Ces descriptions partagées constituent un support pour la transmission d'informations⁴⁵ entre contextes. Chaque incrément d'interface relatif à une fonction définit les modalités de fourniture de valeur des propriétés de la description de base. Ceci permet de spécifier les contextes fournisseurs des valeurs de chaque propriété et ceux qui les consomment. Chaque description de base constitue ainsi, au sein des classes du domaine, un lieu d'interaction entre contextes.

Entre deux contextes et au sein d'une classe, il est possible de définir un accès inter-plans. Un accès inter-plans enrichit les descriptions relatives au contexte consommateur avec des descriptions partagées issues du contexte fournisseur. Il répond à l'objectif de transmission d'informations fonctionnelles dans un sens précis : du contexte fournisseur au contexte consommateur. Chaque accès inter-plans constitue donc un mécanisme d'interaction entre contextes.

Évaluation des extensions fonctionnelles dans les contextes consommateurs

L'évaluation de l'extension fonctionnelle d'une classe dans un contexte est d'abord basée sur le contenu de l'extension de cette classe. L'extension d'une classe est fournie par un ou plusieurs contextes d'instanciation et elle est exploitée par les autres contextes fonctionnels.

43. généralement par projection et sélection ou par augmentation et sélection

44. Rappelons que nous avons fait le choix de contraintes d'intégrité décrites au sein des classes (cf 3.6.1).

45. Rappelons qu'il s'agit d'informations communes à l'ensemble des fonctions.

Dans ces contextes d'instanciation, l'extension fonctionnelle est égale à l'extension de la classe (cf 3.5.5.3), ce qui correspond bien au fait que leurs clients créent et détruisent les instances. Un contexte d'instanciation d'une classe est spécifié par une clause de l'incrément d'interface correspondant. A l'image de sa description de base, l'extension d'une classe constitue un support d'interaction entre les contextes d'instanciation et ceux consommateurs de cette extension.

La sélection des instances dans un contexte consommateur pour déterminer son extension fonctionnelle est réalisée à l'aide de plusieurs critères :

- la vérification du prédicat de participation ;
- la participation aux contextes fournisseurs des accès inter-plans ;
- la vérification des contraintes de cardinalité des relations inter-entités consommées.

Ces trois critères montrent que l'évaluation de l'extension fonctionnelle d'un contexte dépend d'un ou de plusieurs autres contextes. D'une part, cette évaluation dépend des contextes fournisseurs des accès inter-plans. D'autre part, la vérification du prédicat de participation et des contraintes de cardinalité des relations consommées dans un contexte ne peut avoir lieu pour une instance que si les clients d'autres contextes ont fixé des valeurs adéquates aux propriétés utilisées dans ce prédicat et, pour chaque relation consommée, associé cette instance avec le nombre adéquat d'instances des autres extrémités.

Les contextes capables de valider un prédicat de participation d'un autre contexte sont les fournisseurs de la valeur des propriétés utilisées dans ce prédicat, que ces propriétés proviennent d'accès inter-plans ou qu'elles soient localisées dans la description de base. Il en va de même pour les contextes capables de valider les contraintes de cardinalités des relations consommées.

Gestion des contraintes d'intégrité

Nous nous intéressons ici plus particulièrement aux contraintes sur les propriétés, l'interaction issue des contraintes de cardinalité des relations consommées ayant été étudiée précédemment. Nous avons vu en 3.6.4.2 que les contraintes d'intégrité sur les propriétés peuvent être violées par des opérations qui ont lieu dans d'autres contextes. Il suffit que le prédicat correspondant contienne des accès inter-plans ou des propriétés du plan de base dont les valeurs sont fournies par d'autres contextes. Une telle violation impose de déclencher un traitement compensatoire dans le contexte où la contrainte est violée. Ceci montre que la vérification des contraintes d'intégrité dans un contexte peut dépendre d'un ou de plusieurs autres contextes.

Comparaison avec l'approche par vues

Dans le cadre de l'approche par les vues des bases de données à objets, chaque schéma-vue est conçu de façon totalement indépendante des autres. Il n'est pas possible de spécifier de mécanismes d'interaction entre schémas-vues. Cependant, à l'exécution, la mise à jour des valeurs d'une instance, la création ou la destruction d'instances provoquent nécessairement des effets de bord entre schémas-vues. Ceux-ci restent difficiles à contrôler : il n'existe pas de notion de schémas-vues fournisseurs et consommateurs.

A l'inverse, en CROME, les contextes qui fournissent les instances, les accès inter-plans et la valeur des propriétés des descriptions de base sont spécifiés. Ces contextes sont à l'origine des différentes interactions que nous avons présentées.

3.7.4 Les contextes fonctionnels ont un statut de schéma de classes

Un contexte fonctionnel a été défini en 3.5.5.2 comme l'ensemble des descriptions fonctionnelles (modalités de fourniture des valeurs des propriétés de base incluses), accès inter-plans et extensions fonctionnelles des classes intervenant dans la fonction correspondante, détenant ainsi la totalité des descriptions de la fonction nécessaires à ses clients⁴⁶.

Un contexte peut ainsi être considéré comme un véritable schéma de classes. En effet, une description fonctionnelle munie d'accès inter-plans et d'une extension fonctionnelle forme une classe relative à une entité telle qu'elle est perçue par la fonction correspondante. A chaque entité du domaine, nous avons associé une classe globale à l'ensemble des fonctions. Mais pour une fonction, la description fonctionnelle de cette classe globale munie des accès inter-plans et de l'extension fonctionnelle associés est aussi une classe, correspondant à la même entité sous l'unique point de vue de cette fonction.

Par conséquent, chaque contexte fonctionnel est considéré comme un schéma de classes par les clients d'une fonction. Ce schéma peut être considéré par ces clients comme celui d'une base de données qu'ils peuvent exploiter classiquement. En effet, les interactions de cette « base » avec les autres « bases »⁴⁷ leur sont transparentes. Cette exploitation se fait donc classiquement par un langage d'interrogation et de manipulation. Dans le cadre du modèle de l'ODMG, cette manipulation se fait par le langage OQL⁴⁸ tel qu'il est défini dans [ODMG97] et sans qu'aucun ajout ou modification à OQL ne soient nécessaires.

Dans le cadre de notre exemple, l'annexe B rend compte de cette propriété des contextes fonctionnels de CROME. Cet annexe montre les descriptions nécessaires aux clients de chaque fonction. Ces descriptions se présentent sous la forme d'interfaces ODL dont la seule modification par rapport au langage ODL tel qu'il est défini dans [ODMG97] est l'introduction des contraintes de cardinalité. Ces interfaces sont donc parfaitement manipulables en OQL et par les opérations de création, suppression et mise à jour d'instances existant dans l'un des langages cibles du modèle de l'ODMG.

Remarques

- L'ensemble des caractéristiques qui forment un contexte fonctionnel a été fourni par les clients de la fonction correspondante. Le fait qu'ils puissent exploiter classiquement ce contexte en tant que schéma de classes répond donc à l'objectif initial des contextes.

- Les contextes fonctionnels qui sont des schémas externes constitués de classes sont directement issus de la conception. Entre l'annexe A qui montre les différentes étapes de la conception de notre exemple et l'annexe B qui donne les interfaces d'exploitation, il n'y a aucune démarche de conception. Le contenu de l'annexe B est généré automatiquement à partir de celui de l'annexe A. Ce dernier point signifie également que la correspondance entre fonctions et descriptions du schéma est préservée de la conception à l'exploitation.

46. ce qui permet de conférer à un contexte fonctionnel le statut de schéma externe.

47. relatives aux autres contextes

48. ainsi que par les opérations de création, suppression et mise à jour d'instance

Chapitre 4

Mises en œuvre

4.1. Introduction

Dans le chapitre précédent, nous avons introduit l'approche CROME. Celle-ci se situe à un niveau logique de conception et se veut exploitable dans de multiples systèmes de bases de données. Dans la première partie de ce chapitre, nous montrons comment il est possible de traduire une conception en CROME par les techniques de vues des bases de données à objets. Cette traduction donne lieu à une formulation des contextes en classes-vues et schémas-vues. Ces classes-vues et schémas-vues sont générés automatiquement à partir de la conception en CROME. Nous expliciterons les limites de cette traduction et établirons quelques conclusions sur les techniques de vues. Dans la seconde partie, nous étudions une mise en œuvre dans le cadre des SGDB à objets du commerce et en particulier, ceux supportant Java comme le SGDB POET [POET97] qui nous a servi de support pour la réalisation d'un prototype. Les SGDB du commerce n'intégrant pas les techniques de vues, il n'est pas possible d'exploiter la traduction des contextes en classes-vues et schémas-vues de la première partie. Cette seconde mise en œuvre utilise donc certaines caractéristiques du langage Java¹ pour réaliser les contextes fonctionnels.

4.2. Formulation des contextes fonctionnels en schémas-vues

Dans le chapitre 2, nous avons étudié les différentes techniques de vues d'exploitation (2.2) et plus précisément celles qui insèrent les classes-vues dans la hiérarchie d'héritage (2.2.3). Nous proposons maintenant à partir d'une conception en CROME à un niveau logique de fournir un guide de traduction par ces techniques de vues et plus particulièrement dans MultiView [Rundensteiner92a]. La mise en œuvre de cette traduction conduit à la génération automatique des classes-vues et schémas-vues pertinents pour les clients des différentes fonctions de l'organisation. Il s'agit de conférer un statut de contexte fonctionnel aux schémas-vues ainsi générés.

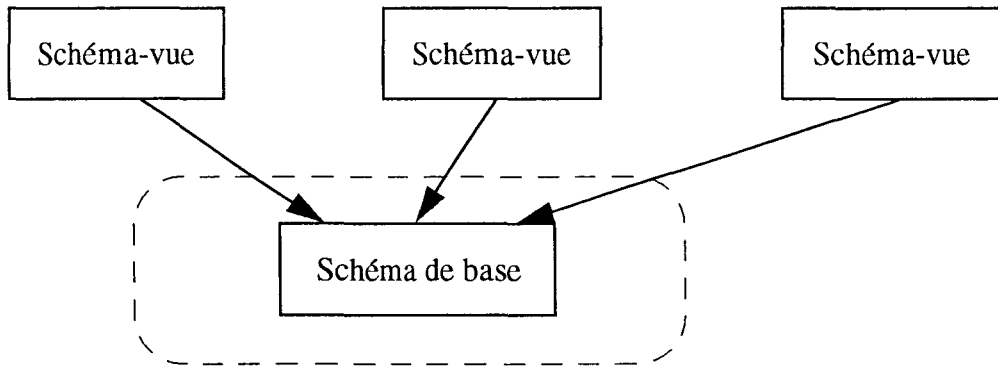
Dans cette section, nous étudions deux principes de formulation des contextes en schémas-vues pour mener deux mises en œuvre distinctes :

- un premier qui consiste à définir les classes-vues formant les schémas-vues par extraction en intension (4.2.1) ;
- un second qui consiste à définir ces classes-vues par augmentation de capacité comme le supporte MultiView [Ra95] (4.2.2).

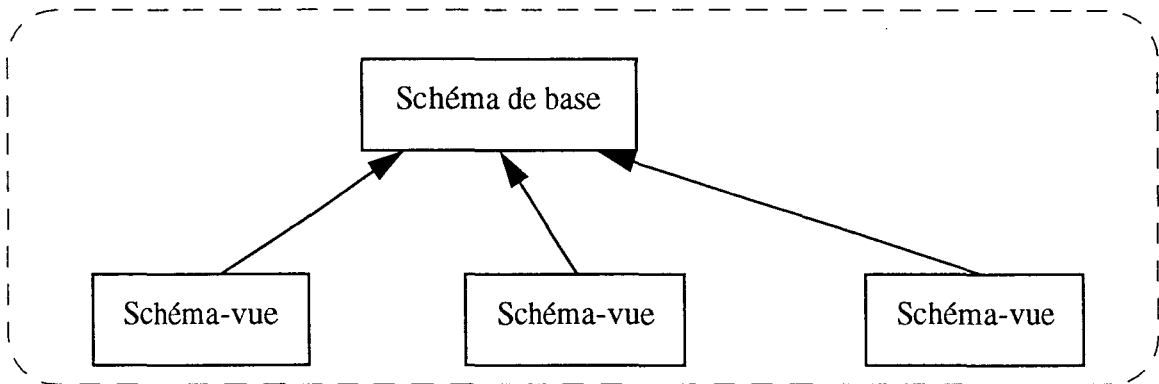
Ces deux principes peuvent être illustrés à la figure 4.1. Dans le premier cas, toutes les descriptions des informations contenues dans la base se trouvent dans le schéma de base, les schémas-vues sont constitués de classes-vues qui sont des extractions de description. Dans le second cas, les schémas-vues sont constitués de classes-vues qui introduisent des descriptions

1. [ODMG97] a introduit le couplage avec Java pour la version 2.0 du modèle.

d'informations contenues dans la base, augmentant ainsi les descriptions présentes dans le schéma de base.



Premier cas : définition des classes-vues composant les schémas-vues par extraction



Second cas : définition des classes-vues composant les schémas-vues par augmentation



Ensemble des descriptions des informations contenues dans la base

Figure 4.1. Localisation des descriptions des informations contenues dans une base selon la définition des classes-vues formant les schémas-vues

Dans un premier temps, nous étudierons ces mises en œuvres sans prendre en compte les contraintes de participation des instances à un contexte fonctionnel telles qu'elles sont explicitées en 3.5. Puis, dans un second temps, nous montrerons comment tenir compte de ces contraintes (4.2.3). Nous verrons que leur prise en compte ne peut se faire directement par l'opérateur de sélection en extension (**select**) et que, par conséquent, cela nécessite l'élaboration d'un schéma global dont la complexité devient alors très importante.

Enfin, nous ferons une étude des propriétés spécifiques de ces deux mises en œuvre et nous les comparerons avant d'établir quelques conclusions et limites sur les techniques de vues (4.2.4).

4.2.1 Mise en œuvre par extraction

4.2.1.1 Définition des classes-vues relatives aux descriptions fonctionnelles

Notre première mise en œuvre est basée sur l'utilisation de l'opérateur d'extraction en intension encore appelé opérateur de masquage (**hide**) ou de projection (**project**). Pour chaque entité du domaine est d'abord générée une classe dans le schéma de base². Dans cette approche, chaque classe du schéma de base correspondant à une entité du domaine détient l'ensemble des descriptions pertinentes pour toutes les fonctions. Son intension est donc constituée par la fusion de la description de base et de toutes les parties fonctionnelles déterminées lors de la conception en CROME.

Pour chaque entité du domaine est ensuite générée dans le schéma global une classe-vue pour chaque fonction à laquelle ladite entité participe. Cette classe-vue est une extraction de la classe du schéma de base correspondant à la même entité. Elle ne doit détenir que les descriptions pertinentes de l'entité pour les clients de la fonction. Son intension est donc constituée de la description fonctionnelle et des accès inter-plans associés relatifs à la fonction. Les autres descriptions sont masquées par l'opérateur d'extraction en intension. Ces autres descriptions correspondent aux attributs et relations relatives aux autres fonctions pour lesquelles il n'existe pas d'accès inter-plans depuis la fonction pour laquelle on élabore la classe-vue.

Exemple

Les figures 4.2, 4.3 et 4.4 montrent un ensemble de conception en CROME de quatre classes. Les clauses relatives aux prédicats de participation et aux contextes d'instanciation ont été omises. Les contraintes de participation sont introduites en 4.2.3. Par ailleurs, il n'est pas possible de spécifier les classes dont les clients d'un schéma-vue posséderaient ou ne posséderaient pas la faculté d'instanciation.

```
interface Part_externe
{
    key nom,telephone;
    attribute string nom;
    attribute string adresse;
    attribute string telephone;
}
interface Client: Part_externe
{
    relationship set<Produit> a_achete inverse Produit::a_ete_achete_par;
}
interface Fournisseur: Part_externe
{
    relationship set<Produit> fournit inverse Produit::est_fourni_par;
}
interface Produit
{
    attribute string designation;
    attribute integer prix_tarif;
    relationship set<Client> a_ete_achete_par inverse Client::a_achete;
    relationship set<Fournisseur> est_fourni_par inverse Fournisseur::fournit;
}
```

Figure 4.2. Plan de base d'un schéma conçu par l'approche CROME

2. Pour la définition du schéma de base et du schéma global dans le cadre de MultiView, voir 2.2.3.3.1.

```

function Commercial interface increment Client
{
  function part:
  {
    attribute integer montant_cmd;
    attribute date date_derniere_cmd;
  }
}

function Commercial interface increment Fournisseur
{
  function part:
  {
    attribute string contact_commercial;
  }
}

function Commercial interface increment Produit
{
  access function Comptable:
  {
    attribute integer chiffre_affaires_produit;
  }
  function part:
  {
    attribute integer nbr_unites_vendues;
  }
}

function Comptable interface increment Part_externe
{
  function part:
  {
    attribute string contact_comptable;
  }
}

function Comptable interface increment Client
{
  function part:
  {
    attribute integer encours_a_regler;
    attribute integer chiffre_affaires;
  }
}

function Comptable interface increment Fournisseur
{
  function part:
  {
    attribute integer sommes_acquittees;
    attribute integer sommes_dues;
  }
}

function Comptable interface increment Produit
{
  function part:
  {
    attribute integer chiffre_affaires_produit;
    attribute integer marge_moyenne;
  }
}

```

Figure 4.3. Plans fonctionnels commercial et comptable

```

function Direction interface increment Part_externe
  function part:
  {
    attribute string interlocuteur;
  }
}
function Direction interface increment Produit
{
  function part:
  {
    attribute integer importance;
  }
}

```

Figure 4.4. Plan fonctionnel « Direction »

Pour chaque entité introduite par le plan de base de la figure 4.2, nous élaborons une classe dans le schéma de base dont l'intension est la fusion de toutes les descriptions : description de base et parties fonctionnelles relatives aux fonctions commerciale et comptable. La figure 4.5 illustre le schéma de base. La syntaxe retenue est celle de COCOON présentée en 2.2.3.1.

```

define type part_externe isa object =
  nom : string,
  adresse : string,
  telephone : string,
  contact_comptable: string,
  interlocuteur : string;
define class Part_externe : part_externe some Objects;

define type client isa part_externe =
  a_achete : set of produit inverse a_ete_achete_par,
  montant_cmd : integer,
  date_derniere_cmd : string;
  encours_a_regler : integer,
  chiffre_affaires : integer;
define class Client : client some Part_externe;

define type fournisseur isa part_externe =
  fournit : set of produit inverse est_fourni_par,
  contact_commercial : string,
  sommes_acquittees : integer,
  sommes_dues : integer;
define class Fournisseur : fournisseur some Part_externe;

define type produit isa object =
  designation : string;
  prix_tarif : integer;
  a_ete_achete_par : set of client inverse a_achete,
  est_fourni_par : set of fournisseur inverse fournit,
  nbr_unites_vendues : integer,
  chiffre_affaires_produit : integer,
  marge_moyenne : integer,
  importance : integer;
define class Produit : produit some Objects;

```

Figure 4.5. Schéma de base obtenu pour chaque classe par fusion des descriptions

Nous définissons maintenant les classes-vues relativement à chaque fonction :

- fonction « Direction » :

```
define view Part_externe_Direction as hide [contact_comptable](Part_externe);  
define view Client_Direction as hide [montant_cmd, date_derniere_cmd, contact_comptable,  
encours_a_regler, chiffre_affaires](Client);  
define view Fournisseur_Direction as hide [contact_commercial, contact_comptable,  
sommes_acquittees, sommes_dues](Fournisseur);  
define view Produit_Direction as hide [nbr_unites_vendues, chiffre_affaires_produit,  
marge_moyenne](Produit);
```

- fonction commerciale :

```
define view Client_Commercial as hide [contact_comptable, encours_a_regler, chiffre_affaires,  
interlocuteur](Client);  
define view Fournisseur_Commercial as hide [contact_comptable, sommes_acquittees,  
sommes_dues, interlocuteur](Fournisseur);  
define view Produit_Commercial as hide [importance, marge_moyenne](Produit);
```

- fonction comptable :

```
define view Part_externe_Comptable as hide [interlocuteur](Part_externe);  
define view Client_Comptable as hide [montant_cmd, date_derniere_cmd, interlocuteur](Client);  
define view Fournisseur_Comptable as hide [contact_commercial, interlocuteur](Fournisseur);  
define view Produit_Comptable as hide [nbr_unites_vendues, importance](Produit);
```

Nous n'avons pas défini de classe-vue pour l'entité « Part_externe » dans la fonction commerciale. En effet, l'entité « Part_externe » n'y participant pas, aucune classe-vue ne doit y être définie pour cette entité.

Nous disposons ainsi des classes-vues nécessaires pour présenter les informations pertinentes aux clients de chaque fonction. Pour la fonction « Direction », il s'agit des classes-vues « Part_externe_Direction », « Client_Direction », « Fournisseur_Direction » et « Produit_Direction ». Pour la fonction commerciale, il s'agit des classes-vues « Client_Commercial », « Fournisseur_Commercial » et « Produit_Commercial ». Enfin, pour la fonction comptable, il s'agit des classes-vues « Part_externe_Comptable », « Client_Comptable », « Fournisseur_Comptable » et « Produit_Comptable ». Ces classes-vues pertinentes pour les différentes fonctions vont nous servir de support pour l'élaboration des schémas-vues par la suite.

Remarque

Les descriptions fonctionnelles et les propriétés définies dans les accès inter-plans forment l'intension des classes-vues. L'intension des classes-vues n'est pas structurée et par conséquent, les propriétés correspondant aux accès inter-plans y sont incluses de la même façon que les propriétés des descriptions fonctionnelles et avec le même statut. Par exemple, l'attribut « chiffre_affaires_produit » de l'entité « Produit » a le même statut dans les classes-vues « Produit_Comptable » et « Produit_Commercial » alors qu'au niveau de la conception, cet attribut est introduit par le contexte comptable qui fournit aussi sa valeur et il est exploité au travers d'un accès inter-plans par le contexte commercial.

4.2.1.2 Classification des classes-vues

La définition des classes-vues entraîne leur classification automatique au sein du schéma global. Nous nous proposons d'étudier cette classification en nous référant à l'algorithme de

classification de MultiView étudié au chapitre 2 (2.2.3.2).

Préparation de la hiérarchie des types

L'algorithme de classification de MultiView impose que l'introduction de chaque propriété soit réalisée par un type unique, ce qui a été formalisé par la notion de fermeture de la hiérarchie de types relativement à l'opérateur « $\wedge$ ». Rappelons que pour être fermée par rapport à l'opérateur « $\wedge$ », une hiérarchie de types doit uniquement vérifier que, quels que soient deux types T_1 et T_2 de cette hiérarchie, il existe un type T_3 appartenant également à cette hiérarchie tel que $T_3 = T_1 \wedge T_2$.

Pour assurer que la hiérarchie des types soit fermée, on insère automatiquement les classes suivantes³ :

$\forall (C, CC) \in V \times V$, s'il n'existe pas une classe C' telle que $\text{type}(C') = \text{type}(C) \wedge \text{type}(CC)$, la classe C' est alors créée et insérée avec :

$\text{type}(C') = \text{type}(C) \wedge \text{type}(CC)$

$\text{extension}(C') = \text{extension}(C) \cup \text{extension}(CC)$

Il convient donc, dans notre cas, d'examiner les classes supplémentaires qu'il faut insérer afin que la hiérarchie des types du schéma global soit fermée. Nous commençons par réaliser cet examen dans le cadre de notre exemple. Le calcul du plus petit majorant pour chaque paire de types permet de découvrir l'absence des classes suivantes⁴ :

C_1 telle que :

$\text{type}(C_1) = \{\text{nom, adresse, telephone}\}$

$\text{extension}(C_1) = \text{extension}(\text{Part_externe})$

C_2 telle que :

$\text{type}(C_2) = \{\text{nom, adresse, telephone, a_achete}\}$

$\text{extension}(C_2) = \text{extension}(\text{Client})$

C_3 telle que :

$\text{type}(C_3) = \{\text{nom, adresse, telephone, fournit}\}$

$\text{extension}(C_3) = \text{extension}(\text{Fournisseur})$

C_4 telle que :

$\text{type}(C_4) = \{\text{designation, prix_tarif, a_ete_achete_par, est_fourni_par}\}$

$\text{extension}(C_4) = \text{extension}(\text{Produit})$

C_5 telle que :

$\text{type}(C_5) = \{\text{designation, prix_tarif, a_ete_achete_par, est_fourni_par, chiffre_affaires_produit}\}$

$\text{extension}(C_5) = \text{extension}(\text{Produit})$

Les quatre premières classes supplémentaires à insérer correspondent aux classes dont l'intension est égale à la description de base des différentes entités du domaine. Ces classes sont issues du calcul du plus petit majorant entre deux classes-vues correspondant à la même entité mais relatives à des fonctions différentes telles qu'il n'existe pas d'accès inter-plans

3. Rappelons que « V » désigne ici l'ensemble des classes et classes-vues du schéma global.

4. Nous ne présentons pas ici tous les calculs.

entre elles pour cette entité.

La classe suivante (C_5) provient du calcul des majorants des deux types suivants :
 $\text{type}(C_5) = \text{type}(\text{Produit_Commercial}) \wedge \text{type}(\text{Produit_Comptable})$

Cette classe est issue de l'existence d'accès inter-plans entre les fonctions commerciales et comptables relativement à l'entité « Produit » sur l'attribut « chiffre_affaires_produit ». Comme nous avons expliqué ci-dessus, les accès inter-plans ont le même statut dans les classes-vues correspondant aux contextes fournisseurs et dans les classes-vues correspondant aux contextes consommateurs : ils font partie de l'intension de chacune de ces classes-vues. Ainsi lors du calcul des plus petits majorants, ces accès inter-plans font apparaître des classes supplémentaires qui vont servir à les partager au travers de la relation d'héritage⁵.

D'une façon générale, il est possible de prouver que les classes supplémentaires à insérer correspondent aux classes dont l'intension est formée uniquement de la description de base de l'une des entités et de classes dont l'intension est aussi formée de la description de base de l'une des entités mais contenant en plus un ou plusieurs attributs partagés au travers d'un accès inter-plans. Tout ceci a pour effet d'engendrer une hiérarchie relativement complexe au sein du schéma global, le nombre de classes supplémentaires générées à cause des accès inter-plans pouvant être important.

Insertion des classes-vues dans le schéma global

Après l'ajout des classes supplémentaires, les classes-vues sont ensuite insérées dans le schéma global en respectant la relation de subsumption de MultiView qui est définie ainsi (cf 2.2.3.2) :

$\text{subsume}(C1, C2) = \text{vrai si } C2 \text{ is-a } C1$

$\text{subsume}(C1, C2) = \text{faux sinon.}$

La figure 4.6 montre le schéma global de notre exemple, une fois les classes-vues insérées.

Propriétés obtenues

- La définition d'une classe-vue par masquage des attributs d'une classe du schéma de base inclut le masquage des attributs de ses surclasses. Par exemple, le masquage des attributs des classes « Client » et « Fournisseur » pour définir les classes-vues « Client_Comptable » et « Fournisseur_Comptable » inclut le masquage de l'attribut « interlocuteur » utilisé pour définir la classe-vue « Part_externe_Comptable ». Ainsi, ces classes-vues entretiennent des relations d'héritage isomorphes à celles entretenues dans le schéma de base par les classes dont elles sont issues. Cette propriété intéressante est due au fait que les classes-vues résultent, de façon automatique, de la conception.

- Les classes du schéma de base sont des sous-classes des classes-vues, ce qui ne correspond pas au résultat de la phase de conception où les différentes descriptions fonctionnelles héritent de la description de base.

5. ce qui nous éloigne considérablement de la sémantique des accès inter-plans de CROME !

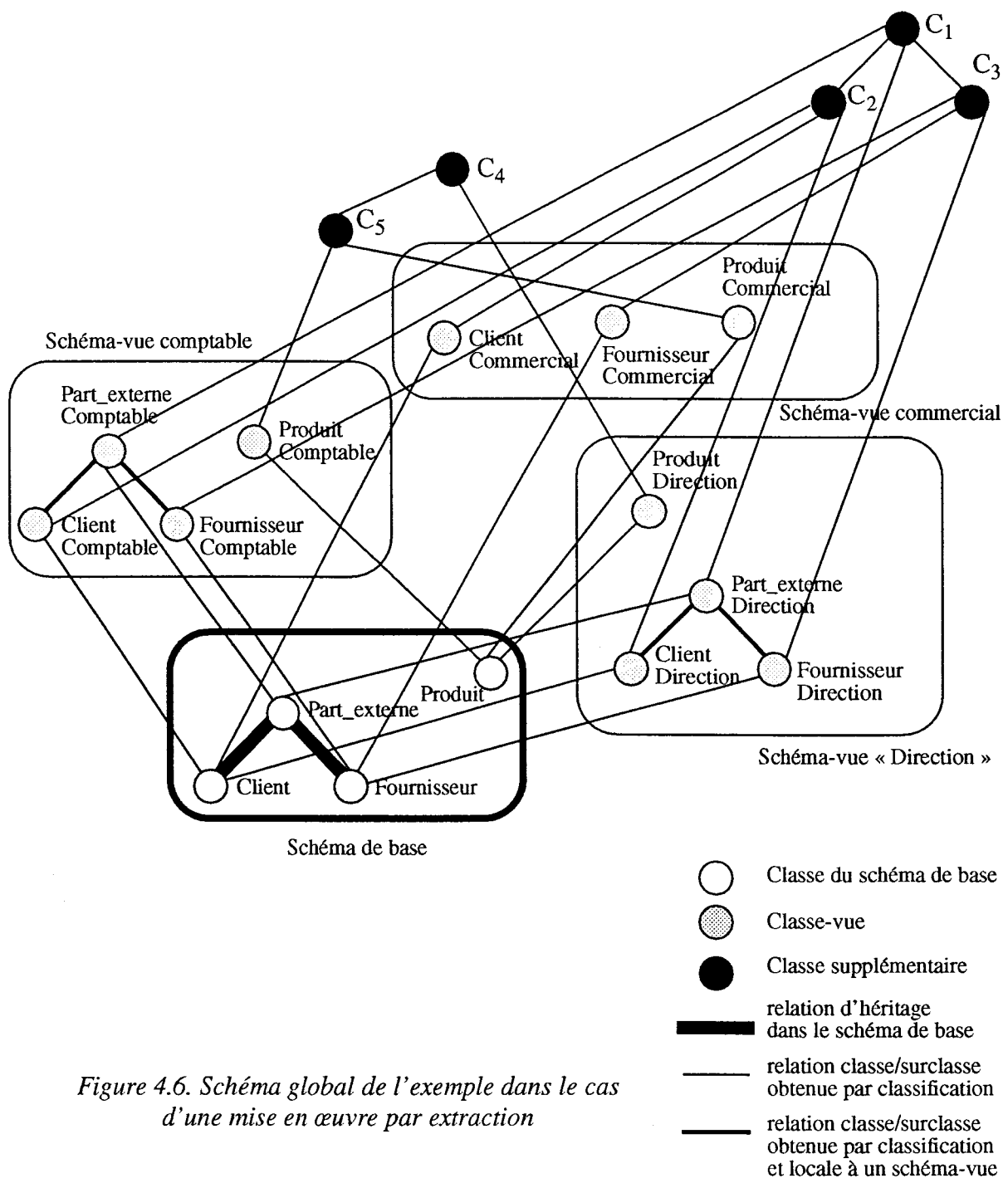


Figure 4.6. Schéma global de l'exemple dans le cas d'une mise en œuvre par extraction

4.2.1.3 Réalisation des contextes par les schémas-vues de MultiView

Nous pouvons définir un schéma-vue pour chaque fonction de l'exemple. Ce schéma-vue contient toutes les classes-vues pertinentes pour les clients de la fonction. Dans le cadre de l'exemple, nous pouvons définir les schémas-vues suivants⁶ :

6. Pour éviter toute confusion, nous renommons la clause « DEFINE-VIEW » de définition de schémas-vues par « DEFINE-SCHEMA-VIEW ».

```
DEFINE-SCHEMA-VIEW Direction;  
ADD-CLASS(Part_externe_Direction);  
ADD-CLASS(Client_Direction);  
ADD-CLASS(Fournisseur_Direction);  
ADD-CLASS(Produit_Direction);  
END-VIEW;
```

```
DEFINE-SCHEMA-VIEW Commercial;  
ADD-CLASS(Client_Commercial);  
ADD-CLASS(Fournisseur_Commercial);  
ADD-CLASS(Produit_Commercial);  
END-VIEW;
```

```
DEFINE-SCHEMA-VIEW Comptable;  
ADD-CLASS(Part_externe_Comptable);  
ADD-CLASS(Client_Comptable);  
ADD-CLASS(Fournisseur_Comptable);  
ADD-CLASS(Produit_Comptable);  
END-VIEW;
```

Bilan

- Chaque schéma-vue détient l'ensemble des descriptions de la fonction nécessaires à ses clients, ce qui lui confère un statut de contexte fonctionnel⁷. Ainsi, les schémas-vues définis ci-dessus sont un moyen de structuration du schéma global de la base de données.

- Chaque schéma-vue possède la même structuration en classes que le schéma de base. En effet, il est constitué des classes-vues issues des classes du schéma de base et relatives à une seule fonction. Or, nous avons vu que ces classes-vues entretiennent des relations d'héritage isomorphes à celles entretenues dans le schéma de base par les classes dont elles sont issues. La figure 4.6 illustre cette préservation dans les différents schémas-vues.

4.2.2 Mise en œuvre par augmentation de capacité

4.2.2.1 Définition des classes-vues relatives aux descriptions fonctionnelles

Notre seconde mise en œuvre s'appuie sur l'opérateur d'augmentation de capacité (**refine**) des classes-vues, opérateur propre à MultiView [Ra95] et dont la sémantique a été présentée au chapitre 2 (2.2.3.1.2) et que la table 4.1 rappelle ici.

7. moyennant de ne pas prendre en compte des contraintes de participation des instances à un contexte fonctionnel

Table 4.1: Sémantique de l'opérateur « refine » spécifique à MultiView

Opérateur	Syntaxe et sémantique
Opérateur d'augmentation définissant une ou plusieurs nouvelles fonctions $g_1, g_2, \dots, g_m$ aux éléments d'une classe C . A la différence de COCOON, ces fonctions peuvent être des propriétés stockées modifiant la structure des instances. Elles peuvent également être une propriété héritée du type d'une autre classe (ou classe-vue).	<u>Syntaxe</u> Cas de l'augmentation par une fonction calculée : refine [p:=expr](C) Cas de l'augmentation par une propriété « p » stockée : refine [p:<domain _p >](C) Cas de l'augmentation par une propriété « p » héritée du type d'une autre classe (ou classe-vue) refine [<classe>:p](C) <u>Sémantique</u> type(résultat) := type(C) $\cup$ {p} extension(résultat) := extension(C) <u>Comparaison résultat/opérande</u> type(résultat) $\leq$ type(C) extension(résultat) $\subseteq$ extension(C) résultat is-a C

Pour chaque entité du domaine et pour chaque fonction à laquelle cette entité participe, il est alors possible d'augmenter l'intension de la classe correspondante du schéma de base de la partie fonctionnelle et des accès inter-plans. Plus précisément, la démarche se décline en deux temps comme dans le cadre de la mise en œuvre par extraction.

Dans un premier temps, pour chaque entité du domaine, est générée une classe dans le schéma de base. Ici, chaque classe du schéma de base correspondant à une entité du domaine détient uniquement les descriptions communes à toutes les fonctions. Son intension est donc constituée par la description de base déterminée lors de la conception en CROME.

Dans un second temps, pour chaque entité du domaine est générée dans le schéma global une classe-vue à augmentation de capacité pour chaque fonction à laquelle l'entité participe. Cette classe-vue augmente la classe du schéma de base correspondant à la même entité. Les propriétés ajoutées sont de trois natures distinctes :

- les propriétés de la partie fonctionnelle non héritées de la partie fonctionnelle d'une surclasse. Il s'agit alors de propriétés nouvelles spécifiées selon la syntaxe **refine**[p:<domain_p>](C) où « p » désigne le nom de la nouvelle propriété ;
- les propriétés de la partie fonctionnelle héritées de la partie fonctionnelle d'une surclasse. Il s'agit alors de propriétés héritées d'une autre classe-vue et spécifiées selon la syntaxe **refine**[<classe>:p](C) où « classe » désigne la classe qui introduit la propriété « p » que l'on veut hériter ;
- les accès inter-plans sont introduits à l'aide d'un attribut calculé qui implante l'accès à l'attribut et dont il retourne la valeur. On utilise ici la syntaxe p:=expr où « p » désigne le nom de l'attribut calculé. Ceci n'est utilisé que lors de l'introduction d'un nouvel accès inter-plans. Les attributs calculés relatifs à des accès inter-plans hérités sont spécifiés selon la syntaxe **refine**[<classe>:p](C) où « classe » désigne la classe qui introduit l'accès inter-plans.

Une telle classe-vue détient alors l'ensemble des descriptions pertinentes de l'entité correspondante pour les clients de la fonction.

Exemple

Nous reprenons l'exemple de la section 4.2.1.1 (figures 4.2, 4.3 et 4.4). Nous commençons par élaborer le schéma de base. Pour chaque entité du domaine, nous introduisons une classe dont l'intension est égale à la description de base de l'entité. Ce schéma est illustré à la figure 4.7. Comme en 4.2.1.1, nous avons retenu la syntaxe de COCOON.

```
define type part_externe isa object =
  nom : string,
  adresse : string,
  telephone : string;
define class Part_externe : part_externe some Objects;

define type client isa part_externe =
  a_achete : set of produit inverse a_ete_achete_par;
define class Client : client some Part_externe;

define type fournisseur isa part_externe =
  fournit : set of produit inverse est_fourni_par;
define class Fournisseur : fournisseur some Part_externe;

define type produit isa object =
  designation : string;
  prix_tarif : integer;
  a_ete_achete_par : set of client inverse a_achete,
  est_fourni_par : set of fournisseur inverse fournit;
define class Produit : produit some Objects;
```

Figure 4.7. schéma de base : chaque classe correspond à la description de base

Nous définissons maintenant les classes-vues relativement à chaque fonction :

- fonction « Direction » :

```
define view Part_externe_Direction as refine[interlocuteur : string](Part_externe);
define view Client_Direction as refine Part_externe_Direction:interlocuteur](Client);
define view Fournisseur_Direction as refine [Part_externe_Direction:interlocuteur](Fournisseur);
define view Produit_Direction as refine [importance : integer](Produit);
```

- fonction commerciale :

```
define view Client_Commercial as refine [
  montant_cmd : integer,
  date_derniere_cmd : string](c:Client);
define view Fournisseur_Commercial as refine [contact_commercial : string](Fournisseur);
define view Produit_Commercial as refine [
  chiffre_affaires_produit := chiffre_affaires_produit(pick(select[pc = p] (pc :Produit_Comptable))),
  nbr_unites_vendues : integer](p:Produit);
```

- fonction comptable :

```
define view Part_externe_Comptable as refine[contact_comptable : string](Part_externe);
define view Client_Comptable as refine [
  Part_externe_Comptable:contact_comptable,
  encours_a_regler : integer;
```

```
chiffre_affaires : integer](Client);
define view Fournisseur_Comptable as refine [
Part_externe_Comptable:contact_comptable,
sommest_acquittees : integer,
sommest_dues : integer](Fournisseur);
define view Produit_Comptable as refine [
chiffre_affaires_produit : integer,
marge_moyenne : integer](Produit);
```

Remarques

- Les descriptions spécifiques à chaque fonction sont définies dans les classes-vues et non dans les classes du schéma de base, à l'image de ce qui se fait dans les plans fonctionnels. Cette réalisation est d'une certaine façon plus proche de la conception que dans le cas de la mise en œuvre par extraction.

- Un attribut fourni par une classe-vue relative à une fonction et exploité dans une classe-vue relative à une autre fonction est défini dans cette dernière comme un attribut calculé de même valeur. Par exemple, l'attribut « chiffre_affaires_produit » de l'entité « Produit » exploité par la fonction commerciale est défini dans la classe-vue « Produit_Commercial » comme un attribut calculé par une requête dont le résultat est la valeur dudit attribut dans la classe-vue « Produit_Comptable ». Ainsi seuls les clients du contexte comptable où est introduit l'attribut « chiffre_affaires_produit » peuvent en fournir la valeur, les clients des fonctions consommatrices ne pouvant qu'exploiter cette valeur. Ceci constitue une différence importante avec la mise en œuvre par extraction.

4.2.2.2 Classification des classes-vues

Comme en 4.2.1.2, nous examinons la classification des classes-vues en nous référant à l'algorithme de classification de MultiView étudié au chapitre 2 (2.2.3.2).

Préparation de la hiérarchie des types

A la différence de la mise en œuvre par classes-vues à extraction, la hiérarchie des types du schéma global composé des classes du schéma de base et des classes-vues à augmentation possède la propriété de fermeture par rapport à l'opérateur « $\wedge$ » sans qu'il soit nécessaire d'introduire de classes supplémentaires. Pour montrer cette propriété, nous faisons l'hypothèse de fermeture de la hiérarchie des types constitués chacun par la description de base, l'ensemble des parties fonctionnelles et l'ensemble des attributs calculés correspondant aux accès inter-plans relatifs à une même entité du domaine⁸.

Cette propriété peut être démontrée aisément en reprenant la formalisation utilisée dans le chapitre 3. Dans le cadre de cette formalisation, nous n'avons pas pris en compte l'affinement du domaine d'attributs et donc l'opérateur « $\wedge$ » équivaut ici à l'intersection entre deux intensions⁹.

Nous avons besoin d'étendre l'ensemble des attributs pour prendre en compte les

8. C'est-à-dire que chaque propriété d'une entité du domaine est introduite dans une seule classe, y compris les attributs calculés relatifs aux accès inter-plans. Les attributs calculés relatifs à des accès inter-plans hérités sont également hérités.

9. Cependant, cette démonstration pourrait être facilement généralisée au cas où l'affinement serait possible.

attributs calculés des accès inter-plans. Nous faisons le choix de laisser inchangée la définition de l'ensemble des attributs $\mathcal{A}$ dans la formalisation de CROME et de prendre en compte un ensemble d'attributs calculés $\mathcal{AC}$. $\mathcal{AC}$ peut être partitionné par la relation d'équivalence qui associe deux attributs de $\mathcal{AC}$ s'ils sont introduits par le même contexte fonctionnel. On obtient ainsi un ensemble quotient, ensemble des classes d'équivalence noté $\{\mathcal{AC}_f, \forall f \in \mathcal{F}\}$. L'ensemble total des attributs est donné par $\mathcal{A} \cup \mathcal{AC}$. L'intension de chaque classe de $\mathcal{C}^{10}$ est un sous-ensemble de $\mathcal{A} \cup \mathcal{AC}$. Notre hypothèse de fermeture se formule maintenant ainsi :

« la hiérarchie des intensions des classes de $\mathcal{C}$ est fermée par rapport à l'intersection ».

Soit C une classe de $\mathcal{C}$. On note $B(C)$ la classe introduite dans le schéma de base correspondant à la classe « C ». Par définition, on a :

$$\text{intension}(B(C)) = D_{\text{base}}(C)$$

Si la classe « C » est enrichie par la fonction « f », on note $V_f(C)$ la classe-vue à augmentation introduite dans le schéma global correspondant à la classe « C » pour la fonction « f ». Par définition, on a :

$\text{intension}(V_f(C)) = DF_f(C) \cup AC_f(C)$ si et seulement si $PF_f(C) \neq \emptyset$. $AC_f(C)$ est l'ensemble des attributs calculés correspondant aux accès inter-plans de $Acc_f(C)^{11}$. $AC_f(C)$ est défini ainsi :

$$\begin{aligned} AC_f: \quad \mathcal{C} &\rightarrow 2^{\mathcal{AC}} \\ C &\rightarrow AC_f \cap \text{intension}(C) \end{aligned}$$

Montrer que la hiérarchie des types du schéma global composé des classes du schéma de base et des classes-vues à augmentation est fermée revient à montrer qu'il existe, pour chacun des cas suivants, une classe ou une classe-vue du schéma global dont l'intension est égale à :

$$\text{cas 1 : } \text{intension}(B(C)) \cap \text{intension}(B(C')), \forall (C, C') \in \mathcal{C}^2$$

$$\text{cas 2 : } \text{intension}(V_f(C)) \cap \text{intension}(B(C')), \forall f \in \mathcal{F}, \forall (C, C') \in \mathcal{C}^2$$

$$\text{cas 3 : } \text{intension}(V_f(C)) \cap \text{intension}(V_{f'}(C')), \forall (f, f') \in \mathcal{F}^2, \forall (C, C') \in \mathcal{C}^2$$

Etudions chaque cas un à un.

cas 1 :

$$\text{intension}(B(C)) \cap \text{intension}(B(C')) = D_{\text{base}}(C) \cap D_{\text{base}}(C') = (\mathcal{D}_{\text{base}} \cap \text{intension}(C)) \cap (\mathcal{D}_{\text{base}} \cap \text{intension}(C')) = \mathcal{D}_{\text{base}} \cap (\text{intension}(C) \cap \text{intension}(C')) = \text{intension}(B(C'))$$

C'' existe et appartient à $\mathcal{C}$ à cause de notre hypothèse de fermeture. Enfin, $B(C'')$ existe car toute classe de $\mathcal{C}$ possède une classe correspondante dans le schéma de base.

cas 2 :

$$\begin{aligned} \text{intension}(V_f(C)) \cap \text{intension}(B(C')) &= (DF_f(C) \cup AC_f(C)) \cap D_{\text{base}}(C') = ((\mathcal{D}_f \cup AC_f) \cap \\ \text{intension}(C)) \cap (\mathcal{D}_{\text{base}} \cap \text{intension}(C')) &= (\mathcal{D}_f \cup AC_f) \cap \mathcal{D}_{\text{base}} \cap (\text{intension}(C) \cap \\ \text{intension}(C')) &= \mathcal{D}_{\text{base}} \cap (\text{intension}(C) \cap \text{intension}(C')) = \text{intension}(B(C')) \end{aligned}$$

Ce cas est donc ramené au cas précédent.

cas 3 :

$$\begin{aligned} \text{intension}(V_f(C)) \cap \text{intension}(V_{f'}(C')) &= (DF_f(C) \cup AC_f(C)) \cap (DF_{f'}(C') \cup AC_{f'}(C')) = ((\mathcal{D}_f \cup \\ AC_f) \cap \text{intension}(C)) \cap ((\mathcal{D}_{f'} \cup AC_{f'}) \cap \text{intension}(C')) &= ((\mathcal{D}_f \cup AC_f) \cap (\mathcal{D}_{f'} \cup AC_{f'})) \cap \\ (\text{intension}(C) \cap \text{intension}(C')) & \end{aligned}$$

10. Rappelons que $\mathcal{C}$ est l'ensemble des classes relatives aux entités du domaine.

11. Rappelons que $Acc_f(C)$ est l'ensemble des accès inter-plans de « C » pour la fonction « f »

Dans ce dernier cas, il convient de distinguer le calcul selon que $f=f'$ ou que $f \neq f'$

Si $f \neq f'$

$$\text{intension}(V_f(C)) \cap \text{intension}(V_{f'}(C')) = \mathcal{D}_{\text{base}} \cap (\text{intension}(C) \cap \text{intension}(C')) = \text{intension}(B(C'))$$

Ce cas est donc ramené au cas 1.

Si $f=f'$

$$\text{intension}(V_f(C)) \cap \text{intension}(V_{f'}(C')) = (\mathcal{D}_f \cup \mathcal{A}C_f) \cap (\text{intension}(C) \cap \text{intension}(C')) = (\mathcal{D}_f \cup \mathcal{A}C_f) \cap \text{intension}(C'')$$

C'' existe et appartient à $\mathcal{C}$ à cause de notre hypothèse de fermeture.

$$\text{Si } PF_f(C'') \neq \emptyset \text{ alors } V_f(C'') \text{ existe et } \text{intension}(V_f(C)) \cap \text{intension}(V_{f'}(C')) = \text{intension}(V_f(C''))$$

$$\text{Si } PF_f(C'') = \emptyset \text{ alors } \mathcal{A}C_f(C) = \emptyset^{12} \text{ et } \text{intension}(V_f(C)) \cap \text{intension}(V_{f'}(C')) = \mathcal{D}_{\text{base}} \cap \text{intension}(C'') = \text{intension}(B(C''))$$

Ce dernier cas est aussi ramené au cas 1.

Insertion des classes-vues dans le schéma global

Le schéma global constitué des classes et classes-vues étant fermé par rapport à l'opérateur « $\wedge$ », les classes-vues sont ensuite insérées dans le schéma global en respectant la relation de subsomption de MultiView.

La figure 4.8 montre le schéma global de notre exemple, une fois les classes-vues insérées.

Propriétés obtenues

- Toutes les propriétés et tous les attributs calculés relatifs à des accès inter-plans ayant augmenté l'intension d'une classe doivent également augmenter l'intension de ses sous-classes. Ceci est réalisé en les incluant dans la définition des classes-vues augmentant l'intension des sous-classes, ce qui est exprimé grâce à la syntaxe **refine**[<classe>:p](C). Par exemple, les classes-vues « Client_Comptable » et « Fournisseur_Comptable » incluent la propriété « contact_comptable » définie dans « Part_externe_Comptable ». Il en résulte que ces deux classes-vues sont classifiées comme sous-classes de « Part_externe_Comptable », préservant les relations d'héritage décrites dans le schéma de base. Comme dans le cas de la réalisation par extraction, cette propriété est issue de la construction automatique des classes-vues guidée par le résultat de la conception.

- Les classes-vues entretiennent une relation d'héritage avec les classes du schéma de base, à l'image de l'héritage des descriptions de base par les différentes descriptions fonctionnelles et à la différence de la mise en œuvre par extraction.

12. Si une classe ne détient pas de partie fonctionnelle, elle ne détient pas non plus d'accès inter-plans relative à la même fonction.

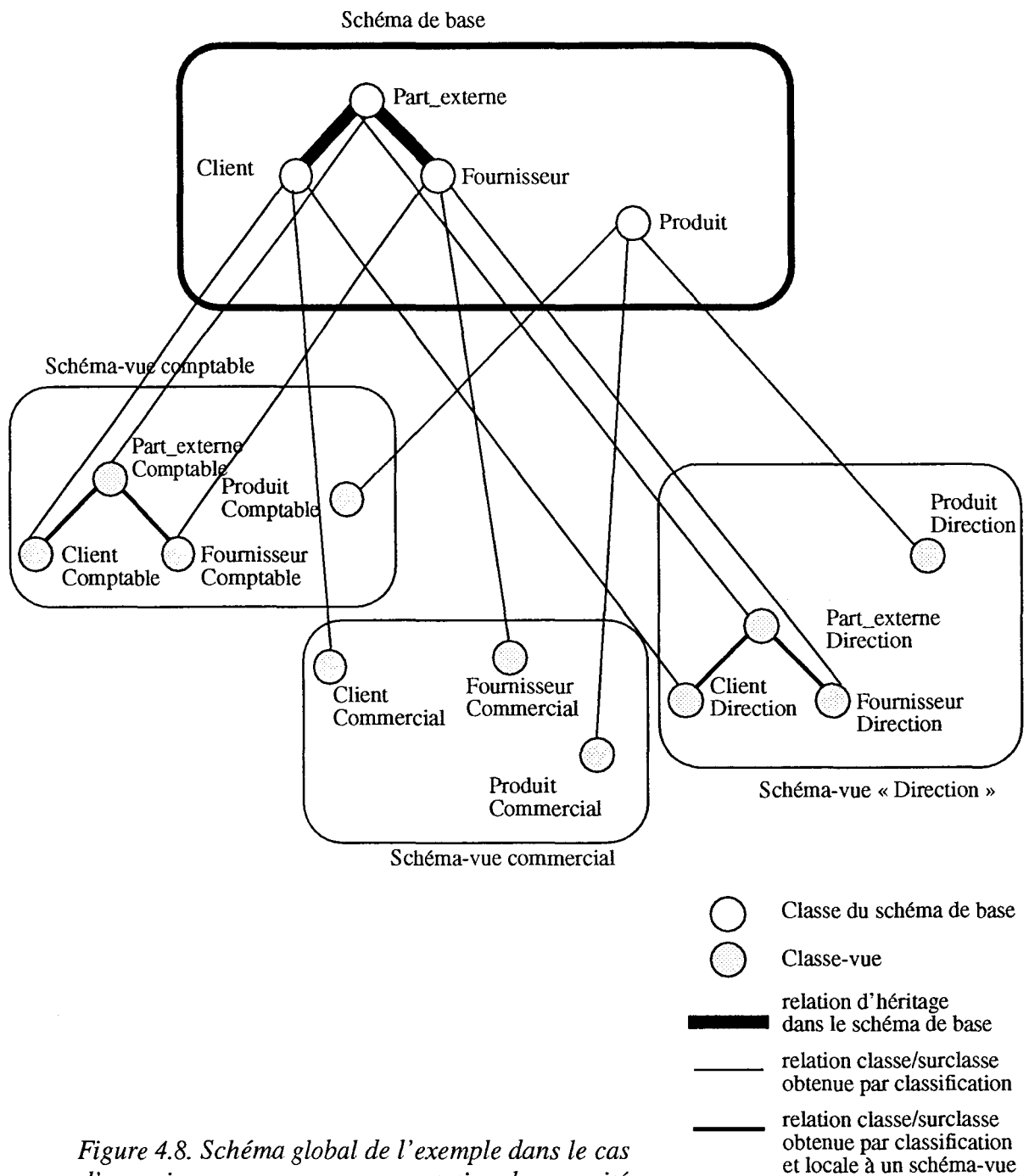


Figure 4.8. Schéma global de l'exemple dans le cas d'une mise en œuvre par augmentation de capacité

4.2.2.3 Réalisation des contextes par les schémas-vues de MultiView

Les schémas-vues sont définis exactement de la même façon que dans le cas de la mise en œuvre par extraction (cf 4.2.1.3). Ils possèdent les mêmes propriétés, à savoir de détenir le statut de contexte fonctionnel et de posséder la même structuration en classes que le schéma de base. La figure 4.8 illustre ces différents schémas-vues et montre qu'ils préservent la structuration en classes du schéma de base.

4.2.3 Réalisation des contraintes de participation des instances

Dans un schéma-vue, les classes-vues par sélection en extension permettent de déterminer les instances pertinentes pour les clients de ce schéma. Il semble donc intéressant d'utiliser cette technique de sélection pour réaliser les contraintes de participation des instances à un contexte fonctionnel. Cependant, ceci pose deux problèmes importants :

- en CROME, c'est le prédicat de la classe d'instanciation qui est utilisé pour déterminer la participation d'une instance¹³ à une fonction. Si l'on regarde finement la sémantique de l'opérateur de sélection en extension (**select**), celui-ci s'applique à l'ensemble des instances de la classe-vue, indépendamment de leur classe d'instanciation¹⁴. La systématisation de l'opération de sélection à un ensemble de classes prenant en compte la classe d'instanciation n'est donc pas immédiate à obtenir ;
- il faut procéder à l'adaptation des relations partagées inter-entités dans les contextes consommateurs à l'aide de mécanismes appropriés afin de garantir l'intégrité référentielle.

L'adaptation des relations de composition dans les schémas-vues où sont effectuées des sélections sur leurs extrémités a été présentée dans le cadre de NO² [Geppert93] en 2.2.4. Cette solution peut être retenue pour les relations inter-entités.

La réalisation des contraintes de participation nous conduit à générer des classes-vues d'extensions égales aux extensions fonctionnelles des classes du domaine à l'aide de l'opérateur de sélection. Ceci se fait de la façon suivante¹⁵ :

- En utilisant l'opérateur de différence, on calcule l'extension stricte de la classe du domaine. L'extension stricte est obtenue par différence entre l'extension de la classe et l'extension de chacune de ses sous-classes directes ;
- On applique l'opérateur de sélection sur cette extension stricte pour obtenir l'extension fonctionnelle stricte de la classe. Pour un contexte donné, le prédicat de sélection doit être la conjonction du prédicat propre à la classe et à toutes ses surclasses, ainsi que du prédicat propre à la classe et à toutes ses surclasses pour tous les contextes fournisseurs d'un accès inter-plans¹⁶. Les contraintes de participation relatives aux contraintes de cardinalité des relations partagées ne sont pas prises en compte. En effet, ni MultiView ni COCOON n'intègrent les contraintes de cardinalité ;
- On calcule l'extension fonctionnelle par l'opérateur d'union. L'extension fonctionnelle est obtenue par l'union de l'extension fonctionnelle stricte de la classe et des extensions fonctionnelles de ses sous-classes directes.

La figure 4.9 montre un exemple de réalisation par les techniques de vues par sélection en extension sur un cas simple avec héritage simple dans le schéma de base¹⁷. Par exemple, la classe-vue « A' » détient l'extension stricte de la classe « A ». La classe-vue « A'' » détient l'extension fonctionnelle stricte de la classe « A ». Ensuite, la classe-vue « A''' » détient

13. Ceci provient de la systématisation des contraintes de participation des instances des classes du domaine relativement aux contextes fonctionnels.

14. Ce qui est tout à fait normal : une classe-vue est uniquement définie relativement à sa classe de base mais pas par rapport à un ensemble de classes. Une sélection relative à la classe d'instanciation des instances est donc impossible.

15. Pour les définitions d'extension stricte, d'extension fonctionnelle stricte et de sous-classe directe, voir 3.5.3.

16. Il faut prendre en compte les contextes fournisseurs en faisant une fermeture transitive, c-à-d en prenant en compte les contextes fournisseurs des contextes fournisseurs, etc.

17. On suppose ici l'absence d'accès inter-plans.

l'extension fonctionnelle de la classe « A ». Il faut noter qu'il n'est pas nécessaire de définir une classe-vue pour obtenir l'extension stricte de la classe « C » car celle-ci ne possède de sous-classe. Ainsi, la classe « C'' » détient directement l'extension fonctionnelle de la classe « C ».

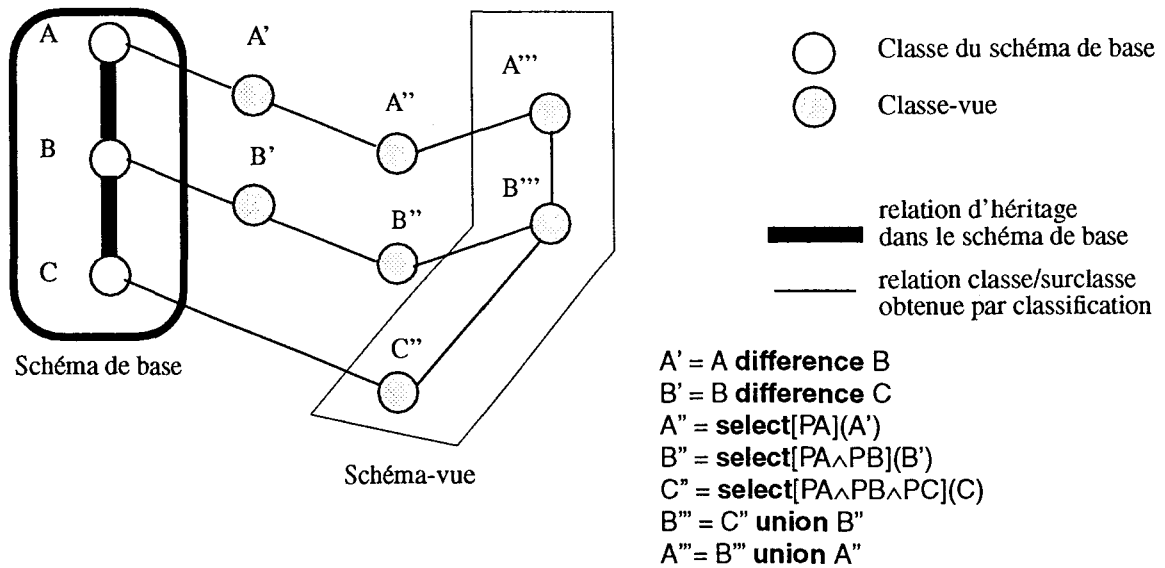


Figure 4.9. Détermination des extensions fonctionnelles dans un cas simple

PA, PB et PC étant respectivement les prédicats de participation des classes A, B et C

Il est ensuite possible de déterminer les classes-vues qui détiennent les descriptions pertinentes pour les clients d'une fonction en procédant à une mise en œuvre par extraction ou par augmentation de capacité. En effet, les calculs d'extension que nous venons de faire peuvent s'appliquer à des classes du schéma de base quelle que soit leur intension. Cette dernière peut être égale à la description de base ou à la fusion de la description de base et toutes les parties fonctionnelles.

Par ailleurs, toutes les classes intermédiaires générées pour le calcul de l'extension fonctionnelle ne changent rien à la problématique de la fermeture de la hiérarchie des types. Toutes ces classes n'engendrent aucun nouveau type. En effet, les opérateurs de différence et de sélection n'engendrent pas de nouveau type et l'opérateur d'union n'est employé qu'avec des sous-classes d'une classe. De ce fait, il est évident de vérifier que, dans ce cas, il n'engendre pas non plus de nouveau type.

Enfin, la réalisation des schémas-vues reste inchangée par rapport à ce qui a été présenté en 4.2.1.3. Les schémas-vues détiennent toujours le statut de contexte fonctionnel.

Exemple

Nous modifions l'exemple des figures 4.2, 4.3 et 4.4 pour que seuls les clients ayant acheté au moins un produit dont le prix est supérieur à 1000 F participent à la fonction comptable. La figure 4.10 montre le prédicat de participation de la classe « Client » dans la fonction comptable.

```

function Comptable interface increment Client
{
  function extent condition:
    exists p in a_achete: p.prix_tarif > 1000;
  function part:
    {
      attribute integer encours_a_regler;
      attribute integer chiffre_affaires;
    }
}
f

```

Figure 4.10. Prédicat de participation de la classe « Client » dans la fonction comptable

Pour pouvoir prendre en compte cette modification dans le cadre des deux mises en œuvre précédentes, il faut définir les classes-vues suivantes¹⁸ :

```

define view Client_Comptable''' as select[(select[(p ∈ a_achete(c)) and (prix_tarif(p) > 1000)](p:Produit)≠∅ ](c:Client));
define view Part_externe_Comptable' as Part_externe difference Client;
define view Part_externe_Comptable''' as Part_externe_Comptable' union Client_Comptable''';

```

Les classes-vues « Client_Comptable''' » et « Part_externe_Comptable''' » doivent ensuite être utilisées à la place des classes « Client » et « Part_externe » pour les définitions des classes-vues « Client_Comptable » et « Part_externe_Comptable » en 4.2.1.1 et 4.2.2.1.

Il faut noter que la relation « a_achete/a_ete_achete_par » doit, en plus, être contextualisée dans le schéma-vue relatif à la fonction comptable à l'aide d'attributs calculés (cf 3.7.2 et 2.2.4).

Si l'on imagine un cas plus général que cet exemple où le schéma de base est plus complexe incluant des relations d'héritage multiple, cette mise en œuvre aboutit alors à un schéma global dont la complexité est considérable à cause du nombre très élevé de classes-vues générées. Cependant, celui-ci peut être généré automatiquement. Les schémas-vues ne sont pas plus complexes que dans les cas où les contraintes de participation ne sont pas prises en compte.

4.2.4 Conclusion

Bilan

Dans cette section, nous avons étudié comment traduire une conception en CROME par les techniques de vues de bases de données à objets, et plus précisément dans le cadre de MultiView. Nous avons retenu deux principes de traduction qui ont donné lieu à deux mises en œuvre distinctes : une mise en œuvre par extraction et une par augmentation de capacité. Ces deux mises en œuvre montrent qu'il est ainsi possible de générer automatiquement les schémas-vues à partir de la conception à un niveau logique.

18. Nous avons simplifié le nombre de classes-vues en tenant compte du fait qu'il n'y a pas de sélection propre aux entités « Part_externe » et « Fournisseur » dans la fonction comptable.

Propriétés obtenues

Dans les deux mises en œuvre, suite à leur classification automatique, les classes-vues entretiennent des relations d'héritage isomorphes à celles entretenues dans le schéma de base par les classes dont elles sont issues. Cette propriété intéressante est due au fait que les classes-vues résultent, de façon automatique, de la conception en CROME. Dans le cas de la mise en œuvre par extraction, ce résultat est dû au fait que les propriétés masquées dans la définition d'une classe-vue relative à une classe du schéma de base et à une fonction sont également masquées dans la définition des classes-vues relatives aux sous-classes de cette classe et à la même fonction. Dans le cas de la mise en œuvre par augmentation de capacité, les propriétés qui augmentent une classe du schéma de base dans la définition de sa classe-vue relativement à une fonction augmentent également les sous-classes de cette classe dans la définition de leur classe-vue relativement à la même fonction.

Les schémas-vues ainsi obtenus détiennent l'ensemble des descriptions et des instances d'une fonction nécessaires à ses clients et détiennent donc un statut de contexte fonctionnel. Par conséquent, ils offrent un moyen de structuration du schéma global de la base de données.

La prise en compte des contraintes de participation des instances aux contextes est possible dans les deux mises en œuvre. Cependant, elle aboutit à un schéma global extrêmement complexe. Cette complexité provient de l'impossibilité d'utiliser directement l'opérateur de sélection pour obtenir les extensions fonctionnelles. Une extension fonctionnelle ne s'obtient pas en appliquant le même prédicat sur toutes les instances d'une classe mais en appliquant le prédicat de participation de la classe d'instanciation de chacune des instances.

Comparaison des deux mises en œuvre

Nous avons pu déterminer que la mise en œuvre par augmentation est plus proche de la conception en CROME par les points suivants :

- le schéma de base ne détient que les descriptions du plan de base. Les attributs relatifs aux plans fonctionnels sont introduits dans les classes-vues et donc dans les schémas-vues. Dans le cas de la mise en œuvre par extraction, une classe du schéma de base détient l'ensemble des descriptions de son entité du domaine, c-à-d la description de base et toutes les parties fonctionnelles ;

- les classes-vues sont des sous-classes des classes du schéma de base. Dans le cas de la mise en œuvre par extraction, c'est l'inverse, ce qui ne correspond pas au résultat de la phase de conception ;

- la réalisation des accès inter-plans diffère dans les schémas-vues correspondant aux contextes fournisseurs et dans ceux correspondant aux contextes consommateurs, reflétant mieux les différences qui existent, au niveau de la conception, entre contextes fournisseurs et contextes consommateurs d'un accès inter-plans. Seuls les clients des schémas-vues correspondant aux contextes fournisseurs peuvent fournir la valeur des propriétés correspondant aux accès inter-plans. Dans le cas de la mise en œuvre par extraction, l'intension d'une classe-vue relative à une entité et à une fonction détient les propriétés de la description fonctionnelle correspondante et les propriétés définies dans les accès inter-plans associés. Cette intension n'est pas structurée et les propriétés définies dans les accès inter-plans ont le même statut que celles de la description fonctionnelle. Il n'y a donc pas de distinction entre contextes fournisseurs et contextes consommateurs des accès inter-plans.

Ces trois points confère à la mise en œuvre par augmentation un caractère plus modulaire et plus évolutif que celle par extraction.

La mise en œuvre par augmentation est plus modulaire que celle par extraction. Pour une entité du domaine, sa description est structurée par une classe du schéma de base et pour chaque schéma-vue relatif à une fonction, par une classe-vue. Cette structuration obtenue est isomorphe à la structuration interne des classes du domaine du niveau logique de conception qui est l'origine de leur modularité interne (cf 3.7.1). Par ailleurs, les fournisseurs des valeurs des attributs spécifiques à une fonction ne peuvent être que les clients de cette fonction, même si ces attributs font l'objet d'accès inter-plans. En effet, les clients du contexte consommateur d'un accès inter-plans ne peuvent fournir la valeur de l'attribut consommé. Ceci permet de respecter l'interaction entre modules relative aux accès inter-plans.

Cette modularité rend la mise en œuvre par augmentation plus évolutive que celle par extraction : l'ajout d'un nouveau contexte fonctionnel ne nécessite pas de modifier le schéma de base. Dans le cas de la mise en œuvre par extraction, chaque classe du schéma de base doit être modifiée pour détenir les descriptions relatives au nouveau contexte.

Conclusions sur les techniques de vues des bases de données à objets

Ces deux mises en œuvre possèdent la propriété de préserver les relations d'héritage du schéma de base dans les schémas-vues générés. Cette propriété, comme nous l'avons vu ci-dessus, est due à la génération automatique des schémas-vues à partir d'une conception en CROME. Dans le cas classique, l'administrateur d'une base de données qui élabore un schéma-vue doit prendre beaucoup de précautions dans la définition des classes-vues pour que cette préservation soit respectée. Au sein d'un schéma-vue, si l'intension d'une classe-vue relative à une classe ne contient pas une propriété contenue dans l'intension de la classe-vue de l'une des surclasses de cette classe, le lien d'héritage entre ces sous-classes est perdu comme l'illustre sur un exemple la figure 4.11. Il en va de même en cas de sélection des instances. Le prédicat de définition de la classe-vue d'une classe doit être plus restrictif que celui de chaque classe-vue relative à une surclasse de cette classe.

Nous avons évoqué le problème de l'utilisation des classes-vues par extraction pour déterminer les extensions fonctionnelles, ces classes-vues ne pouvant être directement utilisées. Ce problème vient de la granularité fine de ces classes-vues : une classe-vue est définie sur une ou plusieurs classes sans prendre en compte la hiérarchie dans laquelle se situent cette classe ou ces classes. La première conséquence de cette granularité fine est le problème de la préservation de la hiérarchie du schéma de base dans les schémas-vues que nous venons de voir. Dans l'exemple de la figure 4.11, le calcul de l'intension de la classe-vue « Produit_grand_public_fab » s'est fait à partir de l'intension de la classe « Produit_grand_public » prise dans sa globalité et sans tenir compte des propriétés héritées de « Produit » ni de leur pertinence vis-à-vis de la fonction « Fabrication ».

Cette première conséquence signifie que, au sein d'un schéma-vue, la définition d'une classe-vue pour une classe ne prend pas automatiquement en compte les définitions des classes-vues relatives aux surclasses de cette classe, qu'il s'agisse d'extraction, d'augmentation de capacité ou de sélection. A l'inverse en CROME, les parties fonctionnelles d'une classe sont héritées dans ses sous-classes ainsi que leur prédicat de participation.

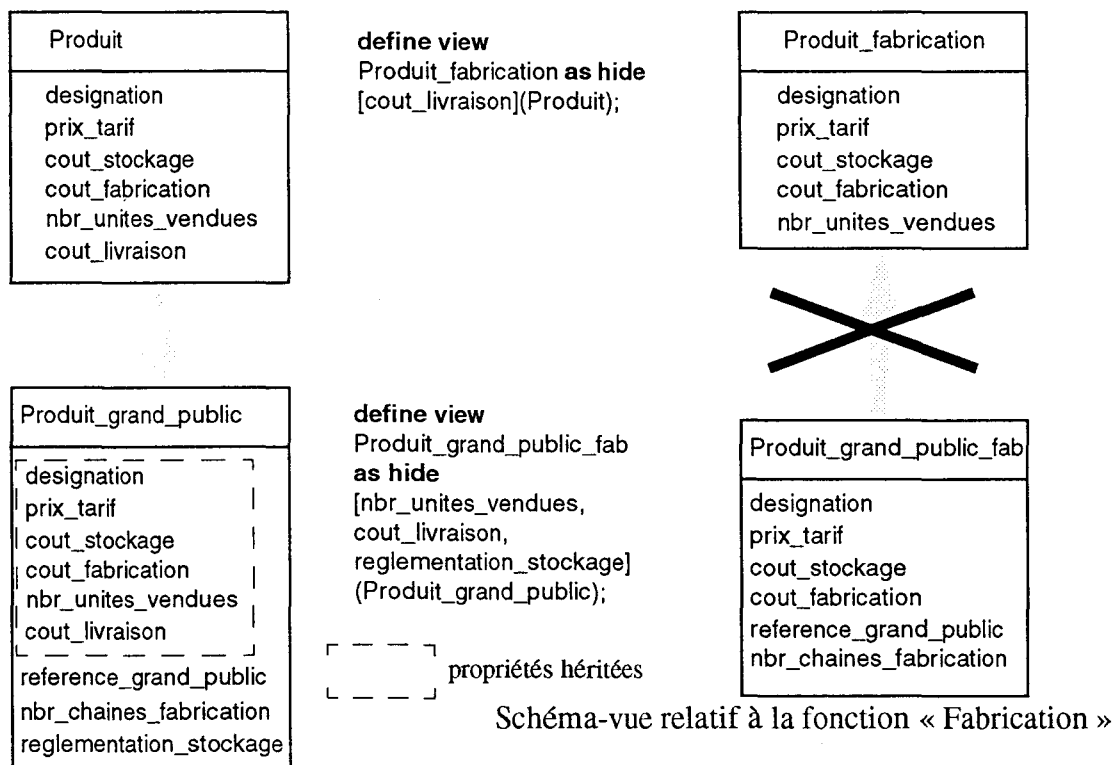


Figure 4.11. Relation d'héritage non préservée entre deux classes-vues

La seconde conséquence concerne le rôle de la classe d'instanciation. En CROME, la classe d'instanciation détermine l'ensemble des descriptions qui sont conférées à ses instances. En effet, d'une part, une classe hérite de toutes les descriptions issues de ses surclasses. D'autre part, c'est la classe d'instanciation qui détermine l'état fonctionnel d'une instance et donc l'ensemble des parties dont elle est dotée.

Les techniques de vues ne préservent nullement le rôle de la classe d'instanciation. Une instance peut appartenir et être décrite par plusieurs classes et classes-vues indépendantes vis-à-vis de la relation d'héritage. La classe d'instanciation n'a aucun rôle privilégié dans cet ensemble¹⁹. La pertinence de la participation d'une instance à un schéma-vue n'est pas déterminé par sa classe d'instanciation ou par une autre classe. Une instance peut participer à certaines classes et classes-vues d'un schéma-vue suivant des critères propres à chacune de ces classes.

Enfin, nous pouvons rappeler que les techniques de vues n'offrent aucun mécanisme d'interaction entre schémas-vues comme nous l'avons vu en 3.7.3. Ceci est dû au fait que chaque schéma-vue est conçu de façon totalement indépendante des autres. Il n'est donc pas possible de spécifier les schémas-vues fournisseurs de valeurs d'attributs communs ou d'instances (c-à-d de spécifier des schémas-vues d'instanciation).

De même, les techniques de vues ne prévoient pas la gestion des contraintes d'intégrité au sein des schémas-vues. L'interaction et les contraintes d'intégrité sont deux techniques qui sortent du cadre classique des vues.

19. La meilleure preuve est qu'en COCOON, l'opérateur « lose » qui permet à une instance de ne plus être décrite par un type peut s'appliquer aussi à son type d'instanciation.

4.3. Mise en œuvre dans les SGBD à objets couplés au langage Java

Dans la section précédente, nous avons proposé une mise en œuvre dans le cadre des techniques de vues des bases de données à objets. Une telle mise en œuvre n'est malheureusement guère envisageable dans le cadre des SGBD à objets du commerce. En effet, à la différence des SGBD relationnels, les SGBD à objets du commerce n'intègrent quasiment pas les techniques de vues. Ceux-ci sont basés sur un couplage (« binding ») avec un ou plusieurs langages de programmation à objets auxquels ils offrent la plupart des propriétés des SGBD comme la persistance, la gestion des extensions, les requêtes, les transactions, la sécurité. Le langage à objets constitue le langage de spécification du schéma de bases de données. Il est également employé comme langage de manipulation des données. Les propriétés des SGBD (persistance, extensions, requêtes, transaction, sécurité) sont offertes au travers d'objets spécifiques mis à la disposition des clients d'une base de données.

La mise en œuvre de CROME dans l'un de ces SGBD doit alors prendre en compte :

- le langage à objets, support pour la spécification et la manipulation des entités. Il sera nécessaire de trouver des caractéristiques qui lui sont propres afin de pouvoir conférer des descriptions multiples aux entités du domaine ;
- les objets spécifiques aux bases de données. Il sera nécessaire d'adapter ceux-ci, notamment ceux liés à la gestion des extensions afin de prendre en compte les extensions fonctionnelles²⁰.

Nous avons choisi comme cible le langage à objets « Java » qui va nous servir de support pour la génération du schéma de la base de données. Ce schéma pourra être utilisé avec tout SGBD à objets couplé avec Java. Par contre, les objets de gestion des extensions étant spécifiques à un SGBD particulier, nous avons choisi ceux du SGBD POET²¹ comme cible pour adaptation à la gestion des extensions fonctionnelles²². La mise en œuvre a donné lieu à un prototype appelé CromeJava qui a été implanté au dessus du SGBD POET. Ce prototype est principalement constitué de deux outils :

- un outil de génération de schémas en Java ;
- un browser pour la conception de schémas orientée « fonctions ».

L'outil de génération construit des schémas en Java à partir de schémas décrits dans le langage ODL modifié pour les besoins de CROME²³ (cf chapitre 3) ou à partir des descriptions fournies par le browser. Ces schémas sont constitués de classes et de paquetages Java. L'outil fournit aussi les adaptations des objets de gestion des extensions du SGBD POET pour la prise en compte des extensions fonctionnelles.

Dans ce chapitre, nous présentons successivement :

- l'utilisation du prototype au travers de l'outil de génération et du browser cités ci-dessus (4.3.1) ;
- une étude des schémas générés (4.3.2). Cette génération s'appuie sur les notions d'interfaces et des paquetages « Java » ;
- l'adaptation de la classe de POET qui permet d'obtenir les extensions des classes pour prendre en compte les extensions fonctionnelles (4.3.3).

20. Nous ne nous intéressons pas ici aux problèmes de transaction et de sécurité.

21. Notons qu'un avantage de POET est le support de l'évolution de schémas.

22. Il faut noter que cette adaptation des objets de gestion des extensions ne représente qu'une petite partie de cette mise en œuvre ce qui faciliterait un éventuel portage vers un autre SGBD.

23. Ce langage est appelé ici « CromeODL ».

Nous introduirons dans le cadre de cette réalisation les classes d'entités locales à un contexte (4.3.1.3).

4.3.1 Utilisation du prototype CromeJava

4.3.1.1 Outil de génération de schémas en Java

La mise en œuvre dans le cadre d'un SGBD couplé avec le langage Java est basée sur un outil²⁴ de génération de schémas décrits dans ce dernier langage. Cet outil peut soit générer un schéma à partir d'un ou de plusieurs fichiers « sources » écrits en CromeODL, soit à partir de descriptions fournies par un browser spécifique. La figure 4.12 illustre ces deux possibilités.

Chaque schéma ainsi généré est constitué d'un ou plusieurs fichiers Java qui sont destinés au préprocesseur de SGBD ainsi qu'au compilateur Java. Une fois préprocessés et compilés, ces fichiers donneront lieu à un ensemble de classes qui pourront être exploitées par les clients des différentes fonctions. Ces clients sont ici d'autres objets « Java ». Nous étudierons plus précisément dans les sections suivantes les schémas générés.

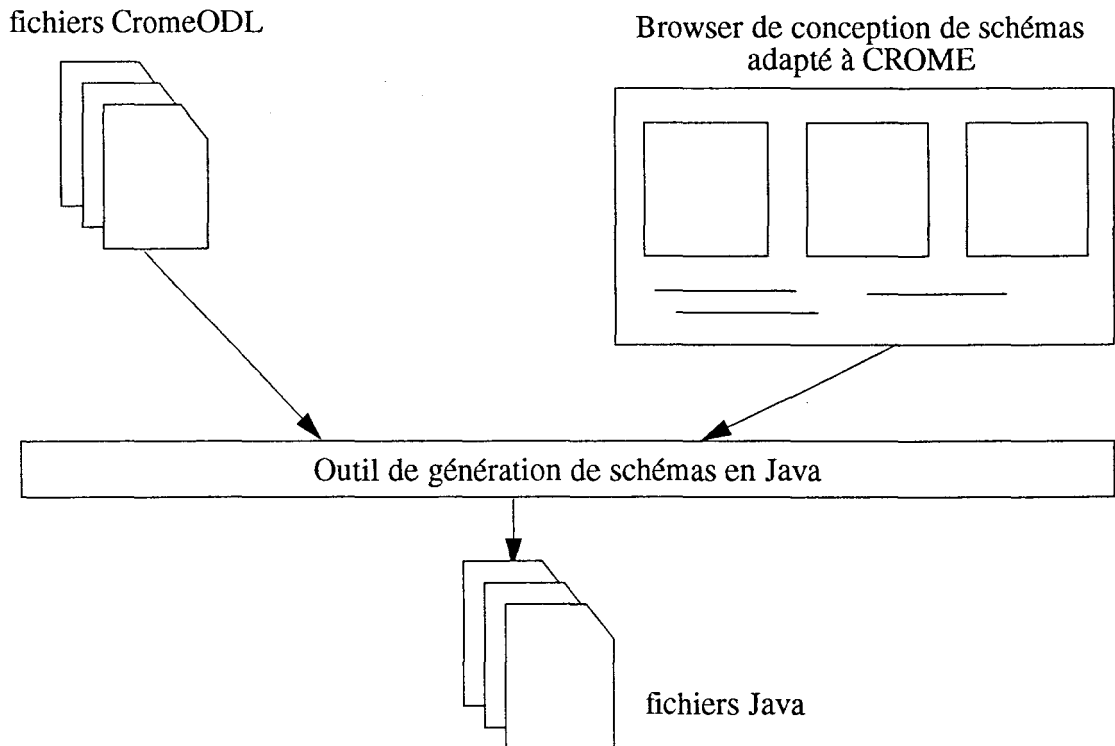


Figure 4.12. l'outil de génération de schémas en Java à partir d'une conception en CROME

4.3.1.2 Le browser de schémas

Le browser de schémas est un outil de conception de schémas orientée « fonctions » qui prend en compte les aspects de structuration des classes du domaine introduits au chapitre 3. La figure 4.13 représente une copie d'écran de ce browser dans le cadre de l'exemple de l'annexe A.

24. Cet outil est disponible à l'adresse Internet <http://www.lifl.fr/DOWNLOAD/>

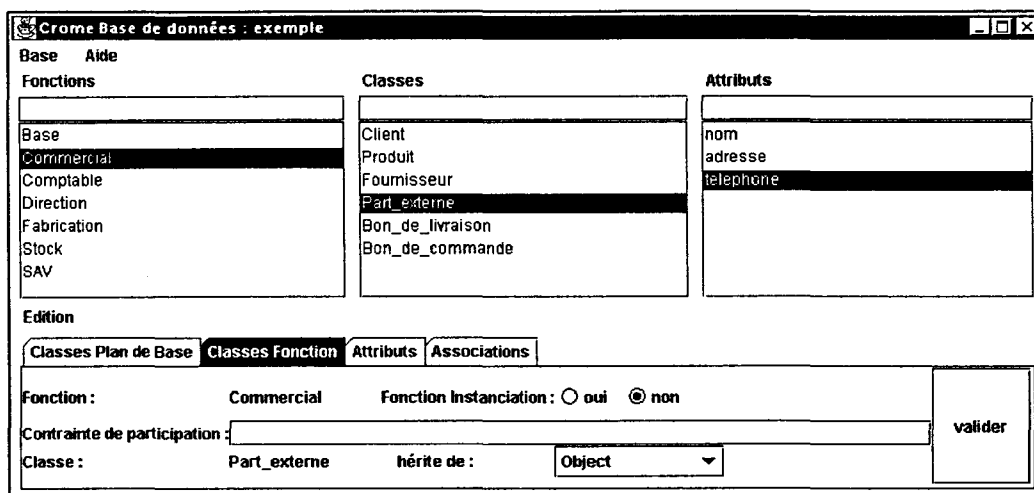


Figure 4.13. Browser orienté « fonction »

Le browser se compose de trois listes verticales et d'une zone d'édition horizontale. La liste en haut à gauche contient la liste des fonctions. La liste du milieu contient la liste des classes dont l'entité correspondante participe à la fonction sélectionnée. La troisième liste à droite contient la liste des attributs de la partie fonctionnelle et des accès inter-plans de la classe sélectionnée correspondant à la fonction sélectionnée. Enfin, la partie inférieure permet d'éditer les descriptions de base, les parties fonctionnelles ainsi qu'à un niveau de granularité plus fin, les attributs et les relations. Les accès inter-plans, les modalités de fourniture de la valeur des attributs du plan de base, les contraintes de participation, les contextes d'instanciation, les contraintes d'intégrité peuvent également être précisés. Pour les relations, il est possible de fixer des extrémités avec une cardinalité minimale de zéro ou de un et une cardinalité maximale de un ou de « n »²⁵. L'ensemble de ces possibilités d'édition confère au browser le statut d'un outil de conception orienté « fonctions ».

L'utilisation du browser facilite et accélère la conception de la base en offrant une vérification automatique d'un certain nombre de contrôles qui limite les erreurs de conception. La liste des contrôles implantés est la suivante :

- le prédicat de participation à un contexte d'instanciation est une tautologie. Par la suite, le browser pourra vérifier qu'il n'y a pas d'autres contraintes de participation (cf 3.5.5.3) ;
- la suppression d'une partie fonctionnelle dans un contexte n'est possible que si celle-ci ne détient plus d'attributs qui soient des extrémités d'une relation locale avec d'autres entités ;
- la suppression d'une propriété d'une partie fonctionnelle exploitée dans d'autres contextes au travers d'accès inter-plans provoque la suppression automatique de ces accès inter-plans ;
- la suppression d'une classe du domaine entraîne la suppression de sa description de base et toutes ses parties fonctionnelles ;
- la suppression d'un plan fonctionnel entraîne la suppression de toutes les parties fonctionnelles qui le composaient.

Il faut noter que le browser dispose d'une capacité de stockage des schémas dans un

25. C'est-à-dire « sans cardinalité maximale ».

format qu'il lui est propre. Il est également capable d'exporter et d'importer des schémas décrits dans le langage CromeODL. Dans ce dernier cas, il peut d'ailleurs être utilisé indépendamment de la génération de schémas en Java pour concevoir des schémas en CromeODL.

4.3.1.3 Prise en compte des entités locales aux fonctions

Le browser comme l'outil de génération offre la possibilité de définir des entités locales aux fonctions [Debrauwer97]. De telles entités ne sont pertinentes qu'à une seule fonction. Elles y sont donc décrites par une classe localisée dans le contexte fonctionnel correspondant et que seuls les clients de ce contexte peuvent exploiter. Cette classe ne possède aucun lien d'héritage avec l'une des classes du domaine. Elle n'est pas structurée. Il est possible de lui associer une ou plusieurs contraintes d'intégrité comme celles qu'il est possible d'associer aux descriptions fonctionnelles (cf 3.6.4).

Il est possible de définir une relation entre une entité du domaine et une entité locale à une fonction. Cette relation est alors une relation locale au contexte qui n'induit pas de contraintes de participation spécifiques. Son intégrité référentielle peut être vérifiée comme celle des relations locales inter-entités du domaine (cf 3.6.2.1.2). Il en va de même pour la vérification de ses contraintes de cardinalité (cf 3.6.3.1.2). Par contre, une telle relation ne peut faire l'objet d'un accès inter-plans depuis une autre fonction. En effet, un tel accès inter-plans supposerait la connaissance de l'entité locale dans la fonction consommatrice, ce qui est contraire à la définition des entités locales.

La figure 4.14 montre un exemple d'entité locale introduite dans le contexte « Fabrication », à savoir l'entité « Machine_fabrication ». La forme « **function** <function_name> **interface** <type_name> » permet d'introduire une classe locale à un contexte et de spécifier son intension.

```
function Fabrication interface Machine_fabrication
{
    attribute integer poids;
    relationship set<Produit_grand_public>[1] fabrique inverse Produit_grand_public::est_fabrique_par;
}
function Fabrication interface increment Produit_grand_public
{
    function extent condition: nbr_unites_vendues > 20000;
    function part:
    {
        attribute integer nbr_chaines_fabrication;
        relationship set<Machine_fabrication>[1] est_fabrique_par inverse Machine_fabrication::fabrique;
    }
}
```

Figure 4.14. Classe locale « Machine_fabrication » au sein du contexte « Fabrication »

4.3.2 Génération des schémas en Java

4.3.2.1 Interfaces fonctionnelles et classes d'implantation

4.3.2.1.1 Principes

Nous utilisons les interfaces du langage Java afin de conférer des descriptions multiples aux entités du domaine. Pour rappel, une interface est une classe totalement abstraite qui ne décrit que des spécifications publiques de méthode [Niemeyer97-p167]. Pour une entité, nous

allons créer une interface « Java » appelée interface fonctionnelle pour chaque fonction où participe ladite entité plus une interface relative au plan de base que nous appelons interface de base. Chaque interface fonctionnelle d’une entité hérite de l’interface de base. Enfin, pour chaque entité, une classe implante les spécifications issues de l’ensemble de ses interfaces fonctionnelles et son interface de base. la figure 4.15 illustre l’ensemble des interfaces et classes « Java » générées pour une entité du domaine.

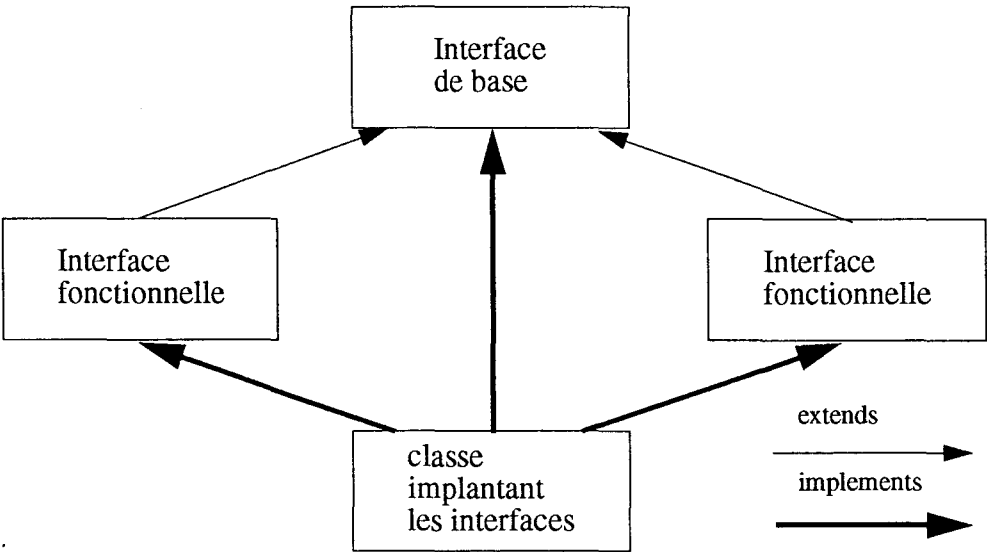


Figure 4.15. Interfaces fonctionnelles et de base d’une entité. Classe d’implantation

Remarque relative aux classes locales

Pour une entité locale à une fonction, il n’est généré qu’une interface destinée aux clients de ladite fonction et une classe d’implantation.

4.3.2.1.2 Interfaces fonctionnelles et de base

Chaque interface fonctionnelle correspond à une description fonctionnelle et aux accès inter-plans associés. Une interface étant une classe abstraite ne pouvant détenir que des spécifications de méthode, une interface fonctionnelle est constituée par un ensemble de spécifications d’accesseurs sur les propriétés de la description fonctionnelle et sur les propriétés relatives aux accès inter-plans. Ces accesseurs sont publics et peuvent donc être invoqués par les clients de la fonction. Chaque description de base détient les spécifications d’accesseurs sur les propriétés contenues dans la description de base.

De façon plus précise, l’interface de base ne contient que les accesseurs en lecture sur les propriétés de la description de base. Ces accesseurs sont hérités dans toutes les interfaces fonctionnelles. Par contre, les accesseurs en écriture ne sont spécifiés que dans les interfaces relatives aux fonctions fournissant la valeur de la propriété de base.

Les propriétés introduites dans les parties fonctionnelles font l’objet dans les interfaces fonctionnelles d’un accesseur en lecture et d’un accesseur en écriture.

Quant aux accès inter-plans, ils sont traduits par la seule présence d’un accesseur en lecture dans les interfaces relatives aux fonctions consommatrices.

La table 4.2 explicite les accesseurs générés dans les différentes interfaces « Java ».

Table 4.2: Accesseurs générés dans les interfaces « Java »

Type de propriété	accesseur en lecture	accesseur en écriture
propriété de la description de base	dans les interfaces de base	dans les interfaces relatives aux fonctions fournisseuses de la valeur de la propriété
propriété d'une partie fonctionnelle	• dans l'interface fonctionnelle correspondante • dans les interfaces relatives aux fonctions consommatrices de cette propriété	dans l'interface fonctionnelle correspondante

La figure 4.16 illustre les interfaces « Java » générées pour l'entité « Part_externe » dans le cadre de l'exemple de l'annexe A. Le nommage des accesseurs respecte le protocole des Javabeans, c'est-à-dire que les accesseurs en lecture sont préfixés par « get » et ceux en écriture par « set ». Les en-têtes de paquetages ont été ajoutées pour montrer les fonctions auxquelles sont associées les différentes interfaces « Java ».

```
package exemple.Base ;

public interface Part_externe {
    String getNom();
    String getAdresse();
    String getTelephone();
}

package exemple.Direction ;

public interface Part_externe extends exemple.Base.Part_externe {
    String getInterlocuteur();
    void setInterlocuteur(String vString);
}

package exemple.Fabrication ;

public interface Part_externe extends exemple.Base.Part_externe {
    String getContact_technique();
    void setContact_technique(String vString);
}
```

Figure 4.16. Ensemble des interfaces fonctionnelles et de l'interface de base pour l'entité « Part_externe »

4.3.2.1.3 Classes d'implantation

- Chaque classe d'implantation détient :
- toutes les propriétés relatives à l'entité associée, c'est-à-dire celles issues de la description de base et des différentes parties fonctionnelles de cette entité ;
 - l'implantation de l'ensemble des accesseurs spécifiés dans les différentes interfaces.

L'en-tête de la classe d'implantation « Part_externe » ainsi que ses propriétés sont montrées à la figure 4.17. Il faut noter que toutes les classes d'implantation héritent de la classe

« CromeObject » qui décrit des mécanismes communs. Par ailleurs, les propriétés sont encapsulées, faisant ainsi que l'accès à ces dernières doive se faire impérativement par les accesseurs .

```
package exemple ;
```

```
public class Part_externe extends CromeObject implements exemple.Base.Part_externe,  
exemple.Direction.Part_externe, exemple.Fabrication.Part_externe, exemple.SAV.Part_externe {
```

```
protected String nom ;  
protected String adresse ;  
protected String telephone ;  
protected String contact_technique ;  
protected String interlocuteur ;  
...
```

Figure 4.17. En-tête et propriétés de classe d'implantation « Part_externe »

L'implantation d'accesseurs sur les attributs ne pose guère de problèmes. Par contre, celle des accesseurs sur les extrémités de relation est nettement plus complexe car il s'agit de vérifier les contraintes d'intégrité référentielle et les contraintes de cardinalité. En effet, tous les SGBD à objets²⁶ ne supportent pas la gestion de l'intégrité référentielle, ni a fortiori la gestion des contraintes de cardinalité²⁷. Notons aussi qu'il s'agit ici de surcroît de vérifier l'intégrité référentielle et les contraintes de cardinalité dans le cas de relations partagées en respectant les règles introduites en 3.6.

Par ailleurs, ces classes possèdent la faculté d'instanciation. Chaque instanciation est effectuée à l'aide de la primitive « new » du langage Java suivie de l'une des classes d'implantation. Par exemple, dans le contexte commercial, il est possible d'instancier un nouvel élément de la classe « Client » ainsi :

```
import exemple.Commercial. ;  
Client unClient = (Client ) new exemple.Client();
```

Il faut noter que « exemple.commercial.Client » est le nom de l'interface pertinente pour les clients de la fonction commerciale et que « exemple.Client » est le nom de la classe d'implantation. Notons aussi que « exemple » représente ici le nom de la base de données. Il faut remarquer que le constructeur d'une classe d'implantation vérifie que le contexte d'où est issue l'instanciation a bien été défini comme un contexte d'instanciation de la classe.

Les instances de ces classes détiennent l'ensemble des données de la base. Ces instances peuvent être rendues persistantes par la méthode « bind » (cf 4.3.3.1).

4.3.2.2 Paquetages fonctionnels

L'outil de génération crée les paquetages « Java » suivants :

- un paquetage de base contenant les interfaces de base relatives à toutes les entités du domaine. Un tel paquetage est désigné par le nom de la base suffixé par le mot « Base » ;
- un paquetage par fonction contenant les interfaces fonctionnelles relatives à toutes les entités participant à cette fonction. Un tel paquetage est appelé un paquetage fonctionnel. Il est désigné par le nom de la base suffixé par le nom de la fonction ;

26. C'est notamment le cas de POET.

27. Rappelons que, pour simplifier, notre prototype ne supporte que les relations avec une cardinalité minimale de zéro ou de un et une cardinalité maximale de un ou de « n ».

- un paquetage contenant les classes d'implantation relatives à toutes les entités du domaine. Ce paquetage est simplement désigné par le nom de la base.

Dans notre exemple, les paquetages générés sont les suivants :

- « exemple.Base » qui est le paquetage de base ;
- « exemple.Commercial », « exemple.Comptable », « exemple.Stock », « exemple.SAV », « exemple.Fabrication », « exemple.Direction » qui sont les différents paquetages fonctionnels ;
- « exemple » qui est le paquetage contenant les classes d'implantation.

Un paquetage fonctionnel détient l'ensemble des descriptions pertinentes pour les clients de la fonction correspondante. Il peut être utilisé par ces clients qui sont ici, rappelons-le, d'autres objets « Java ». Cette utilisation se fait au travers de l'importation du paquetage fonctionnel dans le paquetage du ou des clients. La figure 4.18 montre un exemple d'utilisation du paquetage fonctionnel commercial par des objets « Java » clients. La clause « import » sert ici à spécifier l'utilisation de ce paquetage fonctionnel. Il faut noter qu'elle permet aussi de simplifier l'écriture, la forme « Client » étant ici équivalente à la forme « exemple.Commercial.Client ».

```
import exemple.Commercial.*;

public class ObjetClient {
    public void doTask() {
        Client unClient = (Client ) new exemple.Client();
        unClient.setNom("Dupont");
    }
    ...
}
```

Figure 4.18. Exemple d'utilisation du paquetage fonctionnel commercial par un client

4.3.2.3 Préservation de la structuration en classes issue de la conception en CROME au sein des paquetages générés

Au sein du paquetage de base, le graphe d'héritage des interfaces de base est isomorphe au graphe d'héritage des classes introduit dans le plan de base lors de la conception en CROME. Rappelons que ce graphe est généré automatiquement au travers des clauses « extend » des interfaces de base.

Au sein d'un paquetage fonctionnel, il en va de même. Le graphe d'héritage entre les interfaces fonctionnelles est isomorphe à celui des interfaces de base. Par ailleurs, chaque interface fonctionnelle hérite de l'interface de base correspondant à la même entité. Par exemple, dans le contexte « Fabrication », l'interface fonctionnelle relative à « Fournisseur » hérite de l'interface fonctionnelle de « Part_externe » et de son interface de base, comme l'illustre la figure 4.19. Ceci montre que l'héritage local entre parties fonctionnelles se traduit ici par un héritage entre interfaces fonctionnelles. Ceci montre également que l'héritage entre une description fonctionnelle et la description de base correspondante est traduit par un héritage entre l'interface fonctionnelle et l'interface de base correspondantes.

Il faut noter qu'il en aurait été de même si l'entité « Fournisseur » n'avait pas été explicitement enrichie. La définition de l'interface n'aurait alors consisté qu'en une en-tête spécifiant l'héritage des deux interfaces. En effet, une telle interface est nécessaire pour exploiter les entités dont la classe n'est pas explicitement mais implicitement enrichie au travers de sa surclasse (cf 3.3.4).

```
package exemple.Fabrication ;

public interface Fournisseur extends exemple.Base.Fournisseur, Part_externe {
    boolean getFournit_seulement_lot_complet_de_pieces();
    void setFournit_seulement_lot_complet_de_pieces(boolean vBoolean);
}
```

Figure 4.19. Interface fonctionnelle de « Fournisseur » dans le contexte « Fabrication »

Enfin, au sein du paquetage contenant les classes d'implantation, le graphe d'héritage entre ces classes d'implantation est isomorphe à celui des paquetages fonctionnels et du paquetage de base. Pour une classe d'implantation, cet héritage lui permet de bénéficier des accesseurs et des propriétés de sa surclasse. Ceci permet de simplifier la génération des classes d'implantation mais engendre alors une contrainte sur le graphe d'héritage dans tous les paquetages : celui-ci ne peut être constitué que de relations d'héritage simple. En effet, le langage « Java » ne supporte pas l'héritage multiple entre classes²⁸.

La figure 4.20 illustre l'isomorphisme entre les graphes d'héritage des différents paquetages et montre aussi les liens entre les interfaces et classes relatives à une même entité (pour ces liens, voir aussi la figure 4.15)

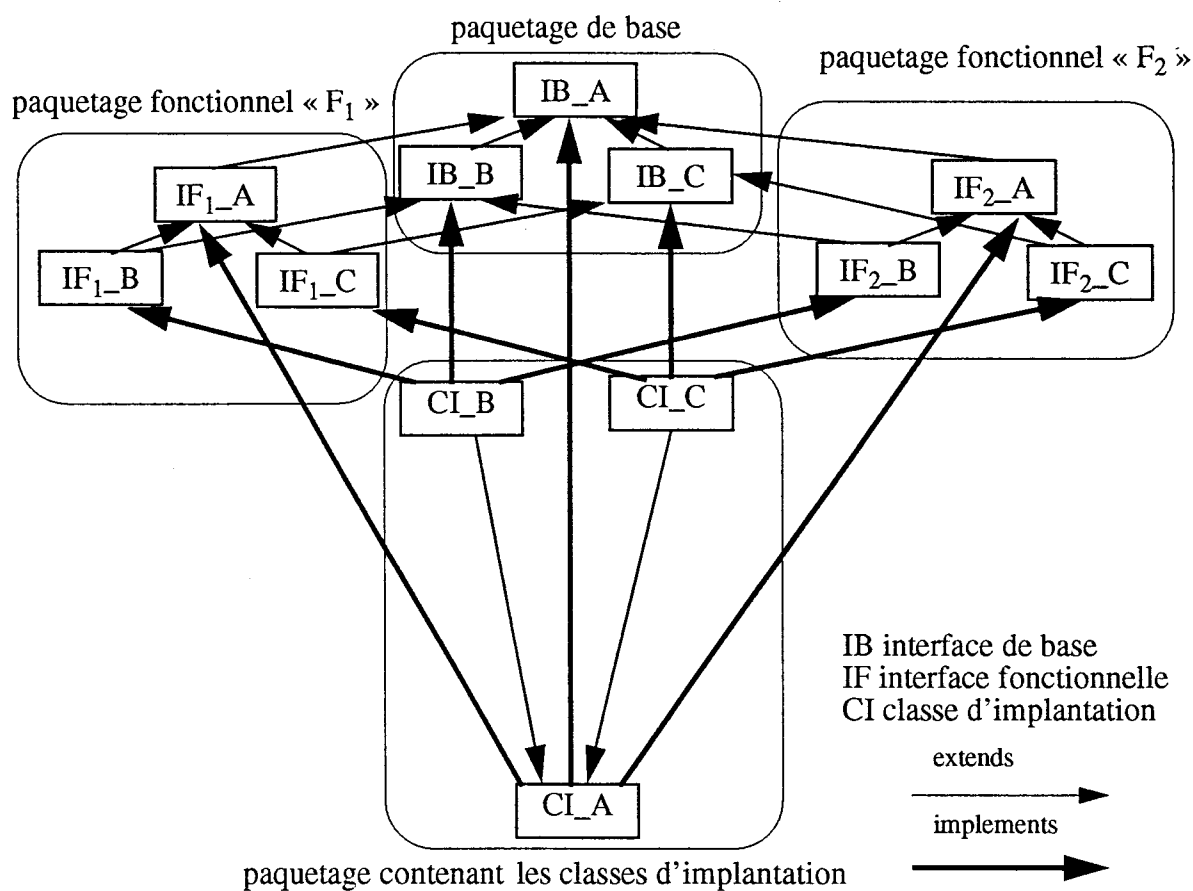


Figure 4.20. Isomorphisme des graphes d'héritage des paquetages. Relations d'héritage entre interfaces fonctionnelles et interfaces de base. Relation d'héritage d'implantation entre classes d'implantation et interfaces.

28. Pour supporter l'héritage multiple, il faudrait alors procéder par recopie au niveau des classes d'implantation, ce qui n'est pas le cas du prototype actuel

4.3.3 Gestion des extensions et des extensions fonctionnelles dans le cadre du SGBD à objets POET

4.3.3.1 Principales caractéristiques du couplage de POET avec Java

Dans le cadre de son couplage avec Java, POET offre la plupart des caractéristiques des SGBD [POET97]. Nous étudions dans cette section trois de ces caractéristiques, à savoir la persistance des instances, la suppression des instances et la gestion des extensions des classes.

Persistance des instances

La persistance en POETJava se décline au niveau des classes et des instances. Une classe doit être spécifiée comme pouvant avoir des instances stockées dans la base de données. Cette spécification ne se fait pas en Java mais dans un fichier de configuration qui doit être fourni au compilateur avec les autres fichiers sources. Dans ce fichier, il faut ajouter la clause « persistent = true » dans l'entrée relative à la classe dont les instances pourront être stockées lors de l'exploitation.

Ensuite, une instance d'une telle classe peut être rendue persistante par l'invocation de la méthode « bind »²⁹ dont la description est ;

```
void bind(Object object,String name)
```

Le premier paramètre contient une référence vers l'instance que l'on veut rendre persistante.

Le second paramètre permet de conférer à l'instance un nom unique au sein de la base de données. Il est alors possible de retrouver l'instance par son nom. Si l'on ne désire pas nommer une instance que l'on veut rendre persistante, il faut passer « null » comme valeur du second paramètre.

Il faut noter que « bind » ne peut être invoquée à n'importe quel moment. Elle doit être invoquée lors d'une transaction.

Par ailleurs, toute instance référencée par une instance persistance est automatiquement rendue persistante par POET.

Suppression des instances

Les instances des classes Java sont automatiquement supprimées lorsqu'elles ne sont plus référencées par aucune autre instance. Ce n'est plus le cas des instances qui ont été rendues persistantes. Celles-ci doivent être supprimées explicitement à l'aide de la méthode « delete »³⁰ dont la description est :

```
void delete(Object object)
```

Le fait de pouvoir détruire explicitement des instances rend alors possible l'existence de références vers de telles instances. Utiliser une telle référence produit alors de la part de POET le déclenchement d'une exception particulière.

Enfin, il faut noter que « delete » ne peut être invoquée que lors d'une transaction.

Gestion des extensions

POETJava fournit la classe « Extent » qui permet d'obtenir l'extension³¹ d'une classe.

29. méthode d'instance de la classe « Database » (voir [POET97] pour créer des bases de données qui sont des instances de cette classe)

30. méthode de classe de la classe « ObjectServices »

Ceci se fait en instanciant la classe « Extent » au travers du constructeur suivant :

```
Extent(Database base,String className )
```

Le premier paramètre est la base de données, le second le nom de la classe dont on désire obtenir l'extension.

L'instance de « Extent » ainsi créée peut fournir chaque instance au travers des méthodes « hasMoreElements » et « nextElement ».

Le figure 4.21 montre un exemple d'utilisation de la classe « Extent ». Elle consiste à invoquer la méthode « ApplyUpdate » à toutes les instances persistantes de la classe « C ». Il convient de remarquer que la classe « Extent » doit également être utilisée lors d'une transaction.

```
import COM.POET.odmg.*;
import COM.POET.odmg.collection.*;
public class ExempleExtension {
    public static void main(String args[]) {
        Database db = Database.open("poet://LOCAL/nomBase",Database.openReadWrite);
        Transaction txn = new Transaction(db);
        C unC;
        txn.begin();
        try {
            Extent C_Extent = new Extent(db,"C");
            while (C_Extent.hasMoreElements()) {
                unC = (C) C_Extent.nextElement();
                unC.applyUpdate();
            }
            txn.commit();
        } catch(ODMGRuntimeException exc) {
            txn.abort();
            throw exc;
        }
    }
}
```

Figure 4.21. Exemple d'utilisation de la classe « Extent » dans POETJava

4.3.3.2 La classe « CromeExtent »

Nous avons vu en 4.3.3.1 l'utilisation de la classe « Extent » pour retrouver toutes les instances persistantes d'une classe. Cette solution ne correspond pas aux besoins des clients des différentes fonctions qui manipulent l'extension fonctionnelle des classes du domaine et non leur extension complète. Il est donc nécessaire de procéder à une adaptation par sous-classement de « Extent ». La sous-classe résultante s'appelle « CromeExtent » et son utilisation est similaire à celle de la classe « Extent ». La figure 4.22 illustre un exemple d'utilisation de « CromeExtent » où un client de la fonction « Direction » obtient l'extension fonctionnelle de la classe « Client » dans le contexte correspondant.

31. l'extension est constituée ici des instances qui ont été rendues persistantes.

```

import COM.POET.odmg.*;
import COM.POET.odmg.collection.*;
import nomBase.Direction.*;
public class ExempleCromeExtension {
    public static void main(String args[]) {
        Database db = Database.open("poet://LOCAL/nomBase",Database.openReadWrite);
        Transaction txn = new Transaction(db);
        Client unClient;
        int encours_a_regler_cumul;
        txn.begin();
        try {
            CromeExtent Direction_Client_Extent = new CromeExtent(db,"Direction", "Client");
            encours_a_regler_cumul = 0;
            while (Direction_Client_Extent.hasMoreElements()) {
                unClient = (Client) Direction_Client_Extent.nextElement();
                encours_a_regler_cumul = encours_a_regler_cumul + unClient.getEncours_a_regler();
            }
            txn.commit();
        }
    }
}
...

```

Figure 4.22. Utilisation de la classe « CromeExtent » dans le contexte « Direction » pour obtenir l'extension fonctionnelle de la classe « Client »

4.3.3.3 Calcul des extensions fonctionnelles

Pour procéder au calcul de l'extension fonctionnelle d'une classe dans un contexte, le constructeur de la classe « CromeExtent » sélectionne les instances de cette classe dont la fonction booléenne « SetParticipe » est vérifiée pour ce contexte. « SetParticipe » est une méthode d'instance. Elle détermine la participation de l'instance au contexte fonctionnel courant³². Au stade actuel de l'implantation du prototype, « SetParticipe » ne vérifie que le prédicat de participation et la participation aux contextes fournisseurs des accès inter-plans. Elle vérifie que les contraintes de participation héritées de la surclasse sont vérifiées en invoquant la méthode « SetParticipe » de cette surclasse.

La méthode « SetParticipe » est générée automatiquement pour chaque classe du domaine. La figure 4.23 montre cette méthode pour la classe du domaine « Produit ».

```

public boolean setParticipe()
{
    String fonction=this.getFonctionCourante();
    participe=true;
    if (fonction.equals("Fabrication") || fonction.equals("SAV"))
        participe = participe && (cout_stockage > 300);
    if (fonction.equals("SAV"))
        participe = participe && (prix_tarif > 1000);
    return participe;
}

```

Figure 4.23. Méthode « SetParticipe » de la classe du domaine « Produit »

Sur cette figure, on peut s'assurer que la vérification d'un prédicat de participation n'a

32. Le nom de fonction courante est fixée par la méthode « setFunctionCourante ».

lieu que pour les contextes « Fabrication » et « SAV ». En outre, l'existence d'un accès inter-plans du contexte « Fabrication » vers le contexte « SAV » fait que la participation à ce dernier contexte nécessite la vérification du prédicat de participation au contexte « Fabrication » qui est d'avoir un coût de stockage supérieur à 300 F.

4.3.3.4 Gestion des exceptions

Lors de l'exécution d'un programme utilisant les différentes interfaces relatives à une même fonction, différentes exceptions peuvent être enclenchées soit par les accesseurs soit par le constructeur d'une classe lors d'une opération d'instanciation. Ces exceptions sont toutes dérivées de l'exception « CromeException » et sont les suivantes :

- « ViolationException » : cette exception intervient lors d'un accès vers une instance qui ne participe plus au contexte courant. La figure 4.24 montre un exemple d'enclenchement de cette exception lors d'une transaction dans le contexte « Fabrication ». En effet, pour participer au contexte « Fabrication », un produit doit avoir un coût de stockage supérieur à 300 F. Ici, pendant la transaction, le coût de stockage du produit référencé par la variable « unProduit » passe de 300 F à 200 F. Lors de l'accès qui suit, l'instance référencée par la variable « unProduit » ne participe donc plus au contexte « Fabrication », ce qui conduit à soulever l'exception « ViolationException » ;

```
txn.begin(); // début de la transaction
...
System.out.println(unProduit.get Cout_stockage()); // affiche 400, le produit participe bien au contexte
unProduit.setCout_stockage(200);
System.out.println(unProduit.getCout_stockage()); // enclenche une exception du type ViolationException
...
txn.commit();
```

Figure 4.24. Enclenchement de l'exception « ViolationException » dans le contexte « Fabrication »

- « AssociationException » : cette exception se produit lorsque, dans un contexte fonctionnel, un client crée une occurrence d'une relation provoquant la violation de l'une des contraintes de cardinalité maximale dans le contexte de base de cette relation sans la violer dans le contexte fonctionnel. Ceci est dû à l'existence dans le contexte fonctionnel d'une sélection sur la classe située à l'extrémité de la relation où la contrainte de cardinalité est violée dans le contexte de base (cf remarques de 3.6.3.2 ainsi que la figure 3.30 du chapitre 3) ;

- « InstanciationException » : cette exception est enclenchée lors d'une tentative d'instanciation d'une classe du domaine dans un contexte qui n'est pas un contexte d'instanciation de cette classe.

4.3.4 Conclusion

Nous avons présenté notre mise en œuvre dans le cadre d'un SGBD couplé avec le langage Java. Cette mise en œuvre a donné lieu à notre prototype CromeJava que nous avons implanté avec le SGBD commercial POET.

Cette mise en œuvre est basée sur les interfaces de Java utilisées pour conférer des descriptions multiples aux entités du domaine. Ensuite, les paquetages sont utilisés pour

regrouper les interfaces relatives à une même fonction. Ceci nous a conduit aux paquetages fonctionnels qui détiennent l'ensemble des descriptions nécessaires pour les clients d'une fonction. Nous avons ensuite adapté la classe « Extent » de POET pour que les clients des fonctions puissent manipuler les extensions fonctionnelles et non les extensions des classes du domaine. Ceci a donné lieu à la classe « CromeExtent », sous-classe de « Extent ». Les clients d'une fonction utilisent donc le paquetage fonctionnel qui leur est destiné et la classe « CromeExtent »³³ pour manipuler les informations fonctionnelles (et communes) qui leur sont pertinentes. L'union formée de cette classe et d'un paquetage fonctionnel constitue donc un contexte fonctionnel tel qu'il est défini au chapitre 3 (3.5.5.2). Il faut aussi remarquer que les paquetages constituent un moyen de structuration du schéma global généré.

La possibilité d'adapter la classe « Extent » et la génération d'accesseurs nous ont permis, par leur souplesse³⁴, de réaliser une mise en œuvre plus complète que dans le cas des techniques de vues. Nous avons pu notamment intégrer la gestion de l'intégrité référentielle et les contraintes de participation des instances.

Enfin, l'utilisation d'un SGBD commercial nous a permis de développer un prototype qu'il sera encore nécessaire d'améliorer, notamment pour prendre en compte les contraintes de participation relatives aux contraintes de cardinalité des relations partagées lors du calcul des extensions fonctionnelles.

4.4. Bilan

Dans ce chapitre, nous avons présenté deux mises en œuvres de CROME. Dans la première partie, nous avons montré comment formuler les contextes fonctionnels schémas-vues. Nous avons étudié cette formulation dans le cas d'une traduction basée sur les classes-vues par extraction puis dans le cas d'une traduction basée sur les classes-vues par augmentation de capacité. Nous avons montré les qualités et limites de cette formulation et avons comparé ces deux principes de traduction. Ces limites sont exprimées par la complexité qui est nécessaire pour formuler les contraintes de participation et par la difficulté de prendre en compte des mécanismes d'interaction comme les accès inter-plans.

Dans la seconde partie, nous avons présenté une mise en œuvre dans les SGBD supportant Java qui a donné lieu à un prototype basé sur le SGBD POET. Celle-ci a montré comment l'utilisation des notions d'interface et de paquetage du langage Java ainsi que l'adaptation de la classe « Extent » de POET pour le support des extensions fonctionnelles ont permis de réaliser les contextes fonctionnels.

33. correctement paramétrée avec le nom de la fonction

34. Cette souplesse est due à l'utilisation d'un langage de programmation au sein des accesseurs et dans l'adaptation de la classe « Extent ».

Chapitre 5

Extensions de l'approche CROME

5.1. Introduction

Dans le chapitre 3, nous avons introduit l'approche CROME dans le cadre des bases de données. Nous proposons maintenant deux extensions à cette approche, ayant pour objectifs de répondre aux problèmes introduits dans les sections 1.8 et 1.9 du premier chapitre, soit :

- la prise en compte du *cycle de vie fonctionnel* des instances au sein d'une base de données lors de la conception en CROME ;
- l'*intégration d'un composant* « base de données » existant et peuplé dans le cadre d'une conception en CROME afin d'exploiter les informations fonctionnelles qu'il détient.

Ces extensions sont entièrement basées sur les notions de CROME que nous avons présentées dans le chapitre 3. Elles ne nécessitent pas de nouvelles notions ni d'enrichir les notions existantes pour répondre à leurs objectifs. Ainsi, le cycle de vie fonctionnel spécifié par un réseau de Petri est traduit en un ensemble de contraintes de participation aux différents contextes. L'intégration d'un composant est réalisée en lui conférant un statut de contexte fonctionnel puis en utilisant les moyens d'interaction entre contextes introduits en 3.4.

Dans la section 5.2.2, nous présentons la notion de contexte sémantique ainsi que la dynamique d'instances en Farandole [Al-Jadir93, Al-Jadir94, Al-Jadir95, Falquet88]¹. Le cycle de vie d'instances est décrit par un contexte dynamique constitué par un ensemble de contextes sémantiques et par un réseau de Petri. De cette notion de contexte dynamique, nous retenons principalement la possibilité d'utiliser un réseau de Petri pour spécifier un cycle de vie fonctionnel. Puis nous expliquons, en 5.2.2, comment traduire une telle spécification.

La section 5.3.2 présente d'abord les différents aspects de l'intégration de composants existants lors de la conception d'une base de données tels que l'on peut les trouver dans la littérature. Nous verrons deux approches d'intégration, à savoir celles basées sur les classes-vues préservant les identités et sur la détermination d'une relation de coréférence et celles basées sur la génération de nouveaux objets.

Lors de l'étude de l'intégration d'un composant existant dans le cadre d'une conception en CROME, nous étudions les problèmes d'articulation entre le composant et la base multi-fonctionnelle en cours de conception. Dans un premier temps, nous faisons l'hypothèse que le schéma du composant a été structuré relativement au même référentiel d'entités que celui de la base multi-fonctionnelle. Nous proposons alors une solution fondée sur l'établissement d'une relation de coréférence entre instances du composant et de la base multi-fonctionnelle. Cette solution nous conduit à réaliser l'intégration au sein des entités du domaine. Dans un second temps, nous relâchons cette hypothèse et proposons une solution de reconstitution du référentiel basée sur les techniques de génération de nouveaux objets.

En conclusion, nous détaillons les limites de ces deux extensions.

1. Le choix de Farandole s'explique par le fait qu'il s'attaque à des problèmes proches de ceux de CROME et notamment les vues et les contextes dans les bases de données à objets.

5.2. Cycle de vie de participation d'instances aux contextes

5.2.1 Notion de contexte et de dynamique d'instance en Farandole

Farandole est un SGBD développé à l'université de Genève [Al-Jadir93, Al-Jadir94, Al-Jadir95, Falquet88] dont il en existe deux versions : Farandole 1 qui est un SGBD relationnel et Farandole 2 qui est un SGBD à objets. L'originalité de Farandole se situe dans la notion de contexte sémantique qui est une abstraction du schéma de la base. Il permet de ne retenir que les aspects pertinents du schéma pour l'application courante. Un contexte contient des contraintes et des méthodes (Farandole 2). Il peut être interrogé par une fonction de connexion.

Les travaux relatifs à Farandole traitent aussi de la dynamique des instances. Cette dynamique est spécifiée par un réseau de Petri. Dans Farandole 1, les différents états possibles ne sont pas liés aux contextes sémantiques. Dans Farandole 2, chaque état correspond à la participation de l'instance à un contexte. Nous verrons que l'utilisation des contextes permet de généraliser la dynamique spécifiée par un réseau de Petri à des groupes d'objets.

5.2.1.1 Contexte sémantique (en Farandole 2)

Un contexte sémantique n'est pas constitué de classes mais de nœuds issus des classes du schéma.

Un contexte sémantique est un quadruplet $\langle c_n, S, R, M \rangle$ où :

- c_n est le nom du contexte ;
- S est la structure du contexte, constituée d'un ensemble de nœuds et d'arcs ;
- R est un ensemble de contraintes que doivent vérifier les instances des classes participant au contexte ;
- M est un ensemble de méthodes attachées au contexte.

La structure « S » détient l'ensemble des descriptions relatives aux informations pertinentes pour les clients du contexte. Elle est constituée d'arcs et de nœuds :

- un nœud est une couple $(N:C)$ où « N » est le nom du nœud et « C » la classe associée. Une même classe peut être associée à plusieurs nœuds. Lors de la définition d'un nœud est précisé l'ensemble des attributs ayant pour domaine un type de littéraux et qui détiennent les informations pertinentes pour les clients du contexte. Les autres attributs pertinents donnent lieu à des arcs ;
- un arc est un triplet $(N_1 \text{ a } N_2)$. Il correspond à l'attribut de « a » de la classe associée au nœud « N_1 » dont le domaine est la classe associée au nœud « N_2 ».

Chaque contrainte de l'ensemble « R » est définie ainsi :

$\langle \text{chemin.attribut opérateur valeur} \rangle$ ou

$\langle \text{chemin.attribut opérateur chemin} \rangle$

Un chemin prend la forme d'une liste $[N_1 A_1 N_2 A_2 \dots N_S A_S N_{S+1}]$ dont tous les nœuds N_i sont distincts et telle que chaque arc A_i aille bien de N_i à N_{i+1} . Au minimum, un chemin ne contient qu'un seul nœud. L'opérateur peut être l'un des opérateurs classiques de comparaison de valeurs. Chaque contrainte est associée à un nœud. Les chemins doivent commencer par ce nœud, c'est-à-dire que N_1 doit correspondre au nœud auquel est associée la contrainte. Pour être instance d'un nœud, une instance de la classe associée à ce nœud doit en vérifier les contraintes. Le fait que plusieurs nœuds puissent être associés à une même classe fait qu'un contexte peut contenir plusieurs sous-ensembles de l'extension d'une classe.

L'ensemble « M » contient des méthodes qui sont attachées au contexte et non pas aux classes. Ceci permet un polymorphisme relatif aux contextes. Ces méthodes peuvent accéder à l'ensemble des attributs visibles dans leur contexte d'introduction.

L'insertion d'une ou de plusieurs instances relativement à un ou plusieurs nœuds dans un contexte doit se faire explicitement par l'utilisation de l'ordre « **assert** ». Dans le cas où plusieurs instances sont insérées simultanément, celles-ci se retrouvent connectées par tous les chemins qui se trouvent entre leurs nœuds respectifs².

Exemple

Nous reprenons l'exemple de l'organisation d'une conférence [Al-Jadir94] dont la hiérarchie des classes (simplifiée ici) est donnée à la figure 5.1.

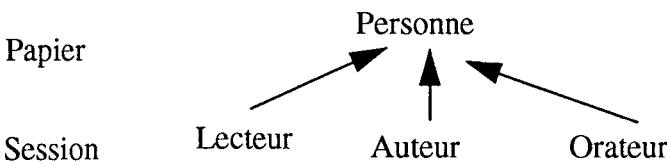


Figure 5.1. Classes du schéma de l'organisation d'une conférence

Le contexte correspondant à la phase de présentation des papiers acceptés est illustré à la figure 5.2.

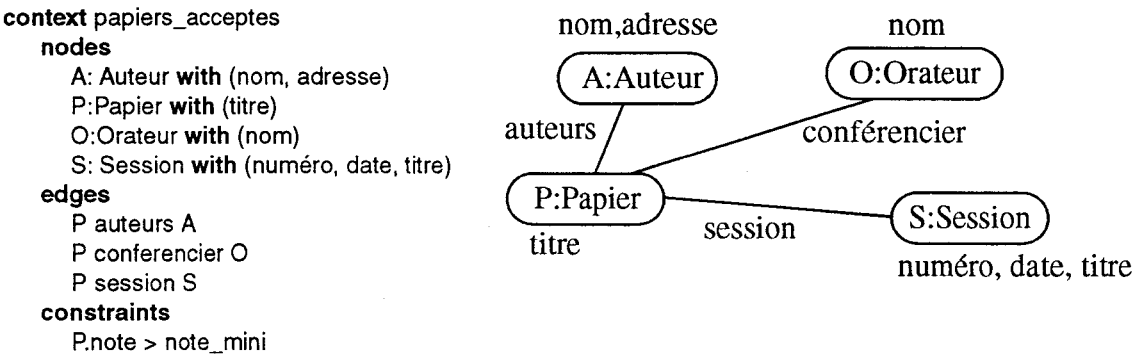


Figure 5.2. Contexte correspondant à la présentation des papiers acceptés

L'insertion dans le contexte « papiers_acceptes » d'une instance d'un papier avec son auteur, son orateur et sa session peut se faire ainsi :

```
assert papiers_acceptes[P:unpapier A:unauteur O:unorateur S:unesession]
```

Les quatre instances sont alors connectées au travers des trois arcs présents dans le contexte.

Fonctions de connexion

Les fonctions de connexion sont utilisées pour l'expression de requêtes sur un contexte.

2. S'il le faut, de nouvelles instances sont créées dans les classes correspondant aux nœuds situés sur les chemins entre les nœuds des instances insérées.

Une fonction de connexion est associée à un sous-ensemble « N » de l'ensemble des nœuds de la structure « S » du contexte. Le résultat de la fonction est composé d'un ensemble de n-uplets composés d'instances des nœuds de « N » telles qu'elles soient liées par tous les chemins présents dans le contexte entre les nœuds de « N ». La définition d'un chemin est la même que dans le cas des contraintes. S'il existe plusieurs chemins distincts entre deux nœuds, les instances d'un n-uplet doivent être liées entre elles pour chaque chemin.

Exemple

Si l'on veut connaître tous les auteurs ayant un papier accepté, il faut utiliser la fonction de connexion suivante :

connect(papiers_acceptes, {P,A})³

Cette fonction retourne une liste de couples de papiers acceptés et de leurs auteurs.

Deux chemins distincts entre deux nœuds « N₁ » et « N₂ » peuvent être séparés en associant l'un deux à un nouveau nœud « N₃ » à la place de « N₁ » ou « N₂ » et tel que « N₃ » soit associé à la même classe que le nœud substitué. En effet, deux nœuds distincts peuvent être associés à la même classe. Ceci permet de d'exprimer différentes sémantiques sur les chemins (conjonction ou alternative).

Exemple

La figure 5.3 montre deux contextes sur un même schéma contenant deux classes C₁ et C₂ liées par deux relations distinctes R₁ et R₂. Dans le cas du premier contexte, toute fonction de connexion sur N₁ et N₂ retourne des instances de C₁ et C₂ liées par R₁ et R₂. Dans le cas du second contexte, toute fonction de connexion sur N₁ et N₂ retourne des instances de C₁ et C₂ liées seulement par R₁ et toute fonction de connexion sur N₁ et N₃ retourne des instances de C₁ et C₂ liées seulement par R₂.

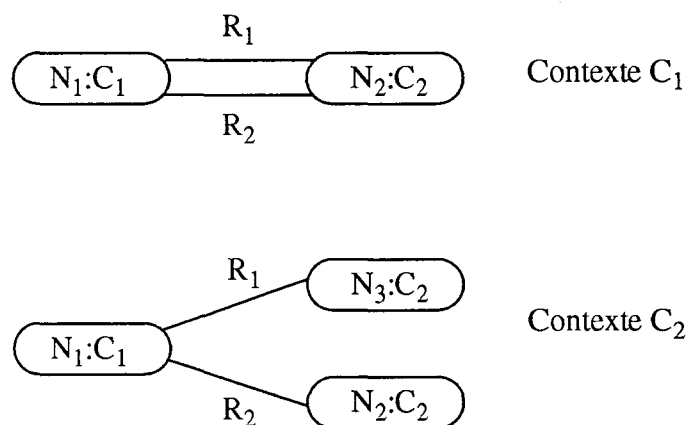


Figure 5.3. Séparation de chemins issus de relations entre deux classes

Remarques

- Les contextes sont formés à partir du schéma de la base. Ils peuvent se recouvrir, provoquant alors un partage de description entre contextes. Ceci peut conduire à des conflits et

3. Notons que pour simplifier, nous avons omis le troisième paramètre d'une fonction de connexion qui est l'état de la base de données

il n'est pas prévu de distinguer les contextes fournisseurs des contextes consommateurs.

- Les contextes ne préservent pas la hiérarchie d'héritage des classes présentes dans le schéma de la base. Cette hiérarchie n'est donc pas présentée aux clients d'un contexte, même partiellement.
- Les contextes ne peuvent pas augmenter les descriptions existantes dans le schéma de la base.

5.2.1.2 Expression de la dynamique en Farandole

5.2.1.2.1 Farandole 1

Dans Farandole 1 [Falquet88], la dynamique des états que peuvent prendre les instances n'est pas nécessairement liée à l'ensemble des contextes sémantiques. Cette dynamique est représentée par un réseau de Petri. Chaque place de ce réseau de Petri correspond à un état particulier. Les transitions correspondent à des événements. Le réseau de Petri est une spécification qui est ensuite traduite, à l'aide d'un outil spécifique, en un squelette de programme gérant les différents états et leurs transitions. Cette traduction est basée sur l'utilisation d'un attribut spécifique à l'entité correspondant au réseau de Petri et qui contient l'état dynamique des instances.

Le réseau de Petri qui se trouve à la figure 5.4 illustre deux états de l'entité « Participant » (participant à une conférence) ainsi que la transition qui les sépare. Le premier état DBR (« Demande de brochure reçue ») est l'état d'un participant après qu'on ait reçu, de sa part, une demande de brochure de la conférence. Le second état BCE (« Brochure de la conférence envoyée ») est l'état du participant après qu'on lui ait envoyé la brochure de la conférence. L'évènement lié à la transition est donc l'envoi de cette brochure.

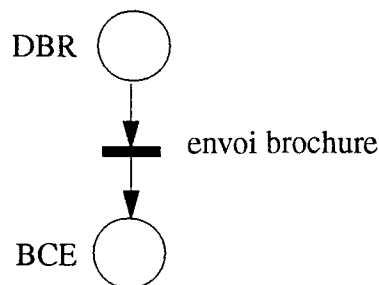


Figure 5.4. Réseau de Petri illustrant la dynamique des états d'un participant à une conférence

Le squelette de programme qui est généré à partir d'une telle spécification est donné à la figure 5.5. Ce squelette a été complété par l'action d'envoi (entre la sélection des instances dans l'état DBR et la mise à jour de leur état en BCE).

Cette génération est basée sur l'adjonction de l'attribut « DYN_STATE_PARTICIPANT » à l'entité « Participant ». Cet attribut est un ensemble qui détient l'état dynamique d'un participant.

```

domain DYN_STATE_PARTICIPANT_TYPE : enumeration (DBR, BCE,...),

relation PARTICIPANT
  ( NAME : TEXT[40],
    ...
    DYN_STATE_PARTICIPANT : DYN_STATE_PARTICIPANT_TYPE[8]);

-- sélection des instances dont l'attribut DYN_STATE_PARTICIPANT contient la valeur DBR
S_PARTICIPANT := PARTICIPANT : {DBR} subset DYN_STATE_PARTICIPANT
-- action d'envoi
for each p in S_PARTICIPANT
  envoi_brochure(p);
end for each;
-- mise à jour de l'état des instances après l'envoi
for each p in S_PARTICIPANT
  p := update p[DYN_STATE_PARTICIPANT := DYN_STATE_PARTICIPANT + {BCE} - {DBR}];
end for each;

```

Figure 5.5. Exemple de traduction du précédent réseau de Petri en un programme

5.2.1.2.2 Contextes dynamiques de Farandole 2

Un contexte dynamique est constitué d'un ensemble de contextes sémantiques qui sont ses versions. Un contexte dynamique est un n -uplet $(c_n, \{V_0 = \langle vn_0, S_0, R_0, M_0 \rangle, V_1 = \langle vn_1, S_1, R_1, M_1 \rangle, \dots, V_s = \langle vn_s, S_s, R_s, M_s \rangle\}, PN)$ où :

- c_n est le nom du contexte dynamique ;
- $V_0, V_1, \dots, V_n$ sont les différentes versions du contexte dynamique. Chaque version est un contexte sémantique. V_0 est le contexte générique qui détient les nœuds, arcs, contraintes et méthodes communs à l'ensemble des versions ;
- PN est le réseau de Petri qui décrit le cycle de vie des objets dont la description est contenue dans le contexte générique V_0 . Il décrit également les transitions correspondant au passage d'une version à la suivante.

Chaque place du réseau de Petri correspond à une version du contexte. Les transitions correspondent à des événements. Lors de leur cycle de vie, les objets restent toujours décrits par le contexte générique V_0 . A ces descriptions minimales s'ajoutent les descriptions fournies par le contexte courant⁴. Dans le cas où V_0 contient plusieurs nœuds, le contexte dynamique s'applique alors à des groupes d'objets qui doivent être liés entre eux par tous les chemins existants entre leurs nœuds⁵. Un élément d'un groupe d'objets ne peut pas franchir seul une transition du réseau de Petri. Seul le groupe entier peut franchir la transition.

Remarque

Le réseau de Petri fait partie de la notion de contexte dynamique en F2. Il sert à vérifier la validité des transitions que l'on veut franchir. Une transition ne peut être franchie pour une instance que si cette instance apparaît dans tous les contextes en entrée de ladite transition.

4. le contexte correspondant à la place courante.

5. l'ensemble des nœuds de V_0 constituent une fonction de connexion qui détermine les groupes d'objets

Exemple

Le réseau de Petri qui se trouve à la figure 5.6 illustre le cycle de vie d'un papier lors d'une conférence. Le contexte générique « V₀ » de ce contexte dynamique est constitué du seul nœud « P » associé à la classe « Papier ».

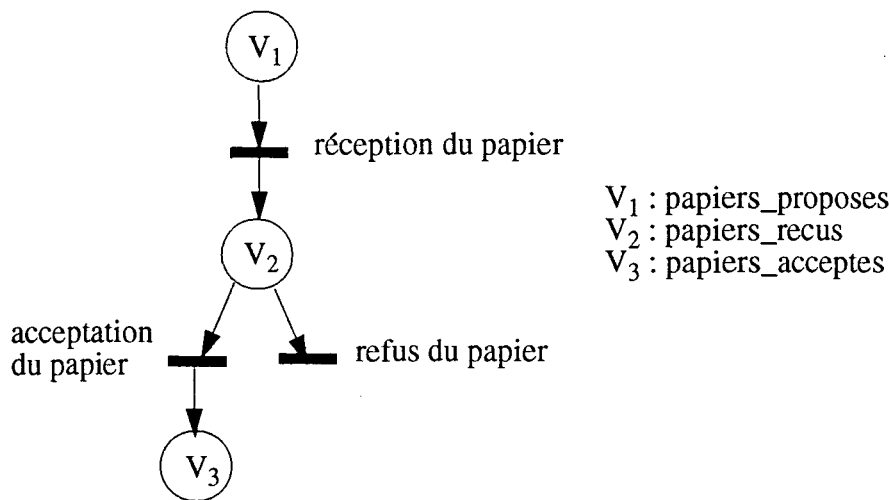


Figure 5.6. Réseau de Petri du contexte dynamique sur le cycle de vie d'un papier lors d'une conférence

La figure 5.2 illustre le contexte « V₃ » de ce contexte dynamique. Les figures 5.7 et 5.8 montrent respectivement les contextes « V₁ » et « V₂ ».

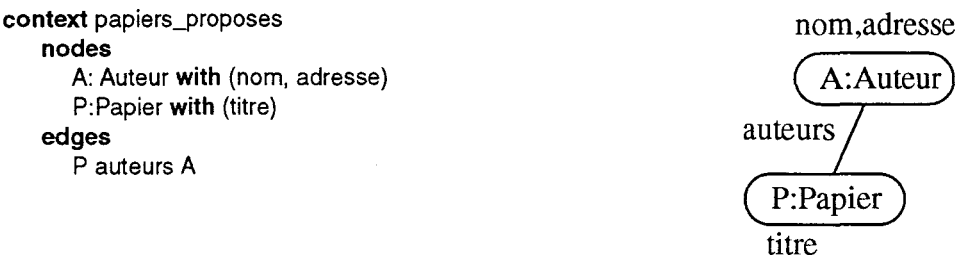


Figure 5.7. Contexte correspond aux papiers proposés

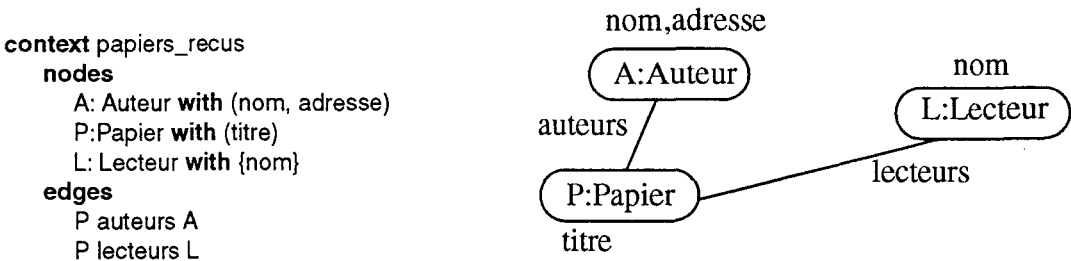


Figure 5.8. Contexte correspondant aux papiers reçus

5.2.2 Prise en compte du cycle de vie fonctionnel des instances dans l'approche CROME

Nous avons vu en 1.8 le besoin de déterminer le cycle de vie fonctionnel d'instances ou de couple d'instances. Un réseau de Petri [Al-Jadir93, Al-Jadir94, Falquet88] permet de représenter un cycle de vie. Nous proposons, dans un premier temps, d'établir finement l'interprétation d'un réseau de Petri pour représenter des cycles de vie fonctionnels, tels que nous les avons définis en 1.8. Cette interprétation nous conduira à établir certaines restrictions sur la forme et le fonctionnement du réseau de Petri.

Puis à partir de la spécification ex-nihilo d'un cycle de vie fonctionnel décrite par un réseau de Petri, nous proposons de traduire celle-ci à l'aide des contraintes de participation de CROME. Cette traduction permet de prendre en compte le cycle de vie fonctionnel des instances d'une classe du domaine au sein de cette classe. Ensuite, nous définirons précisément la notion de couple d'instances et expliquerons comment traduire en CROME le cycle de vie fonctionnel d'un tel couple.

5.2.2.1 Interprétation du réseau de Petri représentant le cycle de vie fonctionnel d'une instance

Nous associons à chaque entité du domaine un réseau de Petri [Peterson77] qui est constitué de places et de transitions. La figure 5.11 à la page 5.186 fournit un exemple de représentation graphique d'un tel réseau. Chacune des places correspond à un contexte fonctionnel où intervient cette entité. Pour chaque instance, la présence d'un jeton dans une place signifie la participation au contexte associé. L'état fonctionnel d'une instance est alors déterminé par l'ensemble des contextes correspondant aux places contenant un jeton.

A chaque transition correspond un événement spécifique dont la réalisation provoque la participation d'une instance à de nouveaux contextes. Pour qu'une transition soit franchie, il faut qu'elle soit validée (c'est-à-dire que ses places en entrée contiennent un jeton) et que l'événement associé se produise. Un tel franchissement va alors correspondre à la participation pour l'instance concernée à de nouveaux contextes fonctionnels et donc à un nouvel état fonctionnel.

Cette notion d'activation d'une transition correspond à l'idée que, pour participer à un nouveau contexte, une instance doit avoir participé à d'autres contextes au préalable. Ceci est souvent nécessaire pour des besoins de transmissions d'information entre contextes, ou encore pour respecter un ordre établi de participation.

5.2.2.2 Restrictions apportées au cycle de vie fonctionnel

Pour simplifier notre étude et pour garantir les états du réseau de Petri représentant un cycle de vie fonctionnel vis-à-vis de l'interprétation donnée ci-dessus, nous nous proposons d'étudier des cycles de vie fonctionnels réduits à des chaînes de participations successives d'une instance. Ces restrictions sont décrites ci-dessous sur le réseau de Petri qui sert à la représentation du cycle de vie fonctionnel.

a) Chaque transition n'a qu'une seule place en entrée. En sortie, elle a soit uniquement une place distincte de celle en entrée, soit deux places distinctes dont celle en entrée. Ces deux types de transition sont illustrés à la figure 5.9 et sont appelés respectivement « alternance » et « dépendance ». Le cas de la dépendance correspond à un besoin où, après franchissement, l'instance continue à participer au contexte en entrée en plus d'un nouveau contexte. Ceci

permet notamment à ce nouveau contexte d’exploiter la partie fonctionnelle relative au contexte en entrée (au travers d’accès inter-plans). Cette exploitation conduit à une dépendance entre ces deux contextes d’où le terme choisi pour désigner ce type de transition. Dans le cas de l’alternance, après franchissement, l’instance ne participe qu’au contexte en sortie.

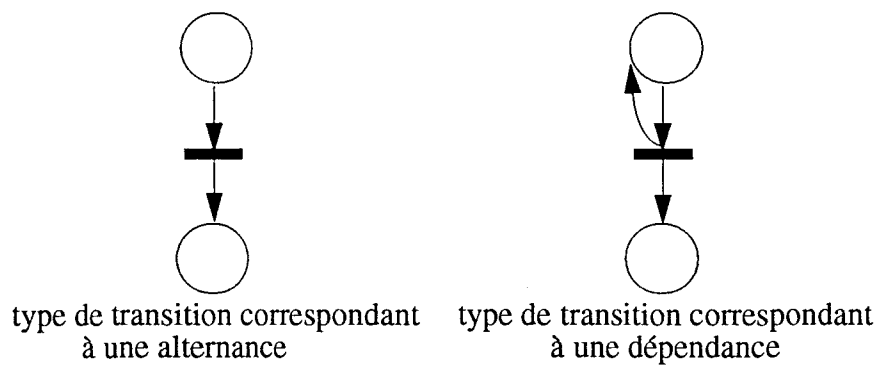


Figure 5.9. Alternance et dépendance

b) Une place ne peut avoir qu’une transition en entrée provenant d’une place distincte d’elle-même. Ceci proscrit la possibilité d’écrire des boucles.

c) Les transitions en sortie d’une place, si elles sont multiples, correspondent à des évènements mutuellement exclusifs. La figure 5.10 illustre le cas de deux transitions en sortie pour une place. Les évènements correspondant à ces deux transitions sont exclusifs, c’est-à-dire qu’une occurrence de A implique l’absence d’occurrence de B et vice-versa. Une instance participant au contexte en entrée ne peut, en cas d’occurrence de A ou de B, participer qu’à l’un des deux nouveaux contextes en sortie. Il convient aussi de remarquer qu’il s’agit ici d’un partitionnement de la chaîne de participation en deux sous-chaînes exclusives.

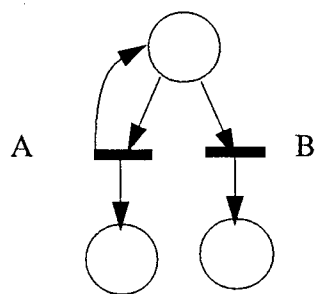


Figure 5.10. Plusieurs transitions en sortie d’une place. Les évènements A et B sont mutuellement exclusifs.

d) Lors de l’exécution du réseau de Petri (correspondant au cycle de vie d’une instance précise), la sortie d’un jeton d’une place ne peut se produire qu’une seule fois. Ceci a pour conséquence d’interdire qu’une transition de dépendance soit franchie plus d’une fois. Dans le cas où une place a plusieurs transitions en sortie, une seule au plus, pourra être franchie. Dans

le cas de la figure 5.10, ceci signifie que si la transition de gauche est franchie, celle de droite ne pourra plus l'être bien qu'un jeton soit à nouveau présent dans la place du haut (c'est-à-dire que l'instance maintient définitivement sa participation dans le contexte en entrée).

Les deux dernières restrictions s'appliquent au fonctionnement du réseau de Petri. Les deux premières contraintes sont relatives à la forme du réseau illustrée par la figure 5.11. L'ensemble des places du réseau forme ainsi un arbre. Chaque place est reliée aux places qui la suivent par une transition d'alternance ou de dépendance.

Ces restrictions garantissent le fait que, lors de l'exécution du réseau, il ne puisse y avoir plus d'un jeton dans une place, ce qui poserait un problème de cohérence vis-à-vis de l'interprétation du réseau de Petri que nous venons de faire en 5.2.2.1. En effet, chaque place a, au plus, une seule autre place distincte qui la précède. Un seul jeton ne pouvant sortir de celle-ci, il ne peut donc y avoir deux jetons dans une place.

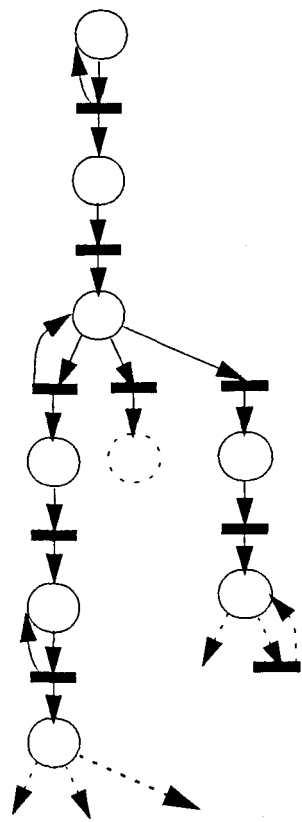


Figure 5.11. Forme générale d'un réseau de Petri représentant un cycle de vie fonctionnel

5.2.2.3 Traduction en CROME du cycle de vie fonctionnel d'une instance

Nous faisons le choix de traduire le cycle de vie fonctionnel d'une instance spécifié par un réseau de Petri en un ensemble de contraintes de participation associées à sa classe. Plus précisément, nous nous inspirons des choix de Farandole 1 (cf 5.2.1.2.1) sur l'utilisation d'un attribut qui va, dans notre cas, détenir l'état fonctionnel de l'instance. Nous appelons cet attribut, l'*attribut du cycle de vie fonctionnel*. Les contraintes de participation utilisées sont

alors des prédicats de participation qui font une sélection par rapport à la valeur de cet attribut. Chaque évènement provoquant le franchissement d'une transition enclenche une mise à jour de la valeur de cet attribut afin de changer l'état fonctionnel de son instance.

Nous faisons l'hypothèse que la mise à jour de la valeur de cet attribut suffit pour changer l'état fonctionnel. Par conséquent, nous supposons que les contraintes de participation autres que celle du prédicat et présentes dans les différents contextes du cycle de vie fonctionnel sont vérifiées dès le début de ce cycle⁶. Prendre en compte d'autres contraintes signifierait qu'un évènement provoquant le franchissement d'une transition devrait, en plus de mettre à jour la valeur de l'attribut du cycle de vie fonctionnel, valider ces autres contraintes du contexte en sortie de la transition.

Nous ne faisons donc pas le choix d'introduire une nouvelle notion dans CROME qui serait ici un réseau de Petri décrivant le cycle de vie fonctionnel d'instances, à la différence de Farandole 2. Un tel choix conduirait à rendre compte dans CROME, par un nouveau mécanisme, des évènements qui provoquent le franchissement des transitions lorsque celles-ci sont actives. Ce nouveau mécanisme pourrait être un ordre explicite qui s'appliquerait à une instance. Nous préférons ici nous restreindre aux notions existantes de CROME : un évènement provoque la mise à jour de la valeur de l'attribut du cycle de vie fonctionnel.

Notre traduction est basée sur des prédicats de participation de la forme :

$val(att-cv(C)) \in Dom_{par}(f,C)$
où $C \in \mathcal{C}$, $att-cv(C) \in D_{base}(C)$, $f \in \mathcal{F} - \{base\}$, $PF_f(C) \neq \emptyset$.

$Dom_{par}(f,C)$ est l'ensemble des valeurs que peut prendre l'attribut « att-cv » d'une instance de C pour que le prédicat de participation au contexte fonctionnel CF_f soit vérifié. « att-cv(C) » est l'attribut du cycle de vie fonctionnel de C . Il détient l'état fonctionnel. $Dom_{par}(f,C)$ est appelé « domaine de participation » de la classe C au contexte CF_f .

Un tel prédicat peut facilement être écrit en OQL.

Exemple

Nous reprenons l'exemple décrit en 1.8.1 relatif au cycle de vie fonctionnel du bon de livraison que nous rappelons ici :

Chaque bon de livraison est produit par la fonction commerciale. Puis lorsqu'il est prêt, il est transmis à la fonction « Stock »⁷. Enfin, lorsque l'ensemble des produits figurant sur le bon est livré au client correspondant, le bon est transmis à la fonction comptable afin d'être comptabilisé. Après la comptabilisation, la fonction comptable n'en garde ensuite plus trace.

Pour traduire ce cycle de vie en CROME, nous utilisons un attribut de la description de base de la classe « Bon_de_livraison » qui est « progression » de type « string » (cf annexe A). Cet attribut voit sa valeur fournie par les différents contextes où intervient l'entité « Bon_de_livraison », c'est-à-dire « Commercial », « Stock » et « Comptable ».

Par exemple, le prédicat de participation au contexte fonctionnel comptable s'écrit :

`progression = 'comptabilisation'`

Il correspond donc bien à la forme générale décrite ci-dessus avec un domaine qui est un singleton constitué de l'unique valeur « 'comptabilisation' ».

6. C'est le cas pour l'exemple pris dans cette section. Il est possible de le vérifier en examinant les contraintes sur les entités utilisées dans cet exemple à l'annexe A.

7. qui assure aussi les livraisons

Domaine de transition

Chaque évènement provoque la mise à jour de la valeur de l'attribut « att-cv » utilisé dans le prédicat de participation. Pour exprimer la séquentialité au niveau de la traduction, nous proposons que :

- la mise à jour de l'attribut « att-cv » soit faite par les clients de la dernière fonction de participation atteinte et
- que cette mise à jour rende vérifié le prédicat de participation à l'un des prochains contextes possibles (un contexte correspondant à l'une des places en sortie de la dernière place atteinte).

Ceci impose que chaque contexte en entrée fournisse la valeur de l'attribut « att-cv » du plan de base. Une modalité de fourniture d'attribut du plan de base doit donc être spécifiée pour chaque contexte en entrée d'une transition.

A chaque transition, nous associons un domaine de valeurs tel que si l'attribut « att-cv » est valué dans ce domaine, le prédicat de participation au contexte en sortie de la transition soit vérifié. La mise à jour de l'attribut « att-cv » faite par les clients du contexte en entrée de la transition à franchir se fait par une valeur prise dans ce domaine.

Nous désignons par $\text{Dom}_{\text{transition}}(f, ff, C)$ avec $C \in \mathcal{C}$, $(f, ff) \in (\mathcal{F} - \{\mathbf{base}\})^2$, $f \neq ff$, $\text{PF}_f(C) \neq \emptyset$ et $\text{PF}_{ff}(C) \neq \emptyset$, le domaine de valeurs de la transition dont CF_f est le contexte fonctionnel en entrée et CF_{ff} celui en sortie. $\text{Dom}_{\text{transition}}(f, ff, C)$ est le domaine de la transition qui va du contexte CF_f au contexte CF_{ff} pour la classe C .

Si un contexte correspond à une place ayant plusieurs transitions en sortie, ses clients peuvent réaliser une mise à jour de l'attribut « att-cv » par une valeur prise dans l'un des domaines relatifs à ces transitions.

Chaque transition correspond à un évènement spécifique. Par conséquent, le domaine d'une transition est disjoint de celui d'une autre transition, ce que nous pouvons formaliser ainsi :

$\forall (f, ff, f', ff') \in (\mathcal{F} - \{\mathbf{base}\})^4$, $f \neq ff$, $f \neq f'$, $f \neq ff'$, $ff \neq f'$, $ff \neq ff'$, $f' \neq ff'$, on a :
 $\text{Dom}_{\text{transition}}(f, ff, C) \cap \text{Dom}_{\text{transition}}(f', ff', C) = \emptyset$ avec $C \in \mathcal{C}$ telle que $\text{PF}_f(C) \neq \emptyset$, $\text{PF}_{ff}(C) \neq \emptyset$, $\text{PF}_{f'}(C) \neq \emptyset$ et $\text{PF}_{ff'}(C) \neq \emptyset$.

L'inexistence de transitions entre deux contextes fonctionnels CF_f et CF_{ff} pour une classe C se formalise par $\text{Dom}_{\text{transition}}(f, ff, C) = \emptyset$.

Exemple

Le cycle de vie fonctionnel des instances de la classe « Bon_de_livraison » peut être représenté par le réseau de Petri illustré à la figure 5.12. Pour les deux premières transitions⁸, y figure le domaine correspondant.

8. Nous étudierons le cas de la dernière transition par la suite

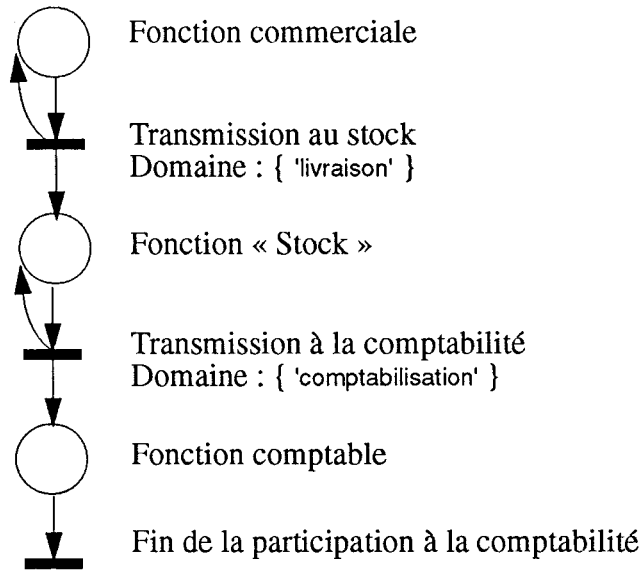


Figure 5.12. Réseau de Petri représentant le cycle de vie fonctionnel d'une instance de « Bon_de_livraison ». Les domaines relatifs à l'attribut « progression » pour les deux transitions sont également indiqués.

Pour qu'une instance franchisse la première transition, les clients du contexte commercial doivent fixer la valeur de son attribut « progression » à « 'livraison' ». Ainsi, cette instance participera au contexte « Stock ». Il en va de même pour la deuxième transition. Pour qu'elle soit franchie et que l'instance participe au contexte comptable, il faut que les clients de « Stock » fixent la valeur de son attribut « progression » à « 'comptabilisation' ».

Nous cherchons maintenant à déterminer les prédicats de participation aux contextes à partir des domaines des différentes transitions. Un tel prédicat étant de la forme précisée ci-dessus, le déterminer revient en fait à déterminer le domaine de participation de l'attribut « att-cv ». Ce domaine de participation est calculé en fonction du domaine de la transition en entrée et des domaines des transitions en sortie du contexte correspondant. Nous étudions d'abord le cas d'un cycle de vie où toutes les transitions sont des transitions d'alternance.

5.2.2.3.1 Traduction dans le cas de transitions d'alternance

Dans le cas d'un cycle de vie dont toutes les transitions sont des transitions d'alternance, le domaine de participation du prédicat de chaque contexte est égal au domaine de la transition en entrée dudit contexte. Ceci garantit l'exclusion entre les prédicats de participation aux différents contextes successifs car les domaines de toutes les transitions sont tous, deux à deux, disjoints.

Soit $C \in \mathcal{C}$ et $(f, ff) \in (\mathcal{F} - \{\mathbf{base}\})^2$ telles que $f \neq ff$, $PF_f(C) \neq \emptyset$, $PF_{ff}(C) \neq \emptyset$, $Dom_{transition}(f, ff, C) \neq \emptyset$ et telles que $\forall fff \in \mathcal{F} - \{\mathbf{base}\}$ vérifiant $Dom_{transition}(ff, fff, C) \neq \emptyset$, la transition qui va de CF_{ff} à CF_{fff} est une transition d'alternance, on obtient :

$$Dom_{par}(ff, C) = Dom_{transition}(f, ff, C)$$

5.2.2.3.2 Traduction dans le cas de transitions d'alternance et de dépendance

Dans le cas d'un cycle de vie dont certaines transitions sont des transitions de dépendance, le domaine de participation du prédicat de chaque contexte est égal à l'union du domaine de la transition en entrée dudit contexte, du domaine de chaque transition de

dépendance en sortie et du domaine de toutes les transitions qui suivent chaque transition de dépendance. Ainsi, si une transition de dépendance en sortie d'un contexte est franchie, la participation à ce contexte n'est pas remise en cause dans la suite du cycle de vie. En effet, l'attribut « att-cv » ne peut pas, par la suite, prendre une valeur qui ne soit pas incluse dans le domaine de l'une des transitions suivantes. Le domaine de participation à chaque contexte qui suit un contexte, par une transition de dépendance, est inclus dans le domaine de participation à ce contexte.

Pour formaliser cette définition, nous commençons par définir la fonction $\text{Domsuiv}(f, ff, C)$ qui renvoie l'union de tous les domaines des transitions suivant celle qui va de CF_f à CF_{ff} ainsi que le domaine de celle-ci.

Cette fonction peut être définie de façon formelle ainsi :

- Cas où la transition qui va de CF_f à CF_{ff} avec $(f, ff) \in (\mathcal{I} - \{\mathbf{base}\})^2$ n'a pas de transitions suivantes, c'est-à-dire qu'il n'existe pas $fff \in \mathcal{I} - \{\mathbf{base}\}$ telle que $\text{Dom}_{\text{transition}}(ff, fff, C) \neq \emptyset$. Dans ce cas, $\text{Domsuiv}(f, ff, C)$ est donné par :

$\text{Domsuiv}(f, ff, C) = \text{Dom}_{\text{transition}}(f, ff, C)$ avec $C \in \mathcal{C}$ telle que $\text{PF}_f(C) \neq \emptyset$ et $\text{PF}_{ff}(C) \neq \emptyset$.

- Cas général où la transition qui va de CF_f à CF_{ff} peut être suivie d'une ou plusieurs autres transitions.

Nous commençons par introduire la fonction « Foncsuiv » : soit $\text{Foncsuiv}(g, C) = \{gg \in \mathcal{I} - \{\mathbf{base}\} \text{ telle que } C \in \mathcal{C}, \text{PF}_g(C) \neq \emptyset, \text{PF}_{gg}(C) \neq \emptyset \text{ et } \text{Dom}_{\text{transition}}(g, gg, C) \neq \emptyset\}$ l'ensemble des fonctions dont le contexte correspondant est en sortie des transitions qui ont CF_g en entrée. Dans le cas général, $\text{Domsuiv}(f, ff, C)$ a pour valeur :

$\text{Domsuiv}(f, ff, C) = \text{Dom}_{\text{transition}}(f, ff, C) \cup_{fff \in \text{Foncsuiv}(ff, C)} \text{Domsuiv}(ff, fff, C)$

Soit, à présent, $\text{Foncsuivdep}(g, C) = \{gg \in \mathcal{I} - \{\mathbf{base}\} \text{ telle que } C \in \mathcal{C}, \text{PF}_g(C) \neq \emptyset, \text{PF}_{gg}(C) \neq \emptyset, \text{Dom}_{\text{transition}}(g, gg, C) \neq \emptyset \text{ et la transition qui va de } CF_g \text{ à } CF_{gg} \text{ est une transition de dépendance}\}$. Foncsuivdep renvoie l'ensemble des fonctions dont le contexte correspondant est en sortie des transitions de dépendance qui ont CF_g en entrée.

Nous pouvons maintenant calculer le domaine de participation à un contexte ayant des transitions d'alternance et de dépendance en sortie :

Soit $C \in \mathcal{C}$ et $(f, ff) \in (\mathcal{I} - \{\mathbf{base}\})^2$ telles que $f \neq ff$, $\text{PF}_f(C) \neq \emptyset$, $\text{PF}_{ff}(C) \neq \emptyset$ et $\text{Dom}_{\text{transition}}(f, ff, C) \neq \emptyset$, on obtient :

$\text{Dom}_{\text{par}}(ff, C) = \text{Dom}_{\text{transition}}(f, ff, C) \cup_{fff \in \text{Foncsuivdep}(ff, C)} \text{Domsuiv}(ff, fff, C)$

La figure 5.13 illustre le calcul du domaine de participation au contexte CF_{ff} . Elle montre par un trait continu les transitions dont l'union des domaines forme le domaine de participation au contexte CF_{ff} . Les domaines des transitions représentées par un trait en pointillés ne sont pas inclus dans ce calcul d'union, soit parce que ces transitions sont antérieures à la transition qui va de CF_f à CF_{ff} soit parce qu'elles suivent CF_{ff} par une alternance.

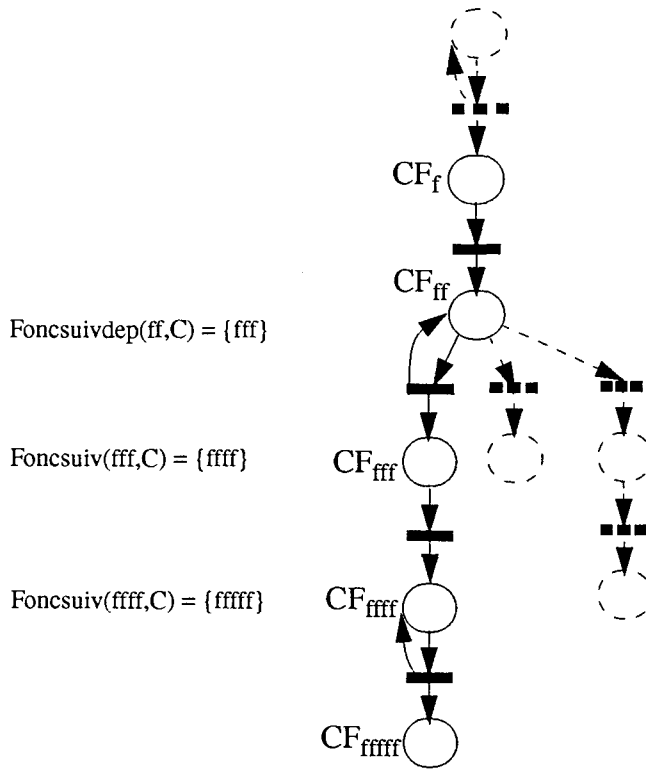


Figure 5.13. Ensemble des transitions dont le domaine concoure au calcul du domaine de participation au contexte CF_{ff}

Exemple

Nous reprenons l'exemple du cycle de vie fonctionnel d'une instance de « Bon_de_livraison ». Nous cherchons à déterminer le domaine de participation relatif à l'attribut « progression » dans les contextes fonctionnels « Stock » et comptable (cf figure 5.12). Nous ne prenons pas, pour l'instant, en compte la troisième transition qui se situe à la sortie de la place correspondant au contexte comptable.

Le domaine de participation pour le contexte fonctionnel « Stock » est donné par :

$$\begin{aligned} \text{Dom}_{\text{par}}(ff, C) &= \text{Dom}_{\text{transition}}(f, ff, C) \cup_{fff \in \text{Foncsuivdep}(ff, C)} \text{Domsuiv}(ff, fff, C) \\ \text{Dom}_{\text{par}}(\text{Stock}, \text{Bon_de_livraison}) &= \text{Dom}_{\text{transition}}(\text{Commercial}, \text{Stock}, \text{Bon_de_livraison}) \cup \\ &\text{Domsuiv}(\text{Stock}, \text{Comptable}, \text{Bon_de_livraison}) \end{aligned}$$

Nous ne prenons donc pas en compte la troisième transition. Par conséquent, nous avons $\text{Domsuiv}(\text{Stock}, \text{Comptable}, \text{Bon_de_livraison}) = \text{Dom}_{\text{transition}}(\text{Stock}, \text{Comptable}, \text{Bon_de_livraison}) = \{ \text{'comptabilisation'} \}$. Par ailleurs, $\text{Dom}_{\text{transition}}(\text{Commercial}, \text{Stock}, \text{Bon_de_livraison}) = \{ \text{'livraison'} \}$. Nous obtenons donc : $\text{Dom}_{\text{par}}(\text{Stock}, \text{Bon_de_livraison}) = \{ \text{'livraison'}, \text{'comptabilisation'} \}$.

Le domaine de participation pour le contexte comptable est donné par :

$$\text{Dom}_{\text{par}}(\text{Stock}, \text{Bon_de_livraison}) = \text{Dom}_{\text{transition}}(\text{Commercial}, \text{Stock}, \text{Bon_de_livraison}) =$$

{ 'comptabilisation' }

5.2.2.3.3 Cas des extrémités

Il existe deux cas d'extrémité à étudier : celle de début du cycle de vie et celle de fin du cycle de vie.

Extrémité de début du cycle de vie

Dans le cadre de la traduction en CROME du cycle de vie, la place du début du cycle, c'est-à-dire celle qui est placée au sommet du réseau de Petri, correspond à un contexte d'instanciation. Seuls les clients d'un contexte d'instanciation peuvent créer des instances et les faire participer à ce contexte, ce qui correspond à la création d'un jeton dans la place du sommet. Cette place étant unique, nous n'associons à chaque classe qu'un seul contexte d'instanciation. Comme il ne peut exister de contraintes de participation pour un contexte d'instanciation, la ou les transitions qui suivent un tel contexte d'instanciation sont nécessairement des transitions de dépendance (cf 3.5.5.3).

Il convient de remarquer que le domaine de participation au contexte d'instanciation est alors égal à l'union des domaines de toutes les transitions du réseau. Ceci correspond bien à un prédicat de participation qui est une tautologie car l'attribut « att-cv » ne peut pas prendre de valeurs ailleurs que dans cette union.

Exemple

Le prédicat de participation des instances de « Bon_de_livraison » au contexte commercial est une tautologie, étant donné qu'il s'agit d'un contexte d'instanciation de cette classe.

Extrémité de fin de cycle de vie

Pour les extrémités qui sont situées en fin de cycle, il existe deux cas :

- elles se terminent par une place. Dans ce cas, l'instance termine son cycle de vie en participant au contexte fonctionnel correspondant à cette place.
- elles se terminent par une transition qui ne possède pas de sortie. Le franchissement d'une telle transition sert à retirer la participation au dernier contexte atteint sans participer à un contexte final. Ces transitions n'ayant pas de sortie, elles ne sont pas des transitions de dépendance mais uniquement d'alternance. Par ailleurs, un contexte ne possède qu'une seule transition de ce type en sortie.

L'existence de transitions sans sortie nous oblige à compléter la formalisation du calcul du domaine de participation à un contexte. Ces transitions étant uniquement d'alternance, elles ne rentrent pas en compte directement dans le calcul du domaine de participation à un contexte mais uniquement dans la définition de la fonction Domsuiv.

Soit $\text{Dom}_{\text{transition}}(f, _, C)$ avec $C \in \mathcal{C}$, $f \in \mathcal{F} - \{\text{base}\}$ $\text{PF}_f(C) \neq \emptyset$, le domaine de valeurs de la transition ne possédant pas de sortie et dont CF_f est le contexte fonctionnel en entrée. S'il n'existe pas une telle transition alors $\text{Dom}_{\text{transition}}(f, _, C) = \emptyset$.

Nous définissons $\text{Domsuiv}(f, _, C)$ comme le domaine de la transition qui a le contexte CF_f en entrée et pas de sortie. Pour une telle transition, il n'existe pas de transitions qui la suivent. Par conséquent, $\text{Domsuiv}(f, _, C)$ est défini ainsi :

$\text{Domsuiv}(f, _, C) = \text{Dom}_{\text{transition}}(f, _, C)$ avec $C \in \mathcal{C}$ telle que $\text{PF}_f(C) \neq \emptyset$.

Prise en compte des extrémités de fin du cycle dans le cas général du calcul des domaines de participation

Dans le cas général, il convient seulement de changer le calcul de $\text{Domsuiv}(f, ff, C)$ qui est maintenant le suivant :

$$\text{Domsuiv}(f, ff, C) = \text{Dom}_{\text{transition}}(f, ff, C) \cup_{ff \in \text{Foncsuiv}(ff, C)} \text{Domsuiv}(ff, fff, C) \cup \text{Domsuiv}(ff, _, C).$$

Exemple

Nous reprenons l'exemple du cycle de vie fonctionnel d'une instance de « Bon_de_livraison ». Nous cherchons à déterminer le domaine de participation relatif à l'attribut « progression » dans les contextes fonctionnels « Stock » et comptable (cf figure 5.12) en prenant en compte, cette fois-ci, la troisième transition à laquelle nous conférons le domaine de transition { 'comptabilise' }.

La prise en compte de cette troisième transition a pour conséquence de modifier le calcul de $\text{Domsuiv}(\text{Stock}, \text{Comptable}, \text{Bon_de_livraison})$ qui devient le suivant :

$$\text{Domsuiv}(\text{Stock}, \text{Comptable}, \text{Bon_de_livraison}) = \text{Dom}_{\text{transition}}(\text{Stock}, \text{Comptable}, \text{Bon_de_livraison}) \cup \text{Domsuiv}(\text{Comptable}, _, \text{Bon_de_livraison}) = \{ \text{'comptabilisation'}, \text{'comptabilise'} \}$$

Par conséquent, le domaine de participation des instances de « Bon_de_livraison » au contexte « Stock » a pour valeur :

$$\text{Dom}_{\text{par}}(\text{Stock}, \text{Bon_de_livraison}) = \{ \text{'livraison'}, \text{'comptabilisation'}, \text{'comptabilise'} \}$$

Le prédicat de participation des instances de « Bon_de_livraison » au contexte « Stock » est donc le suivant (cf annexe A) :

$$(\text{progression} = \text{'livraison'}) \text{ OR } (\text{progression} = \text{'comptabilisation'}) \text{ OR } (\text{progression} = \text{'comptabilise'})$$

5.2.2.4 Cycle de vie fonctionnel d'un couple d'instances

Nous nous intéressons maintenant à la traduction en CROME du cycle de vie fonctionnel d'un couple d'instances (cf la problématique introduite en 1.8.2). Nous commençons par la définition d'un couple d'instances et de la notion de cycle de vie fonctionnel d'un tel couple. Ensuite, il sera possible de traduire en CROME le cycle de vie fonctionnel d'un couple à l'aide des contraintes de participation des instances du couple.

5.2.2.4.1 Définition de la notion de couple d'instances

Un couple d'instances est défini à partir de deux classes et d'une relation entre ces deux classes ayant un unique contexte fournisseur et telle que la cardinalité minimale et maximale à chaque extrémité soit égale à un. Un couple d'instances est alors constitué par une instance de chacune des deux classes connectées par une occurrence de la relation. Cette relation doit être partagée dans tous les contextes où les deux instances doivent participer en tant que couple.

Les propriétés de CROME sur les relations partagées nous permettent d'établir que, dans tous les contextes consommateurs de la relation du couple, chaque instance de l'une des deux classes doit être connectée à une instance de l'autre classe pour participer à ce contexte. Ceci est dû à la cardinalité minimale de un à chaque extrémité de la relation. Par ailleurs, chaque instance d'une classe ne peut être connectée à plus d'une instance de l'autre classe à cause de la cardinalité maximale de un à chaque extrémité dans le contexte fournisseur de la relation. Chaque instance d'une classe reste donc connectée avec la même instance de l'autre classe

dans tous les contextes consommateurs de la relation ou n'y participe pas.

Couple d'instances ayant le même cycle de vie fonctionnel

Pour que les instances aient le même cycle de vie fonctionnel, la relation qui connecte les deux classes doit être localisée dans le contexte de base. Les deux classes ont alors le même contexte d'instanciation qui est le seul fournisseur des occurrences de la relation⁹. Une telle relation étant partagée par tous les contextes fonctionnels, la participation des deux instances à un contexte se fait toujours de façon simultanée. Les deux instances ont alors toujours le même état fonctionnel et par conséquent, le même cycle de vie fonctionnel. Ce cycle de vie fonctionnel commun constitue le cycle de vie fonctionnel du couple.

Exemple

La figure 5.14 montre que les instances des classes « Bon_de_livraison » et « Bon_de_commande » forment des couples d'instances. En effet, la relation du plan de base « correspond » entre « Bon_de_livraison » et « Bon_de_commande » détermine une cardinalité minimale et maximale de un pour chacune de ses extrémités. Par ailleurs, ses occurrences ne sont fournies que par le contexte commercial, seul contexte d'instanciation des classes « Bon_de_livraison » et « Bon_de_commande ».

```
interface Bon_de_livraison
{
  ...
  relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
}
interface Bon_de_commande
{
  ...
  relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
}

function Commercial interface increment Bon_de_livraison
{
  instantiation function;
  write base part:
  {
    ...
    relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
  }
  ...
}

function Commercial interface increment Bon_de_commande
{
  instantiation function;
  write base part:
  {
    ...
    relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
  }
  ...
}
```

Figure 5.14. Ensemble des caractéristiques nécessaires pour que les instances de « Bon_de_livraison » et « Bon_de_commande » forment des couples d'instances

9. Le contexte d'instanciation d'une classe doit fournir les occurrences des relations ayant cette classe pour extrémité avec une cardinalité minimale supérieure ou égale à un (cf remarques du 3.6.3.2).

Chaque bon de livraison correspond ainsi à un unique bon de commande. Ce dernier est reçu par la fonction commerciale, puis est transmis successivement aux fonctions « Stock » et comptable avec le bon de livraison. De ce fait, les instances de « Bon_de_livraison » et « Bon_de_commande » qui forment un couple possèdent effectivement le même cycle de vie fonctionnel représenté à la figure 5.12.

Couple d'instances possédant un cycle de vie fonctionnel commun sur un ensemble de contextes

Dans ce second cas, la relation qui connecte les deux classes doit être localisée dans le contexte fonctionnel qui est le premier du cycle de vie fonctionnel commun. Chaque occurrence de la relation doit être créée lorsqu'une instance de l'une des deux classes vient à participer à ce contexte. Cette participation peut être due à une instanciation dans ce contexte ou à la vérification des contraintes de participation à ce contexte d'une instance existante. La création d'une occurrence de la relation doit se faire immédiatement après la participation d'une nouvelle instance pour respecter la contrainte de cardinalité minimale (cf 3.6.5). Généralement, ceci demande la création d'une instance de l'autre classe¹⁰ et conduit à la formation d'un nouveau couple au sein du contexte fonctionnel.

Une telle relation peut être partagée au travers d'accès inter-plans dans un certain nombre de contextes consommateurs. Dans tous ces contextes, la participation des deux instances se fait toujours de façon simultanée. Ainsi, la restriction de l'état fonctionnel de ces deux instances au contexte fournisseur de la relation et à ses contextes consommateurs est toujours identique. Elles possèdent donc un cycle de vie fonctionnel commun sur cet ensemble de contextes qui constitue le cycle de vie fonctionnel du couple.

Exemple

Pour cet exemple, nous considérons la base de données incluant la fonction « Achats » du client. Au sein de cette base, nous considérons à nouveau les entités « Bon_de_livraison » et « Bon_de_commande ». Cette fois-ci, un bon de commande est créé par la fonction « Achats » du client puis est transmis à la fonction commerciale. Celle-ci crée le bon de livraison associée au bon de commande reçu. Puis le couple ainsi formé est transmis successivement aux fonctions « Stock » et comptable. Le cycle de vie fonctionnel des instances de ces deux classes est illustré à la figure 5.16.

La figure 5.15 montre comment les instances des classes « Bon_de_livraison » et « Bon_de_commande » forment des couples d'instances dans le contexte commercial. La relation « correspond » du contexte commercial entre « Bon_de_livraison » et « Bon_de_commande » détermine une cardinalité minimale et maximale de un pour chacune de ses extrémités. Cette relation est partagée dans les contextes « Stock » et comptable¹¹. Par conséquent, dans ces contextes, les instances des classes « Bon_de_livraison » et « Bon_de_commande » participent par couple.

10. Il est difficile de faire autrement à cause de la cardinalité maximale de un à chaque extrémité de la relation.

11. Bien que ce ne soit pas explicité sur cette figure, les incréments de description des deux classes dans le contexte comptable contiennent, comme pour le contexte « Stock », un accès inter-plans vers la relation « correspond » introduite dans le contexte commercial.

```

function Achats interface increment Bon_de_commande
{
    instantiation function;
    ...
}

function Commercial interface increment Bon_de_commande
{
    function extent condition:
        (progression = 'transmission_commercial') OR
        (progression = 'livraison') OR (progression = 'comptabilisation') OR (progression = 'comptabilise');
    function part:
        {
            ...
            relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
        }
        ...
}

function Commercial interface increment Bon_de_livraison
{
    instantiation function;
    function part:
        {
            ...
            relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
        }
        ...
}

function Stock interface increment Bon_de_commande
{
    function extent condition:
        (progression = 'livraison') OR (progression = 'comptabilisation') OR (progression = 'comptabilise');
    access function Commercial:
        {
            ...
            relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
        }
        ...
}

function Stock interface increment Bon_de_livraison
{
    access function Commercial:
        {
            ...
            relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
        }
        ...
}

function Comptable interface increment Bon_de_commande
{
    function extent condition: progression = 'comptabilisation';
    ...
}

```

Figure 5.15. Les instances de « Bon_de_livraison » et « Bon_de_commande » forment des couples d'instances dans les contextes commercial, « Stock » et comptable

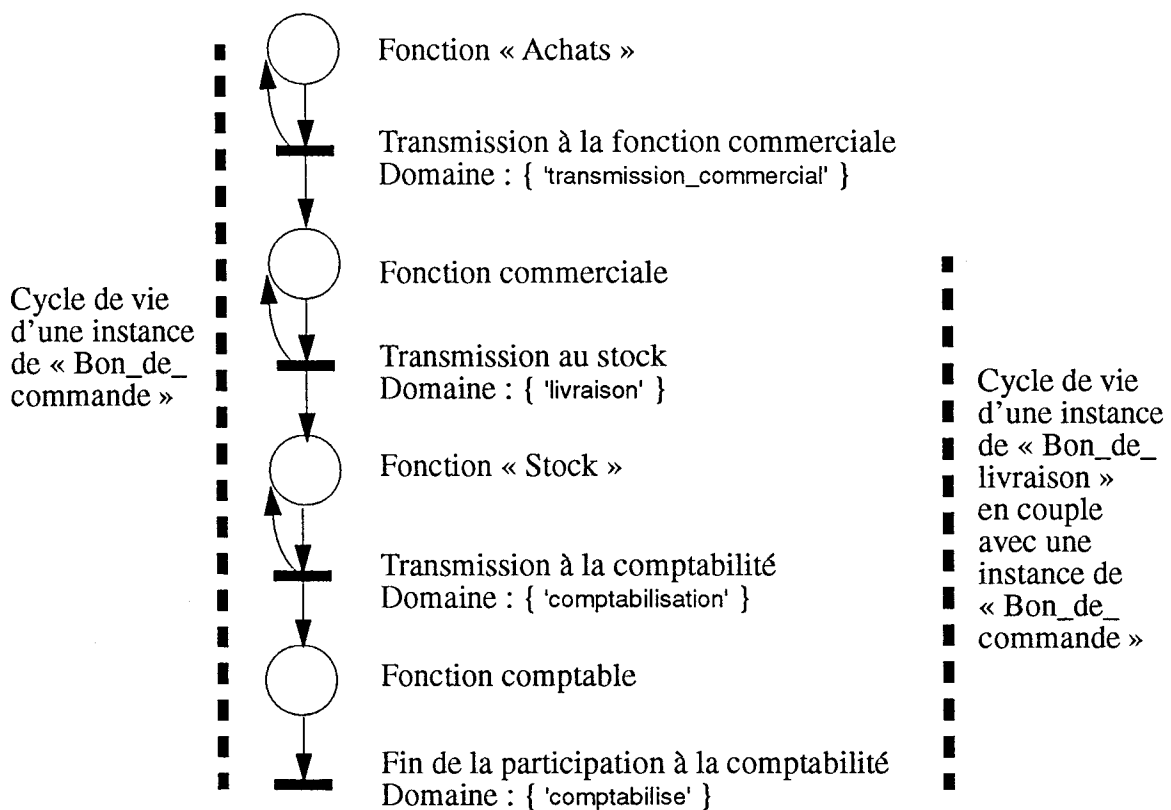


Figure 5.16. Réseau de Petri représentant le cycle de vie fonctionnel d'une instance de « Bon_de_commande » et d'une instance de « Bon_de_livraison ».

5.2.2.4.2 Traduction en CROME du cycle de vie fonctionnel d'un couple d'instances

De la même façon que nous avons traduit en CROME le cycle de vie fonctionnel d'une instance, nous proposons de traduire en CROME le cycle de vie fonctionnel d'un couple d'instances. En fait, la mise en œuvre est très simple à réaliser à cause d'une propriété de la relation qui connecte les deux instances : les contraintes de participation à un contexte consommateur de cette relation de l'une des deux instances s'appliquent à l'autre instance. Ainsi, nous pouvons utiliser la traduction du réseau de Petri en un ensemble de contraintes telle qu'elle a été décrite en 5.2.2.3 et l'appliquer à l'une des instances. On détermine ainsi le cycle de vie fonctionnel du couple¹².

Exemple du couple d'instances ayant le même cycle de vie fonctionnel

Les instances des classes « Bon_de_livraison » et « Bon_de_commande » forment des couples ayant un seul cycle de vie fonctionnel. Nous avons traduit en CROME le cycle de vie

12. Il convient de remarquer que la relation qui unit les deux instances du couple constitue une contrainte de participation aux différents contextes du cycle de vie fonctionnel. Mais nous respectons notre hypothèse de 5.2.2.3 : cette contrainte est vérifiée dès le début du cycle de vie fonctionnel du couple.

fonctionnel de « Bon_de_livraison ». Par conséquent, ce cycle de vie fonctionnel devient le cycle de vie des couples d'instances de « Bon_de_livraison » et « Bon_de_commande ». Nous ne devons ajouter aucune contrainte de participation à l'une de ces deux classes autre que la relation « correspond » introduite dans le plan de base (cf figure 5.14) pour que la traduction soit effective.

Exemple du couple d'instances possédant un cycle de vie fonctionnel commun sur un ensemble de contextes

La figure 5.15 montre que la classe « Bon_de_commande » détient les prédicats de participation issus de la traduction du cycle de vie fonctionnel de ses instances¹³. Dès qu'une instance de « Bon_de_commande » forme un couple avec une instance de « Bon_de_livraison » dans le contexte commercial, le cycle de vie fonctionnel de « Bon_de_commande » devient le cycle de vie fonctionnel d'un couple d'instances de « Bon_de_commande » et « Bon_de_livraison ». Ceci est vérifié dans les contextes commercial, « Stock » et comptable où la relation « correspond » entre les deux classes est introduite ou partagée.

Ceci correspond bien au fait que le cycle de vie fonctionnel des instances de « Bon_de_commande » est commun avec celui des instances de « Bon_de_livraison » dans les contextes commercial, « Stock » et comptable.

5.3. Intégration d'un composant « base de données » existant lors de la conception

5.3.1 Démarche d'intégration de composants « bases de données » existants lors de la conception

5.3.1.1 Besoins et objectifs

Lors de la conception d'une base de données, il peut être nécessaire d'intégrer une ou plusieurs bases de données existantes, appelées par la suite « composants BD » ou plus simplement composants. Cette intégration a pour principaux objectifs :

- reprendre des parties du schéma des composants dans le schéma conceptuel de la nouvelle base ;
- exploiter les informations des composants dans la nouvelle base, en totalité ou en partie.

Cette intégration doit préserver l'existence, la conception et l'autonomie des composants. Ces composants participent à la nouvelle base de données mais leur intégration ne provoque aucun changement pour leurs clients. Elle doit donc être transparente pour les clients des composants et ne provoquer pratiquement aucune modification de leur administration¹⁴.

Cette intégration doit également être transparente pour les clients de la nouvelle base de données. Ceux-ci n'ont pas à connaître l'existence des composants sous-jacents et encore

13. Il faut noter que l'attribut « progression » fait partie, dans cet exemple, de la description de base de « Bon_de_commande ».

14. à part bien sûr tout ce qui est relatif à l'intégration.

moins leur modèle de données, leur schéma, etc. Le fait que certaines informations ne soient pas directement issues de cette nouvelle base mais proviennent d'un composant existant doit être pris en charge par des mécanismes appropriés et encapsulés mais en aucun cas, par les clients de la nouvelle base¹⁵.

La différence avec l'intégration de vues étudiée à la section 2.3 du chapitre 2 provient principalement de l'exploitation d'une partie ou de la totalité des informations des composants au sein de la nouvelle base [Spaccapietra92]. Les schémas-vues employés lors de l'intégration de vues ne sont que des schémas non peuplés qui sont ensuite perdus. Ici, les composants ne sont pas perdus mais continuent à exister et être exploités dans leur mode originel.

Dans la plupart des travaux sur l'intégration de composants, la nouvelle base de données ne détient pas d'informations propres. Elle exploite les informations issues des composants à la façon d'un schéma-vue (cf 2.2.3.3). Elle est désignée sous le nom de base de données fédérée ou de *fédération*. Une fédération offre à ses clients la possibilité de percevoir et de manipuler de façon transparente les données d'un ensemble de composants « BD » décrites par un schéma intégré, appelé *schéma fédéré*. [Sheth90] indique qu'une fédération peut également détenir ses propres informations décrites par le schéma d'un composant spécifique appelé « composant auxiliaire » qui doit être intégré avec les autres composants.

Les travaux sur la fédération sont très nombreux et abordent les nouvelles problématiques en plus du problème de l'intégration de schémas-vues. Nos références sont des travaux exclusivement réalisés dans le cadre du modèle « objet » au niveau du schéma fédéré. Nous en donnerons les principales caractéristiques en 5.3.1.3 après avoir étudié la problématique spécifique de l'intégration de composants et l'architecture d'une fédération en 5.3.1.2.

5.3.1.2 Problématique de l'intégration de composants : architecture d'une base de données fédérée

5.3.1.2.1 Problèmes spécifiques à l'intégration de composants : différences avec l'intégration de schémas-vues non peuplés

Composant peuplé

Les composants à intégrer détiennent des informations, ce qui signifie que les extensions des classes composant leur schéma sont peuplées d'instances. Ceci constitue une première différence avec l'intégration de vues où la problématique d'intégration s'arrêtait aux schémas. Il faut donc mettre en œuvre des mécanismes d'intégration des instances en plus de ceux d'intégration des schémas. Cette intégration des instances est nécessaire pour :

- n'avoir qu'une seule identité (OID) pour un référent du monde réel ;
- déterminer pour les instances de la nouvelle base fédérée l'ensemble des informations qui lui sont relatives et qui sont issues des différents composants ;
- éventuellement résoudre les conflits de valeurs (changement d'échelles, par exemple).

Cette intégration assure la transparence vis-à-vis des clients de la nouvelle base fédérée.

Traduction dans le modèle de données commun

De par leur conception séparée, il est fort probable que le schéma des différents composants à intégrer n'ait pas été conçu dans le même modèle de données. Le schéma fédéré

15. comme c'est le cas des « multi-databases » faiblement couplées [Sheth90]

est lui exprimé dans un modèle de données unique, commun à l'ensemble des composants. Il est donc nécessaire de mettre en œuvre des mécanismes de traduction qui permettent de passer du modèle du composant à celui de la fédération lors de l'exploitation. Cette traduction concerne les requêtes et leurs résultats ainsi que les ordres de mises à jour avec leurs informations associées.

Préservation du composant

Un composant, une fois intégré à une base fédérée, doit continuer à mener sa propre existence. Il peut d'ailleurs être alors intégré à d'autres bases fédérées. Son autonomie doit donc être préservée le plus possible, à l'exception des concessions faites à la fédération. Pour fédérer, il n'est donc pas possible d'importer l'ensemble des informations du composant dans la nouvelle base puis de supprimer ce composant. Ceci impose de mettre en place une articulation entre le composant et la nouvelle base fédérée. Cette articulation est réalisée lors de l'exploitation pour le composant.

Les requêtes issues des clients de la fédération sont distribuées dans les différents composants qui détiennent les informations pertinentes pour former le résultat au travers de cette articulation. Les différents résultats partiels d'une requête distribuée obtenus dans les différents composants doivent être intégrés pour former le résultat.

Un autre aspect de la préservation de l'autonomie du composant est qu'il est seul à déterminer les informations¹⁶ qu'il met à disposition de la fédération.

5.3.1.2.2 Architecture d'une base fédérée

L'architecture adoptée par la plupart des travaux dans le domaine des bases fédérées est donnée dans [Sheth90]. A l'image de l'architecture ANSI/X3/SPARC [ANSI75] à trois niveaux, l'architecture de [Sheth90] contient cinq niveaux :

- niveau local : niveau composé des schémas des composants (encore appelés *schémas locaux*) ;
- niveau « composant » : niveau composé des schémas « composants ». Un schéma « composant » est un schéma local traduit dans son intégralité dans le modèle de données commun. Entre le niveau local et le niveau « composant », un processeur de transformation assure la traduction des requêtes et des données dans les deux sens¹⁷ ;
- niveau « export » : niveau composé des schémas exportés. Un schéma exporté est un schéma-vue d'un composant. Il est exprimé dans le modèle de données commun. Il décrit l'ensemble des informations d'un composant pertinentes pour la fédération. Un processeur de filtrage assure l'extraction des informations pertinentes entre le niveau « composant » et le niveau « export » ;
- niveau fédéré : le schéma fédéré est le résultat de l'intégration complète des schémas exportés. On peut donc considérer qu'il s'agit aussi d'un schéma-vue sur l'union des schémas exportés. Un processeur de construction s'occupe de la distribution des requêtes provenant des clients de la base fédérée et de l'intégration des résultats issus des ces requêtes distribuées. Lors de l'exploitation, il assure l'articulation entre le niveau fédéré et le niveau « export » ;
- niveau externe : niveau composé de schémas-vues externes. Un schéma-vue du niveau externe décrit les informations de la fédération pertinentes pour un de ses clients. On retrouve un processeur de filtrage entre le niveau fédéré et le niveau externe.

16. avec leurs descriptions

17. un processeur relève de la phase d'exploitation.

La figure 5.17 illustre cette architecture à cinq niveaux.

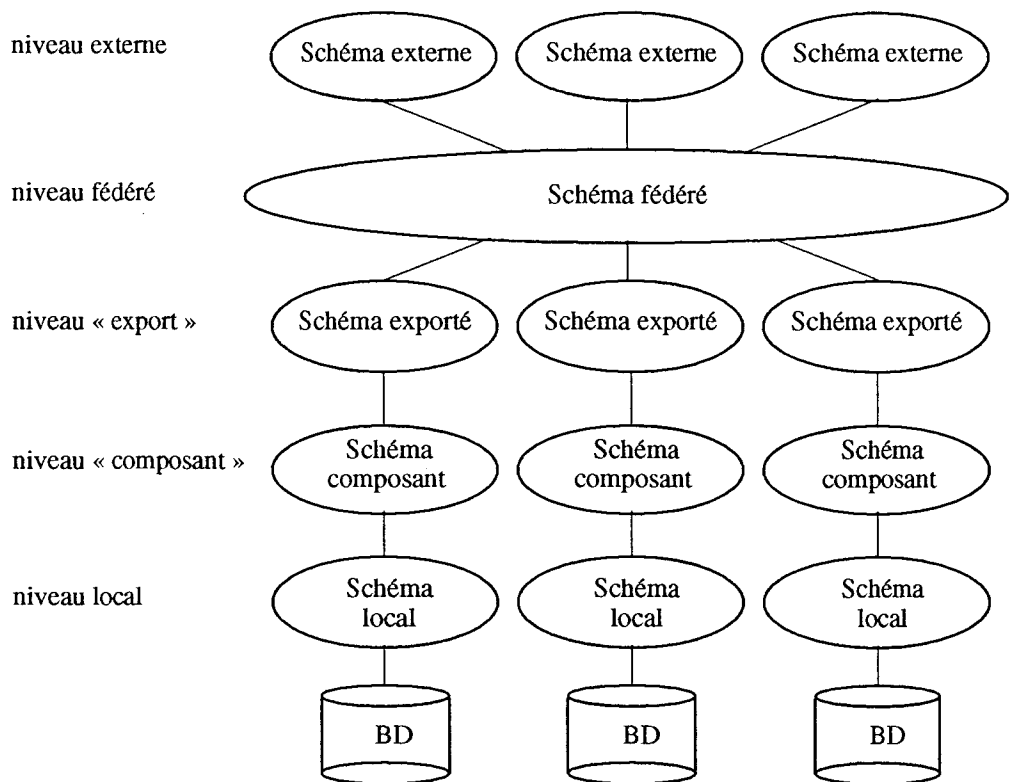


Figure 5.17. Architecture à cinq niveaux d'une base de données fédérée

Tous les niveaux de cette architecture ne sont pas forcément présents dans une base de données fédérée. Par exemple, un schéma local déjà exprimé dans le modèle commun se situe au niveau « composant ». Le niveau « export » n'est pas nécessaire si le composant exporte l'intégralité de ses données. Son schéma exporté est alors égal à son schéma « composant ».

La base de données fédérée peut contenir des descriptions pour de nouvelles informations qui lui sont propres. Ces descriptions sont incluses dans le schéma fédéré. La présence de ces descriptions peut être considérée comme provenant de l'intégration du schéma d'un composant auxiliaire. La figure 5.18 illustre une architecture fédérée incluant un tel composant.

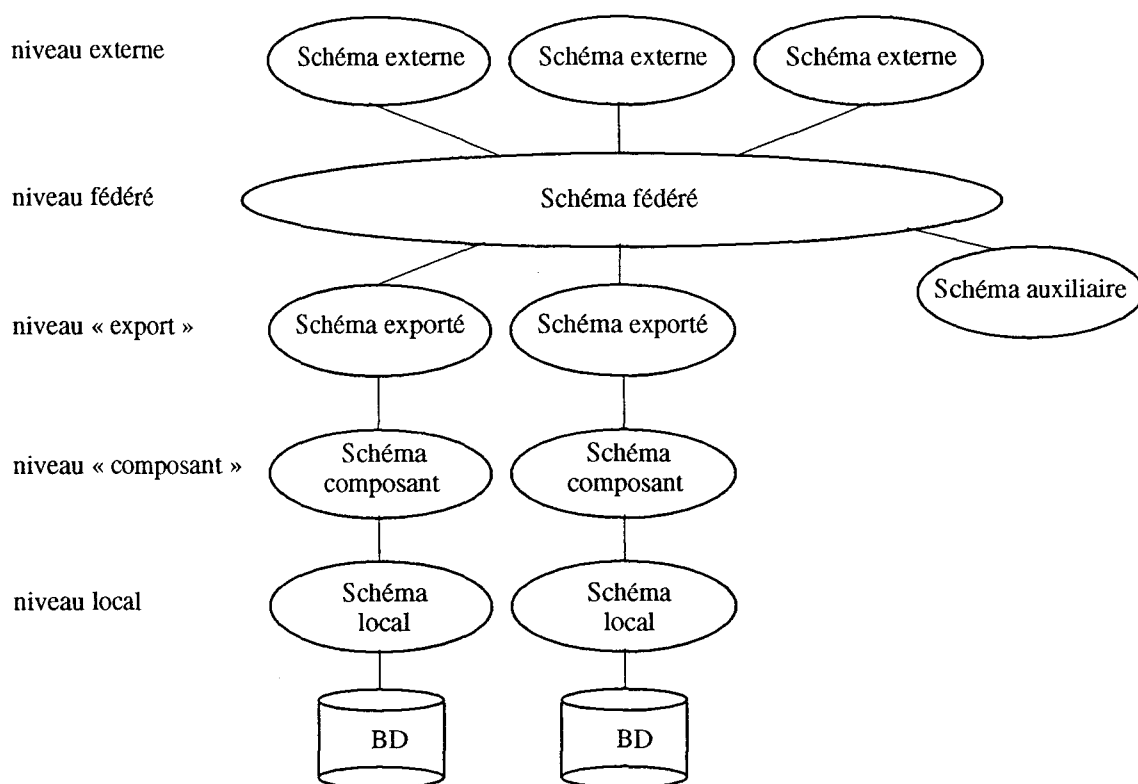


Figure 5.18. Architecture d'une base de données fédérée incluant un schéma auxiliaire décrivant les informations spécifiques de la base fédérée

5.3.1.3 Techniques d'intégration de composants existants

Nous présentons ici les principales techniques d'intégration dans le cadre du modèle « objet » sans nous préoccuper de la phase de traduction. Nous verrons cependant que certaines techniques peuvent aisément inclure cette phase qui n'est alors plus préalable à l'intégration.

Les techniques d'intégration dans le cadre du modèle « objet » relèvent de différentes approches pour intégrer schémas et instances. Nous avons étudié en détail les deux approches suivantes¹⁸ :

- l'approche par classes-vues préservant l'identité des instances des différents composants. Pour préserver l'identité, il faut préalablement étendre cette notion au travers des composants. Ainsi deux instances de deux composants distincts sont dites identiques si elles sont coréférentes (cf [Ferber89]). Cette coréférence est établie par une fonction qui est indépendante des instances finales¹⁹. Les classes-vues du schéma fédéré sont généralement des unions²⁰ (cf table 2.1 du chapitre 2) de chaque classe correspondante dans les composants. Cette approche est prônée dans COCOON [Andersson93a, Andersson93b, Scholl93, Tresch94], dans Pegasus [Ahmed91] et dans Vodak [Kaul90, Klas90]. COCOON emploie également cette technique au niveau méta pour intégrer les fonctions et les classes, c'est-à-dire pour intégrer les schémas. Cette approche est présentée de façon plus détaillée en 5.3.1.3.1 ;

18. voir aussi [Pitoura96] pour une synthèse sur les bases de données fédérées à objets.

19. Elle ne peut contenir d'identifiants d'une instance finale (OID) ou de valeurs d'une instance finale.

20. On trouve aussi le terme « généralisation » qui est, ici, équivalent.

- l'approche par génération d'instances au niveau fédéré. Ces instances détiennent des valeurs issues des différents composants à l'aide de requêtes. Leur identité est alors donnée par un ensemble d'attributs qui forment une clef. Le schéma fédéré est constitué de classes-vues qui décrivent ces instances et la façon de calculer leurs valeurs au travers de requêtes. Pour chaque classe, une requête donne la façon de déterminer son extension. Cette approche est prônée dans IRO-DB [Busse94] ainsi que dans les objets imaginaires de [Abiteboul91]. Il faut noter que les techniques par héritage ascendant citées dans [Boulanger97] relèvent de cette approche que nous étudions de façon plus précise en 5.3.1.3.2.

Une troisième approche serait l'approche par assertions sémantiques au niveau des instances finales. Au niveau méta, ces assertions déterminent l'identité des éléments des schémas entre composants. Nous avons étudié ces assertions lors de l'étude de l'intégration de schémas-vues au chapitre 2 (section 2.3). Il pourrait être envisagé de généraliser ces assertions au niveau des instances finales mais un tel travail est quasiment impossible car il faut spécifier une assertion pour chaque ensemble d'instances finales²¹ coréférentes.

Nous pouvons aussi citer [Boulanger97] qui propose une approche basée sur la conception d'objets et de méta-objets « application ». Ceux-ci sont capables de répartir une requête provenant d'un client de la fédération dans les différents composants et de déterminer le résultat de cette requête par intégration des différents résultats obtenus. Citons encore la possibilité d'utiliser des logiques terminologiques pour réaliser l'intégration des schémas des composants [Bertino96].

5.3.1.3.1 Approche par classes-vues préservant l'identité dans le cadre de COCOON

Dans le cadre de COCON, le schéma fédéré est constitué de classes-vues préservant l'identité (cf 2.2.3). Cette préservation oblige à étendre l'identité des instances au travers des composants. Deux instances de deux composants distincts sont dites identiques si elles sont coréférentes. Soit « GOB » le nom de la base de données fédérée et « LOB » le nom de l'un de ses composants. La fonction d'identité globale « =_{GOB} » est définie en conséquence de la façon suivante (d'après la définition de [Scholl93] que nous avons ici simplifiée) :

$$\begin{aligned} =_{GOB}: \quad & \text{object@GOB} \times \text{object@GOB} \rightarrow \text{boolean} \\ & (o, o') \rightarrow (\exists \text{ LOB tel que } (o, o') \in \text{Objects@LOB} \times \text{Objects@LOB} \text{ et } o =_{LOB} o') \\ & \text{ou} \\ & (\exists C@LOB, \exists C'@LOB' \text{ telles que } (o, o') \in C@LOB \times C'@LOB' \text{ et} \\ & \exists \text{ same}_{C@LOB, C'@LOB'} \text{ telle que } o' =_{LOB'} \text{same}_{C@LOB, C'@LOB'}(o)) \end{aligned}$$

object@GOB désigne le type maximal de toutes les instances de la fédération. Objects@GOB est la classe de toutes les instances de la fédération.

Si LOB est l'un des composants de la fédération, object@LOB désigne le type maximal de toutes les instances de ce composant. Objects@LOB est la classe de toutes les instances de ce composant. C@LOB désigne une classe de composant.

L'identité globale est donc donnée soit par l'identité locale si les deux instances font partie du même composant (cette identité est notée « =_{LOB} » dans le composant LOB) soit au travers d'une fonction « same_{C@LOB, C'@LOB'} ». Cette fonction renvoie pour une instance de la classe « C'@LOB' » l'instance coréférente de la classe « C@LOB ».

Une fonction « same » doit être définie explicitement par l'utilisateur pour chaque classe

21. Chacune des instances de cet ensemble se trouve bien sûr dans un composant distinct.

de chaque composant vis-à-vis de toutes les autres classes correspondantes dans les autres composants. Elle établit la coréférence entre les instances de toutes ces classes. Une telle fonction est basée sur les clefs présentes dans chacune de ces classes. Cette fonction doit être ajoutée dans chaque classe par l'intermédiaire de l'opérateur d'augmentation « extend » de COCOON (cf table 2.1 du chapitre 2).

Exemple

Pour exprimer qu'une instance de la classe « Etudiants@UnivDB » est identique avec une instance de la classe « Employes@TravailDB » si leurs attributs « nom » sont égaux, on peut introduire la fonction « same_{Etudiants@UnivDB,Employes@TravailDB} » ainsi :

define view Etudiants'@UnivDB

as

extend [same_{Etudiants@UnivDB,Employes@TravailDB} := **pick**(**select**[nom(e) = nom(s)]

(e : Employes@TravailDB))] (s : Etudiants@UnivDB);

La fonction inverse doit aussi être introduite dans la classe « Employes@TravailDB » :

define view Employes'@TravailDB

as

extend [same_{Employes@TravailDB,Etudiants@UnivDB} := **pick**(**select**[nom(e) = nom(s)]

(e : Etudiants@UnivDB))] (s : Employes@TravailDB);

Remarquons que le nom a été choisi ici comme clef commune pour les étudiants et les employés.

L'identité entre instances des schémas des composants définie à l'aide d'une fonction « same » s'applique également aux instances de leurs métaschémas. Il est par conséquent possible d'unifier les fonctions, les types ou les classes. Ceci permet d'unifier l'approche d'intégration des instances et des éléments des schémas.

Exemple

Pour unifier les fonctions nom@UnivDB et nom@TravailDB²² d'une part et age@UnivDB et age@TravailDB²³ d'autre part, la fonction suivante peut être définie dans le méta-schéma du composant « UnivDB » :

define view Functions'@UnivDB

as

extend[same_{UnivDB,TravailDB} := **pick**(**select**[(fname(f) = "nom" and fname(g) = "nom") or ((fname(f) = "age" and fname(g) = "age")) (g : Functions@TravailDB))] (f : Functions@UnivDB);

Il faut remarquer que la fonction « same » contient ici les valeurs d'instances, à savoir les noms des fonctions à la différence d'une fonction « same » relative à des instances finales.

Une fonction inverse doit également être définie dans le composant « TravailDB » :

define view Functions'@TravailDB

as

extend[same_{TravailDB,UnivDB} := **pick**(**select**[(fname(f) = "nom" and fname(g) = "nom") or ((fname(f) = "age" and fname(g) = "age")) (g : Functions@UnivDB))] (f : Functions@TravailDB);

Une fois déterminée l'identité entre les instances et entre les fonctions de leurs types

22. Ces fonctions ont respectivement pour domaine Etudiants@UnivDB et Employes@TravailDB.

23. Ces fonctions ont aussi respectivement pour domaine Etudiants@UnivDB et Employes@TravailDB.

respectifs, il est possible de définir une classe-vue union de deux classes-vues augmentées de la fonction « same » dans l'un des composants ou dans les deux :

```
define view Personnes as Etudiants'@UnivDB union Employes'@TravailDB;
```

Le type de la classe-vue « Personnes » contient les propriétés unifiées « nom » et « age ». Son extension est peuplée des instances des classes Etudiants@UnivDB et Employes@TravailDB, les instances identiques n'apparaissant qu'une seule fois dans l'extension du fait des fonctions « same_{Etudiants@UnivDB,Employes@TravailDB} » et « same_{Employes@TravailDB,Etudiants@UnivDB} ».

5.3.1.3.2 Approche par génération d'instances

Dans cette approche, le schéma fédéré est constitué de classes-vues²⁴ dont l'extension est peuplée de nouvelles instances obtenues par des requêtes. Ces instances sont le pur produit de requêtes, tant pour la valeur de leurs attributs que pour leur identité. Un client de la base fédérée peut ensuite manipuler cet ensemble de classes-vues ainsi que leur extension.

L'extension d'une classe-vue est donnée par une requête. Le type d'une classe-vue est défini explicitement [Busse94] ou est égal au type du résultat de la requête donnant l'extension [Abiteboul91].

IRO-DB

Les classes-vues de IRO-DB[Busse94] sont spécifiées par une interface écrite en ODL [ODMG93, ODMG97] standard et par un « mapping » qui constitue une extension d'ODL. L'identité est donnée par une ou plusieurs clefs qu'il est possible de spécifier dans une interface « ODL ». Pour rappel, une interface « ODL » se présente sous la forme syntaxique suivante :

```
interface type_name {
  extent <extension_name>;
  keys <attribute_name> {, <attribute_name>;}
  {attribute <domain_type> <attribute_name>;}
  {relationship <targ_type> <attr_name> inverse <targ_type>::<inv_attr_name>;}
}
```

Une fois l'interface d'une classe-vue spécifiée, il faut ensuite spécifier un « mapping » qui va donner l'extension et pour chaque instance, la ou les valeurs de chaque attribut. La forme syntaxique d'un mapping est la suivante :

```
mapping [ imported ] type_name {
  origin <origin_type> <origin_name>;
  {origin <origin_type> <origin_name>;}
  def_ext <extension_name> as <query>;
  {def_att <attribute_name> as <query>;}
  {def_rel <attribute_name> as <query>;}
}
```

La clause « imported » est utilisée lorsque le mapping de la classe-vue utilise des classes de base importés depuis d'autres composants « BD ». La clause « origin » permet de définir les types utilisés dans les clauses « def_ext », « def_att » et « def_rel ». La clause « def_ext » spécifie la requête qui détermine l'extension de la classe-vue. Les clauses « def_att » et « def_rel » spécifient les requêtes qui déterminent les valeurs des différents attributs et extrémités de relation des instances de la classe-vue.

24. classes-vues qu'il faut donc distinguer des classes-vues de COCOON dont l'extension est peuplée d'instances des classes de base.

O₂ [Abiteboul91]

Les objets ou instances imaginaires de [Abiteboul91]²⁵ sont introduits par une requête qui va donner le type et l'extension de la classe-vue de ces objets. L'exemple suivant montre une requête définissant une classe d'objets imaginaires :

class Family includes imaginary

(select [Husband:H, Wife: H.Spouse] from H in Person where H.sex = 'male')

Le type et l'extension de la classe-vue « Family » sont donnés par le type et l'extension du résultat de la requête qui la définit. Tous les attributs du type du résultat de la requête forment une clef des instances de « Family », leur conférant ainsi une identité de valeur.

Une requête de définition d'une classe d'objets imaginaires ne doit donc plus préserver l'identité des instances de la classe de base. Elle peut donc contenir tous les opérateurs présents dans l'algèbre relationnelle comme la jointure. Ceci constitue une différence importante avec l'algèbre « objet » de COCOON et MultiView (cf 2.2.3.1.2).

Le type de la classe « Family » peut ensuite être enrichi par l'ajout d'un attribut calculé, par exemple :

attribute Children in class Family has value

(select P from Person where P in self.Husband.Children or P in self.Wife.Children)

Il faut noter que l'attribut « Children » ne fait pas partie de la clef des instances de « Family ».

Le choix de la requête de définition ne doit pas être fait au hasard, le type de son résultat constituant la clef des instances imaginaires de la classe-vue. La possibilité d'ajouter ensuite des attributs calculés qui ne font pas partie de la clef permet de choisir finement la requête. [Abiteboul91] donne des exemples de requêtes bien et mal construites selon que la clef des instances de la classe-vue correspondante est correcte ou non.

Remarque

Dans le cadre de cette approche par génération d'instances, les instances sont des n-uplets du modèle relationnel. Ainsi, il est possible avec cette approche d'intégrer des composants dont le modèle de données est le modèle relationnel sans devoir passer par une phase de traduction.

5.3.2 Intégration d'un composant lors de la conception d'une base de données multi-fonctionnelle en CROME

Nous nous proposons maintenant d'étudier la possibilité d'intégrer lors de la conception d'une base de données multi-fonctionnelle en CROME un composant « base de données » (BD) existant, peuplé, relatif au même domaine et correspondant à l'une des fonctions de l'organisation. Cette intégration a pour but de permettre la récupération, lors de l'exploitation, les informations fonctionnelles issues de ce composant dans les contextes en cours de conception. Elle doit préserver l'autonomie du composant existant (cf 5.3.1.2.1), c'est-à-dire qu'elle doit être transparente pour les clients de ce dernier, tant à la conception que lors de l'exploitation²⁶.

25. Ces travaux n'entrent pas directement dans le cadre des travaux de la fédération mais constituent une contribution pour l'élaboration des classes-vues d'un schéma fédéré.

Pour récupérer les informations fonctionnelles du composant, son intégration doit lui conférer un statut de contexte fonctionnel. Ainsi, il sera possible de mettre en œuvre le mécanisme des accès inter-plans pour récupérer les informations fonctionnelles désirées. Notre démarche se rapproche ainsi de la conception d'intégration de composants existants par classes-vues préservant l'identité expérimentée en COCOON (cf [Andersson93a, Andersson93b, Scholl93, Tresch94]) et présentée ci-dessus (5.3.1.3.1).

Dans un premier temps, nous faisons l'hypothèse que :

- le schéma du composant a été conçu dans le modèle de données « objet ». Ce modèle doit introduire les relations comme dans celui de l'ODMG, c'est-à-dire au travers d'un lien bi-directionnel garantissant l'intégrité référentielle ;
- les classes de ce schéma incarnent les mêmes entités du domaine que le référentiel de la base de données multi-fonctionnelle²⁷ ;
- l'organisation de ces classes suivant la relation d'héritage est isomorphe à celle de la base multi-fonctionnelle²⁸.

Chaque entité de ce référentiel détermine une classe du composant à intégrer. L'intégration sera ainsi diffusée au sein des entités du domaine.

Nous chercherons donc un moyen pour déterminer la coréférence entre les instances des deux bases. Nous verrons ensuite comment articuler le composant au sein de la base multi-fonctionnelle en lui conférant un statut de contexte fonctionnel.

Puis, dans un second temps, nous relâcherons l'hypothèse d'un composant organisé suivant le même référentiel que la base multi-fonctionnelle. En nous inspirant des techniques d'intégration par génération d'instances (cf 5.3.1.3.2), nous montrerons alors une solution pour reconstituer au sein du composant, les entités de la base multi-fonctionnelle et organiser de façon isomorphe les classes correspondantes. Nous verrons aussi que cette technique permet de s'affranchir de la contrainte du modèle « objet » et de la nécessité du lien bi-directionnel pour les relations.

5.3.2.1 Etablissement de la coréférence entre instances du composant et de la base multi-fonctionnelle

Les instances des différentes classes détiennent dans le composant les informations fonctionnelles que l'on cherche à récupérer. Cependant, l'extension de chacune de ces classes dans le composant est disjointe de l'extension de la classe relative à la même entité dans la base multi-fonctionnelle. En effet, pour un même référent du monde réel, il existe deux instances distinctes dans chaque base. Il faut donc établir, relativement à chaque entité, une relation de même référent dite « de coréférence » [Ferber89] entre les instances du composant et celles de la base multi-fonctionnelle. Ainsi, il sera possible, lors de l'exploitation, pour une instance de la base multi-fonctionnelle de déterminer l'instance du composant à laquelle elle correspond et ainsi d'accéder aux informations fonctionnelles voulues.

Dans le cas présent, cette relation est sous-tendue, pour chaque entité, par une paire de clefs. Une clef est présente dans la classe de la base multi-fonctionnelle et l'autre clef est présente dans la classe du composant. Chaque clef a une valeur unique pour chaque référent, ce

26. autonomie de conception et d'exécution

27. Notons qu'il est possible, comme pour les autres fonctions, que toutes les entités du domaine n'interviennent pas dans le composant.

28. organisation introduite dans le plan de base

qui permet de l'identifier²⁹. Une instance de la base multi-fonctionnelle est coréférente avec une instance du composant si et seulement si elles dénotent le même référent, c'est-à-dire si leurs clefs respectives correspondent au même référent. On utilise alors un prédicat basé sur ces deux clefs et qui détermine si elles correspondent au même référent. Nous appelons ce prédicat, le prédicat de coréférence. Il est commun aux deux classes relatives à une même entité³⁰.

Une clef identifie un référent de façon unique. Elle est donc liée à des informations propres à ce référent et donc communes à l'ensemble des fonctions. C'est pourquoi, au sein de la base multi-fonctionnelle, les clefs relatives aux différentes classes sont détenues par leur description de base. Elles sont constituées par une ou plusieurs propriétés de ces descriptions de base. Dans le composant, les clefs sont constituées par une ou plusieurs propriétés de l'intension des classes correspondant aux mêmes entités. Par conséquent, pour une entité, le prédicat exprimant la relation de coréférence des instances sera exprimé à l'aide de propriétés de la description de base de la classe correspondante dans la base multi-fonctionnelle et de propriétés de l'intension de la classe correspondante dans le composant.

Un prédicat de coréférence peut être spécifié en OQL. Il doit être paramétré par deux instances respectives des deux classes relatives à la même entité.

Remarques

- Pour une entité du référentiel, il existe une classe dans la base multi-fonctionnelle et une classe dans le composant. Ces deux classes ne sont généralement pas désignées par le même nom dans le composant et dans la classe. Par conséquent, il convient de spécifier par une assertion que ces deux classes désignent la même entité.
- Un prédicat de coréférence n'est établi que pour les classes pour lesquelles une telle assertion existe. En effet, pour une entité du référentiel, il est indispensable de pouvoir déterminer la relation de coréférence pour intégrer la classe correspondante du composant. S'il est impossible de déterminer une clef dans chaque base, alors l'intégration ne pourra être réalisée pour cette entité.
- Un prédicat de coréférence utilise généralement des propriétés dont le type est un type de littéraux au sens du modèle de l'ODMG (cf 3.2.1.1). En effet, il n'est pas nécessaire d'établir de relation de coréférence pour les littéraux.
- Un prédicat de coréférence peut utiliser des expressions de chemin s'il existe déjà, pour chaque classe des instances obtenues par ces expressions, un prédicat de coréférence.
- Il n'est évidemment pas nécessaire de déterminer une relation de coréférence pour des classes abstraites.
- La clef d'identification du référent des instances d'une classe est une clef d'identification pour les instances de ses sous-classes. Un prédicat de coréférence établi entre deux classes est donc valide pour toutes les sous-classes respectives de ces deux classes. Clefs et prédicats de coréférence sont donc hérités dans les sous-classes.

Par conséquent, une sous-classe peut détenir de nouvelles clefs ou prédicats de coréférence par rapport à l'une de ses surclasses, notamment si elle en hérite d'autres

29. cf la définition des clefs dans le modèle de l'ODMG en 3.2.1.1

30. Plusieurs approches sont également basées sur des assertions de coréférence (cf [Ahmed91, Andersson93a, Andersson93b, Busse94, Klas90, Scholl93, Tresch94])

surclasses. Toutes ces clefs doivent désigner le même référent. Tous les prédicats doivent associer les mêmes instances dans la base multi-fonctionnelle et dans le composant.

Exemple

Nous considérons pour les exemples que la base multi-fonctionnelle est celle dont le schéma est donné à l'annexe A. Nous cherchons à y intégrer un composant « BD » relatif à la fonction « Marketing » qui n'a pas été prise en compte jusqu'à présent. Le schéma de ce composant décrit en ODL est donné à la figure 5.19.

```
interface Supplier
{
  key name,phone;
  attribute string name;
  attribute string address;
  attribute string phone;
  attribute string advertising_info;
  relationship set<Product> makes_advertising_for inverse Product::is_promoted_by;
}

interface Product
{
  attribute string name;
  attribute integer price;
  relationship set<Supplier> :is_promoted_by inverse Supplier::makes_advertising_for;
}

interface Consumer_Product: Product
{
  key consumer_code_product;
  attribute integer consumer_code_product;
  attribute integer advertising_expenses;
}

interface Professional_Product: Product
{
  key professional_code_product;
  attribute string professional_code_product;
  attribute integer mailing_per_year;
}

interface Mixed_Product: Consumer_Product, Professional_Product
{
  attribute string reglementation;
  attribute string advertising_specific_requirements;
}
```

Figure 5.19. Schéma du composant « BD » relatif à la fonction « Marketing » décrit en ODL

Les différentes classes de ce schéma correspondent aux mêmes entités que plusieurs classes de la base multi-fonctionnelle. Les assertions de la figure 5.20 rendent compte de ces correspondances.

```

same-entity(Fournisseur,Supplier);
same-entity(Produit,Product),
same-entity(Produit_Grand_Public,Consumer_Product);
same-entity(Produit_Professionnel,Professional_Product);
same-entity(Produit_Mixte,Mixed_Product);

```

Figure 5.20. Assertions de « même entité » entre les classes de la base multi-fonctionnelle et le composant

Nous nous intéressons maintenant à établir la relation de coréférence pour les classes « Produit_Grand_Public » et « Consumer_Product ». L'intension de cette dernière classe contient l'attribut « consumer_code_product » qui est une clef. Dans la description de base de « Produit_grand_public » dans la base multi-fonctionnelle, on trouve l'attribut « référence_grand_public » qui est aussi une clef (cf annexe A).

Le prédicat de coréférence peut être simplement l'égalité entre les valeurs des deux attributs qui sont des littéraux (entier). On peut l'écrire ainsi en « OQL »

```

coreference(prod_gp:Produit_grand_public,cons_prod:Consumer_Product) :
prod_gp.reference_grand_public = cons_prod.consumer_code_product

```

« prod_gp » est une instance de la classe « Produit_grand_public » dans la base multi-fonctionnelle et « cons_prod » est une instance de la classe « Consumer_Product » dans le composant.

Ici, les deux clefs sont comparables directement. Supposons maintenant que les valeurs de l'attribut « consumer_code_product » ne soient comparables avec les valeurs de l'attribut « référence_grand_public » qu'au travers d'une fonction de conversion. Supposons aussi que « Conv » soit une instance d'une classe « Conversion_Code »³¹. Le prédicat peut alors s'écrire :

```

coreference(prod_gp:Produit_grand_public,cons_prod:Consumer_Product) :
prod_gp.reference_grand_public = Conv.CodeToRef(cons_prod.consumer_code_product)

```

« CodeToRef » est une méthode de conversion d'un code utilisé dans le composant vers une référence. Elle correspond à une fonction bijective entre les valeurs de la clef « référence_grand_public » et les valeurs de la clef « consumer_code_product ».

La figure 5.21 montre les prédicats de coréférence retenus pour intégrer le composant « BD » relatif à la fonction « Marketing ».

31. Cette classe est une classe « outil » qui n'est pas relative à une entité du domaine

```

coreference(un_fournisseur:Fournisseur, a_supplier:Supplier) :
(un_fournisseur.nom = a_supplier.name) AND (un_fournisseur.telephone = a_supplier.phone)

coreference(prod_gp:Produit_grand_public, cons_prod:Consumer_Product) :
prod_gp.reference_grand_public = cons_prod.consumer_code_product

coreference(prod_prof:Produit_Professionnel, prof_prod:Professional_Product) :
prod_prof.reference_professionnelle = prof_prod.professional_code_product

```

Figure 5.21. Prédicats de coréférence pour intégrer le composant « BD » relatif à la fonction « Marketing »

Il convient de remarquer qu'il n' a pas été introduit de prédicat de coréférence pour la classe « Produit » qui est une classe abstraite.

La sous-classe « Produit_mixte » détient, dans la base multi-fonctionnelle, deux clefs héritées, à savoir la référence « grand public » et la référence professionnelle. Il est possible d'utiliser une clef comme l'autre, un produit mixte détenant les deux références. Il va de même dans le composant pour la sous-classe « Mixed_Product ». Enfin, il est possible d'utiliser indifféremment l'un de deux prédicats de coréférence hérités et il n'y a pas lieu d'introduire un prédicat spécifique. En effet, les valeurs des deux clefs d'un « mixed product » dans le composant doivent être égales aux valeurs des deux clefs d'un même produit mixte dans la base multi-fonctionnelle³².

5.3.2.2 Intégration du composant

5.3.2.2.1 Statut de contexte fonctionnel pour le composant

Le composant est, par définition, une base de données correspondant à l'une des fonctions de l'organisation. Il détient donc les informations correspondant à cette fonction. Par ailleurs, comme il a été conçu de façon autonome, il détient également les informations communes à l'ensemble des fonctions³³. A l'instar d'un contexte fonctionnel, le composant détient les informations communes et fonctionnelles qu'il décrit et présente à ses clients (cf 3.5.5.2). Nous pouvons donc lui conférer un statut de contexte fonctionnel.

Le composant détient les intensions et extensions des différentes classes qu'il détermine. Son statut de contexte fonctionnel fait que l'intension de chaque classe du composant équivaut ainsi à une description fonctionnelle, constituée de l'union d'une description de base et d'une partie fonctionnelle. L'ensemble de ces parties fonctionnelles forme le plan du composant.

Il est nécessaire de déterminer ce plan afin de circonscrire les propriétés des différentes classes qui sont relatives à la fonction du composant. Ce plan est déterminé en extrayant au sein de l'intension de chaque classe, les propriétés formant la partie fonctionnelle. Cette extraction est, en quelque sorte, un travail de rétroconception comparativement à la conception en CROME que nous avons présentée dans ce chapitre 3.

Par ailleurs, l'extension de chaque classe du composant constitue une extension fonctionnelle. Notons que cette dernière est fournie localement et non pas par sélection comme dans le cas d'une base multi-fonctionnelle. Ceci est dû à la conception et à l'exploitation autonomes du composant.

Nous proposons d'utiliser la forme « interface increment » pour chaque classe du

32. à une fonction bijective de conversion près.

33. qui sont relatives à des aspects propres aux entités du domaine

composant afin de déterminer le plan du composant. Seule la clause **function part** est nécessaire pour déterminer chaque partie fonctionnelle.

Exemple

La figure 5.22 montre le plan du composant « BD » relatif à la fonction « Marketing » et dont le schéma a été donné à la figure 5.19. Ce plan détient l'ensemble des parties fonctionnelles relatives à la fonction « Marketing » des différentes classes du composant.

Par ailleurs, l'extension des différentes classes du composant est égale à l'extension des classes correspondantes de la base multi-fonctionnelle, moyennant les relations de coréférence définies ci-dessus. Tous les produits et tous les fournisseurs sont pertinents pour la fonction « Marketing ».

```
function Marketing interface increment Supplier
{
  function part:
  {
    attribute string advertising_info;
    relationship set<Product> makes_advertising_for inverse Product::is_promoted_by;
  }
}

function Marketing interface increment Product
{
  function part:
  {
    relationship set<Supplier> :is_promoted_by inverse Supplier::makes_advertising_for;
  }
}

function Marketing interface increment Consumer_Product
{
  function part:
  {
    attribute integer advertising_expenses;
  }
}

function Marketing interface increment Professional_Product
{
  function part:
  {
    attribute integer mailing_per_year;
  }
}

function Marketing interface increment Mixed_Product
{
  function part:
  {
    attribute string advertising_specific_requirements;
  }
}
```

Figure 5.22. Plan du composant « BD » relatif à la fonction « Marketing »

5.3.2.2.2 *Articulation du composant*

Ayant conféré le statut de contexte fonctionnel au composant, il nous est maintenant possible d'articuler celui-ci au sein de l'ensemble des contextes de la base multi-fonctionnelle. Cette articulation a pour objectif d'offrir aux autres contextes les moyens d'accéder aux informations fonctionnelles du composant. Classiquement, le moyen d'accéder aux informations fonctionnelles d'un autre contexte se fait au sein des entités au travers d'accès inter-plans. Nous proposons donc d'employer ces accès en leur donnant le composant comme contexte fournisseur. Le plan du composant obtenu précédemment permet de déterminer les propriétés auxquelles il est possible d'accéder. L'articulation entre les contextes de la base multi-fonctionnelle et le composant est ainsi diffusée et réalisée au sein des différentes entités du domaine.

Pour une classe de la base multi-fonctionnelle, il est possible de définir dans un contexte fonctionnel un accès inter-plans vers la classe du composant relative à la même entité. De plus, il faut qu'une relation de coréférence ait été établie entre ces deux classes au travers d'un prédicat de coréférence, hérité ou non.

La différence entre ces accès inter-plans vers le composant et les accès classiques réside dans le fait que les premiers s'appuient sur la relation de coréférence entre les instances explicitée en 5.3.2.1. Lors de l'exploitation, l'utilisation d'un accès inter-plans vers le plan du composant au sein d'une instance de la base multi-fonctionnelle oblige à retrouver l'instance correspondante dans le composant. Ceci rend compte de la nécessité d'utiliser la relation de coréférence.

Pour pouvoir exploiter un accès inter-plans vers une relation inter-entités du composant, il faut qu'une relation de coréférence ait été établie, pour chaque entité de la relation, entre les deux classes correspondantes. Dans ce cas, il est également utile d'utiliser la relation de coréférence pour retrouver une instance de la base multi-fonctionnelle depuis son instance coréférente dans le composant, celle-ci ayant été obtenue au travers d'une occurrence de la relation inter-entités.

Exemple

La figure 5.23 montre les accès inter-plans qui ont été ajoutés au sein du contexte fonctionnel « Direction » afin que ses clients puissent accéder à des propriétés du composant « BD » relatif à la fonction « Marketing ».

On peut remarquer l'accès vers la relation « *makes_advertising_for/is_promoted_by* » dans les classes « Fournisseur » et « Produit ». Notons que la définition des extrémités de la relation a été modifiée pour prendre en compte le fait qu'il s'agit d'occurrences entre des instances de la base multi-fonctionnelle³⁴.

Notons que ces accès inter-plans provoquent des dépendances entre le contexte « Direction » et le contexte du composant « Marketing ». Cependant, ces dépendances ne créent pas de sélections supplémentaires dans le contexte « Direction » car tous les produits et tous les fournisseurs sont pertinents pour la fonction « Marketing ».

34. occurrences obtenues à l'aide de la relation de coréférence

```

function Direction interface increment Fournisseur
{
    access function Marketing:
    {
        relationship set<Produit> makes_advertising_for inverse Produit::is_promoted_by;
    }
    ...
}

function Direction interface increment Produit
{
    access function Marketing:
    {
        relationship set<Fournisseur> :is_promoted_by inverse Fournisseur::makes_advertising_for;
    }
    ...
}

function Direction interface increment Produit_grand_public
{
    access function Marketing:
    {
        attribute integer advertising_expenses;
    }
}

function Direction interface increment Produit_professionnel
{
    access function Marketing:
    {
        attribute integer mailing_per_year;
    }
}

function Direction interface increment Produit_mixte
{
    access function Marketing:
    {
        attribute string advertising_specific_requirements;
    }
}

```

Figure 5.23. Accès inter-plans vers le composant « Marketing » ajoutés au sein du contexte « Direction »

5.3.2.3 Reconstitution de la structure de la base multi-fonctionnelle au sein du composant

Dans le cas où le schéma du composant ne vérifie pas les hypothèses énoncées en 5.3.2, il est possible d'utiliser la technique des « classes imaginaires³⁵ » [Abiteboul91, Ye95] pour essayer, au sein du composant, de reconstituer le référentiel et obtenir une structuration identique à celle du plan de base. Cette technique permet également de s'affranchir de la contrainte du modèle « objet » et pour les relations, de l'obligation d'avoir des liens bi-directionnels et la gestion de l'intégrité référentielle. Les extensions des « classes

35. cf 5.3.1.3.2

imaginaires » de [Abiteboul91] ou des classes-vues de [Ye95] sont constituées d'instances dépourvues d'identifiant propre. Ces classes correspondent aux types de littéraux de l'ODMG et plus particulièrement, aux ensembles de structures (type ODL « set<struct> »). Nous n'utilisons donc pas ici le terme de « classe imaginaire » mais uniquement celui de type de littéraux.

Le référentiel est reconstitué au travers de requêtes dont le résultat correspond à une entité du domaine. Le résultat d'une telle requête est un littéral de type « set<struct> ». Chaque élément de cet ensemble est également un littéral pour lequel il est possible d'établir la coréférence avec une instance de la base multi-fonctionnelle. En effet, pour établir la relation de coréférence, nous n'avons pas besoin de l'identité propre des instances du composant. Nous pouvons donc l'établir avec des littéraux à condition qu'il existe pour ceux-ci une ou plusieurs clefs d'identification.

L'articulation du composant peut donc être réalisée avec des littéraux, éléments du résultat de requêtes correspondant à une entité du domaine. L'articulation se fait alors par intégration de ces requêtes et définition d'accès inter-plans vers leurs propriétés.

Exemple

Nous supposons maintenant que le schéma de notre composant « BD » relatif à la fonction ait été décrit dans le modèle de données relationnel et ait été structuré comme l'illustre la figure 5.24. Remarquons que nous avons utilisé les structures du langage ODL pour décrire les intensions des tables³⁶. Leurs extensions sont introduites à l'aide de variables globales de type « set<struct> ». Ces variables globales contiennent donc les n-uplets des différentes tables.

Précisons que :

- les attributs « name » et « phone » constituent une clef pour la table « Supplier_Rec » ;
- l'attribut « consumer_code_product » est une clef pour la table « Consumer_Product_Rec » ;
- l'attribut « professional_code_product » est une clef pour la table « Professional_Product_Rec ».

Les n-uplets de « Rel_advertising_Consumer » et « Rel_advertising_Professional » sont des occurrences des relations « makes_advertising_for/is_promoted_by » qui existent entre les fournisseurs et les produits. L'intension de ces deux tables contient les clefs des tables mises en relation. La table « Rel_Mixed_Product » est une relation dont chaque occurrence lie un n-uplet de la table « Consumer_Product_Rec » avec un n-uplet de la table « Professional_Product_Rec » si ces deux n-uplets ont le même référent qui est donc un produit mixte. Pour chaque produit mixte, il existe donc une seule occurrence de « Rel_Mixed_Product ».

Un tel schéma présente de fortes différences de structuration avec celui de la figure 5.19. D'abord, le modèle de données utilisé est différent. Il n'existe pas de table correspondant ni à l'entité « Produit » ni à l'entité « Produit_Mixte ». Les relations sont introduites par des tables spécifiques et basées sur les clefs.

36. en ne typant les attributs qu'avec des types de littéraux atomiques

```

struct Supplier_Rec
{
    string name;
    string address;
    string phone;
    string advertising_info;
}

struct Rel_advertising_Consumer
{
    string name;
    string phone;
    integer consumer_code_product;
}

struct Consumer_Product_Rec
{
    string name;
    integer price;
    integer consumer_code_product;
    integer advertising_expenses;
}

struct Rel_advertising_Professional
{
    string name;
    string phone;
    integer professional_code_product;
}

struct Professional_Product_Rec
{
    string name;
    integer price;
    string professional_code_product;
    integer mailing_per_year;
}

struct Rel_Mixed_Product
{
    string professional_code_product;
    integer consumer_code_product;
    string reglementation;
    string advertising_specific_requirements;
}

set <Supplier_Rec> the_suppliers;
set <Rel_advertising_Consumer> adv_cons_occurrences;
set <Consumer_Product_Rec> the_cons_products;
set <Rel_advertising_Professional> adv_prof_occurrences;
set <Professional_Product_Rec> the_prof_products;
set <Rel_Mixed_Product> mixed_product_occurrences;

```

Figure 5.24. schéma du composant « BD » relatif à la fonction « Marketing » dans le modèle relationnel décrit à l'aide des structure d'ODL

Nous cherchons maintenant à déterminer les requêtes basées sur les tables de la figure 5.24 et dont le résultat va correspondre aux classes du schéma de la figure 5.19. La figure 5.25 montre les requêtes qu'il est possible de définir pour reconstituer le référentiel. Chaque requête correspond à une entité du domaine. Nous avons utilisé le langage OQL pour définir chacune des requêtes.

La clef des éléments du résultat de la requête « `Supplier()` » est constituée des attributs « `name` » et « `phone` ». Pour les éléments du résultat des requêtes « `Consumer_Product()` » et « `Professional_Product()` », les clefs sont respectivement constituées de l'attribut « `consumer_code_product` » et de l'attribut « `professional_code_product` ». Les éléments du résultat de la requête « `Product()` » n'ont pas de clef, ceux de la requête « `Mixed_Product()` » ont deux clefs, soit l'attribut « `consumer_code_product` » soit l'attribut « `professional_code_product` ». Ces clefs sont issues des clefs des tables à partir desquelles est calculé le résultat de ces requêtes.

L'ensemble des littéraux du résultat des requêtes « `Supplier()` », « `Consumer_Product()` » et « `Professional_Product()` » correspond à l'extension des tables dont elles sont issues. Il faut remarquer que la forme **`select distinct`** permet de s'assurer que le résultat est bien un ensemble ne contenant pas plusieurs fois le même littéral.

Pour la requête « `Product()` », le résultat est un ensemble union des résultats de « `Consumer_Product()` » et de « `Professional_Product()` »³⁷. Son type est de la forme « `set<struct>` » où la structure est la plus petite borne supérieure des types des éléments de « `Consumer_Product()` » et de « `Professional_Product()` »³⁸.

Pour la requête « `Mixed_Product` », chaque élément du résultat correspond à une occurrence de la relation « `Rel_Mixed_Product` » et porte en plus les valeurs des attributs de « `Consumer_Product()` » et de « `Professional_Product()` ». Le résultat de cette requête est inclus dans celui des requêtes « `Consumer_Product()` » et de « `Professional_Product()` »³⁹.

La relation entre « `makes_advertising_for/is_promoted_by` » est reconstituée dans les requêtes « `Supplier()` », « `Consumer_Product()` » et « `Professional_Product()` ». Chaque extrémité est un ensemble de littéraux dont le type est donné par la requête. Pour « `Supplier()` », l'extrémité est une union de deux ensembles dont la plus petite borne supérieure du type des éléments est égale au type du résultat de la requête « `Product()` ».

37. Cette union est possible en OQL car les éléments de ces deux ensembles ont des types compatibles.

Deux types de littéraux « `struct` » sont compatibles s'il possèdent une plus petite borne supérieure.

38. soit :

```
struct {
    string name;
    integer price;
    struct {...} is_promoted_by;
}
```

39. les types de leurs éléments étant compatibles, une telle comparaison est possible.

```

define Supplier() as
  select distinct name: s.name, address: s.address, phone: s.phone, advertising_info: s.advertising_info,
  makes_advertising_for: (
    (select distinct cp from cp in Consumer_Product()
     where exists (select occ from occ in adv_cons_occurrences where (occ.name = s.name) and
      (occ.phone = s.phone) and (occ.consumer_code_product = cp.consumer_code_product)))
    union
    (select distinct pp from pp in Professional_Product()
     where exists (select occ from occ in adv_prof_occurrences where (occ.name = s.name) and
      (occ.phone = s.phone) and (occ.professional_code_product = pp.professional_code_product))))
  from s in the_suppliers;

define Consumer_Product() as
  select distinct name: cp.name, price: cp.price, consumer_code_product: cp.consumer_code_product,
  advertising_expenses: cp.advertising_expenses,
  is_promoted_by: (
    select distinct s from s in Supplier()
    where exists
      (select occ from occ in adv_cons_occurrences where (occ.name = s.name) and
      (occ.phone = s.phone) and (occ.consumer_code_product = cp.consumer_code_product)))
  from cp in the_cons_products;

define Professional_Product() as
  select distinct name: pp.name, price: pp.price,
  professional_code_product: pp.professional_code_product, mailing_per_year: pp. mailing_per_year,
  is_promoted_by: (
    select distinct s from s in Supplier()
    where exists
      (select occ from occ in adv_prof_occurrences where (occ.name = s.name) and
      (occ.phone = s.phone) and (occ.professional_code_product = pp.professional_code_product)))
  from pp in the_prof_products;

define Mixed_Product() as
  select distinct name: pp.name, price: pp.price,
  professional_code_product: pp.professional_code_product, mailing_per_year: pp. mailing_per_year,
  is_promoted_by: pp.is_promoted_by, consumer_code_product: cp.consumer_code_product,
  advertising_expenses: cp.advertising_expenses, reglementation: occ.reglementation,
  advertising_specific_requirements: occ.advertising_specific_requirements,
  from cp in Consumer_Product(), pp in Professional_Product(), occ in the mixed_product_occurrences
  where
    (occ.consumer_code_product= cp.consumer_code_product) and
    (occ.professional_code_product= pp.professional_code_product);

define Product() as
  Consumer_Product()
  union
  Professional_Product();

```

Figure 5.25. Requêtes reconstituant le référentiel d'identités

A partir de ces requêtes, il est donc possible de reprendre les étapes décrites en 5.3.2.2.1 et en 5.3.2.2.2, c'est-à-dire :

- établir les assertions de « même entité » avec les classes du plan de base ;
- établir les relations de coréférence entre les littéraux ainsi obtenus et les instances de la base multi-fonctionnelle ;
- circonscrire au sein du résultat des différentes requêtes les parties fonctionnelles ;
- déterminer les accès inter-plans.

5.4. Conclusion

Dans ce chapitre, nous avons proposé deux extensions de l'approche CROME. La particularité de ces deux extensions est de se baser sur les notions existantes de CROME sans en introduire de nouvelles, ni d'enrichir celles existantes.

Ainsi, la traduction du cycle de vie fonctionnel d'instances ou de couple d'instances a été réalisée au travers des notions de contrainte de participation, modalité de fourniture d'attribut du plan de base et de partage de relations introduites dans le plan de base.

L'intégration s'est basée sur la notion de contexte fonctionnel conférée au composant à intégrer, puis sur les possibilités d'interaction entre contextes, notamment les accès inter-plans.

Il faut remarquer que les notions utilisées par les deux extensions sont des caractéristiques propres aux contextes fonctionnels.

Par ailleurs, ces extensions imposent certaines limitations dont :

- un réseau de Petri restreint à un arbre pour la traduction du cycle de vie fonctionnel d'instances avec une solution de traduction basée sur une forme restreinte des prédicats de participation ;
- la prise en compte du cycle de vie fonctionnel commun à plusieurs instances restreinte à des couples d'instances ;
- une intégration limitée à un composant qui doit correspondre à l'une des fonctions de l'organisation. Par ailleurs, nous n'avons pas étudié les besoins d'interaction de ce composant avec les contextes de la base multi-fonctionnelle après l'intégration.

Lors de la conclusion générale qui suit, nous présenterons un certain nombre de perspectives relatives à ces deux extensions.

Conclusion générale

Bilan

Dans le premier chapitre, nous avons examiné les besoins spécifiques de conception de la base de données à objets sous-tendant le SI d'une entreprise. Ces besoins proviennent de la structuration en fonctions de son organisation. Nous avons établi que cette structuration de l'organisation doit être prise en compte au sein du schéma de la base de données, tant à la conception qu'à l'exploitation. Puis, nous avons constaté l'orthogonalité entre fonctions et entités du domaine. Ayant adopté l'approche « objet », nous avons pu alors montrer que la structuration du schéma nécessite de conférer, de façon systématisée, de multiples descriptions aux entités du domaine relativement aux fonctions. Chacune des descriptions multiples d'une entité peut détenir soit des informations communes soit des informations fonctionnelles.

Plus finement, les informations relatives à une fonction sont détenues par les instances appartenant aux différentes extensions fonctionnelles. La notion d'extension fonctionnelle nous a amenés à établir la notion d'état fonctionnel d'une instance et par conséquent, à envisager le problème de description multiple et évolutive des instances. Nous avons également abordé les problèmes du partage et de transmission d'informations entre fonctions, de représentation du cycle de vie fonctionnel d'une instance et de l'intégration d'un composant « base de données » existant et peuplé.

Dans le deuxième chapitre, nous avons d'abord présenté les techniques de vues « utilisateurs » des bases de données à objets et les méthodes d'intégration de vues. Nous avons constaté que les techniques de vues « utilisateurs » procèdent principalement par extraction des descriptions et par sélection des instances lors de l'exploitation. D'autre part, elles ne procurent aucun moyen pour systématiser les descriptions multiples des entités du domaine. Quant aux méthodes d'intégration de vues, elles prennent en compte, lors de la conception, les descriptions issues des différentes catégories d'utilisateurs¹ mais elles aboutissent à l'élaboration d'un schéma conceptuel canonique sans préservation de la structuration vis-à-vis de ces catégories d'utilisateurs.

Nous avons présenté au chapitre 3 l'approche CROME dans le cadre des bases de données à objets. Nous y avons introduit les différentes composantes des contextes fonctionnels :

- les descriptions de base, parties et descriptions fonctionnelles des entités systématisées par les plans ;
- les accès inter-plans et les modalités de fourniture des propriétés du plan de base comme support de partage et de transmission d'informations entre fonctions, ;
- les extensions fonctionnelles contenant l'ensemble des instances d'une classe participant à une fonction ;
- les contraintes d'intégrité référentielle, de cardinalité et les contraintes sur les propriétés ;
- les différentes contraintes de participation qui déterminent l'état fonctionnel des instances.

La conception du schéma de la base de données se traduit par la conception de chaque contexte fonctionnel. Lors de l'exploitation, chaque contexte peut être considéré comme une base de données que les clients de la fonction correspondante peuvent exploiter classiquement.

1. qui sont les fonctions dans notre cas

Cette approche a été présentée dans le cadre du modèle standard des bases de données à objets de l'ODMG. Elle a donné lieu à une adaptation du langage de spécification ODL, appelée « CromeODL ». Ceci confère une certaine généralité à l'approche car CromeODL comme ODL se veulent indépendants d'un SGBD spécifique.

L'approche CROME se situant à un niveau logique de conception, elle offre la possibilité de réaliser des mises en œuvre dans différents SGBD. Nous avons donc, dans le chapitre 4, étudié une traduction d'une conception en CROME par les techniques de vues des SGBD à objets et une mise en œuvre dans les SGBD supportant Java qui a donné lieu à un prototype basé sur le SGBD POET. La mise en œuvre par les techniques de vues a montré ses limites, notamment dans la prise en compte des contraintes de participation, des mécanismes d'interaction et des contraintes d'intégrité sur les propriétés. Cette réalisation a mis en avant la granularité fine des techniques de vues et a montré les limites de ces dernières. La seconde mise en œuvre a montré comment utiliser les caractéristiques du langage Java et comment enrichir les classes de gestion des extensions de POET pour réaliser les contextes fonctionnels.

Nous avons enfin proposé deux extensions dans le chapitre 5 :

- la traduction du cycle de vie fonctionnel d'une instance par un ensemble de prédicats de participation de cette instance aux différents contextes ;
- l'intégration d'un composant « bases de données » existant qui permet l'exploitation des informations détenues par celui-ci dans les autres contextes. Notons que cette intégration est réalisée au sein des entités du domaine au travers d'accès inter-plans adaptés pour prendre en compte l'existence de plusieurs instances coréférentes.

Dans ce cinquième chapitre, nous avons également présenté les différentes approches pour intégrer des composants « bases de données » lors de la conception, et notamment les approches de fédération.

Perspectives

Extensions du modèle

Plusieurs extensions du modèle peuvent être envisagées :

- la prise en compte des sous-relations inter-entités qui sont une spécialisation des relations (cf [Morejon94-p131]). Chaque extrémité d'une sous-relation est une sous-classe de l'extrémité correspondante dans la relation d'origine. De plus, une sous-relation peut restreindre les cardinalités à chaque extrémité. Il serait intéressant d'étudier les conséquences de la notion de sous-relation sur le partage des relations ;

- la définition de classes-vues locales qui serviraient de filtres au sein de contextes fonctionnels. Ce travail pourrait se baser sur les travaux de COCOON [Scholl91, Scholl94] et de MultiView [Rundensteiner92a]. Ceci permettrait de disposer d'un outil de rémanence de requêtes au sein des contextes, requêtes qui ne porteraient que sur les informations pertinentes pour les clients de ces contextes ;

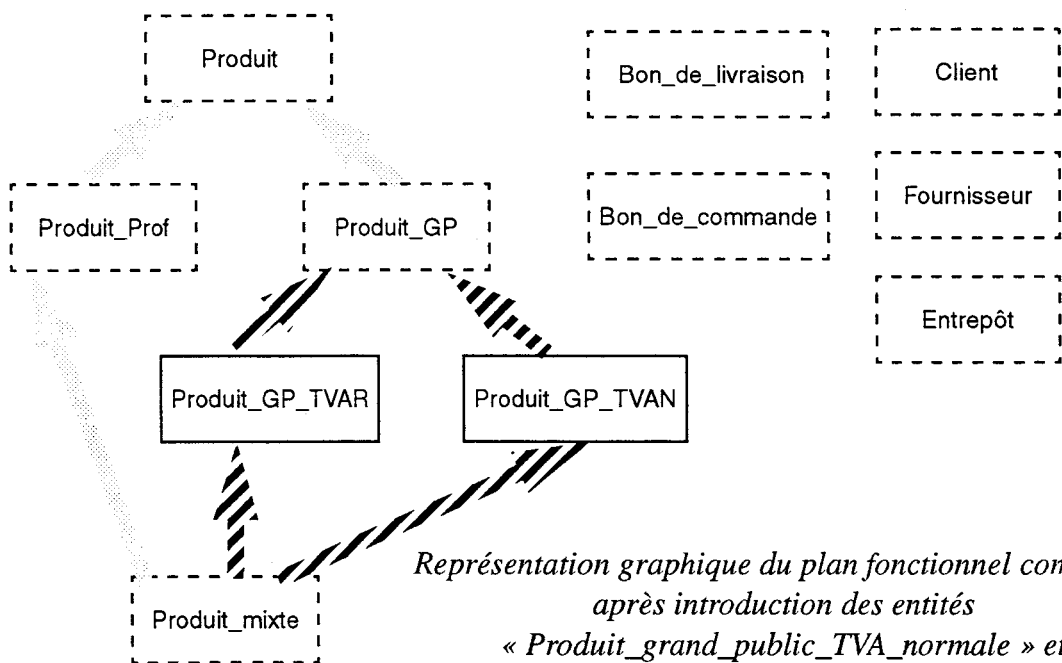
- la gestion de l'homonymie entre propriétés des parties fonctionnelles lors de l'exploitation. Cette homonymie provient de l'indépendance de conception des plans et contextes fonctionnels (cf 3.7.1) ;

- la définition des contraintes d'intégrité sur les propriétés des entités dans le contexte de base. Celles-ci devraient être vérifiées dans le contexte de base et dans chaque contexte

fonctionnel. Si une contrainte de ce type comprend des expressions de chemin sur des relations, sa vérification dans le contexte de base n'implique pas sa vérification dans les contextes où les instances des extrémités de cette relation sont sélectionnées. Une étude approfondie reste nécessaire pour déterminer si une telle contrainte d'intégrité devrait devenir alors une contrainte de participation comme c'est le cas des contraintes de cardinalité des relations partagées (cf 3.6.3.2).

- l'expression de cycles de vie fonctionnels plus complets comme dans [Al-Jadir93] que ceux présentés au chapitre 5 (section 5.2.2) où nous avons introduit plusieurs restrictions. Il serait notamment intéressant d'étudier la possibilité d'utiliser un réseau de Petri pour spécifier et décrire de façon générale le cycle de vie d'une ou plusieurs entités au sein de la base de données sans imposer une solution de traduction. Par ailleurs, la généralisation d'un couple d'instances à des groupes de plusieurs instances reste à accomplir.

- la possibilité d'introduire une nouvelle entité spécifique à une fonction telle que sa classe associée enrichisse dans le plan correspondant la hiérarchie issue du plan de base. Cet enrichissement signifie que la nouvelle classe est insérée dans cette hiérarchie en induisant une sous-classification plus fine de cette dernière. Par exemple, il peut être pertinent au sein de la fonction comptable de distinguer les produits « grand public » par rapport à leur taux de TVA : normal ou réduit. Ceci conduit à introduire deux nouvelles entités au sein du plan comptable : « Produit_grand_public_TVA_normale » et « Produit_grand_public_TVA_réduite » dont les classes associées sont sous-classes de « Produit_grand_public » et surclasses de « Produit_mixte ». La figure suivante illustre le plan comptable tel qu'il pourrait l'être dans le cadre d'une extension de l'approche CROME. Rappelons que les parties fonctionnelles sont représentées comme des classes mais avec un encadré pointillé. Il faut noter que nous avons également représenté les entités qui participent implicitement à la fonction comptable suite à l'enrichissement d'une de leurs surclasses. C'est le cas de toutes les entités relatives à une sous-classe de « Produit ». Par ailleurs, les flèches noires hachurées représentent les liens d'héritage locaux entre parties fonctionnelles et nouvelles classes.



Représentation graphique du plan fonctionnel comptable
après introduction des entités
« Produit_grand_public_TVA_normale » et
« Produit_grand_public_TVA_réduite »

Hiérarchie des fonctions au sein de l'organisation

Nous avons pris pour hypothèse la conception de la base de données d'une organisation structurée en fonctions. Nous avons également fait l'hypothèse que toutes les fonctions sont disjointes entre elles. Il pourrait maintenant être intéressant d'envisager le cas d'organisations où une ou plusieurs fonctions sont elles-mêmes structurées en sous-fonctions. Ceci pourrait conduire à la constitution d'une hiérarchie de plans et de contextes fonctionnels. Ceci donnerait aussi lieu à des hiérarchies entre parties fonctionnelles au sein des classes du domaine.

Ceci permettrait de réaliser un partage plus fin des descriptions. Les descriptions d'un plan fonctionnel seraient partagées par ses « sous-plans ». Au niveau des contextes, l'extension fonctionnelle d'une classe relative à une fonction contiendrait l'extension fonctionnelle de cette même classe relative à chaque sous-fonction. Notons que ceci conduirait à une certaine forme de hiérarchisation de l'état fonctionnel : la participation à une fonction devenant une condition nécessaire de participation à l'une de ses sous-fonctions.

Prise en compte des aspects comportementaux

Dans ce mémoire, nous avons considéré des objets serveurs d'information et n'avons pas développé l'aspect « traitement de l'information ». Les besoins de traitement qui sont fréquents dans le cadre des bases de données sont relégués aux clients des contextes fonctionnels. Un atout de l'approche « objet » par rapport à l'approche « relationnelle » est l'intégration du langage de programmation au sein du SGBD, offrant ainsi une solution au problème de l'« impedance mismatch » [Bertino93-p82]. Cette intégration du langage de programmation, dans le cadre de l'approche « objet », donne lieu à un regroupement des aspects « données » et des aspects comportementaux au sein des objets de la base de données. Dans le cadre de CROME, la prise en compte des aspects comportementaux doit se faire en tenant compte des descriptions multiples et systématisées des entités.

Les travaux de Gilles Vanwormhoudt portent sur l'approche CROME dans un cadre de programmation [Vanwormhoudt97a-b, Vanwormhoudt99]. D'une façon très synthétique, on peut dire de cette approche qu'elle reprend les mêmes principes de structuration en plans de la description des entités que dans le cadre des bases de données. Les méthodes peuvent être introduites au niveau du plan de base ou des plans fonctionnels. Celles introduites dans le plan de base possèdent un caractère générique vis-à-vis des fonctions. Elles peuvent être enrichies dans les différents plans fonctionnels. L'affinement d'une méthode dans une sous-classe doit se faire localement au plan d'introduction de cette méthode. Par ailleurs, l'approche prend en compte les besoins d'interaction entre contextes au travers des méthodes : une méthode locale à un contexte peut invoquer une méthode d'un autre contexte².

Il convient de confronter ces travaux relativement aux besoins de prise en compte des traitements dans les objets des bases de données. Ces travaux sur la programmation ne s'attaquent pas aux aspects spécifiques des bases de données. En particulier, ils n'abordent pas les problèmes d'extension : la notion d'extension fonctionnelle n'est pas prise en compte au sein des contextes. De ce fait, les solutions d'interaction entre contextes au travers des méthodes ne nous semblent pas immédiatement applicables aux objets des bases de données. D'autres aspects relatifs aux bases de données comme les contraintes d'intégrité, l'utilisation

2. par un self-message qualifié par le contexte de la méthode invoquée

des méthodes au sein du langage de requêtes, les problèmes d'encapsulation doivent également être étudiés.

Nouveaux problèmes d'intégration de composants et généralisation à la fédération

Nous avons proposé l'intégration d'un composant « base de données » lors de la conception d'une base de données multi-fonctionnelle en CROME. Nous avons pris pour hypothèse que ce composant corresponde à l'une des fonctions de l'organisation. L'articulation entre ce composant et la base de données est réalisée en conférant un statut de contexte fonctionnel au composant. Les clients des autres contextes de la base de données peuvent ensuite exploiter les informations fonctionnelles du composant au travers d'accès inter-plans.

Une première extension possible de ce travail serait d'enrichir le composant intégré pour qu'il puisse à son tour exploiter les informations fonctionnelles issues des nouveaux contextes. Ceci suppose un enrichissement du composant par des mécanismes qui seraient des accès inter-plans depuis la base multi-fonctionnelle vers le composant. Cet enrichissement peut éventuellement induire d'autres modifications dans le composant car celui-ci pouvait gérer lui-même ces informations fonctionnelles des autres contextes avant son intégration. Remarquons qu'il s'agit ici de nouveaux problèmes d'intégration, les approches par fédération excluant de telles modifications des composants pour préserver leur autonomie (cf [Sheth90]).

Par ailleurs, il pourrait être intéressant de généraliser cette approche pour la conception de bases de données fédérées multi-fonctionnelles en partant exclusivement d'un ensemble de composants existants sans imposer que ceux-ci correspondent aux différentes fonctions de l'organisation. Le but d'une telle approche serait alors de « retrouver » les contextes fonctionnels au sein de cet ensemble de composants par rapport à un référentiel commun. Ce référentiel pourrait être déterminé à partir des descriptions présentes dans les schémas des différents composants. Cette démarche permettrait de s'approcher de l'architecture à cinq niveaux des bases de données fédérées de [Sheth90] (cf 5.3.1.2.2) en associant chaque schéma externe à une fonction de l'organisation.

Annexe A

SI d'une entreprise de fabrication et distribution

Cet annexe contient l'ensemble du schéma de la base de données sous-tendant le SI d'une entreprise industrielle qui achète, fabrique et distribue ses produits. Les entités du domaine de l'organisation sous-jacente sont :

- les partenaires externes décrits par la classe abstraite « Part_externe » ;
- les clients décrits par la classe « Client » sous-classe de « Part_externe » ;
- les fournisseurs décrits par la classe « Fournisseur » une sous-classe de « Part_externe » ;
- les produits décrits par la classe abstraite « Produits » ;
- les produits « grand public » décrits par la classe « Produit_grand_public » sous-classe de « Produits » ;
- les produits professionnels décrits par la classe « Produit_professionnel » sous-classe de « Produits » ;
- les produits mixtes à la fois « grand public » et professionnels décrits par la classe « Produit_mixte » sous-classe de « Produit_grand_public » et de « Produit_mixte » ;
- les bons de livraison décrits par la classe « Bon_de_livraison » ;
- les bons de commandes décrits par la classe « Bon_de_commande » ;
- les entrepôts décrits par la classe « Entrepot ».

Les contextes fonctionnels de l'organisation sont les suivants :

- le contexte commercial dont le plan associé enrichit explicitement les entités « Produit », « Client », « Fournisseur », « Bon_de_livraison » et « Bon_de_commande » ;
- le contexte « Fabrication » dont le plan associé enrichit explicitement les entités « Produit », « Produit_grand_public », « Produit_professionnel », « Part_externe », « Fournisseur » et « Entrepot » ;
- le contexte « Stock » dont le plan associé enrichit explicitement les entités « Produit », « Produit_grand_public », « Produit_professionnel », « Produit_mixte », « Client », « Fournisseur », « Bon_de_livraison », « Bon_de_commande » et « Entrepot » ;
- le contexte comptable dont le plan associé enrichit explicitement les entités « Produit », « Client », « Fournisseur », « Bon_de_livraison », « Bon_de_commande » et « Entrepot » ;
- le contexte « SAV » dont le plan associé enrichit explicitement les entités « Produit », « Client », « Fournisseur » et « Entrepot » ;
- le contexte « Direction » dont le plan associé enrichit explicitement les entités « Part_externe », « Client », « Fournisseur » et « Produit ».

Les pages suivantes contiennent la description en ODL adapté pour nos besoins (encore appelé CromeODL) des interfaces correspondant aux descriptions de base des classes du domaine ainsi que la description de leurs incréments de description pour chaque contexte fonctionnel. Ces descriptions en CromeODL sont complétées à chaque fois par une représentation graphique des incréments. La notation utilisée provient de celle employée dans [ODMG97] et que nous avons enrichie ainsi :

- nous avons ajouté les cardinalités minimales et maximales à chaque extrémité d'une relation inter-entités ;
- les incréments de descriptions sont représentés par un rectangle en trait pointillé ;

- au sein d'un incrément de description, on trouve respectivement le nom de l'entité associé, les accès inter-plans, le prédicat de participation et la partie fonctionnelle. Les propriétés du plan de base dont le contexte est fournisseur de la valeur ainsi que les contraintes d'intégrité n'ont pas été représentées pour simplifier les schémas ;

- le nom de l'entité est suivi d'une étoile si le contexte est un contexte d'instanciation de la classe correspondant à cette entité ;

- chaque accès inter-plans prend la forme « **contexteFournisseur:nomPropriété** ». Lorsque cette notation est utilisée pour une relation, cette forme est employée à chaque extrémité de la relation. Lorsqu'elle est employée pour les autres attributs, la partie « **contexteFournisseur:** » peut être factorisée à plusieurs attributs (à l'intérieur du rectangle représentant un incrément) ;

- chaque relation d'héritage issue du plan de base entre parties fonctionnelles est notée avec une flèche épaisse grise (dans la représentation graphique du contexte de base, les flèches sont en noir).

Contexte de base

```
interface Part_externe
{
    key nom,telephone;
    attribute string nom;
    attribute string adresse;
    attribute string telephone;
}

interface Client: Part_externe
{
    relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
}

interface Fournisseur: Part_externe
{
    relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
}

interface Produit
{
    attribute string designation;
    attribute integer prix_tarif;
    relationship set<Client> a_ete_achete_par inverse Client::a_achete;
    relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
}

interface Produit_grand_public: Produit
{
    key reference_grand_public;
    attribute integer reference_grand_public;
}

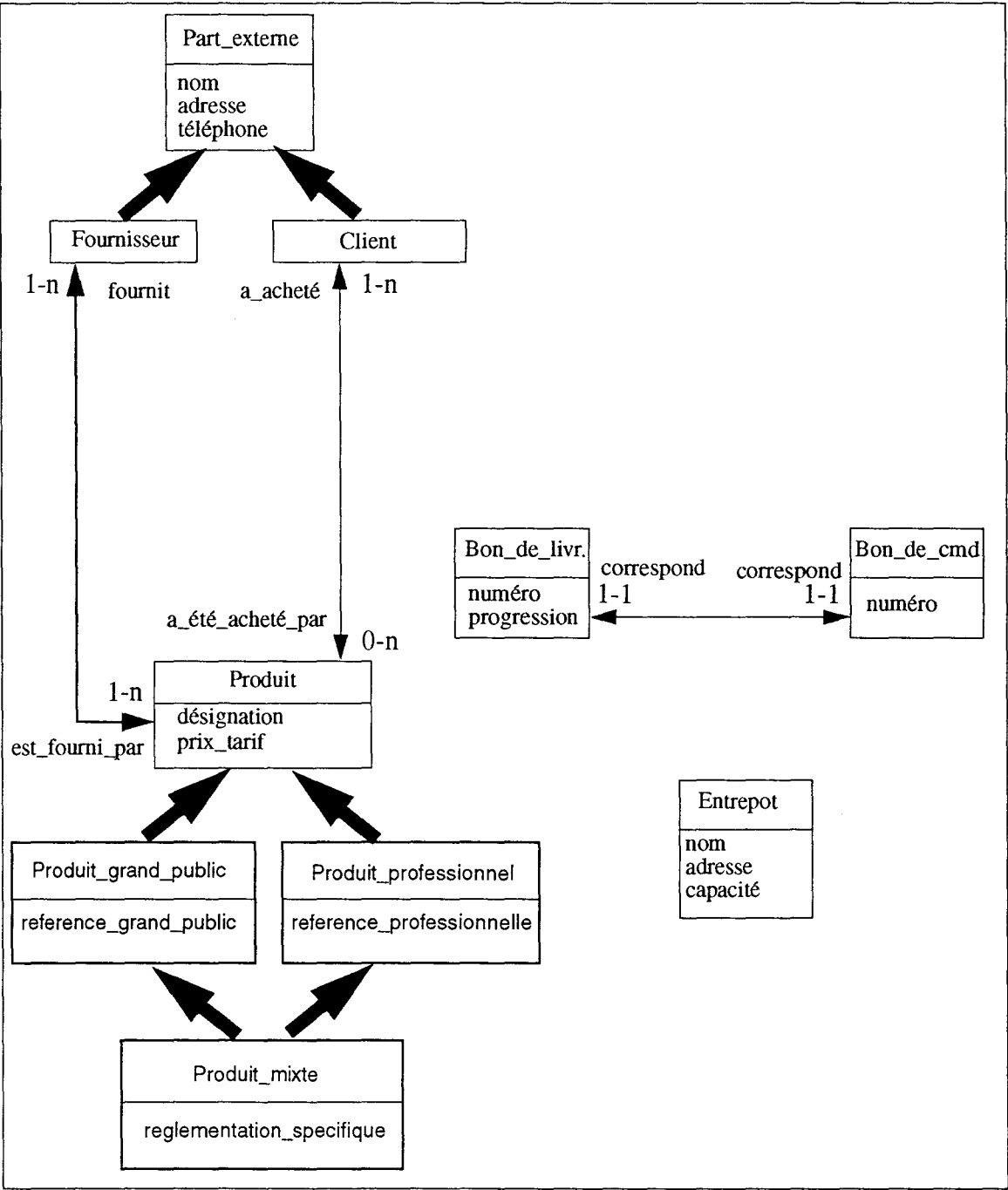
interface Produit_professionnel: Produit
{
    key reference_professionnelle;
    attribute string reference_professionnelle;
}

interface Produit_mixte: Produit_grand_public, Produit_professionnel
{
    attribute string reglementation_specifique;
}

interface Entrepot
{
    attribute string nom;
    attribute string adresse;
    attribute integer capacite;
}

interface Bon_de_livraison
{
    attribute integer numero;
    attribute string progression;
    relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
}
```

```
interface Bon_de_commande
{
  attribute string numero;
  relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
}
```



Interfaces relatives au contexte de base

Contexte commercial

function Commercial interface increment Client

```
{
  access function Comptable:
  {
    attribute integer encours_a_regler;
  }
  instantiation function;
  write base part:
  {
    attribute string nom;
    attribute string adresse;
    attribute string telephone;
    relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
  }
  function part:
  {
    attribute integer montant_cmd;
    attribute date date_derniere_cmd;
    relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
    relationship set<Bon_de_commande> est_concerne_par_BC inverse Bon_de_commande::concerne;
  }
}
```

function Commercial interface increment Fournisseur

```
{
  function part:
  {
    attribute string contact_commercial;
  }
}
```

function Commercial interface increment Produit

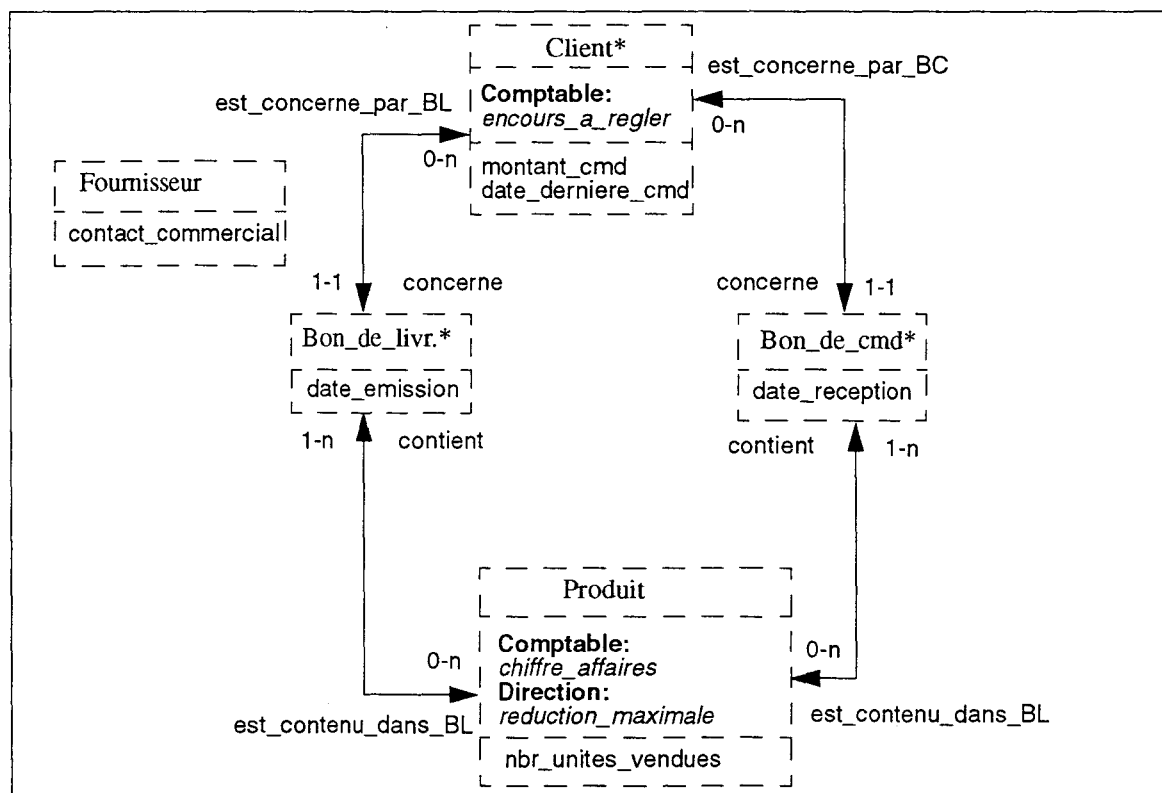
```
{
  access function Comptable:
  {
    attribute integer chiffre_affaires;
  }
  access function Direction:
  {
    attribute integer reduction_maximale;
  }
  write base part:
  {
    attribute integer prix_tarif;
    relationship set<Client> a_ete_achete_par inverse Client::a_achete;
  }
  function part:
  {
    attribute integer nbr_unites_vendues;
    relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
    relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
  }
  function integrity constraint:
  {
    prix_tarif > 0;
    nbr_unites_vendues > 0;
  }
}
```

```

function Commercial interface increment Bon_de_livraison
{
  instantiation function;
  write base part:
  {
    attribute integer numero;
    attribute string progression;
    relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
  }
  function part:
  {
    attribute date date_emission;
    relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BL;
    relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BL;
  }
}

function Commercial interface increment Bon_de_commande
{
  instantiation function;
  write base part:
  {
    attribute string numero;
    relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
  }
  function part:
  {
    attribute date date_reception;
    relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BC;
    relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BC;
  }
}

```



Incréments d'interface relatifs au contexte commercial

Contexte « Fabrication »

function Fabrication **interface increment** Part_externe

```
{  
  function part:  
  {  
    attribute string contact_technique;  
  }  
}
```

function Fabrication **interface increment** Fournisseur

```
{  
  function part:  
  {  
    attribute boolean fournit_seulement_lot_complet_de_pieces;  
  }  
}
```

function Fabrication **interface increment** Produit

```
{  
  access function Commercial:  
  {  
    attribute integer nbr_unites_vendues;  
  }  
  access function Stock:  
  {  
    attribute integer cout_stockage;  
    relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;  
  }  
  function extent condition: cout_stockage > 300;  
  function part:  
  {  
    attribute integer cout_fabrication;  
  }  
}
```

function Fabrication **interface increment** Produit_grand_public

```
{  
  function extent condition: nbr_unites_vendues > 20000;  
  function part:  
  {  
    attribute integer nbr_chaines_fabrication;  
  }  
}
```

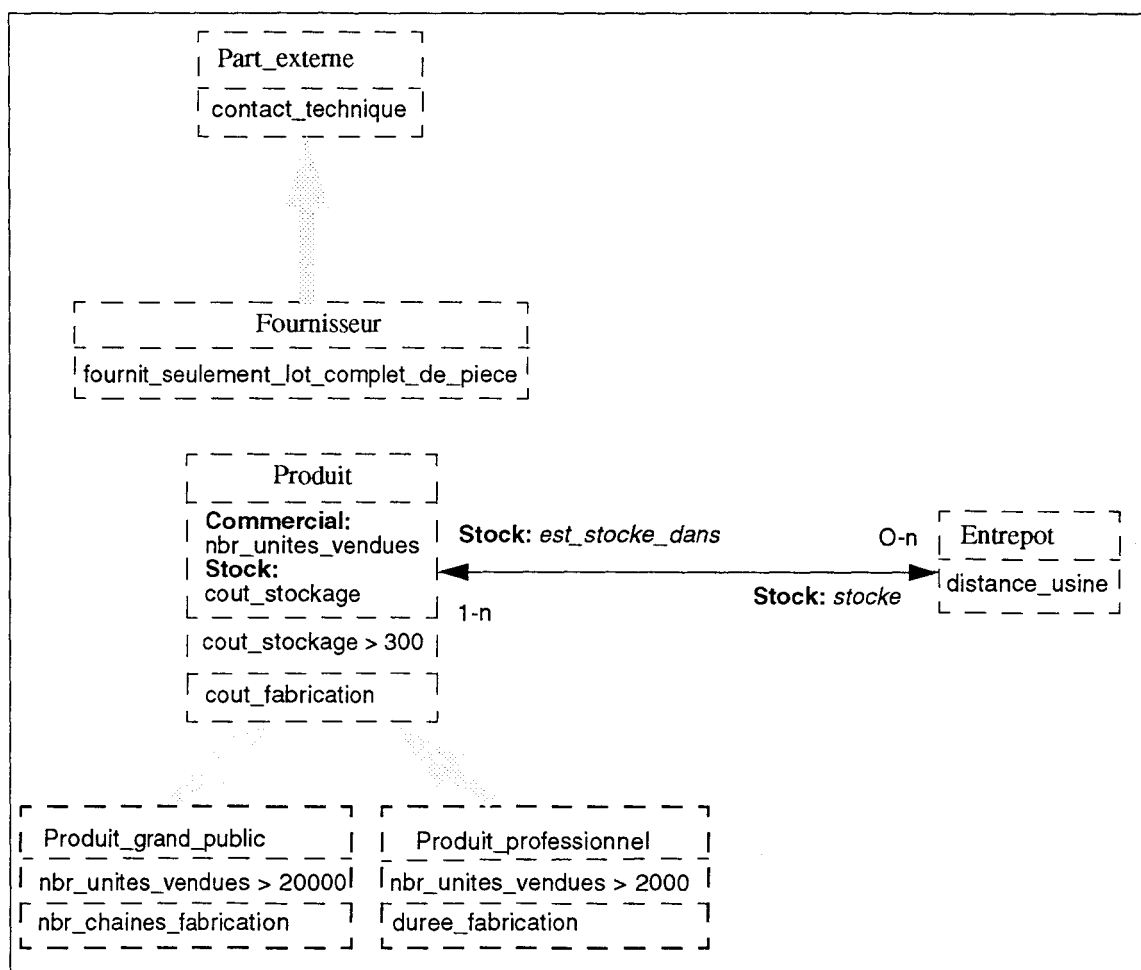
function Fabrication **interface increment** Produit_professionnel

```
{  
  function extent condition: nbr_unites_vendues > 2000;  
  function part:  
  {  
    attribute integer duree_fabrication;  
  }  
}
```

```

function Fabrication interface increment Entrepot
{
  access function Stock:
  {
    relationship set<Produit> stocke inverse Produit::est_stocke_dans;
  }
  function part:
  {
    attribute integer distance_usine;
  }
}

```



Incréments d'interface relatifs au contexte « Fabrication »

Contexte « Stock »

function Stock interface increment Client

```
{
  access function Commercial:
  {
    relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
    relationship set<Bon_de_commande> est_concerne_par_BC inverse Bon_de_commande::concerne;
  }
  function part:
  {
    attribute string adresse_livraison;
    attribute string telephone_livraison;
  }
}
```

function Stock interface increment Fournisseur

```
{
  instantiation function;
  write base part:
  {
    attribute string nom;
    attribute string adresse;
    attribute string telephone;
    relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
  }
  function part:
  {
    attribute string responsable_livraison;
  }
}
```

function Stock interface increment Produit

```
{
  access function Commercial:
  {
    relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
    relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
  }
  write base part:
  {
    attribute string designation;
    relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
  }
  function part:
  {
    attribute integer cout_achat;
    attribute integer cout_stockage;
    attribute integer cout_livraison;
    attribute integer quantite_stockee;
    relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
  }
}
```

function Stock interface increment Produit_grand_public

```
{
  instantiation function;
  write base part:
  {
    attribute integer reference_grand_public;
  }
  function part:
  {
```

```

    attribute string reglementation_stockage;
  }
}

function Stock interface increment Produit_professionnel
{
  instantiation function;
  write base part:
  {
    attribute string reference_professionnelle;
  }
  function part:
  {
    attribute string condition_stockage;
  }
}

function Stock interface increment Produit_mixte
{
  instantiation function;
  write base part:
  {
    attribute string reglementation_specifique;
  }
  function part:
  {
    attribute boolean stockage_unique;
  }
}

function Stock interface increment Entrepot
{
  instantiation function;
  write base part:
  {
    attribute string nom;
    attribute string adresse;
    attribute integer capacite;
  }
  function part:
  {
    relationship set<Produit> stocke inverse Produit::est_stocke_dans;
  }
  function integrity constraint:
  {
    capacite > 1000;
  }
}

function Stock interface increment Bon_de_livraison
{
  access function Commercial:
  {
    relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BL;
    relationship set<Produit>[1] contient inverse Produit::est_contenu_dans_BL;
  }
  function extent condition:
    (progression = 'livraison') OR (progression = 'comptabilisation') OR (progression = 'comptabilise');
  write base part:
  {
    attribute string progression;
  }
  function part:

```

```

{
  attribute date date_livraison;
}
}

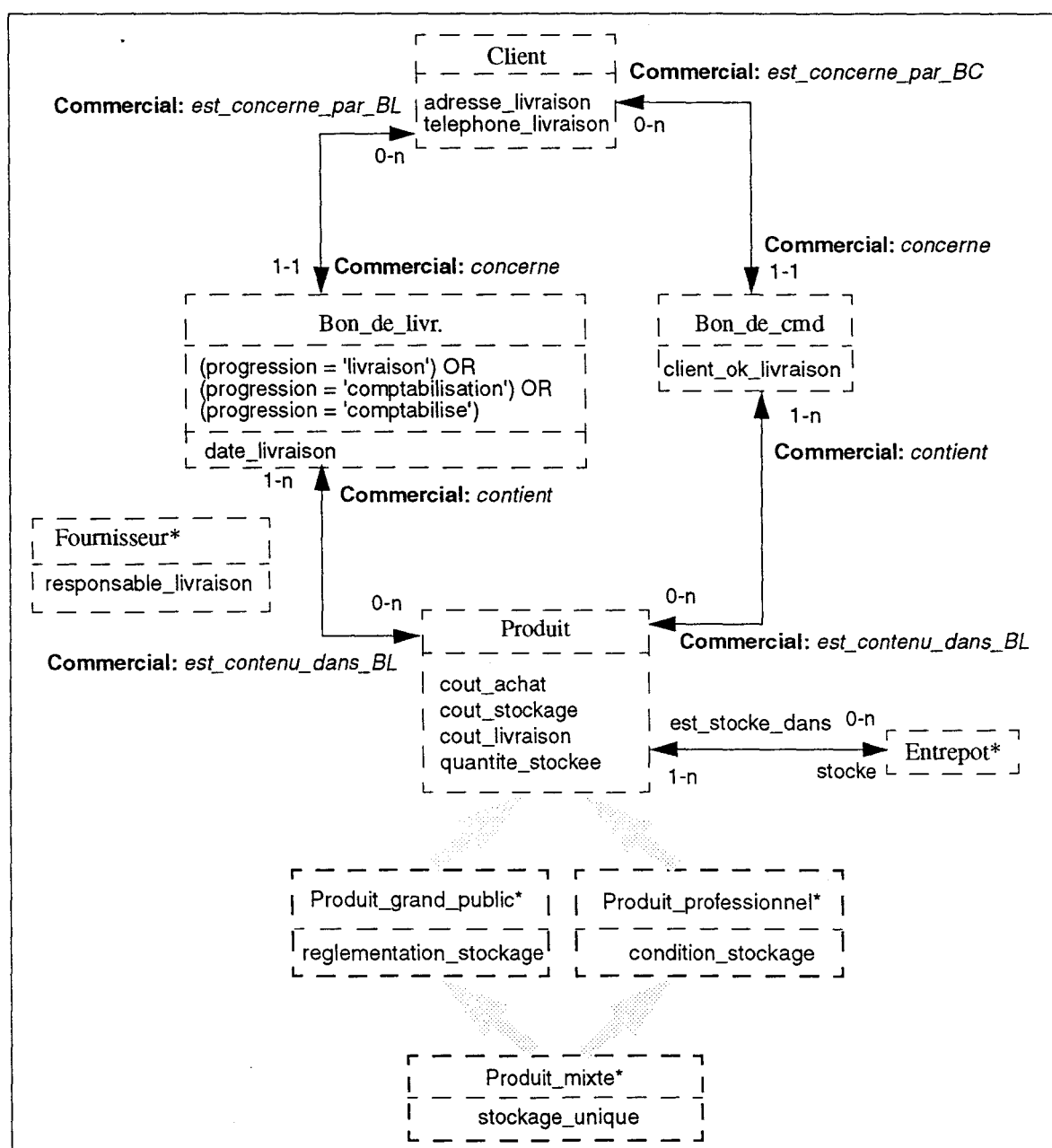
```

function Stock interface increment Bon_de_commande

```

{
  access function Commercial:
  {
    relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BC;
    relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BC;
  }
  function part:
  {
    attribute boolean client_ok_livraison;
  }
}

```



Incréments d'interface relatifs au contexte « Stock »

Contexte comptable

function Comptable **interface increment** Client

```
{
  access function Commercial:
  {
    relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
    relationship set<Bon_de_commande> est_concerne_par_BC inverse Bon_de_commande::concerne;
  }
  function part:
  {
    attribute integer encours_a_regler;
    attribute integer chiffre_affaires;
  }
  function integrity constraint:
  {
    encours_a_regler < chiffre_affaires;
  }
}
```

function Comptable **interface increment** Fournisseur

```
{
  function part:
  {
    attribute integer sommes_acquittees;
    attribute integer sommes_dues;
  }
}
```

function Comptable **interface increment** Produit

```
{
  access function Commercial:
  {
    attribute integer nbr_unites_vendues;
    relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
    relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
  }
  access function Stock:
  {
    attribute integer quantite_stockee;
  }
  access function Direction:
  {
    attribute integer reduction_maximale;
  }
  function part:
  {
    attribute integer chiffre_affaires;
    attribute integer marge_moyenne;
  }
  function integrity constraint:
  {
    (chiffre_affaires >= nbr_unites_vendues*prix_tarif*reduction_maximale) AND
    (chiffre_affaires <= nbr_unites_vendues*prix_tarif);
  }
}
```

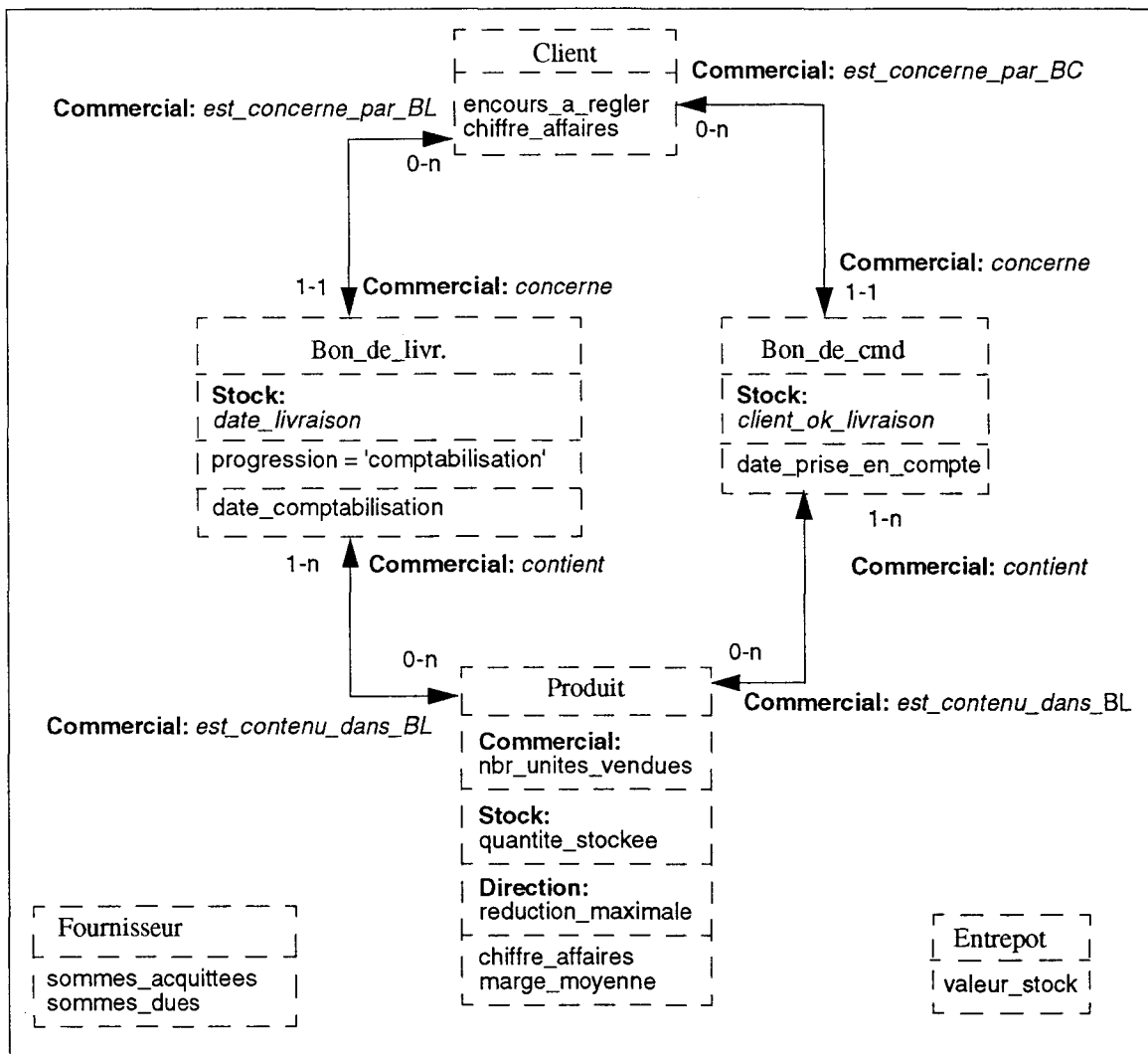
```

function Comptable interface increment Entrepot
{
  function part:
  {
    attribute integer valeur_stock;
  }
}

function Comptable interface increment Bon_de_livraison
{
  access function Commercial:
  {
    relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BL;
    relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BL;
  }
  access function Stock:
  {
    attribute date date_livraison;
  }
  function extent condition: progression = 'comptabilisation';
  write base part:
  {
    attribute string progression;
  }
  function part:
  {
    attribute date date_comptabilisation;
  }
}

function Comptable interface increment Bon_de_commande
{
  access function Commercial:
  {
    relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BC;
    relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BC;
  }
  access function Stock:
  {
    attribute boolean client_ok_livraison;
  }
  function part:
  {
    attribute date date_prise_en_compte;
  }
}

```



Incréments d'interface relatifs au contexte comptable

Contexte « SAV »

function SAV interface increment Client

```
{
  function part:
  {
    attribute date date_dernier_depot;
    attribute string contact_SAV;
  }
}
```

function SAV interface increment Fournisseur

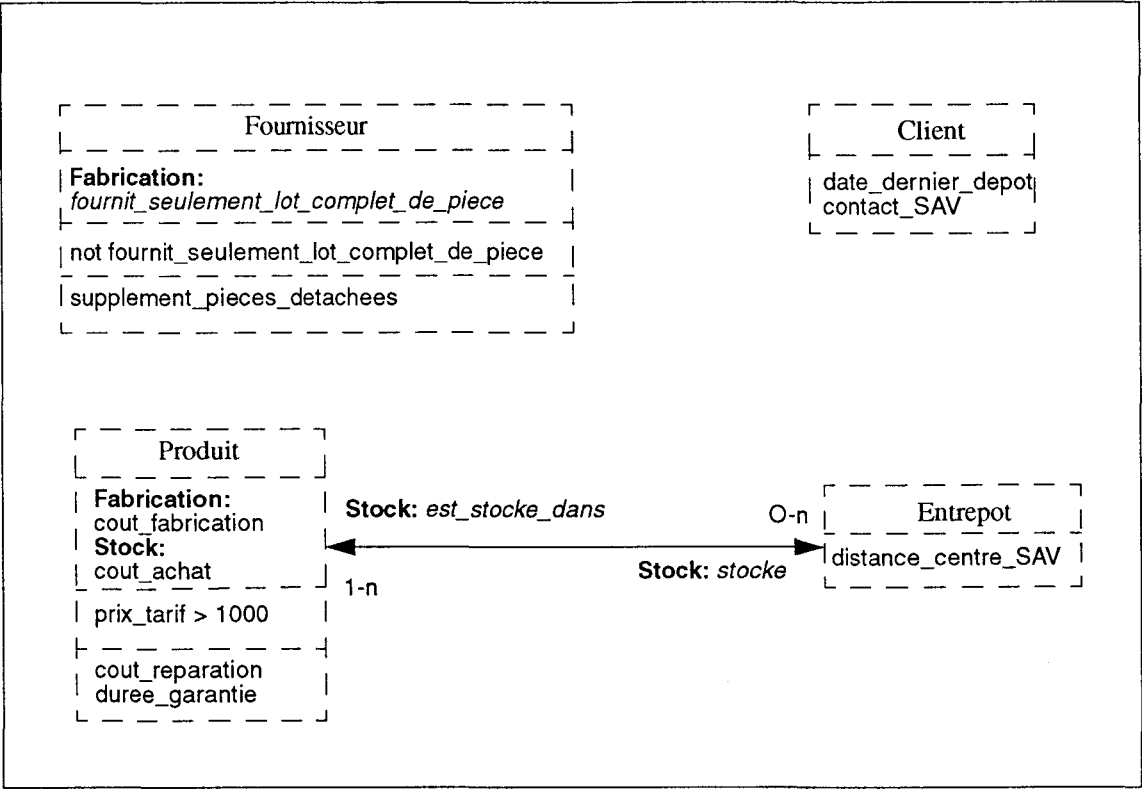
```
{
  access function Fabrication:
  {
    attribute boolean fournit_seulement_lot_complet_de_pieces;
  }
  function extent condition: not fournit_seulement_lot_complet_de_pieces;
  function part:
  {
    attribute integer supplement_pieces_detachees;
  }
}
```

function SAV interface increment Produit

```
{
  access function Fabrication:
  {
    attribute integer cout_fabrication;
  }
  access function Stock:
  {
    attribute integer cout_achat;
    relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
  }
  function extent condition: prix_tarif > 1000;
  function part:
  {
    attribute integer cout_reparation;
    attribute integer duree_garantie;
  }
}
```

function SAV interface increment Entrepot

```
{
  access function Stock:
  {
    relationship set<Produit> stocke inverse Produit::est_stocke_dans;
  }
  function part:
  {
    attribute integer distance_centre_SAV;
  }
}
```



Incréments d'interface relatifs au contexte « SAV »

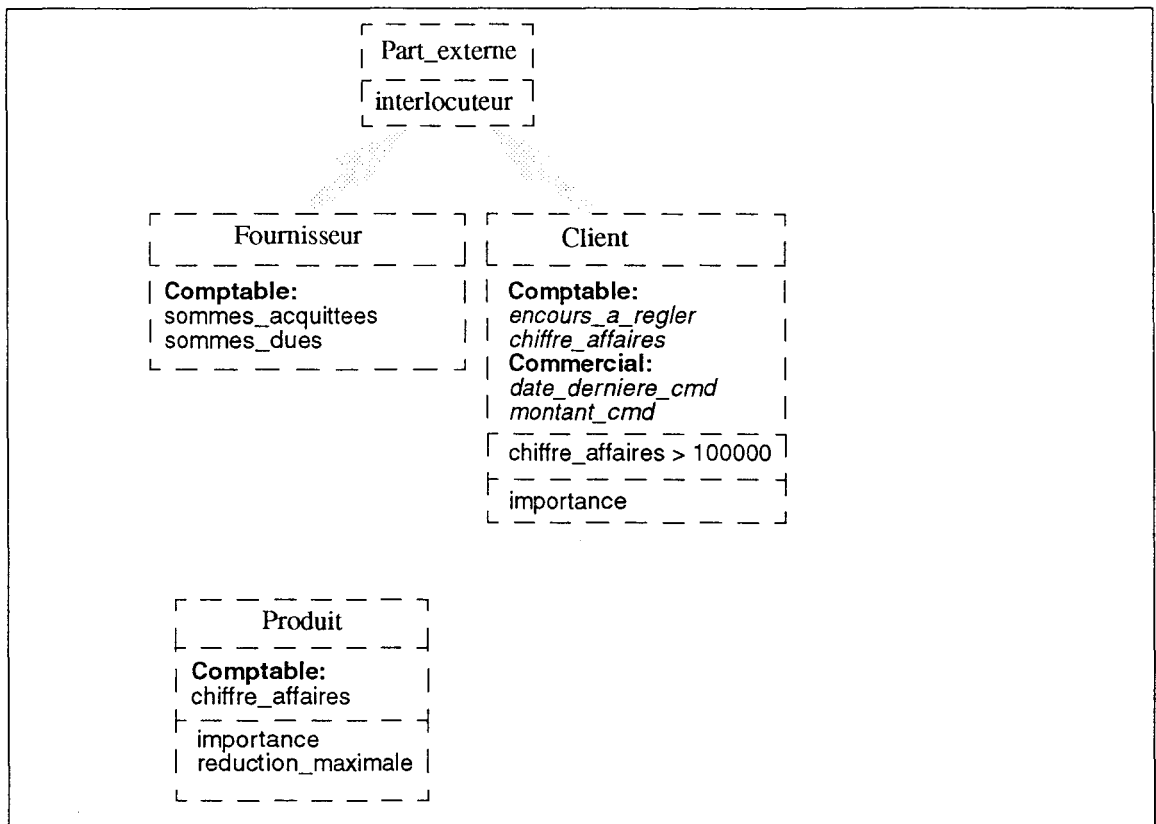
Contexte « Direction »

```
function Direction interface increment Part_externe
function part:
{
  attribute string interlocuteur;
}
}
```

```
function Direction interface increment Client
{
  access function Comptable:
  {
    attribute integer encours_a_regler;
    attribute integer chiffre_affaires;
  }
  access function Commercial:
  {
    attribute date date_derniere_cmd;
    attribute integer montant_cmd;
  }
  function extent condition: chiffre_affaires > 100000;
  function part:
  {
    attribute string importance;
  }
}
```

```
function Direction interface increment Fournisseur
{
  access function Comptable:
  {
    attribute integer sommes_acquittees;
    attribute integer sommes_dues;
  }
  function part:
  {
  }
}
```

```
function Direction interface increment Produit
{
  access function Comptable:
  {
    attribute integer chiffre_affaires;
  }
  function part:
  {
    attribute integer importance;
    attribute integer reduction_maximale;
  }
  function integrity constraint:
  {
    (reduction_maximale >= 0.5) AND (reduction_maximale <= 1.0);
  }
}
```



Incréments d'interface relatifs au contexte « Direction »

Annexe B

Interfaces destinées aux clients des contextes du SI

Cet annexe contient l'ensemble des interfaces destinées aux clients des contextes d'une entreprise de fabrication et de distribution dont le schéma est décrit à l'annexe A. Ces interfaces sont représentées par la forme syntaxique « interface » d'ODL [ODMG97] modifiée uniquement par l'ajout des spécifications de contraintes de cardinalité.

Chaque interface contenue dans cet annexe est générée automatiquement à partir de l'interface de la description de base et de l'incrément d'interface correspondant à la même entité.

Pour chaque contexte, nous donnons la liste des entités participant à ce contexte et que les clients du contexte peuvent exploiter au travers des langages de requêtes et de manipulation. L'interface correspondant à chaque surclasse de l'une de ces entités est également présentée, même si l'entité relative à cette surclasse n'intervient pas dans ce contexte. L'interface d'une surclasse, dans un cas de non-intervention, correspond uniquement à sa description de base. Ceci permet de présenter l'ensemble des propriétés d'une classe présentes dans un contexte, y compris les propriétés introduites par ses surclasses.

Pour chaque contexte, nous donnons également la liste des classes dont ce contexte est un contexte d'instanciation. Enfin, les contraintes d'intégrité sur les propriétés sont décrites mais en dehors des interfaces ODL.

Contexte commercial

Entités participant au contexte = {Client, Fournisseur, Produit, Produit_grand_public, Produit_professionnel, Produit_mixte, Bon_de_livraison, Bon_de_commande}

Classe dont le contexte est un contexte d'instanciation = {Client, Bon_de_livraison, Bon_de_commande}

interface Part_externe

```
{
  key nom,telephone;
  read-only attribute string nom;
  read-only attribute string adresse;
  read-only attribute string telephone;
}
```

interface Client: Part_externe

```
{
  attribute string nom;
  attribute string adresse;
  attribute string telephone;
  relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
  read-only attribute integer encours_a_regler;
  attribute integer montant_cmd;
  attribute date date_derniere_cmd;
  relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
  relationship set<Bon_de_commande> est_concerne_par_BC inverse Bon_de_commande::concerne;
}
```

interface Fournisseur: Part_externe

```
{
  read-only relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
  attribute string contact_commercial;
}
```

interface Produit

```
{
  read-only attribute string designation;
  attribute integer prix_tarif;
  relationship set<Client> a_ete_achete_par inverse Client::a_achete;
  read-only relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
  read-only attribute integer chiffre_affaires;
  read-only attribute integer reduction_maximale;
  attribute integer nbr_unites_vendues;
  relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
  relationship set<Bon_de_commande> est_contenu_dans_BC inverse Bon_de_commande::contient;
}
```

Contraintes d'intégrité sur « Produit »

```
prix_tarif > 0;
nbr_unites_vendues > 0;
```

interface Produit_grand_public: Produit

```
{
  key reference_grand_public;
  read-only attribute integer reference_grand_public;
}
```

interface Produit_professionnel: Produit

```
{
  key reference_professionnelle;
  read-only attribute string reference_professionnelle;
}
```

```

interface Produit_mixte: Produit_grand_public, Produit_professionnel
{
  read-only attribute string reglementation_specifique;
}

```

```

interface Bon_de_livraison
{
  attribute integer numero;
  attribute string progression;
  relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
  attribute date date_emission;
  relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BL;
  relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BL;
}

```

```

interface Bon_de_commande
{
  attribute string numero;
  relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
  attribute date date_reception;
  relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BC;
  relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BC;
}

```

Contexte « Fabrication »

Entités participant au contexte = {Part_externe, Client, Fournisseur, Produit, Produit_grand_public, Produit_professionnel, Produit_mixte, Entrepot}

interface Part_externe

```
{
  key nom,telephone;
  read-only attribute string nom;
  read-only attribute string adresse;
  read-only attribute string telephone;
  attribute string contact_technique;
}
```

interface Client: Part_externe

```
{
  read-only relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
}
```

interface Fournisseur: Part_externe

```
{
  read-only relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
  attribute boolean fournit_seulement_lot_complet_de_pieces;
}
```

interface Produit

```
{
  read-only attribute string designation;
  read-only attribute integer prix_tarif;
  read-only relationship set<Client> a_ete_achete_par inverse Client::a_achete;
  read-only relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
  read-only attribute integer nbr_unites_vendues;
  read-only attribute integer cout_stockage;
  read-only relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
  attribute integer cout_fabrication;
}
```

interface Produit_grand_public: Produit

```
{
  key reference_grand_public;
  read-only attribute integer reference_grand_public;
  attribute integer nbr_chaines_fabrication;
}
```

interface Produit_professionnel: Produit

```
{
  key reference_professionnelle;
  read-only attribute string reference_professionnelle;
  attribute integer duree_fabrication;
}
```

interface Produit_mixte: Produit_grand_public, Produit_professionnel

```
{
  read-only attribute string reglementation_specifique;
}
```

interface Entrepot

```
{
  read-only attribute string nom;
  read-only attribute string adresse;
  read-only attribute integer capacite;
  read-only relationship set<Produit> stocke inverse Produit::est_stocke_dans;
  attribute integer distance_usine;
}
```

Contexte « Stock »

Entités participant au contexte = {Client, Fournisseur, Produit, Produit_grand_public, Produit_professionnel, Produit_mixte, Entrepot, Bon_de_livraison, Bon_de_commande}

Classes dont le contexte est un contexte d'instanciation = {Fournisseur, Produit_grand_public, Produit_professionnel, Produit_mixte, Entrepot}

interface Part_externe

```
{
  key nom,telephone;
  read-only attribute string nom;
  read-only attribute string adresse;
  read-only attribute string telephone;
}
```

interface Client: Part_externe

```
{
  read-only relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
  read-only relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
  read-only relationship set<Bon_de_commande> est_concerne_par_BC
    inverse Bon_de_commande::concerne;
  attribute string adresse_livraison;
  attribute string telephone_livraison;
}
```

interface Fournisseur: Part_externe

```
{
  attribute string nom;
  attribute string adresse;
  attribute string telephone;
  relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
  attribute string responsable_livraison;
}
```

interface Produit

```
{
  attribute string designation;
  read-only attribute integer prix_tarif;
  read-only relationship set<Client> a_ete_achete_par inverse Client::a_achete;
  relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
  read-only relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
  read-only relationship set<Bon_de_commande> est_contenu_dans_BC
    inverse Bon_de_commande::contient;
  attribute integer cout_achat;
  attribute integer cout_stockage;
  attribute integer cout_livraison;
  attribute integer quantite_stockee;
  relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
}
```

interface Produit_grand_public: Produit

```
{
  key reference_grand_public;
  attribute integer reference_grand_public;
  attribute string reglementation_stockage;
}
```

interface Produit_professionnel: Produit

```
{
  key reference_professionnelle;
  attribute string reference_professionnelle;
  attribute string condition_stockage;
}
```

```

interface Produit_mixte: Produit_grand_public, Produit_professionnel
{
    attribute string reglementation_specifique;
    attribute boolean stockage_unique;
}

```

```

interface Entrepot
{
    attribute string nom;
    attribute string adresse;
    attribute integer capacite;
    relationship set<Produit> stocke inverse Produit::est_stocke_dans;
}

```

Contraintes d'intégrité sur « Entrepot »

capacite > 1000;

```

interface Bon_de_livraison
{
    read-only attribute integer numero;
    attribute string progression;
    read-only relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
    read-only relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BL;
    read-only relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BL;
    attribute date date_livraison;
}

```

```

interface Bon_de_commande
{
    read-only attribute string numero;
    read-only relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
    read-only relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BC;
    read-only relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BC;
    attribute boolean client_ok_livraison;
}

```

Contexte comptable

Entités participant au contexte = {Client, Fournisseur, Produit, Produit_grand_public, Produit_professionnel, Produit_mixte, Entrepot, Bon_de_livraison, Bon_de_commande}

interface Part_externe

```
{
  key nom,telephone;
  read-only attribute string nom;
  read-only attribute string adresse;
  read-only attribute string telephone;
}
```

interface Client: Part_externe

```
{
  read-only relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
  read-only relationship set<Bon_de_livraison> est_concerne_par_BL inverse Bon_de_livraison::concerne;
  read-only relationship set<Bon_de_commande> est_concerne_par_BC
    inverse Bon_de_commande::concerne;
  attribute integer encours_a_regler;
  attribute integer chiffre_affaires;
}
```

Contraintes d'intégrité sur « Client »

encours_a_regler < chiffre_affaires;

interface Fournisseur: Part_externe

```
{
  read-only relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
  attribute integer sommes_acquittees;
  attribute integer sommes_dues;
}
```

interface Produit

```
{
  read-only attribute string designation;
  read-only attribute integer prix_tarif;
  read-only relationship set<Client> a_ete_achete_par inverse Client::a_achete;
  read-only relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
  read-only attribute integer nbr_unites_vendues;
  read-only relationship set<Bon_de_livraison> est_contenu_dans_BL inverse Bon_de_livraison::contient;
  read-only relationship set<Bon_de_commande> est_contenu_dans_BC
    inverse Bon_de_commande::contient;
  read-only attribute integer quantite_stockee;
  read-only attribute integer reduction_maximale;
  attribute integer chiffre_affaires;
  attribute integer marge_moyenne;
}
```

Contraintes d'intégrité sur « Produit »

(chiffre_affaires >= nbr_unites_vendues*prix_tarif*reduction_maximale) AND
(chiffre_affaires <= nbr_unites_vendues*prix_tarif);

interface Produit_grand_public: Produit

```
{
  key reference_grand_public;
  read-only attribute integer reference_grand_public;
}
```

interface Produit_professionnel: Produit

```
{
  key reference_professionnelle;
  read-only attribute string reference_professionnelle;
}
```

```

interface Produit_mixte: Produit_grand_public, Produit_professionnel
{
    read-only attribute string reglementation_specifique;
}

interface Entrepot
{
    read-only attribute string nom;
    read-only attribute string adresse;
    read-only attribute integer capacite;
    attribute integer valeur_stock;
}

interface Bon_de_livraison
{
    read-only attribute integer numero;
    attribute string progression;
    read-only relationship set<Bon_de_commande>[1..1] correspond inverse Bon_de_commande::correspond;
    read-only relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BL;
    read-only relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BL;
    read-only attribute date date_livraison;
    attribute date date_comptabilisation;
}

interface Bon_de_commande
{
    read-only attribute string numero;
    read-only relationship set<Bon_de_livraison>[1..1] correspond inverse Bon_de_livraison::correspond;
    read-only relationship set<Client>[1..1] concerne inverse Client::est_concerne_par_BC;
    read-only relationship set<Produit>[1] contient inverse Produit:: est_contenu_dans_BC;
    read-only attribute boolean client_ok_livraison;
    attribute date date_prise_en_compte;
}

```

Contexte « SAV »

Entités participant au contexte = {Client, Fournisseur, Produit, Produit_grand_public, Produit_professionnel, Produit_mixte, Entrepot}

interface Part_externe

```
{
  key nom,telephone;
  read-only attribute string nom;
  read-only attribute string adresse;
  read-only attribute string telephone;
}
```

interface Client: Part_externe

```
{
  read-only relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
  attribute date date_dernier_depot;
  attribute string contact_SAV;
}
```

interface Fournisseur: Part_externe

```
{
  read-only relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
  read-only attribute boolean fournit_seulement_lot_complet_de_pieces;
  attribute integer supplement_pieces_detachees;
}
```

interface Produit

```
{
  read-only attribute string designation;
  read-only attribute integer prix_tarif;
  read-only relationship set<Client> a_ete_achete_par inverse Client::a_achete;
  read-only relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
  read-only attribute integer cout_fabrication;
  read-only attribute integer cout_achat;
  read-only relationship set<Entrepot>[1] est_stocke_dans inverse Entrepot::stocke;
  attribute integer cout_reparation;
  attribute integer duree_garantie;
}
```

interface Produit_grand_public: Produit

```
{
  key reference_grand_public;
  read-only attribute integer reference_grand_public;
}
```

interface Produit_professionnel: Produit

```
{
  key reference_professionnelle;
  read-only attribute string reference_professionnelle;
}
```

interface Produit_mixte: Produit_grand_public, Produit_professionnel

```
{
  read-only attribute string reglementation_specifique;
}
```

interface Entrepot

```
{
  read-only attribute string nom;
  read-only attribute string adresse;
  read-only attribute integer capacite;
  read-only relationship set<Produit> stocke inverse Produit::est_stocke_dans;
  attribute integer distance_centre_SAV;
}
```

Contexte « Direction »

Entités participant au contexte = {Part_externe, Client, Fournisseur, Produit, Produit_grand_public, Produit_professionnel, Produit_mixte}

interface Part_externe

```
{
  key nom,telephone;
  read-only attribute string nom;
  read-only attribute string adresse;
  read-only attribute string telephone;
  attribute string interlocuteur;
}
```

interface Client: Part_externe

```
{
  read-only relationship set<Produit>[1] a_achete inverse Produit::a_ete_achete_par;
  read-only attribute integer encours_a_regler;
  read-only attribute integer chiffre_affaires;
  read-only attribute date date_derniere_cmd;
  read-only attribute integer montant_cmd;
  attribute string importance;
}
```

interface Fournisseur: Part_externe

```
{
  read-only relationship set<Produit>[1] fournit inverse Produit::est_fourni_par;
  read-only attribute integer sommes_acquittees;
  read-only attribute integer sommes_dues;
}
```

interface Produit

```
{
  read-only attribute string designation;
  read-only attribute integer prix_tarif;
  read-only relationship set<Client> a_ete_achete_par inverse Client::a_achete;
  read-only relationship set<Fournisseur>[1] est_fourni_par inverse Fournisseur::fournit;
  read-only attribute integer chiffre_affaires;
  attribute integer importance;
  attribute integer reduction_maximale;
}
```

Contraintes d'intégrité sur « Produit »

(reduction_maximale >= 0.5) AND (reduction_maximale <= 1.0);

interface Produit_grand_public: Produit

```
{
  key reference_grand_public;
  read-only attribute integer reference_grand_public;
}
```

interface Produit_professionnel: Produit

```
{
  key reference_professionnelle;
  read-only attribute string reference_professionnelle;
}
```

interface Produit_mixte: Produit_grand_public, Produit_professionnel

```
{
  read-only attribute string reglementation_specifique;
}
```

Annexe C

Modèle CROME dans le cadre des bases de données

Modèle « objet » de base pour la formalisation de CROME

Notions relatives aux classes

Soit $\mathcal{C}$ l'ensemble des classes relatives aux entités du domaine, $\mathcal{O}$ l'ensemble de toutes les instances de toutes les classes du domaine et $\mathcal{A}$ l'ensemble de tous les attributs de toutes les classes appartenant à $\mathcal{C}$, C'est-à-dire $\forall a \in \mathcal{A}, \exists C \in \mathcal{C}$ tel que $a \in \text{intension}(C)$. L'ensemble $\mathcal{A}$ contient également les attributs correspondant à des extrémités des relations.

Intension

L'intension d'une classe est une fonction qui associe à cette classe un ensemble d'attributs décrivant ses instances :

$$\begin{aligned} \text{intension: } \mathcal{C} &\rightarrow 2^{\mathcal{A}} \\ C &\rightarrow \{a_1, a_2, \dots, a_n\} \end{aligned}$$

Extension

L'extension d'une classe est une fonction qui associe à cette classe l'ensemble de ses instances :

$$\begin{aligned} \text{extension: } \mathcal{C} &\rightarrow 2^{\mathcal{O}} \\ C &\rightarrow \{o_1, o_2, \dots, o_n\} \end{aligned}$$

Relation « sous-classe »

La relation d'héritage entre intensions est définie ainsi :

$$\text{Soit } i \text{ et } i' \in 2^{\mathcal{A}}, i \leq i' \Leftrightarrow i \supseteq i'$$

La relation « sous-classe » entre deux classes est définie par :

$$\begin{aligned} \text{Soit } (C, C') \in \mathcal{C}^2, \\ C \text{ sous-classe de } C' &\Leftrightarrow \text{intension}(C) \leq \text{intension}(C') \wedge \text{extension}(C) \subseteq \text{extension}(C') \end{aligned}$$

Relation « sous-classe-directe »

La relation « sous-classe-directe » entre deux classes est définie par

$$\begin{aligned} \text{Soit } (C, C') \in \mathcal{C}^2, \\ C \text{ sous-classe-directe de } C' &\Leftrightarrow C \text{ sous-classe de } C' \wedge \neg(\exists C'' \in \mathcal{C} \text{ telle que } C \text{ sous-classe de } \\ &C'' \wedge C'' \text{ sous-classe de } C') \end{aligned}$$

Extension stricte

L'extension stricte d'une classe est une fonction qui associe à cette classe l'ensemble des instances qui lui sont propres, à l'exclusion des instances de ses sous-classes :

$$\begin{aligned} \text{extension}_{\text{stricte}}: \mathcal{C} &\rightarrow 2^{\mathcal{O}} \\ C &\rightarrow \{o_1, o_2, \dots, o_p\} \end{aligned}$$

L'extension peut être définie récursivement à partir de l'extension stricte :

$\text{extension}(C) = \text{extension}_{\text{stricte}}(C) \cup_{SC \in \text{sous-classe-directe}(C)} \text{extension}(SC)$
 $\text{sous-classe-directe}(C)$ est l'ensemble des sous-classes directes de C .

Fonction « classe »

La fonction « classe » associe à un élément de $\mathcal{O}$ sa classe d'instanciation

classe: $\mathcal{O} \rightarrow \mathcal{C}$

$o \rightarrow C$ telle que $o \in \text{extension}_{\text{stricte}}(C)$

Notions relatives aux relations

Nous définissons $\mathcal{R}$ comme l'ensemble des relations entre deux classes de $\mathcal{C}$. Un élément R de $\mathcal{R}$ peut être représenté par le couple $((C,a),(C',b))$ tel que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in \text{intension}(C) \times \text{intension}(C')$. a et b sont les attributs des extrémités de R . C et C' sont respectivement les classes d'introduction des extrémités a et b , c'est-à-dire que :

$\neg (\exists SC \in \mathcal{C}, (\text{intension}(C) \leq \text{intension}(SC)) \wedge (a \in \text{intension}(SC)))$

$\neg (\exists SC' \in \mathcal{C}, (\text{intension}(C') \leq \text{intension}(SC')) \wedge (b \in \text{intension}(SC')))$

Extension d'une relation et vérification de l'intégrité référentielle

Nous définissons l'extension d'une relation par l'ensemble de ses occurrences. L'extension d'une relation est un sous-ensemble du produit cartésien de l'extension des classes situées à ses extrémités :

extension: $\mathcal{R} \rightarrow 2^{\mathcal{O} \times \mathcal{O}}$

$R \rightarrow \{(o_1, o_1'), (o_1, o_2'), (o_2, o_3'), \dots, (o_n, o_m')\}$ telle que

• $\{o_1, o_2, \dots, o_n\} \subseteq \text{extension}(C)$

• $\{o_1', o_2', o_3', \dots, o_m'\} \subseteq \text{extension}(C')$

Par conséquent :

$\text{extension}(R) \subseteq \text{extension}(C) \times \text{extension}(C')$

Cette inclusion garantit l'intégrité référentielle, traduit ici par le fait que tout couple de la relation appartient au produit cartésien des extensions des deux classes.

Vérification de la cardinalité minimale

Soit $R \in \mathcal{R}$ une relation représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in \text{intension}(C) \times \text{intension}(C')$.

$\text{card-min}_C(R)$ est la cardinalité minimale de R pour l'extrémité C .

$\text{card-max}_C(R)$ est la cardinalité maximale de R pour l'extrémité C .

$\text{card-min}_{C'}(R)$ est la cardinalité minimale de R pour l'extrémité C' .

$\text{card-max}_{C'}(R)$ est la cardinalité maximale de R pour l'extrémité C' .

La notation $|E|$ désigne le cardinal de l'ensemble E . La vérification des contraintes de cardinalité se formalise par la vérification des inégalités suivantes :

$\forall o \in \text{extension}(C), \text{card-min}_C(R) \leq |\{(u,v) \in \text{extension}(R) \text{ tel que } u=o\}| \leq \text{card-max}_C(R)$

$\forall o' \in \text{extension}(C'), \text{card-min}_{C'}(R) \leq |\{(u,v) \in \text{extension}(R) \text{ tel que } v=o'\}| \leq \text{card-max}_{C'}(R)$

Modèle CROME

Plan et description de base

Le plan de base détient la structuration du référentiel constitué des classes du domaine ainsi que la description de base de ces dernières. Cette structuration et ces descriptions de base sont partagées par les différents contextes fonctionnels au travers des entités du domaine.

Plans et parties fonctionnels

Un plan fonctionnel détient les parties fonctionnelles relatives à son contexte qui sont destinées à enrichir les seules entités du domaine participant à ce contexte.

Toute relation doit être introduite dans un plan, c'est-à-dire que les attributs correspondant à ses extrémités doivent être introduits par le même plan, qu'il soit de base ou fonctionnel.

Les entités situées aux deux extrémités d'une relation introduite dans le plan de base doivent participer aux mêmes contextes et leurs classes doivent donc être enrichies par les plans associés.

Définition formelle des plans, d'une description de base et d'une partie fonctionnelle

Soit $\mathcal{F}$ un ensemble de symboles désignant les différentes fonctions et le symbole **base** se rapportant au plan de base.

Soit RF la relation d'équivalence qui associe deux attributs de l'ensemble $\mathcal{A}$ s'ils sont définis par le même plan.

$$\begin{aligned} RF: \quad & \mathcal{A} \times \mathcal{A} \rightarrow \{0,1\} \\ & (a,b) \rightarrow 1 \text{ si } a \text{ et } b \text{ sont définis par le même plan} \\ & (a,b) \rightarrow 0 \text{ sinon} \end{aligned}$$

Soit $\{\mathcal{D}_f, \forall f \in \mathcal{F}\}$ l'ensemble quotient¹ $\mathcal{A}/RF$. $\mathcal{A}/RF$ détermine une partition de $\mathcal{A}$. $\mathcal{D}_{base}$ est, par convention, la classe d'équivalence des attributs du plan de base.

Description de base

La description de base d'une classe $D_{base}(C)$ est définie par la fonction suivante :

$$\begin{aligned} D_{base}: \quad & \mathcal{C} \rightarrow 2^{\mathcal{A}} \\ & C \rightarrow \mathcal{D}_{base} \cap \text{intension}(C) \end{aligned}$$

Partie fonctionnelle

Une partie fonctionnelle d'une classe $PF_f(C)$ ($f \in \mathcal{F} - \{\mathbf{base}\}$) est définie par la fonction suivante :

$$\begin{aligned} PF_f: \quad & \mathcal{C} \rightarrow 2^{\mathcal{A}} \\ & C \rightarrow \mathcal{D}_f \cap \text{intension}(C) \end{aligned}$$

L'intersection illustre l'orthogonalité entre fonctions et entités.

1. ensemble des classes d'équivalence

Obligation de définir une relation dans un plan

Une relation est obligatoirement introduite par un plan, ce qui se traduit par :

$\forall R \in \mathcal{R}$ et représentée par $((C,a),(C',b))$

$a \in D_{\text{base}}(C) \wedge b \in D_{\text{base}}(C')$ ou

$\exists f \in \mathcal{F}-\{\text{base}\}$ tel que $a \in PF_f(C) \wedge b \in PF_f(C')$

Les entités situées aux deux extrémités d'une relation du plan de base sont enrichies par les mêmes plans fonctionnels

$\forall R \in \mathcal{R}$ et représentée par $((C,a),(C',b))$ avec $a \in D_{\text{base}}(C) \wedge b \in D_{\text{base}}(C')$

$PF_f(C) \neq \emptyset \Leftrightarrow PF_f(C') \neq \emptyset$

$PF_f(C) \neq \emptyset$ signifiant que C est enrichie par le plan fonctionnel correspondant à la fonction f.

Propriété 1 : l'intension d'une classe est partitionnée par $\mathcal{F}$

Cette propriété peut être formulée ainsi : $\text{intension}(C) = D_{\text{base}}(C) \cup_{f \in \mathcal{F}-\{\text{base}\}} PF_f(C)$

En effet :

$$\begin{aligned} D_{\text{base}}(C) \cup_{f \in \mathcal{F}-\{\text{base}\}} PF_f(C) &= (\mathcal{D}_{\text{base}} \cap \text{intension}(C)) \cup_{f \in \mathcal{F}-\{\text{base}\}} (\mathcal{D}_f \cap \text{intension}(C)) \\ &= (\mathcal{D}_{\text{base}} \cup_{f \in \mathcal{F}-\{\text{base}\}} \mathcal{D}_f) \cap \text{intension}(C) \\ &= \mathcal{A} \cap \text{intension}(C) \\ &= \text{intension}(C) \end{aligned}$$

Plan de base

Le plan de base est défini par :

$$P_{\text{base}} = \{D_{\text{base}}(C), C \in \mathcal{C}\}$$

Plan fonctionnel

Un plan fonctionnel est défini par :

$$P_f (f \in \mathcal{F}-\{\text{base}\}) = \{PF_f(C), C \in \mathcal{C} \wedge (PF_f(C) \neq \emptyset)\}$$

Héritage local dans un plan fonctionnel

Au sein d'un plan fonctionnel, la partie fonctionnelle qui enrichit une classe du plan de base est héritée localement dans ses sous-classes.

Une sous-classe d'une classe enrichie dans un plan fonctionnel est également enrichie dans ce plan. Elle l'est soit explicitement, soit implicitement par l'héritage de la partie fonctionnelle de sa surclasse.

Propriété 2 : Au sein d'un plan, les parties et descriptions fonctionnelles d'une classe sont héritées par ses sous-classes.

La relation d'héritage entre parties fonctionnelles peut être prouvée ainsi :

Soit $(C,C') \in \mathcal{C}^2$, C sous-classe de C' et soit $f \in \mathcal{F}-\{\text{base}\}$

C sous-classe de C' implique que $\text{intension}(C) \leq \text{intension}(C')$

Ceci est vrai à une intersection près.

Donc $\mathcal{D}_f \cap \text{intension}(C) \leq \mathcal{D}_f \cap \text{intension}(C')$ et donc $PF_f(C) \leq PF_f(C')$

Descriptions fonctionnelles

Une description fonctionnelle d'une classe est définie par l'union d'une partie fonctionnelle et de la description de base. Une description fonctionnelle d'une classe entretient une relation d'héritage avec la description de base correspondante pour permettre le partage de cette description de base.

Définition formelle d'une description fonctionnelle

Une description fonctionnelle d'une classe $DF_f(C)$ ($f \in \mathcal{F} - \{\mathbf{base}\}$) est définie par la fonction suivante :

$$DF_f: \quad \begin{aligned} \mathcal{C} &\rightarrow 2^A \\ C &\rightarrow (\mathcal{D}_{\mathbf{base}} \cup \mathcal{D}_f) \cap \text{intension}(C) \end{aligned}$$

Propriété 3 : Toute description fonctionnelle d'une classe hérite de la description de base de cette même classe.

Cette propriété s'écrit par conséquent :

$$\forall f \in \mathcal{F} - \{\mathbf{base}\}, \forall C \in \mathcal{C}, DF_f(C) \leq D_{\mathbf{base}}(C)$$

En effet :

$$\begin{aligned} \forall f \in \mathcal{F} - \{\mathbf{base}\}, \forall C \in \mathcal{C} \\ DF_f(C) &= (\mathcal{D}_{\mathbf{base}} \cup \mathcal{D}_f) \cap \text{intension}(C) \\ &= (\mathcal{D}_{\mathbf{base}} \cap \text{intension}(C)) \cup (\mathcal{D}_f \cap \text{intension}(C)) \\ &= D_{\mathbf{base}}(C) \cup PF_f(C) \end{aligned}$$

Par conséquent, $DF_f(C) \leq D_{\mathbf{base}}(C)$

Accès inter-plans

Un accès inter-plans spécifie un attribut ou une relation d'un plan fonctionnel dont la valeur est exploitée dans un autre contexte fonctionnel. Chaque accès inter-plans est défini au sein de la classe où est introduit l'attribut correspondant éventuellement à l'une des extrémités de la relation.

Les accès inter-plans relatifs à une classe du domaine sont hérités dans ses sous-classes.

Définition formelle d'un accès inter-plans

$Acc_f(C)$ est défini comme l'ensemble des accès aux attributs d'une classe C fournis par les autres contextes que celui correspondant à f .

$$Acc_f(C): \quad \begin{aligned} \mathcal{C} &\rightarrow 2^{A \times (\mathcal{F} - \{\mathbf{base}\})} \\ C &\rightarrow \{ (a, ff), a \in PF_{ff}(C) \wedge \\ &\quad \neg (\exists SC \in \mathcal{C}, (PF_{ff}(C) \leq PF_{ff}(SC)) \wedge (a \in PF_{ff}(SC))) \} \end{aligned}$$

Soit une relation $R \in \mathcal{R}$ et représentée par $((C, a), (C', b))$ avec $a \in PF_{ff}(C)$ et $b \in PF_{ff}(C')$. Le fait que R soit rendue accessible au contexte correspondant à f , se traduit par $(a, ff) \in Acc_f(C)$ et $(b, ff) \in Acc_f(C')$.

Héritage des accès inter-plans d'une classe par ses sous-classes

Les sous-classes d'une classe héritent au sein de chaque contexte de ses accès inter-plans

$$\forall (C, C') \in \mathcal{C}^2, \forall f \in \mathcal{F} - \{\mathbf{base}\}, C \text{ sous-classe de } C' \Rightarrow Acc_f(C) \supseteq Acc_f(C')$$

Modalités de fourniture des valeurs des attributs et des relations du plan de base

Lors de l'héritage entre une description de base et une description fonctionnelle, il est possible de spécifier une modalité optionnelle de « fourniture de valeur » pour chaque attribut et extrémité de relations. Les deux extrémités d'une relation doivent être héritées avec la même modalité dans un plan fonctionnel. Une modalité de « fourniture de valeur » est héritée dans ses sous-classes au sein du contexte fonctionnel où elle s'applique.

Définition formelle des modalités de fourniture d'attributs et de relations du plan de base

$WB_f(C)$ est défini comme l'ensemble des attributs de la description de base d'une classe dont la valeur est fournie par un contexte fonctionnel.

$WB_f(C): \mathcal{C} \rightarrow 2^A$

$$C \rightarrow \{a_1, a_2, \dots, a_n \text{ tels que } \forall i \in 1..n, a_i \in D_{\text{base}}(C)\}$$

Si les occurrences d'une relation du plan de base $R \in \mathcal{R}$ et représentée par $((C, a), (C', b))$ avec $a \in D_{\text{base}}(C)$ et $b \in D_{\text{base}}(C')$ sont fournies par le contexte correspondant à f à partir des classes CC et CC' respectivement sous-classes de C et C' alors $a \in WB_f(CC)$ et $b \in WB_f(CC')$.

Héritage des modalités de fourniture d'une classe par ses sous-classes

Les sous-classes d'une classe héritent dans chaque fonction de ses modalités de fourniture de propriétés du plan de base.

$$\forall (C, C') \in \mathcal{C}^2, \forall f \in \mathcal{F} - \{\text{base}\}, C \text{ sous-classe de } C' \Rightarrow WB_f(C) \supseteq WB_f(C')$$

Etat fonctionnel d'une instance

L'état fonctionnel d'une instance est l'ensemble des contextes fonctionnels auxquels elle participe. Celui-ci est déterminé pour chaque contexte fonctionnel par un ensemble de trois contraintes de participation qui sont pour une instance :

- la vérification du prédicat de participation de sa classe d'instanciation pour ce contexte ;
- la participation aux contextes fournisseurs des accès inter-plans définis et hérités dans sa classe d'instanciation et dont le contexte dont on cherche à déterminer la participation est consommateur ;
- la vérification de toutes les contraintes de cardinalité des relations dont sa classe d'instanciation ou une de ses surclasses est à une extrémité, et qui sont partagées dans le contexte dont on cherche à déterminer la participation. Cette vérification s'opère en ne prenant en compte que les instances participant à ce contexte des classes situées à l'autre extrémité des relations partagées.

Définition formelle de l'état fonctionnel d'une instance

L'état fonctionnel d'une instance est défini par la fonction suivante :

$$\begin{aligned}
 \mathcal{EF}: \quad & \mathcal{O} \rightarrow 2^{\mathcal{F}-\{\mathbf{base}\}} \\
 & o \rightarrow \{f \in \mathcal{F}-\{\mathbf{base}\} \text{ telle que} \\
 & \quad \bullet \text{PF}_f(\text{classe}(o)) \neq \emptyset \\
 & \quad \bullet \text{Pred}_f(\text{classe}(o)) \text{ soit vérifié pour } o \\
 & \quad \bullet \forall ff \in \mathcal{F}-\{\mathbf{base}\} \text{ telle que} \\
 & \quad \exists a \in \text{PF}_{ff}(\text{classe}(o)) \text{ tel que } (a, ff) \in \text{Acc}_f(\text{classe}(o)), \\
 & \quad ff \in \mathcal{EF}(o) \\
 & \quad \bullet \forall R \in \mathcal{R} \text{ représentée par } ((C, a), (C', b)) \text{ telle que } (C, C') \in \mathcal{C}^2 \wedge \\
 & \quad \text{classe}(o) \text{ sous-classe de } C \wedge \\
 & \quad (a, b) \in D_{\text{base}}(C) \times D_{\text{base}}(C'), \\
 & \quad \text{card-min}_C(R) \leq | \{ (u, v) \in \text{extension}(R) \text{ tel que } u = o \wedge f \in \mathcal{EF}(v) \} | \\
 & \quad \bullet \forall R \in \mathcal{R} \text{ représentée par } ((C, a), (C', b)) \text{ telle que } (C, C') \in \mathcal{C}^2 \wedge \\
 & \quad \text{classe}(o) \text{ sous-classe de } C \wedge \\
 & \quad \exists ff \in \mathcal{F}-\{f, \mathbf{base}\}, (a, b) \in \text{PF}_{ff}(C) \times \text{PF}_{ff}(C') \wedge \\
 & \quad (a, ff) \in \text{Acc}_f(C), (b, ff) \in \text{Acc}_f(C'), \\
 & \quad \text{card-min}_C(R) \leq | \{ (u, v) \in \text{extension}(R) \text{ tel que } u = o \wedge f \in \mathcal{EF}(v) \} | \\
 & \quad \}
 \end{aligned}$$

Extension fonctionnelle d'une classe

L'extension fonctionnelle d'une classe dans un contexte est l'ensemble de ses instances participant à ce contexte.

Définition formelle de l'extension fonctionnelle d'une classe

L'extension fonctionnelle d'une classe $\text{EF}_f(C)$ ($f \in \mathcal{F}-\{\mathbf{base}\}$ et $\text{PF}_f(C) \neq \emptyset$) est définie par la fonction suivante :

$$\begin{aligned}
 \text{EF}_f: \quad & \mathcal{C} \rightarrow 2^{\mathcal{O}} \\
 & C \rightarrow \{o \in \text{extension}(C) \text{ tel que } f \in \mathcal{EF}(o)\}
 \end{aligned}$$

Propriété 4 : *L'extension fonctionnelle d'une sous-classe d'une classe est incluse dans l'extension fonctionnelle de cette classe.*

Cette propriété s'écrit :

$$\begin{aligned}
 & \forall (C, C') \in \mathcal{C}^2, \forall f \in \mathcal{F}-\{\mathbf{base}\} \text{ avec } \text{PF}_f(C') \neq \emptyset \\
 & C \text{ sous-classe de } C' \Rightarrow \text{EF}_f(C) \subseteq \text{EF}_f(C')
 \end{aligned}$$

En effet, on peut écrire :

$$\text{EF}_f(C) = \{o \in \text{extension}(C) \text{ tel que } f \in \mathcal{EF}(o)\} \text{ car on sait que } \text{PF}_f(C) \neq \emptyset \text{ vu que } \text{PF}_f(C) \leq \text{PF}_f(C') \text{ et que } \text{PF}_f(C') \neq \emptyset.$$

De même, on peut écrire :

$$\begin{aligned}
 & \text{EF}_f(C') = \{o \in \text{extension}(C') \text{ tel que } f \in \mathcal{EF}(o)\} \\
 & \text{Comme } \text{extension}(C) \subseteq \text{extension}(C'), \text{ si } o \in \text{EF}_f(C) \text{ alors } o \in \text{EF}_f(C') \\
 & \text{Par conséquent, } \text{EF}_f(C) \subseteq \text{EF}_f(C')
 \end{aligned}$$

Héritage du prédicat de participation

Pour garantir que, au sein d'un contexte, les instances d'une classe vérifient le prédicat de participation de toutes ses surclasses, il faut que le prédicat de participation de chacune de ses surclasses soit hérité dans cette classe.

Règle d'héritage du prédicat de participation

Le prédicat de participation d'une classe à un contexte est la conjonction de tous les prédicats de participation à ce même contexte de ses surclasses et du prédicat de participation qui lui est propre.

Contexte de base

Le contexte de base est défini par l'ensemble des descriptions de base des classes du domaine auxquelles on associe leurs extensions.

Définition formelle du contexte de base

Le contexte de base est défini formellement de la façon suivante :

$$C_{\text{base}} = \{(D_{\text{base}}(C), \text{extension}(C)), C \in \mathcal{C}\}$$

Contextes fonctionnels

Un contexte fonctionnel est formé par l'ensemble des descriptions fonctionnelles (modalités de fourniture des valeurs des propriétés de base incluses), accès inter-plans et extensions fonctionnelles des classes intervenant dans la fonction correspondante.

Définition formelle d'un contexte fonctionnel

Un contexte fonctionnel est défini formellement de la façon suivante :

$$CF_f (f \in \mathcal{F} - \{\mathbf{base}\}) = \{(DF_f(C), WB_f(C), Acc_f(C), EF_f(C)), C \in \mathcal{C} \wedge (PF_f(C) \neq \emptyset)\}$$

Contextes d'instanciation d'une classe

Pour une classe du domaine, il existe un ou plusieurs contextes dont les clients peuvent instancier cette classe ainsi que détruire ses instances.

Un contexte d'instanciation d'une classe est aussi un contexte d'instanciation pour ses sous-classes.

Dépendance entre contextes due aux accès inter-plans

Pour participer à un contexte fonctionnel, une instance doit participer à tous les contextes fournisseurs des accès inter-plans définis et hérités dans sa classe d'instanciation et dont le contexte dont on cherche à déterminer la participation est consommateur. Par conséquent, pour une classe du domaine, tout contexte consommateur d'un accès inter-plans dépend du contexte fournisseur de cet accès inter-plans.

Intégrité référentielle

Vérification de l'intégrité référentielle pour une relation du plan de base

Soit $R \in \mathcal{K}$ une relation du plan de base représentée par le couple $((C,a),(C',b))$ tel que

$$(C, C') \in \mathcal{C}^2 \text{ et } (a, b) \in D_{\text{base}}(C) \times D_{\text{base}}(C')$$

L'intégrité référentielle doit être à nouveau vérifiée après une transaction dont l'action est de supprimer une instance o de C pour laquelle il existe des occurrences de R l'ayant pour extrémité.

Nous préfixons par « old » un ensemble ou une propriété pour désigner sa valeur avant la transaction. De même, nous préfixons par « new » la valeur d'un ensemble ou d'une propriété après la transaction.

Soit $o \in \text{extension}(C)$. La suppression de o a pour conséquence :

$$\text{new extension}(C) = \text{old extension}(C) - \{o\}$$

La gestion de l'intégrité référentielle au sein du contexte de base provoque la suppression de toutes les occurrences de R ayant o pour extrémité :

$$\text{new extension}(R) = \text{old extension}(R) - \{(u, v) \in \text{old extension}(R) \text{ tel que } u = o\}$$

Enfin, l'extension de C' reste inchangée :

$$\text{new extension}(C') = \text{old extension}(C')$$

L'inclusion suivante est par conséquent vérifiée après la transaction, et donc l'intégrité référentielle de R :

$$\text{new extension}(R) \subseteq \text{new extension}(C) \times \text{new extension}(C')$$

Vérification de l'intégrité référentielle pour une relation d'un plan fonctionnel

Soit $R \in \mathcal{R}$ une relation introduite dans un plan fonctionnel P_f ($f \in \mathcal{F} - \{\text{base}\}$) représentée par le couple $((C, a), (C', b))$ tel que $(C, C') \in \mathcal{C}^2$ et $(a, b) \in PF_f(C) \times PF_f(C')$

R vérifie au sein du contexte fonctionnel CF_f l'inclusion suivante :

$$\text{extension}(R) \subseteq EF_f(C) \times EF_f(C')$$

Cette inclusion garantit l'intégrité référentielle au sein de CF_f , traduite ici par le fait que tout couple de la relation appartient au produit cartésien des extensions fonctionnelles des deux classes.

Soit $o \in EF_f(C)$. La suppression de o ou le retrait de participation de o au contexte CF_f a pour conséquence sa suppression de l'extension fonctionnelle de C :

$$\text{new } EF_f(C) = \text{old } EF_f(C) - \{o\}$$

La gestion de l'intégrité référentielle au sein du contexte fonctionnel CF_f provoque la suppression de toutes les occurrences de R ayant o pour extrémité :

$$\text{new extension}(R) = \text{old extension}(R) - \{(u, v) \in \text{old extension}(R) \text{ tel que } u = o\}$$

Enfin, l'extension fonctionnelle de C' reste inchangée :

$$\text{new } EF_f(C') = \text{old } EF_f(C')$$

L'inclusion suivante est par conséquent vérifiée après la transaction, et donc l'intégrité référentielle de R au sein de CF_f :

$$\text{new extension}(R) \subseteq \text{new } EF_f(C) \times \text{new } EF_f(C')$$

Vérification de l'intégrité référentielle d'une relation partagée dans un contexte consommateur

Définition formelle de l'extension fonctionnelle d'une relation

Soit $R \in \mathcal{R}$ une relation représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in \text{intension}(C) \times \text{intension}(C')$. R peut être définie dans le plan de base ou dans un plan fonctionnel.

L'extension fonctionnelle $EF_f(R)$ ($f \in \mathcal{F} - \{\mathbf{base}\}$) telle que le contexte CF_f soit fournisseur ou consommateur de la relation R est définie par ;

$$EF_f: \quad \mathcal{R} \rightarrow 2^{\mathcal{O} \times \mathcal{O}}$$
$$R \rightarrow \text{extension}(R) \cap (EF_f(C) \times EF_f(C'))$$

L'extension fonctionnelle est la restriction de l'extension d'une R dans le contexte CF_f aux seules occurrences dont les deux extrémités participent à CF_f .

Si R est une relation locale au contexte CF_f et y vérifiant l'intégrité référentielle, son extension fonctionnelle dans CF_f est égale à son extension :

$$EF_f(R) = \text{extension}(R) \cap (EF_f(C) \times EF_f(C')) = \text{extension}(R) \text{ car } \text{extension}(R) \subseteq EF_f(C) \times EF_f(C'), R \text{ vérifiant l'intégrité référentielle dans le contexte } CF_f.$$

Utilisation de l'extension fonctionnelle d'une relation partagée pour vérifier son intégrité référentielle dans les contextes consommateurs

Soit $R \in \mathcal{R}$ une relation représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in \text{intension}(C) \times \text{intension}(C')$. R peut être définie dans le plan de base ou dans un plan fonctionnel.

La vérification de l'intégrité référentielle de R dans CF_f ($f \in \mathcal{F} - \{\mathbf{base}\}$) et telle que le contexte CF_f consomme la relation R se fait en considérant l'extension fonctionnelle $EF_f(R)$ à la place de son extension. Ainsi, on a la garantie qu'il n'existe pas d'occurrence de R ayant une extrémité absente de CF_f .

Contraintes de cardinalité

Vérification des contraintes de cardinalité dans le contexte de base

Soit $R \in \mathcal{R}$ une relation du plan de base représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in D_{\text{base}}(C) \times D_{\text{base}}(C')$.

$\text{card-min}_C(R)$, $\text{card-max}_C(R)$, $\text{card-min}_{C'}(R)$, $\text{card-max}_{C'}(R)$ sont définies comme dans le cas de la vérification des contraintes de cardinalité dans le cadre classique.

La vérification des contraintes de cardinalité dans le contexte de base se formalise par la vérification des inégalités suivantes :

$$\forall o \in \text{extension}(C), \text{card-min}_C(R) \leq |\{(u,v) \in \text{extension}(R) \text{ tel que } u=o\}| \leq \text{card-max}_C(R)$$
$$\forall o' \in \text{extension}(C'), \text{card-min}_{C'}(R) \leq |\{(u,v) \in \text{extension}(R) \text{ tel que } v=o'\}| \leq \text{card-max}_{C'}(R)$$

La violation est due à une transaction issue de l'un des contextes fournisseurs des occurrences de la relation. La violation peut provoquer soit un rejet de la transaction, soit l'invocation d'un traitement compensatoire qui est généralement réalisé dans le contexte fournisseur dont est issue la transaction.

Vérification des contraintes de cardinalité d'une relation localisée dans un contexte fonctionnel

Soit $R \in \mathcal{R}$ une relation localisée d'un plan fonctionnel représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in PF_f(C) \times PF_f(C')$.

$\text{card-min}_C(R)$, $\text{card-max}_C(R)$, $\text{card-min}_{C'}(R)$, $\text{card-max}_{C'}(R)$ sont définies comme dans le cas de la vérification des contraintes de cardinalité dans le cadre classique.

La vérification des contraintes de cardinalité dans un contexte fonctionnel est formalisée par la vérification des inégalités suivantes :

$$\forall o \in EF_f(C), \text{card-min}_C(R) \leq |\{(u,v) \in \text{extension}(R) \text{ tel que } u=o\}| \leq \text{card-max}_C(R)$$

$$\forall o' \in EF_f(C'), \text{card-min}_{C'}(R) \leq |\{(u,v) \in \text{extension}(R) \text{ tel que } v=o'\}| \leq \text{card-max}_{C'}(R)$$

La relation étant localisée dans un contexte fonctionnel, les clients de ce contexte en sont les seuls fournisseurs et il leur incombe de réaliser le traitement compensatoire, si le choix du rejet de la transaction n'est pas retenu.

Vérification des contraintes de cardinalité d'une relation partagée dans un contexte consommateur

Soit $R \in \mathcal{R}$ une relation représentée par le couple $((C,a),(C',b))$ telle que $(C,C') \in \mathcal{C}^2$ et $(a,b) \in \text{intension}(C) \times \text{intension}(C')$. R peut être définie dans le plan de base ou dans un plan fonctionnel.

$\text{card-min}_C(R)$, $\text{card-max}_C(R)$, $\text{card-min}_{C'}(R)$, $\text{card-max}_{C'}(R)$ sont définies comme dans le cas de la vérification des contraintes de cardinalité dans le cadre classique.

La vérification des contraintes de cardinalité dans un contexte fonctionnel CF_f ($f \in \mathcal{F} - \{\text{base}\}$) et telle que le contexte CF_f consomme la relation R , se formalise par la vérification des inégalités suivantes :

$$\forall o \in EF_f(C), \text{card-min}_C(R) \leq |\{(u,v) \in EF_f(R) \text{ tel que } u=o\}| \leq \text{card-max}_C(R)$$

$$\forall o' \in EF_f(C'), \text{card-min}_{C'}(R) \leq |\{(u,v) \in EF_f(R) \text{ tel que } v=o'\}| \leq \text{card-max}_{C'}(R)$$

Ces contraintes de cardinalité sont automatiquement vérifiées dans le contexte fonctionnel CF_f car les instances qui ne les vérifient pas, n'y participent pas. En effet, la vérification des contraintes de cardinalité des relations consommées par le contexte est l'une des trois contraintes de participation présentes dans la définition de l'état fonctionnel.

Contraintes d'intégrité statiques sur les propriétés des classes au sein d'un contexte fonctionnel

Chaque contrainte relative à une classe et à un contexte est définie sous la forme d'un prédicat qui ne peut être exprimé que sur les propriétés de la description fonctionnelle et les accès inter-plans de cette classe dans ce contexte.

Ce prédicat peut porter sur les propriétés d'autres classes s'il comporte des expressions de chemin sous-tendues par les relations inter-entités. Ces propriétés doivent être incluses dans les descriptions fonctionnelles et les accès inter-plans de ces autres classes relatives au même contexte.

Une contrainte d'intégrité statique sur les propriétés d'une classe au sein d'un contexte

fonctionnel doit être valide. Une telle contrainte est valide si et seulement si, en cas de violation, les clients du contexte où elle est définie peuvent toujours réaliser un traitement compensatoire qui la rende à nouveau vérifiée.

Combinaison des contraintes d'intégrité d'une base de données structurée en CROME

On note $CI_{C,f}$ ($C \in \mathcal{C}$, $f \in \mathcal{F} - \{\text{base}\}$) le prédicat qui représente la conjonction de toutes les contraintes d'intégrité du contexte fonctionnel CF_f pour les instances de la classe C (contraintes relatives aux relations du contexte et contraintes d'intégrité du contexte sur les propriétés). $CI_{C,\text{base}}$ représente la conjonction de toutes les contraintes d'intégrité du contexte de base pour les instances de la classe C (contraintes relatives aux relations du contexte de base).

Soit CI_C le prédicat qui représente la combinaison de toutes les contraintes des instances de la classe C vis-à-vis de l'ensemble des contextes.

La combinaison des contraintes d'intégrité peut s'écrire ainsi :

$\forall o \in \text{extension}(C),$

$$CI_C(o) = CI_{C,\text{base}}(o) \wedge_{f \in \mathcal{F} - \{\text{base}\}} ((f \in \mathcal{E}\mathcal{F}(o)) \Rightarrow CI_{C,f}(o))$$

Ainsi, si $f \in \mathcal{E}\mathcal{F}(o)$ est faux, $((f \in \mathcal{E}\mathcal{F}(o)) \Rightarrow CI_{C,f}(o))$ est vrai sans qu'il faille vérifier la valeur des contraintes d'intégrité pour ce contexte.

Prédicat de participation et contraintes de participation des contextes d'instanciation

Nous synthétisons ici les précisions concernant les prédicats de participation et les contraintes de participation des contextes d'instanciation.

Prédicat de participation

Un prédicat de participation ne s'exprime que sur les propriétés de la classe de l'instance qui sont soit héritées du plan de base soit définies dans une autre fonction. L'utilisation de propriétés définies dans un autre contexte se fait au travers d'accès inter-plans. Il n'est pas possible d'accéder au travers d'un accès inter-plans à une propriété d'une partie fonctionnelle relative à un contexte dépendant par d'autres accès inter-plans du contexte du prédicat de participation (cf 3.5.6.2).

Le prédicat peut porter sur des propriétés d'autres entités s'il comporte des expressions de chemin sous-tendues par des relations entre ces entités et qui sont partagées dans ce contexte. Ces propriétés doivent être incluses dans les descriptions fonctionnelles ou les accès inter-plans correspondant au même contexte.

L'utilisation d'une expression de chemin dans le prédicat de participation peut commencer mais ne doit pas passer ou se terminer par la classe de ce prédicat (cf 3.6.2.3).

Le prédicat de participation d'une classe à un contexte ne peut comporter une expression de chemin passant par une autre classe dont les instances sont sélectionnées dans ce même contexte par l'existence d'une relation partagée entre cette autre classe et la classe du prédicat. Cette sélection est due à la vérification des contraintes de cardinalité minimale des relations partagées (cf 3.6.3.3).

D'une façon générale, le prédicat de participation ne peut comporter de propriétés dont il détermine la valeur ou l'existence. Il ne peut donc porter sur l'extension fonctionnelle qu'il détermine ni sur les propriétés des parties fonctionnelles correspondantes.

Absence de contraintes de participation dans les contextes d'instanciation

Les contextes d'instanciation d'une classe font nécessairement participer toutes les instances de celle-ci.

Tout contexte d'instanciation d'une classe a pour prédicat de participation une tautologie (cf 3.5.5.3).

S'il existe des accès inter-plans pour une classe dans un contexte d'instanciation, ils doivent avoir pour contextes fournisseurs, des contextes qui vérifient aussi l'égalité entre extension fonctionnelle et extension, c'est-à-dire qui n'introduisent pas de contraintes de participation (cf 3.5.6.1).

Un contexte d'instanciation d'une classe détenant une ou plusieurs extrémités de relations du plan de base avec une cardinalité minimale supérieure ou égale à un, doit être fournisseur de ces relations (cf 3.6.3.2).

Un contexte d'instanciation d'une classe ne peut pas être consommateur au travers d'accès inter-plans d'une relation localisée dans un autre contexte fonctionnel, relation ayant cette classe pour extrémité avec une cardinalité minimale supérieure ou égale à un (cf 3.6.3.2).

Annexe D

Algorithme de calcul des extensions fonctionnelles

Rappel sur les contraintes de participation des instances

Nous avons établi en 1.6.2 que l'extension fonctionnelle est constituée par l'ensemble des instances d'une classe du domaine participant à la fonction correspondante. A partir de la définition de la fonction (cf 3.6.3.2) qui calcule l'état fonctionnel¹ d'une instance, nous pouvons rappeler les contraintes de participation d'une instance « o » à un contexte « CF » :

- « o » doit vérifier le prédicat de participation à « CF » qui lui est attribué par sa classe d'instanciation. Rappelons que ce prédicat est constitué de la conjonction du prédicat propre à la classe d'instanciation et du prédicat de toutes les surclasses de cette classe relatifs à « CF » ;
- pour tous les accès inter-plans de sa classe d'instanciation ayant « CF » pour contexte consommateur, « o » doit participer aux contextes fournisseurs de ces accès inter-plans ;
- pour chaque relation inter-entités partagée dont la classe d'instanciation de « o » est à l'une des extrémités, « o » vérifie pour cette extrémité la cardinalité minimale. Cette vérification prend en compte la vérification de l'intégrité référentielle des relations partagées dans les contextes consommateurs.

Ces contraintes de participation ont été formalisées au chapitre 3 (cf 3.6.3.2).

Détail de l'algorithme

L'algorithme est donné à la figure suivante. Il s'agit d'un algorithme qui évalue toutes les extensions fonctionnelles de toutes les classes du domaine.

L'algorithme est basé sur les extensions et les extensions fonctionnelles strictes. Rappelons que l'extension stricte d'une classe ne contient que les instances dont elle est la classe d'instanciation à l'exclusion des instances de ses sous-classes. L'extension fonctionnelle stricte ne contient que les instances de l'extension stricte participant à la fonction correspondante.

Remarque

Chaque mise à jour d'une instance peut provoquer une modification des extensions fonctionnelles de sa classe, voire des extensions fonctionnelles d'autres classes. Il convient donc d'effectuer, par conséquent, une mise à jour de ces extensions fonctionnelles. Or l'algorithme présenté n'est pas adapté à une telle tâche car il recalcule l'ensemble des extensions fonctionnelles au lieu de déterminer les instances à ajouter et à supprimer dans celles-ci.

1. Rappelons que l'état fonctionnel est l'ensemble des fonctions auxquelles participe une instance

```

-- Chaque classe du domaine est désignée par la lettre « C » suivie d'un indice de 1 à n.
-- Chaque fonction est désignée par la lettre « f » suivie d'un indice de 1 à m.
-- L'extension de la classe  $C_i$  est notée  $\text{extension}(C_i)$ 
-- L'extension stricte de la classe  $C_i$  est notée  $\text{extension\_stricte}(C_i)$ 
-- L'extension fonctionnelle de la classe  $C_i$  relative à la fonction  $f_j$  est notée  $\text{EF}(C_i, f_j)$ 
-- L'extension fonctionnelle stricte de la classe  $C_i$  relative à la fonction  $f_j$  est notée  $\text{EFS}(C_i, f_j)$ 

pour chaque classe C de {  $C_1, C_2, \dots, C_n$  } faire
  pour chaque fonction f de {  $f_1, f_2, \dots, f_m$  } telle que C soit enrichie relativement à f faire
    début
       $\text{EFS}(C, f) = \{\}$ 
       $\text{EF}(C, f) = \{\}$ 
      pour chaque instance i de  $\text{extension\_stricte}(C)$  faire
        si i vérifie les prédicats(*) de participation de C dans f alors
--(*) signifie qu'il faut prendre en compte les prédicats de participation des surclasses de C
           $\text{EFS}(C, f) = \text{EFS}(C, f) \cup \{i\};$ 
           $\text{EF}(C, f) = \text{EF}(C, f) \cup \{i\};$ 
          Ajouter i à l'extension fonctionnelle relative à f de toutes les sous-classes de C
      fin
    fin

répéter
  DoitEncoreCalculer = faux
  pour chaque classe C de {  $C_1, C_2, \dots, C_n$  } faire
    pour chaque fonction f de {  $f_1, f_2, \dots, f_m$  } telle que C soit enrichie relativement à f faire
      début
        pour chaque instance i de  $\text{EFS}(C, f)$  faire
          début
            pour chaque accès inter-plans(*) de C dans f faire
--(*) signifie qu'il faut prendre en compte les accès inter-plans hérités dans C
              si i n'appartient pas à  $\text{EFS}(C, \text{fonction\_fournisseuse\_accès})$  alors
                début
                   $\text{EFS}(C, f) = \text{EFS}(C, f) - \{i\};$ 
                   $\text{EF}(C, f) = \text{EF}(C, f) - \{i\};$ 
                  Retirer i de l'extension fonctionnelle relative à f de toutes les sous-classes de C
                  DoitEncoreCalculer = vrai;
                  Passer au calcul de l'instance suivante de  $\text{EFS}(C, f)$ ;
                fin
              fin
            pour chaque relation_partagee(*) R entre C et une classe C'
--(*) signifie qu'il faut prendre en compte les relations partagés dont une extrémité est héritée dans C
              si ( $\text{cardinalite\_minimale de extr}(R, C) > 0$ ) et
              --  $\text{extr}(R, C)$  désigne l'extrémité de la relation R dans la classe C
                ( $\text{cardinalite}(i.\text{extr}(R, C) \cap \text{EF}(C', f)) < \text{cardinalite\_minimale de extr}(R, C)$ )
              -- i. $\text{extr}(R, C)$  désigne l'ensemble des instances de C' avec lesquelles i est associée dans le contexte fournisseur
              -- de la relation, c'est-à-dire la valeur de l'extrémité pour i dans le contexte fournisseur
                début
                   $\text{EFS}(C, f) = \text{EFS}(C, f) - \{i\};$ 
                   $\text{EF}(C, f) = \text{EF}(C, f) - \{i\};$ 
                  Retirer i de l'extension fonctionnelle relative à f de toutes les sous-classes de C
                  DoitEncoreCalculer = vrai;
                  Passer au calcul de l'instance suivante de  $\text{EFS}(C, f)$ ;
                fin
              fin
            fin
          fin
        fin
      fin
    fin
  fin
  jusqu'à non(DoitEncoreCalculer);

```

Algorithme de calcul des extensions fonctionnelles des classes du domaine

Bibliographie

- [Abiteboul91] S. Abiteboul and A. Bonner, *Objects and Views*, Proceedings of SIGMOD, pp 238-247, May 1991
- [Ahmed91] R. Ahmed, P. De Smedt, W. Du, W. Kent, M. A. Ketabchi, W. A. Litwin, A. Rafii and M-C. Shan, *The Pegasus Heterogeneous Multidatabase System*, IEEE computer, vol 24 n°12, pp 19-27, December 1991
- [Albano93] A. Albano, R. Bergamini, G. Ghelli and R. Orsini, *An Object Data Model with Roles*, Proceedings of the 19th International Conference on Very Large Databases (VLDB'93), pp 39-51, Dublin, Ireland, August 1993
- [Al-Jadir93] L. Al-Jadir, G. Falquet, M. Léonard, *Context Versions in an Object-Oriented Model*, Proceedings of the DEXA Conference, 1993
- [Al-Jadir94] L. Al-Jadir, A. Le Grand, M. Léonard, O. Parchet, *Contribution to the evolution of information systems*, in *Methods and Associated Tools for the Information Systems Lifecycle*, A.A. Verrijn-Stuart and T.W. Olle (Editors), IFIP, Elsevier, 1994
- [Al-Jadir95] L. Al-Jadir, T. Estier, G. Falquet, M. Léonard, *Evolution Features of the F2 OODBMS*, Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95), T. Wang Ling and Y. Masunaga (Editors), Singapore, 1995
- [Andersson93a] M. Andersson, Y. Dupont, S. Spaccapietra, K. Yétongnon, M. Tresch, H. Ye, *The FEMUS Approach in Building a Federated Multilingual Database System*, Third International Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS'93), Vienna, Austria, April 1993
- [Andersson93b] M. Andersson, Y. Dupont, S. Spaccapietra, K. Yétongnon, M. Tresch, H. Ye, *FEMUS: A Federated Multilingual Database System*, Advanced Database Systems, N.R. Adam and B.K. Bhargava (Editors), Springer-Verlag, New York, 1993
- [ANSI75] ANSI/X3/SPARC, *Study Group on Database Management Systems*, ACM Sigmod, vol 7 n°2, 1975.
- [Bancilhon92] F. Bancilhon and C. Delobel, *Building an Object Oriented Database System : The story of O₂*, Morgan-Kaufmann, 1992
- [Batini92] C. Batini, S. Ceri and S. Navathe, *Conceptual Database Design. An Entity-Relationship Approach*, The Benjamin/Cummings Publishing Company, 1992
- [Bauzer Medeiros91] C. Bauzer Medeiros and P. Pfeffer, *Object Integrity Using Rules*, Proceedings of ECOOP'91, P. America (Editor), Springer-Verlag, pp 219-230, Geneva, Switzerland, July 1991
- [Bertino92] E. Bertino, *A View Mechanism for Object-Oriented Databases*, Proceedings of the third Conference on Extending Database Technology, pp 136-151, Vienna, 1992

- [Bertino93] E. Bertino and L. Martino, *Object-Oriented Database Systems. Concepts and Architectures*, International Computer Science Series, Addison-Wesley, 1993
- [Bertino96] E. Bertino and A. Illarramendi, *The Integration of Heterogeneous Data Management Systems: Approaches Based on the Object-Oriented Paradigm*, in *Object-Oriented Multidatabase Systems. A solution for Advanced Applications*, O.A. Bukhres and A.K. Elmagarmid (Editors), Prentice-Hall, 1996
- [Biskup86] J. Biskup and B. Convent, *A Formal View Integration Method*, Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 141-156, 1986
- [Bloom87] T. Bloom and S.B. Zdonik, *Issues in the Design of Object-Oriented Database Programming Languages*, Proceeding of OOPSLA'87, ACM, pp 441-451, 1987
- [Bouaziz95] T. Bouaziz, *Classification des contraintes d'intégrité dans les bases de données orientées-objet*, Ingénierie des systèmes d'information, vol 3 n°6, Hermès (Editeur), pp 713-737, 1995
- [Borgida89] A. Borgida, R. Brachman, D. McGuinness and L.A. Resnick, *CLASSIC: A Structural Data Model for Objects*, SIGMOD Record, vol 18 n°2, pp 59-67, 1989
- [Boulanger97] D. Boulanger et G. Dubois, *Objets et coopération de systèmes d'information*, dans *Ingénierie objet. Concepts et techniques.*, C. Oussalah et al. (Editeurs), InterEditions, 1997
- [Brodie92] M. Brodie and S. Ceri, *On Intelligent and Cooperative Information Systems: a workshop summary*, Int. Journal of Intelligent and Cooperative Information Systems, vol 1 n° 2, pp 249-289, 1992
- [Busse94] R. Busse, P. Fankhauser and E. J. Neuhold, *Federated Schemata in ODMG*, 2nd International East/West Database Workshop, pp 356-379, 1994
- [Carré89] B. Carré, *Méthodologie orientée objet pour la représentation des connaissances, concepts de point de vue, de représentation multiple et évolutive d'objet*, Thèse d'informatique, LIFL, Université de Lille 1, janvier 1989
- [Carré90a] B. Carré and J.M. Geib, *The Point of View notion for Multiple Inheritance*, Proceedings of OOPSLA'90, pp 312-321, 1990
- [Carré90b] B. Carré, L. Dekker and J.M. Geib, *Multiple and Evolutive Representation in the ROME Language*, Proceedings of the Second International Conference TOOLS, J. Bezivin, B. Meyer, J.M. Nerson (Editors), Paris, 1990
- [Carré91] B. Carré and L. Dekker, *Inheriting Object-Oriented Features through Meta-Programming, A Frame Extension to ROME*, Proceedings of the East European Conference on Object-Oriented Programming, Bratislava, Czecho-Slovakia, 1991
- [Carré96] B. Carré et G. Vanwormhoudt, *Conception modulaire en CROME. Le cas de l'interfaçage graphique d'objets*, IHM'96, <http://www-ihm96.imag.fr>, 1996

- [Cattel97] R.G.G. Cattel, *Bases de données orientées objets*, 2e édition, Addison-Wesley, 1997
- [Chambers93] C. Chambers, *Predicate Classes*, Proceedings of ECOOP'93, pp 268-296, 1993
- [Charpentier97] P. Charpentier, *Organisation et gestion de l'entreprise*, Editions Nathan, 1997
- [Chan95] D.K.C. Chan and D.A.Kerr, *Improving One's Views of Object-Oriented Databases*, in *Object-Oriented Technology for Database and Software Systems*, V.S. Alagar and R. Missaoui (Editors), World Scientific Publishing, pp 186-205, 1995
- [Chen76] P. P. Chen, *The Entity-Relationship Model: Towards a Unified View of Data*, ACM Transactions on Database Systems, vol 1 n°1, pp 9-37, March 1976
- [Codd70] E.F. Codd, *A relational model for large shared data banks*, Communication of the ACM 13:6, pp 377-387, 1970
- [Dayal89] U. Dayal, *Queries and Views in an Object-Oriented Data Model*, 2nd International Workshop on Database Programming Languages, 1989
- [Debrauwer97] L. Debrauwer, G. Vanwormhoudt et B. Carré, *Un cadre de conception par contextes fonctionnels de systèmes d'information à objets*, XVème Congrès INFORSID, Toulouse, juin 1997
- [Dekker92] L. Dekker and B. Carré, *Multiple and Dynamic Representation of frames with Points of View in FROME*, Actes de la 1ère Conférence sur la Représentation par Objets, La Grande Motte, EC2 (Editeur), juin 1992
- [Dekker94] L. Dekker, *FROME : Représentation multiple et classification d'objets avec points de vue*, Thèse d'informatique, LIFL, Université de Lille 1, 1994
- [Delannoy94] X. Delannoy, *La cohérence dans les bases de données*, Rapport de recherche RR 936 I, IMAG, octobre 1994
- [Dobrovnik94] M. Dobrovnik and J. Eder, *Adding View Support to ODMG-93*, Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS'94), Moscow, Russia, May 1994
- [Ducournau95] R. Ducournau, M. Habib, M. Huchard, M.L. Mugnier et A. Napoli, *Le point sur l'heritage multiple*, Techniques et Sciences Informatiques, vol 14 n°3, pp 309-345, 1995
- [Falquet88] G. Falquet, J. Guyot, M. Junet, M. Léonard, R. Bursens, P. Crausaz and I. Prince, *Concept Integration as an Approach to Information System Design*, in *Computerized Assistance during the information systems life cycle*, T.W. Olle, A.A. Verrijn-Stuart and I. Bhabuta (Editors), IFIP, North-Holland, pp 19-65, 1988
- [Ferber89] J. Ferber, *Objets et agents : une étude des structures de représentation et de communications en Intelligence Artificielle*, Thèse d'Etat, Université Paris VI, juin 1989

- [García-Solaco96] M. García-Solaco, F. Saltor and M. Castellanos, *Semantic Heterogeneity in Multidatabase Systems*, in *Object-Oriented Multidatabase Systems. A solution for Advanced Applications*, O.A. Bukhres and A.K. Elmagarmid (Editors), Prentice-Hall, 1996
- [Gardarin93] G. Gardarin, *Maîtriser les base de données*, Editions Eyrolles, 1993
- [Geppert93] A. Geppert, S. Scherrer, K. Dittrich, *Derived Types and Subschemas: Towards Better Support for Logical Data Independence in Object-Oriented Data Models*, Technical Report 93.27, Institut für Informatik, Universität Zürich, 1993
- [Golberg83] A. Golberg and D. Robson, *Smalltalk-80, the Language and its Implementation*, Massachusetts, Addison-Wesley, 1983
- [Heiler90] S. Heiler and S. Zdonik, *Object Views: Extending the Vision*, Proceedings of IEEE Data Engineering Conference, Los Angeles, pp 86-93, 1990
- [Kaul90] M. Kaul, K. Drosten and E.J. Neuhold, *ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views*, IEEE Data Engineering, pp 2-10, July 1990
- [Kent91] W. Kent, *A rigorous model of object reference, identity and existence*, Journal of Object-Oriented Programming, vol 4 n°3, pp 28-36, June 1991
- [Khoshafian86] S. Khoshafian and G.P. Copeland, *Object Identity*, Proceedings of OOPSLA'86, pp 406-416, 1986
- [Kim90] W. Kim, *Introduction to Object-Oriented Databases*, The MIT Press, 1990
- [Kim93] W. Kim, *Observations on the ODMG-93 Proposal for an Object-Oriented Database Language*, SIGMOD Record, vol 23, n°1, March 1994
- [Kim95] W. Kim (Editor), *Modern Database System. The Object Model, Interoperability, and Beyond*, Addison-Wesley, 1995
- [Klas90] W. Klas, E.J. Neuhold and M. Schrefl, *Metaclasses in VODAK and their Application in Database Integration*, GMD Technical Report n°462, Sankt Augustin, 1990
- [Larson89] J.. Larson, S. Navathe and R. Elmasri, *A Theory of Attribute Equivalence in Databases with Application to Schema Integration*, IEEE Transactions on Software Engineering, vol 15 n°4, April 1989
- [Mamou91] J.-C. Mamou and C. Bauzer Medeiros, *Interactive Manipulation of Object-Oriented Views*, Proceedings of the 7th International Conference of Data Engineering, pp 60-69, Kobe, Japan, April 1991 in IEEE Computer Society Press, Loas Alamitos, California, 1991
- [Meyer88] B. Meyer, *Object-Oriented Software Construction*, Englewood Cliffs NJ Prentice Hall, 1988
- [Missikoff89] M. Missikoff and M. Scholl, *An Algorithm for Insertion into a Lattice: Application to Type Classification*, Proceedings of the third international Conference on Foundations of Data Organisation and Algorihtms (FODO), pp 64-82, Paris, June 1989

- [Morejon94] J. Morejon, *Merise. Vers une modélisation orientée objet*, Collection Ingénierie des Systèmes d'Information, Les Editions d'Organisation, 1994
- [Nassif91] R. Nassif, Y. Qiu and J. Zhu, *Extending The Object-Oriented Paradigm to Support Relationships and Constraints*, in *Object-Oriented Databases: analysis, Design & Construction (DS-4)*, R.A. Meersman, W. Kent and S. Khosla (Editors), IFIP, Elsevier Science Publishers, 1991
- [Navathe86] S. Navathe, R. Elmasri and J. Larson, *Integrating User Views in Database Design*, IEEE computer, vol 19 n°1, pp 50-62, 1986
- [Nguyen91] Hong-Mai Nguyen und Christian Laasch, *Einordnung von Klassen und Views in eine Klassenhierarchie*, Diplomarbeit, ETH Zürich, 1991
- [Niemeyer97] Patrick Niemeyer and Joshua Peck, *Exploring Java, 2nd Edition*, O'REILLY, 1997
- [ODMG93] R.G.G. Cattell (Editor), T. Atwood, J. Dubl, G. Ferran, M. Loomis and D. Wade, *The Object Database Standard: ODMG-93*, Morgan Kaufmann, 1993
- [ODMG97] R.G.G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland and D. Wade, *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, 1997
- [Orfali96] R. Orfali, D. Harkey and J. Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, Inc., 1996
- [Parent89] C. Parent, H. Rolin, K. Yetongnon and S. Spaccapietra, *An ER Calculus for the Entity-Relationship Complex Model*, Proceedings of the 8th International Conference on Entity-Relationship Approach, Toronto, pp 361-384, 1989
- [Peterson77] J.L. Peterson, *Petri Nets*, Computing Surveys, vol 9 n°3, pp 223-252, September 1977
- [Pernici90a] B. Pernici, *Requirements specifications for object-oriented systems*, Nouvelles perspectives des systèmes d'information, A. Flory et C. Rolland (Editeurs), INFORSID, pp 41-63, 1990
- [Pernici90b] B. Pernici, *Objects with Roles*, Proceedings of the IEEE Conference on Office Information Systems, pp 205-215, 1990
- [Pitoura96] E. Pitoura, O. Bukhres and A.K. Elmagarmid, *Object-Oriented Multidatabase Systems: An Overview*, in *Object-Oriented Multidatabase Systems. A solution for Advanced Applications*, O.A. Bukhres and A.K. Elmagarmid (Editors), Prentice-Hall, 1996
- [POET97] POET Software Corporation, *POET Java SDK - Programmer's Guide*, 1997
- [Ra95] Y.G. Ra and E.A. Rundensteiner, *A Transparent Object-Oriented Schema Change Using View Evolution*, Proceedings of the IEEE International Conference on Data Engineering, (ICDE-11), Taipei, Taiwan, March 1995

- [Richardson91] J. Richardson and P. Schwarz, *Aspects: Extending Objects to Support Multiple, Independant Roles*, Proceedings of the ACM-SIGMOD International Conference on Management of Data, ACM. Sigmod Record, vol 20, pp 298-307, Denver, Colorado, May 1991
- [Rumbaugh87] J. Rumbaugh, *Relations as Semantic Constructs in an Object-Oriented Language*, Proceeding of OOPSLA'87, ACM, pp 466-481, 1987
- [Rumbaugh95] J. Rumbaugh et al., *OMT, Modélisation et conception orientées objet*, Masson et Prentice Hall, 1995
- [Rumbaugh96] J. Rumbaugh, *A search for values: Attributes and associations*, Journal of Object-Oriented Programming, vol 9 n°3, June 1996
- [Rundensteiner92a] E.A. Rundensteiner, *A methodology for Supporting Multiple Views in Object-Oriented Databases*, Proceedings of the 18th International Conference on very large databases (VLDB'92), pp 187-198, Vancouver, Canada, August 1992
- [Rundensteiner92b] E.A. Rundensteiner, *MultiView : A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases*, Information and Computer Science Department, University of California, Irvine, Tech. Rep. 92-07, 1992
- [Rundensteiner92c] E.A. Rundensteiner, *A Class Classification Algorithm For Supporting Consistent Object Views*, Information and Computer Science Department, University of California, Irvine, Tech. Rep. 92-50, 1992
- [Rundensteiner92d] E.A. Rundensteiner and L. Bic, *Automatic View Schema Generation in Object-Oriented Databases*, Information and Computer Science Department, University of California, Irvine, Tech. Rep. 92-15, 1992
- [Rundensteiner94] E.A. Rundensteiner, *A Classification Algorithm for Supporting Object-Oriented Views*, Proceedings of the ACM third International Conference on Information and Knowledge Management (CIKM'94), pp 18-25, November 1994
- [Saltor91] F. Salort, M. Castellanos and M. García-Solaco, *Suitability of data models as canonical models for federated databases*, SIGMOD Record, vol 20 n°4, pp 44-48, 1991
- [Scholl91] M. H. Scholl, C. Laash and M. Tresch, *Updatable Views in Object-Oriented Databases*, Proceedings of the Deductive and Object-Oriented Databases, Second International Conference, C. Delobel, M. Kifer and Y. Masunaga (Editors), Springer-Verlag, pp 189-205, Munich, December 1991
- [Scholl93] M. H. Scholl, H.-J. Schek and M. Tresch, *Object Algebra and Views for Multi-Objectbases*, in *Distributed Object Management*, M.T. Özsu, U. Dayal, P. Valduriez (Editors), pp 352-373, Morgan Kaufmann, 1993
- [Scholl94] M. H. Scholl, C. Laash, C. Rich, H.-J. Schek and M. Tresch, *The COCOON Object Model*, Technical Report 193, Departement of Computer Science, ETH Zürich, 1994
- [Shilling89] J. Shilling and P. Sweeney, *Three steps to views: extending the object-oriented paradigm?*, Proceedings of OOPSLA'89, pp 353-361, October 1989

- [Sheth90] A. Sheth and J. Larson, *Federated Database Systems for Managing Distributed Heterogeneous, and Autonomous Databases*, ACM Computing Surveys, vol 22 n°3, pp 183-236, September 1990
- [Snyder93] A. Snyder, *The essence of objects: concepts and terms*, IEEE Software, vol 10 n°1, pp 31-42, January 1993
- [Souza Dos Santos94] C. Souza Dos Santos, S. Abiteboul and C. Delobel, *Virtual schemas and bases*, Proceedings of the International Conference on Extending Database Technology (EDBT), 1994
- [Spaccapietra92] S. Spaccapietra, C. Parent and Y. Dupont, *Model Independent Assertions for Integration of Heterogeneous Schemas*, Very large databases Journal, vol 1 n°1, July 1992
- [Spaccapietra93] S. Spaccapietra and C. Parent, *View Integration: a step forward in solving structural conflicts*, IEEE Transactions on Data and Knowledge Engineering, 1993
- [Tanaka88] K. Tanaka, M. Yoshikawa and K. Ishihara, *Schema Virtualization in Object-Oriented Databases*, Proceedings of the IEEE Data Engineering Conference, 1988
- [Tardieu83] H. Tardieu, A. Rochfeld et R. Colletti, *La méthode MERISE, principes et outils*, Les éditions d'organisation, 1983
- [Tresch92] M. Tresch and M H. Scholl, *Meta Object Management and its Application to Database Evolution*, Proceedings of the 11th International Conference on Entity Relationship Approach, pp 299-321, 1992
- [Tresch93] M. Tresch and M H. Scholl, *Schema Transformation without Database Reorganization*, SigmodRecord, vol 22 n° 1, March 1993
- [Tresch94] M. Tresch and M H. Scholl, *A Classification of Multi-Database Languages*, Proceedings of the third International Conference on Parallel and Distributed Information Systems (PDIS), Austin, Texas, September 1994
- [Ullman89] J. D. Ullman, *Principles of database and knowledge-base systems*, Volume 1, Computer Science Press, 1989
- [Urban89] S. Urban and L. Delcambre, *Constraint Analysis for Specifying Perspectives of Class Objects*, Proceedings of the fifth International Conference on Data Engineering, pp 10-17, Los Angeles, IEEE, 1989
- [Vanwormhoudt97a] G. Vanwormhoudt, B. Carré et L. Debrauwer, *Programmation par objets et contextes fonctionnels. Application de CROME à Smalltalk*, Langages et modèles à objet, R. Ducournau et S. Garlatti (Editeurs), Roscoff, octobre 1997
- [Vanwormhoudt97b] G. Vanwormhoudt, *Programmation par contextes en Smalltalk*, Revue « L'Objet », Editions Hermes, vol 3 n°4, pp 429-444, décembre 1997
- [Vanwormhoudt99] G. Vanwormhoudt, *CROME : un cadre de programmation par objets structurés en contextes*, Thèse en cours, LIFL, Université de Lille 1, 1999

- [Wierenga91] R.J. Wieringa and W. de Jonge, *The identification of objects and roles*, Technical Report, IR-267, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1991
- [Ye95] X. Ye, C. Parent and S. Spaccapietra, *On the Specification of Views in DOOD Systems*, Proceeding of the Deductive and Object-Oriented Databases, Fourth International Conference, T. Wang Ling, A.O. Mendelzon and L. Vieille (Editors), Springer-Verlag, Singapore, December 1995